

UNIX™
for the
68000

VOLUME II
Program Development Tools

8/23/82

UNISON
CORPORATION
2405 Fourth Street © Berkeley, CA 94710

Copyright 1981, Bell Telephone Laboratories, Incorporated.
Holders of a UNIX(tm) software license are permitted to copy
this document, or any portion of it, as necessary for
licensed use of the software, provided this copyright notice
and statement of permission are included.

CODATA P/N: 03-0305-01 REV D 10/83

VOLUME II

Program Development Tools

Table of Contents

Part 1: System Programming Tools

1. An Introduction to the UNIX Shell
2. An Introduction to the C Shell
3. UNIX Programming - Second Edition
4. YACC: Yet Another Compiler-Compiler
5. SED: A Non-Interactive Text Editor
6. AWK: A Pattern Scanning and Processing Language (Second Edition)
7. LEX: A Lexical Analyzer Generator
8. DC: An Interactive Desk Calculator
9. BC: An Arbitrary Precision Desk-Calculator Language

Part 2: Program Maintenance

1. As Assembler Reference Guide
2. "C" Interface Notes for 68000 UNIX
3. The C Programming Language - Reference Manual
4. Screen updating and Cursor Movement Optimization: a Library Package
5. A Tutorial Introduction to ADB
6. LINT: A Program Checker
7. MAKE: A program for Maintaining Computer Programs
8. Source Code Control System User's Guide; Function and Use of an SCCS Interface Program

Part 3: UNIX Maintenance and Information

1. UNIX Implementation
2. FSCK: the UNIX File System Check Program
3. The UNIX I/O System
4. Regenerating System Software
5. A Tour Through the portable C Compiler
6. Uucp Implementation Description
7. A Dial-up Network of UNIX Systems

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The *shell* is a command programming language that provides an interface to the UNIX† operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while*, *if then else*, *case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through 'pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

November 12, 1978

†UNIX is a Trademark of Bell Laboratories.

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

1.0 Introduction

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section: see, for example, "UNIX for beginners".¹ Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form "see *pipe* (2)" are to a section of the UNIX manual.²

1.1 Simple commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument *-l* tells *ls* to print status information, size and the creation date for each file.

1.2 Background commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing *&* is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps* command.

1.3 Input output redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
ls -l >file
```

The notation *>file* is interpreted by the shell and is not passed as an argument to *ls*. If *file* does not exist then the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls*. Output may be appended to a file using the notation

```
ls -l >>file
```

In this case *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

```
wc <file
```

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words and lines found. If only the number of lines is required then

```
wc -l <file
```

could be used.

1.4 Pipelines and filters

The standard output of one command may be connected to the standard input of another by writing the 'pipe' operator, indicated by *|*, as in,

```
ls -l | wc
```

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

```
ls -l >file; wc <file
```

except that no *file* is used. Instead the two processes are connected by a pipe (see *pipe* (2)) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep*, selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from *ls* that contain the string *old*. Another useful filter is *sort*. For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints the number of file names in the current directory containing the string *old*.

1.5 File name generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints information relating to the file *main.c*.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates, as arguments to *ls*, all file names in the current directory that end in *.c*. The character *** is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

- * Matches any string of characters including the null string.
 - ? Matches any single character.
 - [...]
- Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters *a* through *z*.

```
/usr/fred/test/?
```

matches all names in the directory */usr/fred/test* that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all *core* files in sub-directories of */usr/fred*. (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of */usr/fred*.

There is one exception to the general rules given for patterns. The character *'.'* at the start of a file name must be explicitly matched.

```
echo *
```

will therefore echo all file names in the current directory not beginning with *'.'*

```
echo .*
```

will echo all those file names that begin with *'.'*. This avoids inadvertent matching of the names *'.'* and *'..'* which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files *'.'* and *'..'*.)

1.6 Quoting

Characters that have a special meaning to the shell, such as *< > * ? ! &*, are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a ** is *quoted* and loses its special meaning, if any. The ** is elided so that

```
echo \?
```

will echo a single *?*, and

```
echo \\\
```

will echo a single **. To allow long strings to be continued over more than one line the sequence *\newline* is ignored.

** is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4.

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is '\$ '. It may be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. If a newline is typed and further input is needed then the shell will issue the prompt '> '. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

```
PS2=more
```

1.8 The shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file *.profile* then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- **ls**
Print the names of files in the current directory.
- **ls > file**
Put the output from *ls* into *file*.
- **ls | wc -l**
Print the number of files in the current directory.
- **ls | grep old**
Print those file names containing the string *old*.
- **ls | grep old | wc -l**
Print the number of files whose name contains the string *old*.
- **cc pgm.c &**
Run *cc* in the background.

2.0 Shell procedures

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file*. Such a file is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in *file* using the positional parameters *\$1*, *\$2*, For example, if the file *wg* contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read*, *write* and *execute*. The UNIX command *chmod* (1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as *\$#*. The name of the file being executed is available as *\$0*.

A special shell parameter *\$** is used to substitute for all positional parameters except *\$0*. A typical use of this is to provide some default arguments, as in,

```
nroff -T450 -ms $*
```

which simply prepends some arguments to those already given.

2.1 Control flow - for

A frequent use of shell procedures is to loop through the arguments (*\$1*, *\$2*, ...) executing commands once for each argument. An example of such a procedure is *rel* that searches the file */usr/lib/telnos* that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *rel* is

```
for i
do grep $i /usr/lib/telnos; done
```

The command

```
rel fred
```

prints those lines in */usr/lib/telnos* that contain the string *fred*.

```
tel fred bert
```

prints those lines containing *fred* followed by those for *bert*.

The for loop notation is recognized by the shell and has the general form

```
for name in w1 w2 ...
do command-list
done
```

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. *name* is a shell variable that is set to the words *w1 w2 ...* in turn each time the *command-list* following **do** is executed. If **in w1 w2 ...** is omitted then the loop is executed once for each positional parameter; that is, **in \$*** is assumed.

Another example of the use of the for loop is the *create* command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation *>file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done**.

2.2 Control flow - case

A multiple way branch is provided for by the **case** notation. For example,

```
case $# in
  1) cat >>$1 ;;
  2) cat >>$2 <$1 ;;
  *) echo 'usage: append [ from ] to' ;;
esac
```

is an *append* command. When called with one argument as

```
append file
```

\$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
  pattern ) command-list ;;
  ...
esac
```

The shell attempts to match *word* with each *pattern*, in the order in which the patterns appear. If a match is found the associated *command-list* is executed and execution of the **case** is complete. Since ***** is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the commands following the second ***** will never be executed.


```
case $# in
  *) ... ;;
  *) ... ;;
esac
```

Another example of the use of the case construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```
for i
do case $i in
  -[ocs]) ... ;;
  -*) echo 'unknown flag $i' ;;
  *.c) /lib/c0 $i ... ;;
  *) echo 'unexpected argument $i' ;;
esac
done
```

To allow the same commands to be associated with more than one pattern the case command provides for alternative patterns separated by a |. For example,

```
case $i in
  -x|-y) ...
esac
```

is equivalent to

```
case $i in
  -[xy]) ...
esac
```

The usual quoting conventions apply so that

```
case $i in
  \?) ...
```

will match the character ?.

2.3 Here documents

The shell procedure *tel* in section 2.1 uses the file */usr/lib/telno* to supply the data for *grep*. An alternative is to include this data within the shell procedure as a *here* document, as in,

```
for i
do grep $i <<!
  ...
  fred mh0123
  bert mh0789
  ...
!
done
```

In this example the shell takes the lines between <<! and ! as the standard input for *grep*. The string ! is arbitrary, the document being terminated by a line that consists of the string following <<.

Parameters are substituted in the document before it is made available to *grep* as illustrated by the following procedure called *edg*.

```
ed $3 <<%  
g/$1/s//$2/g  
w  
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%  
g/string1/s//string2/g  
w  
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character \$ as in

```
ed $3 << +  
1,\$s/$1/$2/g  
w  
+
```

(This version of *edg* is equivalent to the first except that *ed* will print a ? if there are no occurrences of the string \$1.) Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<\  
...  
#
```

The document is presented without modification to *grep*. If parameter substitution is not required in a *here* document this latter form is more efficient.

2.4 Shell variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables *user*, *box* and *acct*. A variable may be set to the null string by saying, for example,

```
null =
```

The value of a variable is substituted by preceding its name with \$; for example,

```
echo $user
```

will echo *fred*.

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin  
mv pgm $b
```

will move the file *pgm* from the current directory to the directory */usr/fred/bin*. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

echo Suser

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of `ps` to the file `/tmp/psa`, whereas,

```
ps a >Stmpa
```

would cause the value of the variable `tmpa` to be substituted.

Except for `S?` the following are set initially by the shell. `S?` is set after executing each command.

- S?** The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under `if` and `while` commands.
- S#** The number of positional parameters (in decimal). Used, for example, in the `append` command to check the number of parameters.
- SS** The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/psSS
...
rm /tmp/psSS
```

- S!** The process number of the last process run in the background (in decimal).
- S-** The current shell flags, such as `-x` and `-v`.

Some variables have a special meaning to the shell and should be avoided for general use.

- SMAIL** When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file `.profile`, in the user's login directory. For example,

```
MAIL=/usr/mail/fred
```

- SHOME** The default argument for the `cd` command. The current directory is used to resolve file name references that do not begin with a `/`, and is changed using the `cd` command. For example,

```
cd /usr/fred/bin
```

makes the current directory `/usr/fred/bin`.

```
cat wn
```

will print on the terminal the file `wn` in this directory. The command `cd` with no argument is equivalent to

```
cd SHOME
```

This variable is also typically set in the the user's login profile.

- SPATH** A list of directories that contain commands (the *search path*). Each time a command is executed by the shell a list of directories is searched for an executable

file. If **\$PATH** is not set then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise **\$PATH** consists of directory names separated by **:**. For example,

```
PATH = :/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first **:**), **/usr/fred/bin**, **/bin** and **/usr/bin** are to be searched in that order. In this way individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a **/** then this directory search is not used; a single attempt is made to execute the command.

- \$PS1** The primary shell prompt string, by default, **'\$ '**.
- \$PS2** The shell prompt when further input is needed, by default, **'> '**.
- \$IFS** The set of characters used by *blank interpretation* (see section 3.4).

2.5 The test command

The *test* command, although not part of the shell, is intended for use by shell programs. For example,

```
test -f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here, see *test* (1) for a complete specification.

<code>test s</code>	true if the argument <i>s</i> is not the null string
<code>test -f file</code>	true if <i>file</i> exists
<code>test -r file</code>	true if <i>file</i> is readable
<code>test -w file</code>	true if <i>file</i> is writable
<code>test -d file</code>	true if <i>file</i> is a directory

2.6 Control flow - while

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

```
while command-list,  
do command-list,  
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop *command-list*₁ is executed; if a zero exit status is returned then *command-list*₂ is executed; otherwise, the loop terminates. For example,

```
while test $1  
do ...  
  shift  
done
```

is equivalent to

```
for i  
do ...  
done
```

shift is a shell command that renames the positional parameters **\$2**, **\$3**, ... as **\$1**, **\$2**, ... and loses **\$1**.

Another kind of use for the `while/until` loop is to wait until some external event occurs and then run some commands. In an `until` loop the termination condition is reversed. For example,

```
until test -f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control flow - if

Also available is a general conditional branch of the form,

```
if command-list
then  command-list
else  command-list
fi
```

that tests the value returned by the last simple command following `if`.

The `if` command may be used in conjunction with the `test` command to test for the existence of a file as in

```
if test -f file
then  process file
else  do something else
fi
```

An example of the use of `if`, `case` and `for` constructions is given in section 2.10.

A multiple test `if` command of the form

```
if ...
then ...
else if ...
      then ...
      else if ...
            ...
            fi
      fi
fi
```

may be written using an extension of the `if` notation as,

```
if ...
then ...
elif ...
then ...
elif ...
...
fi
```

The following example is the `touch` command which changes the 'last modified' time for a list of files. The command may be used in conjunction with `make (1)` to force recompilation of a list of files.

```
flag=
for i
do case $i in
  -c) flag=N ;;
  *) if test -f $i
     then ln $i junk$$; rm junk$$
     elif test $flag
     then echo file \"$i\" does not exist
     else >$i
     fi
  esac
done
```

The `-c` flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable `flag` is set to some non-null string if the `-c` argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```
if command1
then command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes `command2` only if `command1` fails. In each case the value returned is that of the last simple command executed.

2.8 Command grouping

Commands may be grouped in two ways,

```
{ command-list ; }
```

and

```
( command-list )
```

In the first `command-list` is simply executed. The second form executes `command-list` as a separate process. For example,

```
(cd x; rm junk )
```

executes `rm junk` in the directory `x` without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory `x`.

2.9 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

```
set -v
```

(*v* for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

```
sh -v proc ...
```

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the *-n* flag which prevents execution of subsequent commands. (Note that saying *set -n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

```
set -x
```

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

```
set -
```

and the current setting of the shell flags is available as *\$-*.

2.10 The man command

The following is the *man* command which is used to print sections of the UNIX manual. It is called, for example, as

```
man sh  
man -t ed  
man 2 fork
```

In the first the manual section for *sh* is printed. Since no section is specified, section 1 is used. The second example will typeset (*-t* option) the manual section for *ed*. The last prints the *fork* manual page from section 2.

```
cd /usr/man

: 'colon is the comment command'
: 'default is nroff ($N), section 1 ($s)'
N=n s=1

for i
do case $i in
  [1-9]*)    s=$i ;;
  -t) N=t ;;
  -n) N=n ;;
  -*) echo unknown flag \"$i\" ;;
  *) if test -f man$s/$i.$s
     then  ${N}roff man0/${N}aa man$s/$i.$s
     else  : 'look through all manual sections'
           found=no
           for j in 1 2 3 4 5 6 7 8 9
           do if test -f man$j/$i.$j
              then man $j $i
              found=yes
           fi
           done
     case $found in
       no) echo '$i: manual page not found'
     esac
  fi
esac
done
```

Figure 1. A version of the man command

3.0 Keyword parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

```
user=fred command
```

will execute *command* with *user* set to *fred*. The *-k* flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain they are available as positional parameters *\$1*, *\$2*,

The *set* command may also be used to set positional parameters from within a procedure. For example,

```
set - *
```

will set *\$1* to the first file name in the current directory, *\$2* to the next, and so on. Note that the first argument, *-*, ensures correct treatment when the first file name begins with a *-*.

3.1 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables *user* and *box* for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command.

```
readonly name ...
```

Subsequent attempts to set *readonly* variables are illegal.

3.2 Parameter substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable *d* is not set

```
echo $d
```

or

```
echo S{$d}
```

will echo nothing. A default string may be given as in

```
echo S{$d-.}
```

which will echo the value of the variable *d* if it is set and *.'* otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo S{$d-'.'}
```

will echo *.'* if the variable *d* is not set. Similarly

```
echo ${d-$1}
```

will echo the value of *d* if it is set and the value (if any) of *\$1* otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.
```

which substitutes the same string as

```
echo ${d-.
```

and if *d* were not previously set then it will be set to the string `'.'`. (The notation `${...=...}` is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable *d* if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
```

```
...
```

Colon (`:`) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables *user*, *acct* or *bin* are not set then the shell will abandon execution of the procedure.

3.3 Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is `/usr/fred/bin` then the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (``...``) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ``` must be escaped using a `\`. For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here documents*) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

```
case SA in
...
*.c)      B=`basename SA .c`
...
esac
```

that sets **B** to the part of **SA** with the suffix `.c` stripped.

Here are some composite examples.

- `for i in `ls -t`; do ...`
The variable `i` is set to the names of files in time order, most recent first.
- `set `date`; echo $6 $2 $3, $4`
will print, e.g., `1977 Nov 1, 23:59:59`

3.4 Evaluation and quoting

The shell is a macro processor that provides parameter substitution, command substitution and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in appendix A. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. `Suser`
- command substitution, e.g. ``pwd``
Only one evaluation occurs so that if, for example, the value of the variable `X` is the string `$y` then

```
echo SX
```

will echo `$y`.

- blank interpretation
Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string `SIFS`. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

```
echo ""
```

will pass on the null string as the first argument to `echo`, whereas

```
echo $null
```

will call `echo` with no arguments if the variable `null` is not set or set to the null string.

- file name generation
Each word is then scanned for the file pattern characters `*`, `?` and `[...]` and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a `for` loop. Only substitution occurs in the `word` used for a `case` branch.

As well as the quoting mechanisms described earlier using ``` and `'...'` a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using `\`.

<code>\$</code>	parameter substitution
<code>`</code>	command substitution
<code>"</code>	ends the quoted string
<code>\</code>	quotes the special characters <code>\$ ` "</code>

For example,

```
echo "Sx"
```

will pass the value of the variable *x* as a single argument to *echo*. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation *\$@* is the same as *\$** except when it is quoted.

```
echo "$@"
```

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

```
echo "$1" "$2" ...
```

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

		<i>metacharacter</i>				
	\	\$	*	'	"	`
'	n	n	n	n	n	t
"	y	n	n	t	n	n
`	y	y	n	y	t	n
t		terminator				
y		interpreted				
n		not interpreted				

Figure 2. Quoting mechanisms

In cases where more than one evaluation of a string is required the built-in command *eval* may be used. For example, if the variable *X* has the value *\$y*, and if *y* has the value *pqr* then

```
eval echo $X
```

will echo the string *pqr*.

In general the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who |grep'
$wg fred
```

is equivalent to

```
who |grep fred
```

In this example, *eval* is required since there is no interpretation of metacharacters, such as *|*, following substitution.

3.5 Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *grty* (2)). A shell invoked with the *-i* flag is also interactive.

Execution of a command (see also 3.7) may fail for any of the following reasons.

- Input output redirection may fail. For example, if a file does not exist or cannot be created.

- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See Figure 2 below for a complete list of UNIX signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- Syntax errors. e.g., `if ... then ... done`
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as `cd`.

The shell flag `-e` causes the shell to terminate if any error is detected.

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from <code>kill(1)</code>)

Figure 3. UNIX signals

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The `trap` command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/psSS; exit' 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/psSS; exit
```

`exit` is another built-in command that terminates execution of a shell procedure. The `exit` is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of

the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file *junk*\$.

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do case $i in
-c) flag=N ;;
*) if test -f $i
then ln $i junk$$; rm junk$$
elif test $flag
then echo file \"$i\" does not exist
else >$i
fi
esac
done
```

Figure 4. The touch command

The *trap* command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in UNIX it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to *trap*. The following fragment is taken from the *nohup* command.

```
trap "" 1 2 3 15
```

which causes *hangup*, *interrupt*, *quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The procedure *scan* (Figure 5) is an example of the use of *trap* where there is no exit in the *trap* command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

```
d=`pwd`
for i in *
do if test -d Sd/Si
then cd Sd/Si
while echo "Si:"
trap exit 2
read x
do trap : 2; eval Sx: done
fi
done
```

Figure 5. The scan command

read x is a built-in command that reads one line from the standard input and places the result in the variable *x*. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap "" 1 2 3 15
exec S*
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is **.c*. Input output specifications are evaluated left to right as they appear in the command.

- > *word* The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.
- >> *word* The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.
- < *word* The standard input (file descriptor 0) is taken from the file *word*.
- << *word* The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted then parameter and command substitution occur and \ is used to quote the characters ' ' and the first character of *word*. In the latter case *newline* is ignored (c.f. quoted strings).
- >& *digit* The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.
- <& *digit* The standard input is duplicated from file descriptor *digit*.

- <&- The standard input is closed.
- >&- The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty file `/dev/null`. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason the UNIX convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

3.8 Invoking the shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file `.profile`.

-c *string*

If the **-c** flag is present then commands are read from *string*.

-s If the **-s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

-i If the **-i** flag is present or if the shell input and output are attached to a terminal (as told by *tty*) then this shell is *interactive*. In this case TERMINATE is ignored (so that `kill 0` does not kill an interactive shell) and INTERRUPT is caught and ignored (so that `wait` is interruptible). In all cases QUIT is ignored by the shell.

Acknowledgements

The design of the shell is based in part on the original UNIX shell³ and the PWB/UNIX shell,⁴ some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System⁵ and of CTSS.⁶

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center and to Joe Maranzano for their comments on drafts of this document.

References

1. B. W. Kernighan. *UNIX for Beginners*. Bell Laboratories internal memorandum (1978).
2. K. Thompson and D. M. Ritchie. *UNIX Programmer's Manual*. Bell Laboratories (1978). Seventh Edition.
3. K. Thompson. "The UNIX Command Language." pp. 375-384 in *Structured Programming—Infotech State of the Art Report*. Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England (March 1975).
4. J. R. Mashey. *PWB/UNIX Shell Tutorial*. Bell Laboratories internal memorandum (September 30, 1977).
5. D. F. Hartley (Ed.). *The Cambridge Multiple Access System — Users Reference Manual*. University Mathematical Laboratory, Cambridge, England (1968).
6. P. A. Crisman (Ed.). *The Compatible Time-Sharing System*. M.I.T. Press, Cambridge, Mass. (1965).

Appendix A - Grammar

item: *word*
 input-output
 name = value

simple-command: *item*
 simple-command item

command: *simple-command*
 (*command-list*)
 {*command-list*}
 for name do command-list done
 for name in word ... do command-list done
 while command-list do command-list done
 until command-list do command-list done
 case word in case-part ... esac
 if command-list then command-list else-part fi

pipeline: *command*
 pipeline | command

andor: *pipeline*
 andor && pipeline
 andor || pipeline

command-list: *andor*
 command-list ;
 command-list &
 command-list ; andor
 command-list & andor

input-output: *> file*
 < file
 >> word
 << word

file: *word*
 & digit
 & -

case-part: *pattern) command-list ;;*

pattern: *word*
 pattern | word

else-part: **elif command-list then command-list else-part**
 else command-list
 empty

empty:

word: a sequence of non-blank characters

name: a sequence of letters, digits or underscores starting with a letter

digit: 0 1 2 3 4 5 6 7 8 9

Appendix B - Meta-characters and Reserved Words

a) syntactic

| pipe symbol
&& 'andf' symbol
|| 'orf' symbol
; command separator
;; case delimiter
& background commands
() command grouping
< input redirection
<< input from a here document
> output creation
>> output append

b) patterns

* match any character(s) including none
? match any single character
[...] match any of the enclosed characters

c) substitution

S{...} substitute shell variable
'...' substitute command output

d) quoting

\ quote the next character
'...' quote the enclosed characters except for '
"..." quote the enclosed characters except for \$ ` \ "

e) reserved words

if then else elif fi
case in esac
for while until do done
()





An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shells capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

January 17, 1979



An introduction to the C shell

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the "UNIX Programmers Manual." The *csh* documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features.

1. Terminal usage of the shell

1 The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *commands* are invoked. While it has a set of *builtin* commands which it performs directly, most useful commands are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system expect a list of strings or *words* as arguments. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to run it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given to the command itself to execute. In this case we specified also the word *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
Bill
%
```

Here we typed a message to send to *bill* and ended this message with a control-d which sent an end-of-file to the mail program. The mail program then transmitted our message. The characters '%' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '%' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file after which the shell noticed that mail had completed and signalled us that it was ready to read from the terminal again by printing another '%' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '-'. Thus the command

```
ls
```

will produce a list of the files in the current directory. The option `-s` is the size option, and

ls -s

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual page for each command in the UNIX programmers manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Many commands may read input or write output to files rather than simply taking input and output from the terminal. Each such command could take special words as arguments indicating where the output is to go. It is simpler, and usually sufficient, to connect these commands to files to which they wish to write, within the shell itself, and just before they are executed.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the *date* command and the *date* command prints the date on its standard output. The shell lets us redirect the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the *date* command in an environment where its standard output is the file 'now' rather than our terminal. Thus this command places the current date and time in the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to our terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to create words which contain *metacharacters* and to thus work without constantly worrying about whether certain characters are metacharacters.

Note that the shell is only reading input when it has prompted with '% '. Thus metacharacters will normally have effect only then. We need not worry about placing shell metacharacters in a letter we are sending via *mail*.

1.5. Input from files; pipelines

We learned above how to route the standard output of a command to a file. It is also possible to route the standard input of a command from a file. This is not often necessary since most commands will read from a file name given as argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads, from the file

'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the *sort* program would sort lines from its *standard input*. Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a control-d to generate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of the next, i.e. to run the commands in a sequence known as a *pipeline*. For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The *-n* option of *sort* specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the *ls* command run with the option *-s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *-r* reverse sort option and the *head* command in combination with the previous command doing

```
ls -s | sort -n -r | head -5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines out. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The metanotation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the output of each is run into the input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX pathnames consist of a number of components separated by '/'. Each component except the last names a directory in which the next component resides. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. Filenames which do not begin with '/' are interpreted starting at the current *working* directory. This directory is, by default, your *home* directory and can be changed dynamically by the *chdir* change directory command.

Most filenames consist of a number of alphanumeric characters and '.'s. In fact, all printing characters except '/' may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' is not a shell-metacharacter and is often used as the prefix with an *extension* of a base name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a *root* portion of a name (a root portion being that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the metanotation

```
prog.*
```

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are sorted into the argument list to the command alphabetically. Thus the command

```
echo prog.*
```

will echo the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in lexicographic order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as an argument directly. The four words were generated by filename expansion of the metasyntax in the one input word.

Other metanotations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently lexicographically sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

```
prog.[co]
```

will match

```
prog.c prog.o
```

in the example above. We can also place two characters astride a '-' in this notation to denote a range. Thus

```
chap.[1-5]
```

might match files

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list*) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

No match.

Another very important point is that the character '.' at the beginning of a filename is treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the current directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' followed by another users login name. For instance the word '~bill' would map to the pathname '/mnt/bill' if the home directory for 'bill' was in the directory '/mnt/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~/mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/mnt/bill/mbox' for me on the Cory Hall UNIX system. This can be very useful if you have used *chdir* to change to another users directory and have found a file you wish to copy using *cp*. You can do

```
cp thatfile ~
```

which will be expanded by the shell to

```
cp thatfile /mnt/bill
```

e.g., which the copy command will interpret as a request to make a copy of 'thatfile' in the directory '/mnt/bill'. The '~' notation doesn't, by itself, force named files to exist. This is useful, for example, when using the *cp* command, e.g.

```
cp thatfile ~/saveit
```

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of word which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.1, as it is used much less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacter pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character '*'. It will either echo an sorted list of filenames in the current directory, or print the message 'No match' if there are no files in the current directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.', or '-' in an argument word to a command is to enclose it with single quotation characters "'", i.e.

```
echo '*'
```

There is one special character '!' which is used by the *history* mechanism of the shell and which cannot be *escaped* in this way. It and the character '~' itself can be preceded by a single '\ ' to prevent their special meaning. These two mechanisms suffice to place any printing character into a word which is an argument to a shell command.

1.8. Terminating commands

When you are running a command from the shell and the shell is dormant waiting for it to complete there are a couple of ways in which you can force such a command to complete. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an INTERRUPT signal to the *cat* command by hitting the DEL or RUBOUT key on your terminal. Actually, hitting this key sends this INTERRUPT signal to all programs running on your terminal, including your shell. The shell normally ignores such signals however, so that the only program affected by the INTERRUPT will be *cat*. Since *cat* does not take any precautions to catch this signal the INTERRUPT will cause it to terminate. The shell notices that *cat* has died and prompts you again with '% '. If you hit INTERRUPT again, the shell will just repeat its prompt since it catches INTERRUPT signals and chooses to continue to execute commands rather than going away like *cat* did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the *mail* program in the first example above was terminated when we hit a control-d which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing 'logout'; UNIX then logs you off the system. Since this means that typing too many control-d's can accidentally log us off, the shell has a mechanism for preventing this. This *ignoreeof* option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a control-d. This is because it read to the end-of-file of our file 'prepared.text' in which we placed a message for 'bill' with an editor. We could also have done

```
cat prepared.text | mail bill
```

since the *cat* command would then have written the text through the pipe to the standard input of the mail command. When the *cat* command completed it would have terminated, closing down the pipeline and the *mail* command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form. These commands could also have been stopped by sending an INTERRUPT.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a QUIT signal, generated by a control-\. This will usually provoke the shell to produce a message like:

```
a.out: Quit -- Core dumped
```

indicating that a file 'core' has been created containing information about the program 'a.out's state when it ran amuck. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore INTERRUPT and QUIT signals at the terminal. To stop them you must use the *kill* program. See section 2.6 for an example.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

`chsh myname /bin/csh`

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. You only have to do this once; it takes effect at next login. You are now ready to try using *csh*.

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is placed by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may create during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login* shell, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
tset -d adm3a -p adm3a
fixexrc
set history=20
set time=3
```

on the CORY Hall UNIX system. This file contains four commands to be executed by UNIX each time I login. The first is a *tset* command which informs the system that I usually dial in on a Lear-Siegler ADM-3A terminal and that if I am on a patchboard port on the fifth floor of Evans Hall I am probably also on an ADM-3A. The second command is a *fixexrc* which manipulates my *ex* startup file in certain ways if I am on a dialup port. We need not be concerned with exactly what this command does. In general you may have certain commands in your *.login* which are particular to you.

The next two *set* commands are interpreted directly by the shell and affect the values of certain shell variables to modify the future behavior of the shell. Setting the variable *time* tells the shell to print time statistics on commands which take more than a certain threshold of machine time (in this case 3 CPU seconds). Setting the variable *history* tells the shell how much history of previous command words it should save in case I wish to repeat or rerun modified versions of previous commands. Since there is a certain overhead in this mechanism the shell does not set this variable by default, but rather lets users who wish to use the mechanism set it themselves. The value of 20 is a reasonably large value to assign to *history*. More casual users of the *history* mechanism would probably set a value of 5 or 10. The use of the *history* mechanism will be described subsequently.

After executing commands from *.login* the shell reads commands from your terminal, prompting for each with '% '. When it receives an end-of-file from the terminal, the shell will print 'logout' and execute commands from the file *.logout* in your home directory. After that the shell will die and UNIX will log you off the system. If the system is not going down, you will receive a new login message. In any case, after the 'logout' message the shell is doomed and will take no further input from the terminal.

2.2. Shell variables

The shell maintains a set of *variables*. We saw above the variables *history* and *time* which had values '20' and '3'. In fact, each shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the *set* command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be reintroduced into commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the shell searches for commands. The *set* command shows the value of all variables currently defined (we usually say *set*) in the shell. The default value for *path* will be shown by *set* to be

```
% set
argv
home /mnt/bill
path (. /bin /usr/bin)
prompt %
shell /bin/csh
status 0
%
```

This notation indicates that the variable *path* points to the current directory '.' and then '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). The most heavily used system commands live in '/bin'. Less heavily used system commands live in '/usr/bin'.

A number of new programs on the system live in the directory '/usr/new'. If we wish, as well we might, all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/new /bin /usr/bin)
```

in our file *.cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.

Other useful built in variables are the variable *home* which shows your home directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal. To logout from UNIX with *ignoreeof* set you must type

```
logout
```

This is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset*. Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

Both *set* and *unset* are built-in commands of the shell.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail*. The metasyntax

```
> filename
```

which redirects the output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your *.login* file. Then trying to do

```
date > now
```

would cause a diagnostic if 'now' existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok.

If you receive mail frequently while you are logged in and wish to be informed of the arrival of this mail you can put a command

```
set mail=/usr/mail/yourname
```

in your *.login* file. Here you should change 'yourname' to your login name. The shell will look at this file every 10 minutes to see if new mail has arrived. If you receive mail only infrequently you are better off not setting this variable. In this case it will only serve to delay the shells response to you when it checks for mail.

The use of shell variables to introduce text into commands, which is most useful in shell command scripts, will be introduced in section 2.4.

2.3. The shell's history list

The shell can maintain a history list into which it places the words of previous commands. It is possible to use a metanotation to reintroduce commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

Consider the following transcript:

```
% where michael
michael is on tty0 dialup 300 baud 642-7927
% write !$
write michael
Long time no see michael.
Why don't you call me at 524-4510.
EOF
%
```

Here we asked the system where *michael* was logged in. It told us he was on 'tty0' and we told the shell to invoke a 'write' command to '\$'. This is a history notation which means the last word of the last command executed, in this case 'michael'. The shell performed this substitution and then echoed the command as it would execute it. Let us assume that we don't hear anything from michael. We might do

```
% ps t0
PID TTY TIME COMMAND
4808 0 0:05 -
% !!
ps t0
PID TTY TIME COMMAND
5104 0 0:00 - 7
% !where
where michael
michael is not logged in
%
```

Here we ran a *ps* on the teletype *michael* was logged in on to see that he had a shell. Repeating this command via the history substitution '!!' we saw that he had logged out and that only a *gerty* process was running on his terminal. Repeating the *where* command showed that he was indeed gone, most likely having hung up the phone in order to be able to call.

This illustrates several useful features of the history mechanism. The form '!!' repeats the last command execution. The form '!string' repeats the last command which began with a word of which 'string' is a prefix. Another useful command form is '[lhs]rhs' performing a

substitute similar to that in *ed* or *ex*. Thus after

```
% cat ~bill/csh/sh..c
/mnt/bill/csh/sh..c: No such file or directory
% ↑..↑.
cat ~bill/csh/sh.c
#include "sh.h"

/*
 * C Shell
 *
 * Bill Joy, UC Berkeley
 * October, 1978
 */

char *pathlist[] = { SRCHP
%
```

here we used the substitution to correct a typing mistake, and then rubbed the command out after we saw that we had found the file that we wanted. The substitution changed the two '.' characters to a single '.' character.

After this command we might do

```
% !! | lpr
cat ~bill/csh/sh.c | lpr
```

to put a copy of this file on the line printer, or (immediately after the *cat* which worked above)

```
% pr !S | lpr
pr ~bill/csh/sh.c | lpr
%
```

to print a copy on the printer using *pr*.

More advanced forms of the history mechanism are also possible. A notion of modification on substitutions allows one to say (after the first successful *cat* above).

```
% cd !S:h
cd ~bill/csh
%
```

The trailing ':h' on the history substitution here causes only the head portion of the pathname reintroduced by the history mechanism to be substituted. This mechanism and related mechanisms are used less often than the forms above.

A complete description of history mechanism features is given in the C shell manual pages in the UNIX Programmers Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to the macro facility of many assemblers.

Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shells environment and commands such as *chdir* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'Mail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail Mail
```

in your *.login* file, the shell will transform an input line of the form

```
mail bill
```

into a call on 'Mail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '-s'. We can do

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command syntax 'dir' which does an 'ls -s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls -s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in "" characters to prevent most substitutions from occurring and the character ';' from being recognized as a parser metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!† /etc/passwd'
```

defines a command which looks up its first argument in the password file.

2.5. Detached commands; >> and >& redirection

There are a few more metanotations useful to the terminal user which have not been introduced yet. The metacharacter '&' may be placed after a command, or after a sequence of commands separated by ';' or '|'. This causes the shell to not wait for the commands to terminate before prompting again. We say that they are *detached* or *background* processes. Thus

```
% pr ~bill/csh/sh.c | lpr &  
5120  
5121  
%
```

Here the shell printed two numbers and came back very quickly rather than waiting for the *pr* and *lpr* commands to finish. These numbers are the process numbers assigned by the system to the *pr* and *lpr* commands.†

†Running commands in the background like this tends to slow down the system and is not a good idea if the system is overloaded. When overloaded, the system will just bog down more if you run a large number of processes at once.

Since `havoc` would result if a command run in the background were to read from your terminal at the same time as the shell does, the default standard input for a command run in the background is not your terminal, but an empty file called `/dev/null`. Commands run in the background are also made immune to `INTERRUPT` and `QUIT` signals which you may subsequently generate at your terminal.*

If you intend to log off the system before the command completes you must run the command immune to `HANGUP` signals. This is done by placing the word `'nohup'` before each program in the command, i.e.:

```
nohup man csh | nohup lpr &
```

In addition to the standard output, commands also have a diagnostic output which is normally directed to the terminal even when the standard output is directed to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

The `'>&'` here tells the shell to route both the diagnostic output and the standard output into `'file'`. of the standard output. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon `lpr.#`

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.†

2.6. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The `alias` command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given an argument such as

```
alias ls
```

to show the current alias for, e.g., `'ls'`.

The `cd` and `chdir` commands are equivalent, and change the working directory of the shell. It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. Thus after you login you can do

*If a background command stops suddenly when you hit `INTERRUPT` or `QUIT` it is likely a bug in the background program.

#A command form

```
command >&! file
```

exists, and is used when `noclobber` is set and `file` already exists.

†If `noclobber` is set, then an error will result if `file` does not exist, otherwise the shell will create `file` if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for `file` to not exist when `noclobber` is set.

```
% pwd
/mnt/bill
% mkdir newspaper
% chdir newspaper
% pwd
/mnt/bill/newspaper
%
```

after which you will be in the directory *newspaper*. You can place a group of related files there. You can return to your 'home' login directory by doing just

```
chdir
```

with no arguments. We used the *pwd* print working directory command to show the name of the current directory here. The current directory will usually be a subdirectory of your home directory, and have it (here '/mnt/bill') at the start of it.

The *echo* command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will yield.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*. By placing a '!' character in its value the shell will there substitute the index of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt = \! % '
```

Note that the '!' character had to be escaped here even within "" characters.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

```
repeat 5 cat one >> five
```

The *setenv* command can be used, on version 7 UNIX systems, to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable *TERM* to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME /usr/bill
SHELL /bin/csh
TERM adm3a
%
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect before the next time you login.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp five five.save
0.0u 0.3s 0:01 26%
% time wc five.save
 1200  6300 37650 five.save
1.2u 0.5s 0:03 55%
%
```

indicates that the *cp* command used less than 1/10th of a second of user time and only 3/10th of a second of system time in copying the file 'five' to 'five.save'. The command word count 'wc' on the other hand used 1.2 seconds of user time and 0.5 seconds of system time in 3 seconds of elapsed time in counting the number of words, character and lines in 'five.save'. The percentage '55%' indicates that over this period of 3 seconds, our command 'wc' used an average of 55 percent of the available CPU cycles of the machine. This is a very high percentage and indicates that the system is lightly loaded.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell.

The *wait* command can be used after starting processes with '&' to quickly see if they have finished. If the shell responds immediately with another prompt, they have. Otherwise you can wait for the shell to prompt at which point they will have finished, or interrupt the shell by sending a RUB or DELETE character. If the shell is interrupted, it will print the names and numbers of the processes it knows to be unfinished. Thus:

```
% nroff paper | lpr &
2450
2451
% wait
 2451 lpr
 2450 nroff
wait: Interrupted.
%
```

You can check again later by doing another *wait*, or see which commands are still running by doing a *ps*. As 'time' will show you, *ps* is fairly expensive. It is thus counterproductive to run many *ps* commands to see how a background process is doing.†

If you run a background process and decide you want to stop it for whatever reason you must use the *kill* program. You must use the number of the processes you wish to kill. Thus to stop the *nroff* in the above pipeline you would do

```
% kill 2450
% wait
2450: nroff: Terminated.
%
```

Here the shell printed a diagnostic that we terminated 'nroff' only after we did a *wait*. If we want the shell to discover the termination of all processes it has created we must, in general, use *wait*.

2.7. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the *foreach* built-in command which can be used to run the same command sequence with a number of different arguments.

†If you do you are usurping with these *ps* commands the processor time the job needs to finish, thereby delaying its completion!

If you intend to use UNIX a lot you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile*. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *cs*h command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *cs*h commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *cs*h at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character '\$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to '1' if name is *set* or to '0' if name is not *set*. It is the fundamental mechanism used

for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

`$#name`

expands to the number of elements in the variable *name*. Thus

```
% set argv = (a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

`$argv[1]`

gives the first component of *argv* or in the example above 'a'. Similarly

`$argv[$#argv]`

would give 'c', and

`$argv[1-2]`

Other notations useful in shell scripts are

`$n`

where *n* is an integer as a shorthand for

`$argv[n]`

the *n*th parameter and

`$*`

which is a shorthand for

`$argv`

The form

`$$`

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names.

One minor difference between '`$n`' and '`$argv[n]`' should be noted here. The form '`$argv[n]`' will yield an error if *n* is not in the range '1-`$#argv`' while '`$n`' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form '`n-`'; if there are less than *n* components of the given variable then no words are substituted. A range of the form '`m-n`' likewise returns an empty vector without giving an error when *m* exceeds the number of elements of the given variable, provided the subscript *n* is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '=' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations.

The shell also allows file enquiries of the form

-? filename

where '?' is replace by a number of single characters. For instance the expression primitive

-e filename

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '\$status' examined in the next command. Since '\$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

    if (Si:r.c != Si) continue      # not a .c file so do nothing

    if (! -r ~/backup/Si:t) then
        echo Si:t not in backup... not cp\`ed
        continue
    endif

    cmp -s Si ~/backup/Si:t        # to set $status

    if ($status != 0) then
        echo new backup of Si
        cp Si ~/backup/Si:t
    endif
end
```

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop

we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '\$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
    command
    ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \  
    command
```

Here we have escaped the newline for the sake of appearance, and the '\' must **immediately**. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else-if* pairs followed by a single *else* and an *endif*, e.g.:

```
if ( expression ) then
    commands
else if ( expression ) then
    commands
...
else
    commands
endif
```

Another important mechanism used in shell scripts is ':' modifiers. We can use the modifier ':r' here to extract a root of a filename. Thus if the variable *i* has the value 'foo.bar' then

†The following two formats are not currently acceptable to the shell:

```
if ( expression )          # Won't work!
then
    command
    ...
endif
```

and

```
if ( expression ) then command endif          # Won't work
```

```
% echo Si Si:r
foo.bar foo
%
```

shows how the `‘:r’` modifier strips off the trailing `‘.bar’`. Other modifiers will take off the last component of a pathname leaving the head `‘:h’` or all but the last component of a pathname leaving the tail `‘:t’`. These modifiers are fully described in the *csh* manual pages in the programmers manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the `‘:’` modification mechanism. Finally, we note that the character `‘#’` lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a `‘#’` are discarded by the shell. This character can be quoted using `‘‘’` or `‘\’` to place it in an argument word.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

```
while ( expression )
    commands
end

and

switch ( word )

case str1:
    commands
    breaksw

...

case strn:
    commands
    breaksw

default:
    commands
    breaksw

endsw
```

For details see the manual section for *csh*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *csh* scripts is to use *break* rather than *breaksw* in switches.

Finally, *csh* allows a *goto* statement, with labels looking like they do in C, i.e.:

#It is also important to note that the current implementation of the shell limits the number of `‘:’` modifiers on a `‘S’` substitution to 1. Thus

```
% echo Si Si:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.

```
loop:
    commands
    goto loop
```

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This it is different from previous shells running under UNIX. It allowing shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly "EOF". The fact that the 'EOF' is enclosed in "" characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,\$' in our editor script we needed to insure that this '\$' was not variable substituted. We could also have insured this by preceding the '\$' here with a '\', i.e.:

```
1,\$s/[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do a *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *-v* and *-x* command line options can be used to help trace the actions of the shell. The *-n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *cs/* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the */bin/sh* on your system may well defer to *cs/* to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using "" which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as "" does.

4. Miscellaneous, less generally useful, shell mechanisms

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, `/bin/sh`, `/bin/nsh`, and `/bin/csh`. To count the number of persons using each shell one could issue the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with `?` when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=(^ls^)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within `""` characters is converted by the shell to a list of words. You can also place the `""` quoted string within `""` characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier `:x` exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters `{` and `}`. These characters specify that the contained strings, separated by `,` are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

Astr1B Astr2B ... AstrnB

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/({hdrs,retrofit,csh})
```

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown bin /usr/({bin/({ex,edit}),lib/({ex1.1strings,how_ex}})
```

4.3. Command substitution

A command enclosed in `` characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable *pwd* or to do

```
ex `grep -l TRACE *.c`
```

to run the editor *ex* supplying as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the shells manual section for a list of these options.

*Command expansion also occurs in input redirected with '<<' and within `` quotations. Refer to the shell manual section for full details.

Appendix — Special characters

The following table lists the special characters of *cs*h and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *cs*h manual section for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
	1.5	separates commands in a pipeline
()	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.3	indicates redirected input
>	1.5	indicates redirected output

Expansion/substitution metacharacters

\$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	3.6	begins a shell comment
-	1.2	prefixes option (flag) arguments to commands

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX programmers manual in section 1. You can get an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

. Your current directory has the name '.' as well as the name printed by the command *pwd*. The current directory '.' is usually the first component of the search path contained in the variable *path*, thus commands which are in '.' are found first (2.2). The character '.' is also used in separating components of filenames (1.6). The character '.' at the beginning of a component of a path-name is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs (1.6).

.. Each directory has a file '..' in it which is a reference to its *parent* directory. After changing into the directory with *chdir*, i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by *pwd* (2.6).

alias An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes aliases and can print their current values. The command *unalias* is used to remove aliases (2.6).

argument Commands in UNIX receive a list of argument words. Thus the command

```
echo a b c
```

consists of a command name 'echo' and three argument words 'a', 'b' and 'c' (1.1).

argv The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).

background Commands started without waiting for them to complete are called *background* commands (1.5).

bin A directory containing binaries of programs and shell scripts to be executed is typically called a 'bin' directory. The standard system 'bin' directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Other binaries are contained in directories such as '/usr/new' where new programs are placed. You can place binaries in any directory. If you wish to execute them often, the name of the directories should be a component of the variable *path*.

break *Break* is a built-in command used to exit from loops within the control structure of the shell (3.6).

builtin A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in 'bin' directories. These commands are accessible because the directories in which they reside are named in the *path* variable.

- case** A *case* command is used as a label in a *switch* statement in the shells control structure, similar to that of the language C. Details are given in the shells documentation 'csh (NEW)' (3.7).
- cat** The *cat* program catenates a list of specified files on the standard output. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).
- cd** The *cd* command is used to change the working directory. With no arguments, *cd* changes your working directory to be your *home* directory (2.3) (2.6).
- chdir** The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.
- chsh** The *chsh* command is used to change the shell which you use on UNIX. By default, you use an older 'standard' version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing
- ```
chsh your-login-name /bin/csh
```
- Thus I would do
- ```
chsh bill /bin/csh
```
- It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).
- cmp** *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.
- command** A function performed by the system, either by the shell (a builtin command) or by a program residing in a file in a directory within the UNIX system is called a *command* (1.1).
- command substitution** The replacement of a command enclosed in "" characters by the text output by that command is called *command substitution* (3.6, 4.3).
- component** A part of a *pathname* between '/' characters is called a *component* of that path-name. A *variable* which has multiple strings as value is said to have several *components*, each string is a *component* of the variable.
- continue** A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).
- core dump** When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This 'core dump' can be examined with the system debuggers 'db (1)' and 'cdb (1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form:
- ```
commandname: Illegal instruction -- Core dumped
```
- (where 'Illegal instruction' is only one of several possible messages) you should report this to the author of the program and save the 'core' file. If this was a system program you should report this with the *trouble* command 'trouble (1)'.
- cp** The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (2.6).

- .cshrc** The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).
- date** The *date* command prints the current date and time (1.3).
- debugging** *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell debugging (4.4).
- default** The label *default*: is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).
- DELETE** The DELETE or RUBOUT key on the terminal is used to generate an INTERRUPT signal in UNIX which stops the execution of most programs (2.6).
- detached** A command run without waiting for it to complete is said to be detached (2.5).
- diagnostic** An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the standard output, since that is often directed away from the terminal (1.3, 1.5). Error messages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus diagnostics will usually appear on the terminal (2.5).
- directory** A structure which contains files. At any time you are in one particular directory whose names can be printed by the command 'pwd'. The *chdir* command will change you to another directory, and make the files in that directory visible. The directory in which you are when you first login is your *home* directory (1.1, 1.6).
- echo** The *echo* command prints its arguments (1.6, 2.6, 3.6, 3.10).
- else** The *else* command is part of the 'if-then-else-endif' control command construct (3.6).
- EOF** An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an end-of-file when the command sending them input completes. Most commands terminate when they receive an end-of-file. The shell has an option to ignore end-of-file from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).
- escape** A character \ used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus
- ```
echo \*
```
- will echo the character '*' while just
- ```
echo *
```
- will echo the names of the file in the current directory. In this example, \ *escapes* '\*' (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some UNIX systems use this character to indicate that output is to be suspended. Other systems use control-s.
- /etc/passwd** This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by ':' characters (2.3). You can look at this file by saying
- ```
cat /etc/passwd
```
- The command *grep* is often used to search for information in this file. See

- 'passwd (5)' and 'grep (1)' for more details.
- exit** The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).
- exit status** A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script to give a non-zero exit status (3.5).
- expansion** The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion*. Thus the replacement of the word '*' by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters '!!' by the text of the last command is a 'history expansion'. Expansions are also referred to as *substitutions* (1.6, 3.4, 4.2).
- expressions** Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell expressions are those of the language C (3.5).
- extension** Filenames often consist of a *root* name and an *extension* separated by the character '.'. By convention, groups of related files often share the same root name. Thus if 'prog.c' were a C program, then the object file for this program would be stored in 'prog.o'. Similarly a paper written with the '-me' nroff macro package might be stored in 'paper.me' while a formatted version of this paper might be kept in 'paper.out' and a list of spelling errors in 'paper.errs' (1.6).
- filename** Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most file names do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the root portion of the filename from an extension (1.6).
- filename expansion** Filename expansion uses the metacharacters '*', '?', and '[' and ']' to provide a convenient mechanism for naming files. Using filename expansion it is easy to name all the files in the current directory, or all files which have a common root name. Other filename expansion mechanisms use the metacharacter '*' and allow files in other users directories to be named easily (1.6, 4.2).
- flag** Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consists of one or more letters preceded by the character '-' (1.2). Thus the *ls* list file commands has an option '-s' to list the sizes of files. This is specified
- ```
ls -s
```
- foreach** The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).
- getty** The *getty* program is part of the system which determines the speed at which your terminal is to run when you first log in. It types the initial system banner and 'login:'. When no one is logged in on a terminal a *ps* command shows a command of the form '- 7' where '7' here is often some other single letter or digit. This '7' is an option to the *getty* command, indicating the type of port which it is running on. If you see a *getty* command running on a terminal in the output of *ps* you know that no one is logged in on that terminal (2.3).

- goto** The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).
- grep** The *grep* command searches through a list of argument files for a specified string. Thus
- ```
grep bill /etc/passwd
```
- will print each line in the file */etc/passwd* which contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed (1)' and 'ex (1)' (2.3). *Grep* stands for 'globally find regular expression and print.'
- hangup** When you hangup a phone line, a HANGUP signal is sent to all running processes on your terminal, causing them to terminate execution prematurely. If you wish to start commands to run after you log off a dialup you must use the command *nohup* (2.6).
- head** The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5, 2.3).
- history** The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (1.7, 2.6).
- home directory** Each user has a home directory, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first log in. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the home directories of other users in forming filenames using a file expansion notation and the character '~' (1.6).
- if** A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).
- ignoreeof** Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can *set* the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off. If the system is slow, this can waste much time, as it may take a long time to log in again (2.2, 2.6).
- input** Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for input from their *standard input* which is, by default, the terminal. This standard input can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in pipelines will read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a file name to use as standard input. Special mechanisms exist for suppling input to commands in shell scripts (1.2, 1.6, 3.8).
- interrupt** An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key. It causes most programs to stop execution. Certain programs such as the shell and the editors handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you hit interrupt

because many commands die when they receive an interrupt (1.8, 2.6, 3.9).

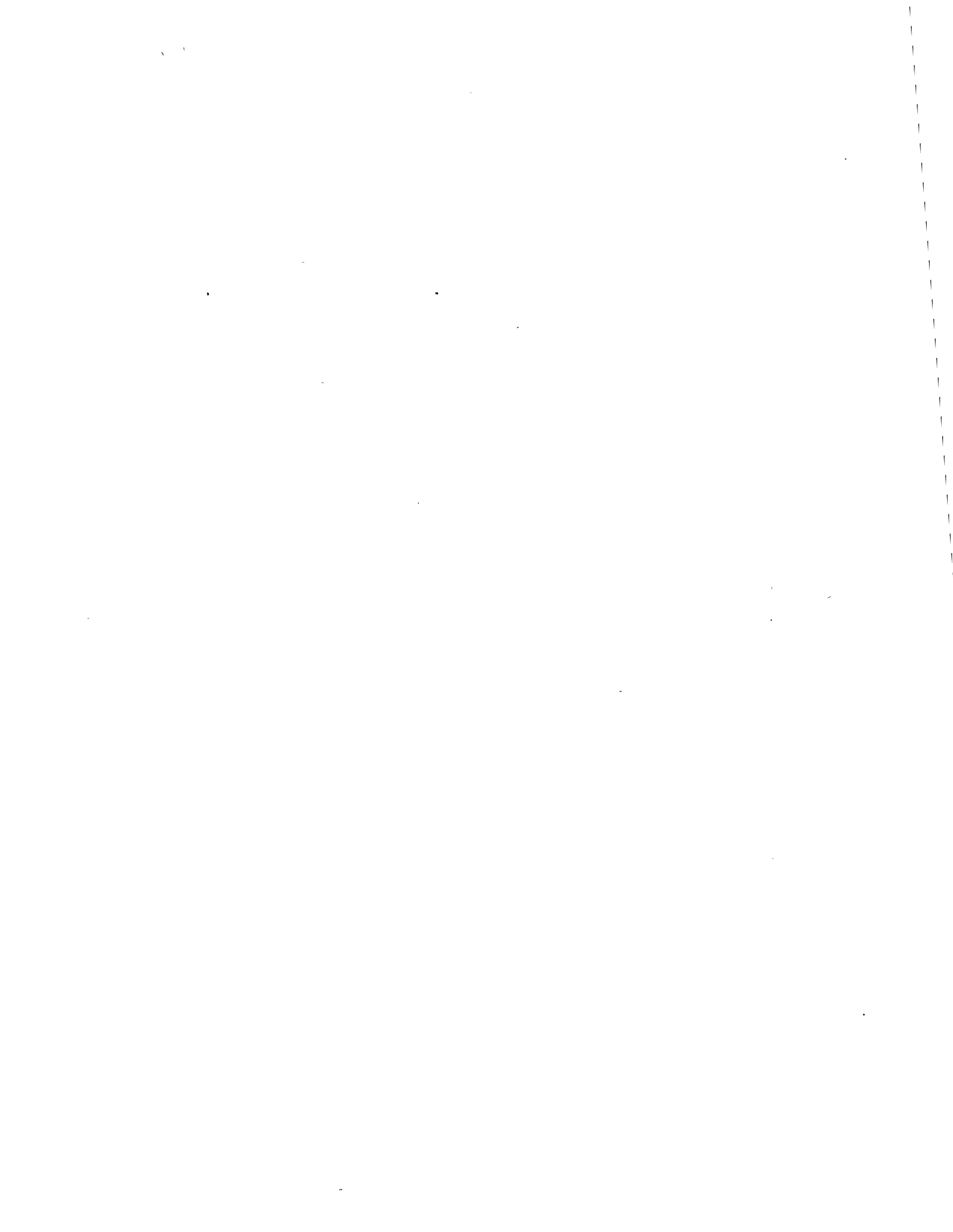
- kill** A program which terminates processes run without waiting for them to complete. (2.6)
- .login** The file *.login* in your *home* directory is read by the shell each time you log in to UNIX and the commands there are executed. There are a number of commands which are usefully placed here especially *tset* commands and *set* commands to the shell itself (2.1).
- logout** The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an end-of-file, but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.2).
- .logout** When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.
- lpr** The command *lpr* is the line printer daemon. The standard input of *lpr* is spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).
- ls** The *ls* list files command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these directories (1.2).
- mail** The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.2).
- make** The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).
- makefile** The file containing command for *make* is called 'makefile' (3.2).
- manual** The 'manual' often referred to is the 'UNIX programmers manual.' It contains a number of sections and a description of each UNIX program. An online version of the manual is accessible through the *man* command. Its documentation can be obtained online via
- man man**
- metacharacter** Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a metacharacter is the character '>' which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted metacharacters form separate words (1.4). The appendix to this user's manual lists the metacharacters in groups by their function.
- mkdir** The *mkdir* command is used to create a new directory (2.6).
- modifier** Substitutions with the history mechanism, keyed by the character '!' or of variables using the metacharacter '\$' are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the modifier itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).

- noclobber** The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '*>*' output redirection metasyntax of the shell (2.2, 2.5).
- nohup** A shell command used to allow background commands to run to completion even if you log off a dialup before they complete. (2.5)
- nroff** The standard text formatter on UNIX is the program *nroff*. Using *nroff* and one of the available *macro* packages for it, it is possible to have documents automatically formatted and to prepare them for phototypesetting using the typesetter program *troff* (3.2).
- onintr** The *onintr* command is built into the shell and is used to control the action of a shell command script when an interrupt signal is received (3.9).
- output** Many commands in UNIX result in some lines of text which are called their *output*. This output is usually placed on what is known as the *standard output* which is normally connected to the users terminal. The shell has a syntax using the metacharacter '*>*' for redirecting the standard output of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '*|*' it is also possible for the standard output of one command to become the standard input of another command (1.5). Certain commands such as the line printer daemon *lpr* do not place their results on the standard output but rather in more useful places such as on the line printer (2.3). Similarly the *write* command places its output on another users terminal rather than its standard output (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the standard output has been sent to a file or another command, but it is possible to direct error diagnostics along with standard output using a special metanotation (2.5).
- path** The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is
- path* (./bin /usr/bin)
- the shell normally looks in the current directory, and then in the standard system directories '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' bits set. This is normally true because a command of the form
- chmod 755 script
- was executed to turn these execute bits on (3.3).
- pathname** A list of names, separated by '/' characters forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next component file resides. Pathnames which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other pathnames are interpreted relative to the current directory as reported by *pwd*. The last component of a pathname may name a directory, but usually names a file.
- pipeline** A group of commands which are connected together, the standard output of each connected to the standard input of the next is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '*|*' (1.5, 2.3).

- pr** The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).
- printenv** The *printenv* command is used on version 7 UNIX systems to print the current setting of variables in the environment. As of this writing, only the VAX/UNIX system on the fifth floor of Evans Hall is running a version 7 UNIX system. The other systems are running version 6, which does not have or need *printenv* (2.6).
- process** A instance of a running program is called a process (2.6). The numbers used by *kill* and printed by *wait* are unique numbers generated for these processes by UNIX. They are useful in *kill* commands which can be used to stop background processes. (2.6)
- program** Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a program.
- programmers manual** Also referred to as the *manual*. See the glossary entry for 'manual'.
- prompt** Many programs will print a prompt on the terminal when they expect input. Thus the editor 'ex (NEW)' will print a ':' when it expects input. The shell prompts for input with '%' and occasionally with '?' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shells main prompt. This is mostly used when debugging the shell (2.6).
- ps** The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.3, 2.6). Login shells, such as the *cs*h you get when you login are shown as '-'.
The *pwd* command prints the full pathname of the current (working) directory.
- quit** The *quit* signal, generated by a control-\ is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).
- quotation** The process by which metacharacters are prevented their special meaning, usually by using the character '"' in pairs, or by using the character '\ ' is referred to as *quotation* (1.4).
- redirection** The routing of input or output from or to a file is known as *redirection* of input or output (1.3).
- repeat** The *repeat* command iterates another command a specified number of times (2.6).
- RUBOUT** The RUBOUT or DELETE key generates an interrupt signal which is used to stop programs or to cause them to return and prompt for more input (2.6).
- script** Sequences of shell commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program in a language such as C, by using the shell to selectively run other programs (3.2, 3.3, 3.10).
- set** The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the *set* command the behavior of the shell can be affected (2.1).

- setenv** On version 7 systems variables in the environment 'environ (5)' can be changed by using the *setenv* builtin command (2.6). The *printenv* command can be used to print the value of the variables in the environment. Currently, only the VAX/UNIX system on the fifth floor of Evans Hall is running version 7 UNIX. The other systems are running version 6, where *setenv* is not necessary and does not exist.
- shell** A shell is a command language interpreter. It is possible to write and run your own shell, as shells are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular shell, called *cs**h*.
- shell script** See *script* (3.2, 3.3, 3.10).
- sort** The *sort* program sorts a sequence of lines in ways that can be controlled by argument flags (1.5).
- source** The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.6).
- special character** See *metacharacters* and the appendix to this manual.
- standard** We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).
- status** A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The shell variable *status* is set to the status returned by the last command. It is most useful in shell command scripts (3.5, 3.6).
- substitution** The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter '!' and variable substitution indicated by '\$'. We also refer to substitutions as *expansions* (3.4).
- switch** The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7).
- termination** When a command which is being executed finished we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified command whose numbers are given (2.6).
- then** The *then* command is part of the shells 'if-then-else-endif' control construct used in command scripts (3.6).
- time** The *time* command can be used to measure the amount of CPU and real time consumed by a specified command (2.1, 2.6).
- troff** The *troff* program is used to typeset documents. See also *nroff* (3.2).
- tset** The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).
- unalias** The *unalias* command removes aliases (2.6).
- UNIX** UNIX is an operating system on which *cs**h* runs. UNIX provides facilities which allow *cs**h* to invoke other programs such as editors and text formatters which you may wish to use.

- unset** The *unset* command removes the definitions of shell variables (2.2, 2.6).
- variable expansion**
See *variables* and *expansion* (2.2, 3.4).
- variables** Variables in *cs* hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. Variables such as *argv* are also used in writing shell programs (shell command scripts) (2.2).
- verbose** The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shells *-v* command line option (3.10).
- wait** The builtin command *wait* causes the shell to pause, and not prompt, until all commands run in the background have terminated (2.6).
- where** The *where* command shows where the users named as arguments are logged into the system (2.3).
- while** The *while* builtin control construct is used in shell command scripts (3.7).
- word** A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '-', '.', or '/' form words all by themselves even if they are not surrounded by blanks. Any sequence of character may be made into a word by surrounding it with "" characters except for the characters '"' and "'" which require special treatment (1.1, 1.6). This process of placing special characters in words without their special meaning is called *quoting*.
- working directory**
At an given time you are in one particular directory, called your working directory. This directories name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change working directories using *chdir*.
- write** The *write* command is used to communicate with other users who are logged in to UNIX (2.3).



An introduction to the C shell
(Revised for the Third Berkeley Distribution)

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

Csh is a new command language interpreter for UNIX† systems. It incorporates good features of other shells and a *history* mechanism similar to the *redo* of INTERLISP. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to *csh* are designed more for the interactive UNIX user.

UNIX users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with *csh* is possible after reading just the first section of this document. The second section describes the shells capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

Back matter includes an appendix listing special characters of the shell and a glossary of terms and commands introduced in this manual.

December 21, 1979

†UNIX is a Trademark of Bell Laboratories.

An introduction to the C shell

(Revised for the Third Berkeley Distribution)

William Joy

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Introduction

A *shell* is a command language interpreter. *Csh* is the name of one particular command interpreter on UNIX. The primary purpose of *csh* is to translate command lines typed at a terminal into system actions, such as invocation of other programs. *Csh* is a user program just like any you might write. Hopefully, *csh* will be a very useful program for you in interacting with the UNIX system.

In addition to this document, you will want to refer to a copy of the "UNIX Programmers Manual." The *csh* documentation in the manual provides a full description of all features of the shell and is a final reference for questions about the shell.

Many words in this document are shown in *italics*. These are important words; names of commands, and words which have special meaning in discussing the shell and UNIX. Many of the words are defined in a glossary at the end of this document. If you don't know what is meant by a word, you should look for it in the glossary.

Acknowledgements

Numerous people have provided good input about previous versions of *csh* and aided in its debugging and in the debugging of its documentation. I would especially like to thank Michael Ubell who made the crucial observation that history commands could be done well over the word structure of input text, and implemented a prototype history mechanism in an older version of the shell. Eric Allman has also provided a large number of useful comments on the shell, helping to unify those concepts which are present and to identify and eliminate useless and marginally useful features. Mike O'Brien suggested the pathname hashing mechanism which speeds command execution.

1. Terminal usage of the shell

1.1. The basic notion of commands

A *shell* in UNIX acts mostly as a medium through which other *commands* are invoked. While it has a set of *builtin* commands which it performs directly, most useful commands are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

Commands in the UNIX system expect a list of strings or *words* as arguments. Thus the command

```
mail bill
```

consists of two words. The first word *mail* names the command to be executed, in this case the mail program which sends messages to other users. The shell uses the name of the command in attempting to run it for you. It will look in a number of *directories* for a file with the name *mail* which is expected to contain the mail program.

The rest of the words of the command are given to the command itself to execute. In this case we specified also the word *bill* which is interpreted by the *mail* program to be the name of a user to whom mail is to be sent. In normal terminal usage we might use the *mail* command as follows.

```
% mail bill
I have a question about the csh documentation.
My document seems to be missing page 5.
Does a page five exist?
    Bill
EOT
%
```

Here we typed a message to send to *bill* and ended this message with a control-d which sent an end-of-file to the mail program. The mail program then echoed the characters 'EOT' and transmitted our message. The characters '%' were printed before and after the mail command by the shell to indicate that input was needed.

After typing the '%' prompt the shell was reading command input from our terminal. We typed a complete command 'mail bill'. The shell then executed the *mail* program with argument *bill* and went dormant waiting for it to complete. The mail program then read input from our terminal until we signalled an end-of-file after which the shell noticed that mail had completed and signalled us that it was ready to read from the terminal again by printing another '%' prompt.

This is the essential pattern of all interaction with UNIX through the shell. A complete command is typed at the terminal, the shell executes the command and when this execution completes prompts for a new command. If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the *tset* command, which sets the default *erase* and *kill* characters on your terminal - the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, the erase character is '#' and the kill character is '@'. Most people who use CRT displays prefer to use the

backspace (control-h) character as their erase character since it is then easier to see what you have typed so far. You can make this be true by typing

```
tset -e
```

which tells the program *tset* to set the erase character, and its default setting for this character is a backspace.

1.2. Flag arguments

A useful notion in UNIX is that of a *flag* argument. While many arguments to commands specify file names or user names some arguments rather specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the character '-'. Thus the command

```
ls
```

will produce a list of the files in the current directory. The option *-s* is the size option, and

```
ls -s
```

causes *ls* to also give, for each file the size of the file in blocks of 512 characters. The manual page for each command in the UNIX programmers manual gives the available options for each command. The *ls* command has a large number of useful and interesting options. Most other commands have either no options or only one or two options. It is hard to remember options of commands which are not used very frequently, so most UNIX utilities perform only one or two functions rather than having a large number of hard to remember options.

1.3. Output to files

Many commands may read input or write output to files rather than simply taking input and output from the terminal. Each such command could take special words as arguments indicating where the output is to go. It is simpler, and usually sufficient, to connect these commands to files to which they wish to write, within the shell itself, and just before they are executed.

Thus suppose we wish to save the current date in a file called 'now'. The command

```
date
```

will print the current date on our terminal. This is because our terminal is the default *standard output* for the *date* command and the *date* command prints the date on its standard output. The shell lets us redirect the *standard output* of a command through a notation using the *metacharacter* '>' and the name of the file where output is to be placed. Thus the command

```
date > now
```

runs the *date* command in an environment where its standard output is the file 'now' rather than our terminal. Thus this command places the current date and time in the file 'now'. It is important to know that the *date* command was unaware that its output was going to a file rather than to our terminal. The shell performed this *redirection* before the command began executing.

One other thing to note here is that the file 'now' need not have existed before the *date* command was executed; the shell would have created the file if it did not exist. And if the file did exist? If it had existed previously these previous contents would have been discarded! A shell option *noclobber* exists to prevent this from happening accidentally; it is discussed in section 2.2.

The system normally keeps files which you create with '>' and all other files. Thus the default is for files to be permanent. If you wish to create a file which will be removed automatically, you can begin its name with a '#' character, this 'scratch' character denotes the fact that the file will be a scratch file.* The system will remove such files after a couple of days, or sooner if file space becomes very tight. Thus, in running the *date* command above, we don't really want to save the output forever, so we would more likely do

```
date > #now
```

1.4. Metacharacters in the shell

The shell has a large number of special characters (like '>') which indicate special functions. We say that these notations have *syntactic* and *semantic* meaning to the shell. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows us to create words which contain *metacharacters* and to thus work without constantly worrying about whether certain characters are metacharacters.

Note that the shell is only reading input when it has prompted with '% '. Thus metacharacters will normally have effect only then. We need not worry about placing shell metacharacters in a letter we are sending via *mail*.

1.5. Input from files; pipelines

We learned above how to route the standard output of a command to a file. It is also possible to route the standard input of a command from a file. This is not often necessary since most commands will read from a file name given as argument. We can give the command

```
sort < data
```

to run the *sort* command with standard input, where the command normally reads, from the file 'data'. We would more likely say

```
sort data
```

letting the *sort* command open the file 'data' for input itself since this is less to type.

We should note that if we just typed

```
sort
```

then the *sort* program would sort lines from its *standard input*. Since we did not *redirect* the standard input, it would sort lines as we typed them on the terminal until we typed a control-d to generate an end-of-file.

A most useful capability is the ability to combine the standard output of one command with the standard input of the next, i.e. to run the commands in a sequence known as a *pipeline*. For instance the command

```
ls -s
```

normally produces a list of the files in our directory with the size of each in

*Note that if your erase character is a '#', you will have to precede the '#' with a '\'. The fact that the '#' character is the old (pre-CRT) standard erase character means that it seldom appears in a file name, and allows this convention to be used for scratch files. If you are using a CRT, your erase character should be a control-h, as we demonstrated in section 1.1 how this could be set up.

blocks of 512 characters. If we are interested in learning which of our files is largest we may wish to have this sorted by size rather than by name, which is the default way in which *ls* sorts. We could look at the many options of *ls* to see if there was an option to do this but would eventually discover that there is not. Instead we can use a couple of simple options of the *sort* command, combining it with *ls* to get what we want.

The *-n* option of *sort* specifies a numeric sort rather than an alphabetic sort. Thus

```
ls -s | sort -n
```

specifies that the output of the *ls* command run with the option *-s* is to be *piped* to the command *sort* run with the numeric sort option. This would give us a sorted list of our files by size, but with the smallest first. We could then use the *-r* reverse sort option and the *head* command in combination with the previous command doing

```
ls -s | sort -n -r | head -5
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the *sort* command asking it to sort numerically in reverse order (largest first). This output has then been run into the command *head* which gives us the first few lines out. In this case we have asked *head* for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

The metanotation introduced above is called the *pipe* mechanism. Commands separated by '|' characters are connected together by the shell and the output of each is run into the input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism; one important use of pipes which is illustrated there is in the routing of information to the line printer.

1.6. Filenames

Many commands to be executed will need the names of files as arguments. UNIX pathnames consist of a number of components separated by '/'. Each component except the last names a directory in which the next component resides. Thus the pathname

```
/etc/motd
```

specifies a file in the directory 'etc' which is a subdirectory of the *root* directory '/'. Within this directory the file named is 'motd' which stands for 'message of the day'. Filenames which do not begin with '/' are interpreted starting at the current *working* directory. This directory is, by default, your *home* directory and can be changed dynamically by the *chdir* change directory command.

Most filenames consist of a number of alphanumeric characters and '.'s. In fact, all printing characters except '/' may appear in filenames. It is inconvenient to have most non-alphabetic characters in filenames because many of these have special meaning to the shell. The character '.' is not a shell-metacharacter and is often used as the prefix with an *extension* of a base name. Thus

```
prog.c prog.o prog.errs prog.output
```

are four related files. They share a *root* portion of a name (a root portion being

that part of the name that is left when a trailing '.' and following characters which are not '.' are stripped off). The file 'prog.c' might be the source for a C program, the file 'prog.o' the corresponding object file, the file 'prog.errs' the errors resulting from a compilation of the program and the file 'prog.output' the output of a run of the program.

If we wished to refer to all four of these files in a command, we could use the metanotation

```
prog.*
```

This word is expanded by the shell, before the command to which it is an argument is executed, into a list of names which begin with 'prog.'. The character '*' here matches any sequence (including the empty sequence) of characters in a file name. The names which match are sorted into the argument list to the command alphabetically. Thus the command

```
echo prog.*
```

will echo the names

```
prog.c prog.errs prog.o prog.output
```

Note that the names are in lexicographic order here, and a different order than we listed them above. The *echo* command receives four words as arguments, even though we only typed one word as an argument directly. The four words were generated by filename expansion of the metasyntax in the one input word.

Other metanotations for *filename expansion* are also available. The character '?' matches any single character in a filename. Thus

```
echo ? ?? ???
```

will echo a line of filenames; first those with one character names, then those with two character names, and finally those with three character names. The names of each length will be independently lexicographically sorted.

Another mechanism consists of a sequence of characters between '[' and ']'. This metasequence matches any single character from the enclosed set. Thus

```
prog.[co]
```

will match

```
prog.c prog.o
```

in the example above. We can also place two characters astride a '-' in this notation to denote a range. Thus

```
chap.[1-5]
```

might match files

```
chap.1 chap.2 chap.3 chap.4 chap.5
```

if they existed. This is shorthand for

```
chap.[12345]
```

and otherwise equivalent.

An important point to note is that if a list of argument words to a command (an *argument list*) contains filename expansion syntax, and if this filename expansion syntax fails to match any existing file names, then the shell considers this to be an error and prints a diagnostic

No match.

Another very important point is that the character '.' at the beginning of a filename is treated specially. Neither '*' or '?' or the '[' ']' mechanism will match it. This prevents accidental matching of the filenames '.' and '..' in the current directory which have special meaning to the system, as well as other files such as *.cshrc* which are not normally visible. We will discuss the special role of the file *.cshrc* later.

Another filename expansion mechanism gives access to the pathname of the *home* directory of other users. This notation consists of the character '~' followed by another users login name. For instance the word '~bill' would map to the pathname '/usr/bill' if the home directory for 'bill' was in the directory '/usr/bill'. Since, on large systems, users may have login directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of a '~' alone, e.g. '~mbox'. This notation is expanded by the shell into the file 'mbox' in your *home* directory, i.e. into '/usr/bill/mbox' for me on Ernie Co-vax, the UCB Computer Science Department Vax machine, where this document was prepared. This can be very useful if you have used *chdir* to change to another users directory and have found a file you wish to copy using *cp*. You can do

```
cp thatfile ~
```

which will be expanded by the shell to

```
cp thatfile /usr/bill
```

e.g., which the copy command will interpret as a request to make a copy of 'thatfile' in the directory '/usr/bill'. The '~' notation doesn't, by itself, force named files to exist. This is useful, for example, when using the *cp* command, e.g.

```
cp thatfile ~/saveit
```

There also exists a mechanism using the characters '{' and '}' for abbreviating a set of word which have common parts but cannot be abbreviated by the above mechanisms because they are not files, are the names of files which do not yet exist, are not thus conveniently described. This mechanism will be described much later, in section 4.1, as it is used less frequently.

1.7. Quotation

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem in that we cannot use them directly as parts of words. Thus the command

```
echo *
```

will not echo the character '*'. It will either echo an sorted list of filenames in the current directory, or print the message 'No match' if there are no files in the current directory.

The recommended mechanism for placing characters which are neither numbers, digits, '/', '.', or '-' in an argument word to a command is to enclose it with single quotation characters "'", i.e.

```
echo '*'
```

There is one special character '!' which is used by the *history* mechanism of the

shell and which cannot be escaped by placing it within ` characters. It and the character `"' itself can be preceded by a single `\'` to prevent their special meaning. Thus

```
echo \\'!
```

prints

```
!
```

These two mechanisms suffice to place any printing character into a word which is an argument to a shell command. They can be combined, as in

```
echo \'"'
```

which prints

```
'"
```

1.8. Terminating commands

When you are running a command from the shell and the shell is dormant waiting for it to complete there are a couple of ways in which you can force such a command to stop. For instance if you type the command

```
cat /etc/passwd
```

the system will print a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an `INTERRUPT` signal to the `cat` command by hitting the `DEL` or `RUBOUT` key on your terminal. Actually, hitting this key sends this `INTERRUPT` signal to all programs running on your terminal, including your shell. The shell normally ignores such signals however, so that the only program affected by the `INTERRUPT` will be `cat`. Since `cat` does not take any precautions to catch this signal the `INTERRUPT` will cause it to terminate. The shell notices that `cat` has died and prompts you again with `%` . If you hit `INTERRUPT` again, the shell will just repeat its prompt since it catches `INTERRUPT` signals and chooses to continue to execute commands rather than going away like `cat` did, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. Thus the `mail` program in the first example above was terminated when we hit a control-d which generates an end-of-file from the standard input. The shell also terminates when it gets an end-of-file printing `'logout'`; `UNIX` then logs you off the system. Since this means that typing too many control-d's can accidentally log us off, the shell has a mechanism for preventing this. This `ignoreeof` option will be discussed in section 2.2.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus if we execute

```
mail bill < prepared.text
```

the mail command will terminate without our typing a control-d. This is because it read to the end-of-file of our file `'prepared.text'` in which we placed a message for `'bill'` with an editor. We could also have done

```
cat prepared.text | mail bill
```

since the `cat` command would then have written the text through the pipe to the standard input of the mail command. When the `cat` command completed it would have terminated, closing down the pipeline and the `mail` command would have received an end-of-file from it and terminated. Using a pipe here is more complicated than redirecting input so we would more likely use the first form.

These commands could also have been stopped by sending an `INTERRUPT`.

If you write or run programs which are not fully debugged then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a `QUIT` signal, generated by a `control-\`. This will usually provoke the shell to produce a message like:

```
a.out: Quit -- Core dumped
```

indicating that a file 'core' has been created containing information about the program 'a.out's state when it ran amuck. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

If you run background commands (as explained in section 2.6) then these commands will ignore `INTERRUPT` and `QUIT` signals at the terminal. To stop them you must use the *kill* program. See section 2.6 for an example.

If you want to examine the output of a command without having it shoot off the screen as the output of the

```
cat /etc/passwd
```

command will, you can use the command

```
more /etc/passwd
```

The *more* program pauses after each complete screenful and types '`--More--`' at which point you can hit a space to get another screenful, a return to get another line, or a 'q' to end the *more* program. You can also use *more* as a filter, i.e.

```
cat /etc/passwd | more
```

works just like the more simple *more* command above.

For stopping output of commands not involving *more* you can use the `control-s` key to stop the typeout. The typeout will resume when you hit `control-q` or any other key, but `control-q` is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type `control-s` and `control-q` fast enough to paginate the output nicely, and a program like *more* is usually used.

1.9. What now?

We have so far seen a number of mechanisms of the shell and learned a lot about the way in which it operates. The remaining sections will go yet further into the internals of the shell, but you will surely want to try using the shell before you go any further. To try it you can log in to UNIX and type the following command to the system:

```
chsh myname /bin/csh
```

Here 'myname' should be replaced by the name you typed to the system prompt of 'login:' to get onto the system. Thus I would use 'chsh bill /bin/csh'. **You only have to do this once; it takes effect at next login.** You are now ready to try using *csh*.

Before you do the 'chsh' command, the shell you are using when you log into the system is '/bin/sh'. In fact, much of the above discussion is applicable to '/bin/sh'. The next section will introduce many features particular to *csh* so you should change your shell to *csh* before you begin reading it.

2. Details on the shell for terminal users

2.1. Shell startup and termination

When you login, the shell is placed by the system in your *home* directory and begins by reading commands from a file *.cshrc* in this directory. All shells which you may create during your terminal session will read from this file. We will later see what kinds of commands are usefully placed there. For now we need not have this file and the shell does not complain about its absence.

A *login* shell, executed after you login to the system, will, after it reads commands from *.cshrc*, read commands from a file *.login* also in your home directory. This file contains commands which you wish to do each time you login to the UNIX system. My *.login* file looks something like:

```
set ignoreeof mail=(/usr/spool/mail/bill)
switch(`tty`)
  case /dev/ttyd*:
    setenv TERM 3a
    breaksw
endsw
tset -e -Q
echo "${prompt}users" ; users
set time=15
msgs -f
if (-e $mail) then
  echo "${prompt}mail"
  mail
endif
```

This file contains several commands to be executed by UNIX each time I login. The first is a *set* command which is interpreted directly by the shell. It sets the shell variable *ignoreeof* which causes the shell to not log me off if I hit control-d. Rather, I use the *logout* command to log off of the system. By setting the *mail* variable, I ask the shell to watch for incoming mail to me. Every 5 minutes the shell looks for this file and tells me if more mail has arrived there.

The next series of commands causes the shell to examine the output of the *'tty'* command, and if this command returns a teletype path name of the form *'/dev/ttyd*'*, then I have logged in on a dialup and the shell sets the terminal type in the environment to *adm3a*, since this is the type of terminal I normally dial in on. If I am not on a dialup, then the system attempts to set the terminal type from a table of types it has for hardwired ports.

The *tset* command next sets up any special initialization I require on the terminal I am using. The *'-e'* option forces *tset* to always use a control-h as my erase character (even on a hardcopy terminal), and the *'-Q'* option causes it to be quiet, not printing any messages (since I am familiar with what it is doing, I don't need to be reminded.)

Next I set the shell variable *'time'* to *'15'* causing the shell to automatically print out statistics lines for commands which execute for at least 15 seconds of machine time. I then run the *'msgs'* program, which provides me with any system messages which I haven't seen before; the *'-f'* option here prevents it from telling me if there are no new messages. Finally, if my mailbox file exists, then I run the *'mail'* program to process my mail.

When the *'mail'* and *'msgs'* programs finish, the shell will complete execution of the *.login* script and begin reading commands from the terminal.

prompting for each with '% '. When it receives an end-of-file from the terminal, the shell will print 'logout' and execute commands from the file '.logout' in your home directory. After that the shell will die and UNIX will log you off the system. If the system is not going down, you will receive a new login message. In any case, after the 'logout' message the shell is doomed and will take no further input from the terminal.

2.2. Shell variables

The shell maintains a set of *variables*. We saw above the variables *history* and *time* which had values '20' and '15'. In fact, each shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the *set* command. It has several forms, the most useful of which was given above and is

```
set name=value
```

Shell variables may be used to store values which are to be reintroduced into commands later through a substitution mechanism. The shell variables most commonly referenced are, however, those which the shell itself refers to. By changing the values of these variables one can directly affect the behavior of the shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the shell searches for commands. The *set* command shows the value of all variables currently defined (we usually say *set*) in the shell. The default value for *path* will be shown by *set* to be

```
% set
argv      ()
home      /usr/bill
path      (. /usr/ucb /bin /usr/bin)
prompt    %
shell     /bin/csh
status    0
%
```

This notation indicates that the variable *path* points to the current directory '.' and then '/bin' and '/usr/bin'. Commands which you may write might be in '.' (usually one of your directories). Other commands, developed at Berkeley, live in '/usr/ucb' while commands developed at Bell Laboratories live in '/bin' and '/usr/bin'.

A number of locally developed programs on the system live in the directory '/usr/local'. If we wish, as well we might, all shells which we invoke to have access to these new programs we can place the command

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in our file *cshrc* in our home directory. Try doing this and then logging out and back in and do

```
set
```

again to see that the value assigned to *path* has changed.*

*The current version of *csh* does not correctly export the path contained in 'path' to the environment as the standard version 7 variable *PATH*, thus you should concoct a string consisting of the words of 'path' and 'setenv' it into the variable *PATH*, i.e.:

```
setenv PATH .:/usr/ucb:/bin:/usr/bin:/usr/local
```

One thing you should be aware of is that the shell examines each directory which you insert into your path and determines which commands are contained there. Except for the current directory '.', which the shell treats specially, this means that if commands are added to a directory in your search path after you have started the shell, they will not necessarily be found by the shell. If you wish to use a command which has been added in this way, you should give the command

```
rehash
```

to the shell, which will cause it to recompute its internal table of command locations, so that it will find the newly added command.

Other useful built in variables are the variable *home* which shows your home directory, the variable *ignoreeof* which can be set in your *.login* file to tell the shell not to exit when it receives an end-of-file from a terminal (as described above). The variable 'ignoreeof' is one of several variables which the shell does not care about the value of, only whether they are *set* or *unset*. Thus to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

These give the variable 'ignoreeof' no value, but none is desired or required.

Finally, some other built-in shell variables of use are the variables *noclobber* and *mail*. The metasyntax

```
> filename
```

which redirects the output of a command will overwrite and destroy the previous contents of the named file. In this way you may accidentally overwrite a file which is valuable. If you would prefer that the shell not overwrite files in this way you can

```
set noclobber
```

in your *.login* file. Then trying to do

```
date > now
```

would cause a diagnostic if 'now' existed already. You could type

```
date >! now
```

if you really wanted to overwrite the contents of 'now'. The '>!' is a special metasyntax indicating that clobbering the file is ok. **The space between the '!' and the word 'now' is critical here, as '!now' would be a invocation of the *history* mechanism, and have a totally different effect.**

2.3. The shell's history list

The shell can maintain a history list into which it places the words of previous commands. It is possible to use a metanotation to reintroduce commands or words from commands in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

in the example above.

The following figure gives a sample session involving typical usage of the history mechanism of the shell.

```
% cat bug.c
main()
{
    printf("hello");
}
% cc !$
cc bug.c
"bug.c", line 3: newline in string or char constant
"bug.c", line 4: syntax error
% ex !$
ex bug.c
"bug.c" 4 lines, 28 characters
:3s/)/;/"&
    printf("hello");
:wq
"bug.c" 4 lines, 29 characters
%!c
cc bug.c
% a.out
hello%!e
ex bug.c
"bug.c" 4 lines, 29 characters
:3s/lo/lo\\n
    printf("hello\n");
:wq
"bug.c" 4 lines, 31 characters
%!c -o bug
cc bug.c -o bug
% size a.out bug
a.out: 2784+364+1028 = 4176b = 0x1050b
bug: 2784+364+1028 = 4176b = 0x1050b
% ls -l !*
ls -l a.out bug
-rwxr-xr-x 1 bill    3932 Dec 19 09:41 a.out
-rwxr-xr-x 1 bill    3932 Dec 19 09:42 bug
% bug
hello
% pr bug.c | sps
sps: Command not found.
% ^sps^ssp
pr bug.c | ssp
Dec 19 09:41 1979 bug.c Page 1

main()
{
    printf("hello\n");
}

% !! | lpr
pr bug.c | ssp | lpr
%
```

In this example we have a very simple C program which has a bug (or two) in it in the file 'bug.c', which we 'cat' out on our terminal. We then try to run the C compiler on it, referring to the file again as '!\$', meaning the last argument to the previous command. Here the '!' is the history mechanism invocation character, and the '\$' stands for the last argument, by analogy to '\$' in the editor which stands for the end of the line. The shell echoed the command, as it would have been typed without use of the history mechanism, and then executed it. The compilation yielded error diagnostics so we now run the editor on the file we were trying to compile, fix the bug, and run the C compiler again, this time referring to this command simply as '!c', which repeats the last command which started with the letter 'c'. If there were other commands starting with 'c' done recently we could have said '!cc' or even '!cc:p' which would have printed the last command starting with 'cc' without executing it.

After this recompilation, we ran the resulting 'a.out' file, and then noting that there still was a bug, ran the editor again. After fixing the program we ran the C compiler again, but tacked onto the command an extra '-o bug' telling the compiler to place the resultant binary in the file 'bug' rather than 'a.out'. In general, the history mechanisms may be used anywhere in the formation of new commands and other characters may be placed before and after the substituted commands.

We then ran the 'size' command to see how large the binary program images we have created were, and then an 'ls -l' command with the same argument list, denoting the argument list '*'. Finally we ran the program 'bug' to see that its output is indeed correct.

To make a listing of the program we ran the 'pr' command on the file 'bug.c'. In order to compress out blank lines in the output of 'pr' we ran the output through the filter 'ssp', but misspelled it as sps. To correct this we used a shell substitute, placing the old text and new text between '^' characters. This is similar to the substitute command in the editor. Finally, we repeated the same command with '!!', but sent its output to the line printer.

There are other mechanisms available for repeating commands. A *history* command will print out a number of previous commands with numbers by which they can be referenced. There is a way to refer to a previous command by searching for a string which appeared in it, and there are other, less useful, ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the C shell manual pages in the UNIX Programmers Manual.

2.4. Aliases

The shell has an *alias* mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The alias facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using shell command files, but these take place in another instance of the shell and cannot directly affect the current shell's environment or involve commands such as *chdir* which must be done in the current shell.

As an example, suppose that there is a new version of the mail program on the system called 'newmail' you wish to use, rather than the standard mail program which is called 'mail'. If you place the shell command

```
alias mail newmail
```

in your *.cshrc* file, the shell will transform an input line of the form

mail bill

into a call on 'newmail'. More generally, suppose we wish the command 'ls' to always show sizes of files, that is to always do '-s'. We can do

```
alias ls ls -s
```

or even

```
alias dir ls -s
```

creating a new command syntax 'dir' which does an 'ls -s'. If we say

```
dir ~bill
```

then the shell will translate this to

```
ls -s /mnt/bill
```

Thus the *alias* mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines, showing where the arguments to the original command are to be substituted using the facilities of the history mechanism. Thus the definition

```
alias cd 'cd \!* ; ls '
```

would do an *ls* command after each change directory *cd* command. We enclosed the entire alias definition in "" characters to prevent most substitutions from occurring and the character ';' from being recognized as a parser metacharacter. The '!' here is escaped with a '\' to prevent it from being interpreted when the alias command is typed in. The '\!*' here substitutes the entire argument list to the pre-aliasing *cd* command, without giving an error if there were no arguments. The ';' separating commands is used here to indicate that one command is to be done and then the next. Similarly the definition

```
alias whois 'grep \!↑ /etc/passwd'
```

defines a command which looks up its first argument in the password file.

Warning: The shell currently parses the *.cshrc* file each time it starts up. If you place a large number of commands there, shells will tend to start slowly. A mechanism for saving the shell environment after reading the *.cshrc* file and quickly restoring it is under development, but for now you should try to limit the number of aliases you have to a reasonable number... 10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up shells, and make the system seem sluggish when you execute commands from within the editor and other programs.

2.5. Detached commands; >> and >& redirection

There are a few more metanotations useful to the terminal user which have not been introduced yet. The metacharacter '&' may be placed after a command, or after a sequence of commands separated by ';' or '|'. This causes the shell to not wait for the commands to terminate before prompting again. We say that they are *detached* or *background* processes. Thus

```
% pr ~bill/csh/sh.c | lpr &  
5120  
5121  
%
```


Here the shell printed two numbers and came back very quickly rather than waiting for the *pr* and *lpr* commands to finish. These numbers are the process numbers assigned by the system to the *pr* and *lpr* commands.

Since *havoc* would result if a command run in the background were to read from your terminal at the same time as the shell does, the default standard input for a command run in the background is not your terminal, but an empty file called *'/dev/null'*. Commands run in the background are also made immune to *INTERRUPT* and *QUIT* signals which you may subsequently generate at your terminal, and are not killed if you hang up a phone connection.

In addition to the standard output, commands also have a diagnostic output which is normally directed to the terminal even when the standard output is directed to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces you can do

```
command >& file
```

The *'>&'* here tells the shell to route both the diagnostic output and the standard output into *'file'*. Similarly you can give the command

```
command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon *lpr*.#

Finally, it is possible to use the form

```
command >> file
```

to place output at the end of an existing file.†

2.6. Useful built-in commands

We now give a few of the useful built-in commands of the shell describing how they are used.

The *alias* command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given an argument such as

```
alias ls
```

to show the current alias for, e.g., *'ls'*.

The *cd* and *chdir* commands are equivalent, and change the working directory of the shell. It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. Thus after you login you can do

#A command form

```
command >&! file
```

exists, and is used when *noclobber* is set and *file* already exists.

†If *noclobber* is set, then an error will result if *file* does not exist, otherwise the shell will create *file* if it doesn't exist. A form

```
command >>! file
```

makes it not be an error for *file* to not exist when *noclobber* is set.

```
% pwd
/usr/bill
% mkdir newspaper
% chdir newspaper
% pwd
/usr/bill/newspaper
%
```

after which you will be in the directory *newspaper*. You can place a group of related files there. You can return to your 'home' login directory by doing just

```
chdir
```

with no arguments. We used the *pwd* print working directory command to show the name of the current directory here. The current directory will usually be a subdirectory of your home directory, and have it (here '/usr/bill') at the start of it.

The *echo* command prints its arguments. It is often used in shell scripts or as an interactive command to see what filename expansions will yield.

The *history* command will show the contents of the history list. The numbers given with the history events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a shell variable called *prompt*. By placing a '!' character in its value the shell will there substitute the index of the current command in the history list. You can use this number to refer to this command in a history substitution. Thus you could

```
set prompt='\! % '
```

Note that the '!' character had to be escaped here even within "" characters.

The *logout* command can be used to terminate a login shell which has *ignoreeof* set.

The *rehash* command causes the shell to recompute a table of where commands are located. This is necessary if you add a command to a directory in the current shells search path and wish the shell to find it, since otherwise the hashing algorithm may tell the shell that the command wasn't in that directory when the hash table was computed.

The *repeat* command can be used to repeat a command several times. Thus to make 5 copies of the file *one* in the file *five* you could do

```
repeat 5 cat one >> five
```

The *setenv* command can be used, on version 7 UNIX systems, to set variables in the environment. Thus

```
setenv TERM adm3a
```

will set the value of the environment variable *TERM* to 'adm3a'. A user program *printenv* exists which will print out the environment. It might then show:

```
% printenv
HOME=/
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=3a
%
```

The *source* command can be used to force the current shell to read commands from a file. Thus

```
source .cshrc
```

can be used after editing in a change to the *.cshrc* file which you wish to take effect before the next time you login.

The *time* command can be used to cause a command to be timed no matter how much CPU time it takes. Thus

```
% time cp five five.save
0.0u 0.3s 0:01 26%
% time wc five.save
 1200  6300 37850 five.save
1.2u 0.5s 0:03 55%
%
```

indicates that the *cp* command used less than 1/10th of a second of user time and only 3/10th of a second of system time in copying the file 'five' to 'five.save'. The command word count 'wc' on the other hand used 1.2 seconds of user time and 0.5 seconds of system time in 3 seconds of elapsed time in counting the number of words, character and lines in 'five.save'. The percentage '55%' indicates that over this period of 3 seconds, our command 'wc' used an average of 55 percent of the available CPU cycles of the machine. This is a very high percentage and indicates that the system is lightly loaded.

The *unalias* and *unset* commands can be used to remove aliases and variable definitions from the shell.

The *wait* command can be used after starting processes with '&' to quickly see if they have finished. If the shell responds immediately with another prompt, they have. Otherwise you can wait for the shell to prompt at which point they will have finished, or interrupt the shell by sending a RUB or DELETE character. If the shell is interrupted, it will print the names and numbers of the processes it knows to be unfinished. Thus:

```
% nroff paper | lpr &
2450
2451
% wait
 2451 lpr
 2450 nroff
wait: Interrupted.
%
```

If you run a background process and decide you want to stop it for whatever reason you must use the *kill* program. You must use the number of the processes you wish to kill. Thus to stop the *nroff* in the above pipeline you would do

```
% kill 2450
% wait
2450: nroff: Terminated.
%
```

Here the shell printed a diagnostic that we terminated 'nroff' only after we did a *wait*. If we want the shell to discover the termination of all processes it has created we must, in general, use *wait*.

If you don't remember the number of a command you have executed, you can also issue a `ps` command, which will print out the numbers and names of the processes (other than shells) which you are running, showing with each the amount of processor time it has used so far.

2.7. What else?

This concludes the basic discussion of the shell for terminal users. There are more features of the shell to be discussed here, and all features of the shell are discussed in its manual pages. One useful feature which is discussed later is the `foreach` built-in command which can be used to run the same command sequence with a number of different arguments.

If you intend to use UNIX a lot you should look through the rest of this document and the shell manual pages to become familiar with the other facilities which are available to you.

3. Shell control structures and command scripts

3.1. Introduction

It is possible to place commands in files and to cause shells to be invoked to read and execute commands from these files, which are called *shell scripts*. We here detail those features of the shell useful to the writers of such scripts.

3.2. Make

It is important to first note what shell scripts are *not* useful for. There is a program called *make* which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance a large program consisting of one or more files can have its dependencies described in a *makefile* which contains definitions of the commands used to create these different files when changes occur. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this *makefile*. This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a *makefile* may be created which defines how different versions of the document are to be created and which options of *nroff* or *troff* are appropriate.

3.3. Invocation and the argv variable

A *csh* command script may be interpreted by saying

```
% csh script ...
```

where *script* is the name of the file containing a group of *csh* commands and '...' is replaced by a sequence of arguments. The shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file 'script' executable by doing

```
chmod 755 script
```

and place a shell comment at the beginning of the shell script (i.e. begin the file with a '#' character) then a '/bin/csh' will automatically be invoked to execute 'script' when you type

```
script
```

If the file does not begin with a '#' then the standard shell '/bin/sh' will be used to execute it. This allows you to convert your older shell scripts to use *csh* at your convenience.

3.4. Variable substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism known as *variable substitution* is done on these words. Keyed by the character '\$' this substitution replaces the names of variables by their values. Thus

```
echo $argv
```

when placed in a command script would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be

unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

`$?name`

expands to '1' if name is *set* or to '0' if name is not *set*. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

`$#name`

expands to the number of elements in the variable *name*. Thus

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus

`$argv[1]`

gives the first component of *argv* or in the example above 'a'. Similarly

`$argv[$#argv]`

would give 'c', and

`$argv[1-2]`

Other notations useful in shell scripts are

`$n`

where *n* is an integer as a shorthand for

`$argv[n]`

the *n*th parameter and

`$*`

which is a shorthand for

`$argv`

The form

`$$`

expands to the process number of the current shell. Since this process number is unique in the system it can be used in generation of unique temporary file names.

One minor difference between '`$n`' and '`$argv[n]`' should be noted here.

The form '\$argv[n]' will yield an error if n is not in the range '1- $\$#argv$ ' while '\$ n ' will never yield an out of range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form ' n -'; if there are less than n components of the given variable then no words are substituted. A range of the form ' m - n ' likewise returns an empty vector without giving an error when m exceeds the number of elements of the given variable, provided the subscript n is in range.

3.5. Expressions

In order for interesting shell scripts to be constructed it must be possible to evaluate expressions in the shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the shell with the same precedence that they have in C. In particular, the operations '==' and '!=' compare strings and the operators '&&' and '||' implement the boolean and/or operations.

The shell also allows file enquiries of the form

-? filename

where '?' is replaced by a number of single characters. For instance the expression primitive

-e filename

tell whether the file 'filename' exists. Other primitives test for read, write and execute access to the file, whether it is a directory, or has non-zero length.

It is possible to test whether a command terminates normally, by a primitive of the form '{ command }' which returns true, i.e. '1' if the command succeeds exiting normally with exit status 0, or '0' if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable '\$status' examined in the next command. Since '\$status' is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

For a full list of expression components available see the manual section for the shell.

3.6. Sample shell script

A sample shell script which makes use of the expression mechanism of the shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)

    if ($i:r.c != $i) continue# not a .c file so do nothing

    if (! -r ~/backup/$i:t) then
    echo $i:t not in backup... not cp\`ed
    continue
    endif

    cmp -s $i ~/backup/$i:t# to set $status

    if ($status != 0) then
    echo new backup of $i
    cp $i ~/backup/$i:t
    endif

end
```

This script makes use of the *foreach* command, which causes the shell to execute the commands between the *foreach* and the matching *end* for each of the values given between '(' and ')' with the named variable, in this case 'i' set to successive values in the list. Within this loop we may use the command *break* to stop executing the loop and *continue* to prematurely terminate one iteration and begin the next. After the *foreach* loop the iteration variable (*i* in this case) has the value at the last iteration.

We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are filenames which have already been expanded or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a '\$' variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form

```
if ( expression ) then
    command
    ...
endif
```

The placement of the keywords here is **not** flexible due to the current implementation of the shell.†

†The following two formats are not currently acceptable to the shell:

```
if ( expression )          # Won't work!
then
    command
    ...
endif

and

if ( expression ) then command endif          # Won't work
```


The shell does have another form of the if statement of the form

```
if ( expression ) command
```

which can be written

```
if ( expression ) \  
    command
```

Here we have escaped the newline for the sake of appearance. The command must not involve '|', '&' or ';' and must not be another control command. The second form requires the final '\' to **immediately** precede the end-of-line.

The more general *if* statements above also admit a sequence of *else-if* pairs followed by a single *else* and an *endif*, e.g.:

```
if ( expression ) then  
    commands  
else if (expression ) then  
    commands  
...  
  
else  
    commands  
  
endif
```

Another important mechanism used in shell scripts is ':' modifiers. We can use the modifier ':r' here to extract a root of a filename. Thus if the variable *i* has the value 'foo.bar' then

```
% echo $i $i:r  
foo.bar foo  
%
```

shows how the ':r' modifier strips off the trailing '.bar'. Other modifiers will take off the last component of a pathname leaving the head ':h' or all but the last component of a pathname leaving the tail ':t'. These modifiers are fully described in the *csh* manual pages in the programmers manual. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the shells environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ':' modification mechanism. Finally, we note that the character '#' lexically introduces a shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a '#' are discarded by the shell. This character can be quoted using '"' or '\' to place it in an argument word.

#It is also important to note that the current implementation of the shell limits the number of ':' modifiers on a '\$' substitution to 1. Thus

```
% echo $i $i:h:t  
/a/b/c /a/b:t  
%
```

does not do what one would expect.

3.7. Other control structures

The shell also has control structures *while* and *switch* similar to those of C. These take the forms

```
while ( expression )
    commands
end
```

and

```
switch ( word )
case str1:
    commands
    breaksw
...
case strn:
    commands
    breaksw
default:
    commands
    breaksw
endsw
```

For details see the manual section for *cs**h*. C programmers should note that we use *breaksw* to exit from a *switch* while *break* exits a *while* or *foreach* loop. A common mistake to make in *cs**h* scripts is to use *break* rather than *breaksw* in switches.

Finally, *cs**h* allows a *goto* statement, with labels looking like they do in C, i.e.:

```
loop:
    commands
    goto loop
```

3.8. Supplying input to commands

Commands run from shell scripts receive by default the standard input of the shell which is running the script. This it is different from previous shells running under UNIX. It allowing shell scripts to fully participate in pipelines, but mandates extra notation for commands which are to take inline data.

Thus we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/^[ ]*//
w
q
'EOF'
end
%
```

The notation '<< 'EOF'' means that the standard input for the *ed* command is to come from the text in the shell script file up to the next line consisting of exactly 'EOF'. The fact that the 'EOF' is enclosed in '' characters, i.e. quoted, causes the shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the '<<' which the shell uses to terminate the text to be given to the command is quoted then these substitutions will not be performed. In this case since we used the form '1,\$' in our editor script we needed to insure that this '\$' was not variable substituted. We could also have insured this by preceding the '\$' here with a '\', i.e.:

```
1,\$s/^[ ]*//
```

but quoting the 'EOF' terminator is a more reliable way of achieving the same thing.

3.9. Catching interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do

```
onintr label
```

where *label* is a label in our program. If an interrupt is received the shell will do a 'goto label' and we can remove the temporary files and then do a *exit* command (which is built in to the shell) to exit from the shell script. If we wish to exit with a non-zero status we can do

```
exit(1)
```

e.g. to exit with status '1'.

3.10. What else?

There are other features of the shell useful to writers of shell procedures. The *verbose* and *echo* options and the related *-v* and *-x* command line options can be used to help trace the actions of the shell. The *-n* option causes the shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that *cs/h* will not execute shell scripts which do not begin with the character '#', that is shell scripts that do not begin with a comment. Similarly, the '/bin/sh' on your system may well defer to 'csh' to interpret shell scripts which begin with '#'. This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using '''' which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as '' does.

4. Other, less commonly used, shell features

4.1. Loops at the terminal; variables as vectors

It is occasionally useful to use the *foreach* control structure at the terminal to aid in performing a number of similar commands. For instance, there were at one point three shells in use on the Cory UNIX system at Cory Hall, `/bin/sh`, `/bin/nsh`, and `/bin/csh`. To count the number of persons using each shell one could have issued the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar we can use *foreach* to do this more easily.

```
% foreach i ('sh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with `?` when reading the body of the loop.

Very useful with loops are variables which contain lists of filenames or other words. You can, for example, do

```
% set a=('ls')
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The *set* command here gave the variable *a* a list of all the filenames in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within `""` characters is converted by the shell to a list of words. You can also place the `""` quoted string within `""` characters to take each (non-empty) line as a component of the variable; preventing the lines from being split into words at blanks and tabs. A modifier `:x` exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

4.2. Braces { ... } in argument expansion

Another form of filename expansion, alluded to before involves the characters '{' and '}'. These characters specify that the contained strings, separated by ',' are to be consecutively substituted into the containing characters and the results expanded left to right. Thus

```
A{str1,str2,...strn}B
```

expands to

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be applied recursively (i.e. nested). The results of each expanded string are sorted separately, left to right order being preserved. The resulting filenames are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not filenames, but which have common parts.

A typical use of this would be

```
mkdir ~/ {hdrs,retrofit,csh}
```

to make subdirectories 'hdrs', 'retrofit' and 'csh' in your home directory. This mechanism is most useful when the common prefix is longer than in this example, i.e.

```
chown bin /usr/ {bin/ {ex,edit},lib/ {ex1.1strings,how_ex}}
```

4.3. Command substitution

A command enclosed in '`' characters is replaced, just before filenames are expanded, by the output from that command. Thus it is possible to do

```
set pwd=`pwd`
```

to save the current directory in the variable *pwd* or to do

```
ex `grep -l TRACE *.c`
```

to run the editor *ex* suppling as arguments those files whose names end in '.c' which have the string 'TRACE' in them.*

4.4. Other details not covered here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the shell. The exact meaning of certain combinations of quotations is also occasionally important. These are detailed fully in its manual section.

The shell has a number of command line option flags mostly of use in writing UNIX programs, and debugging shell scripts. See the shells manual section for a list of these options.

*Command expansion also occurs in input redirected with '<<' and within ''' quotations. Refer to the shell manual section for full details.

Appendix - Special characters

The following table lists the special characters of *cs*h and the UNIX system, giving for each the section(s) in which it is discussed. A number of these characters also have special meaning in expressions. See the *cs*h manual section for a complete list.

Syntactic metacharacters

;	2.4	separates commands to be executed sequentially
	1.5	separates commands in a pipeline
()	2.2,3.6	brackets expressions and variable values
&	2.5	follows commands to be executed without waiting for completion

Filename metacharacters

/	1.6	separates components of a file's pathname
?	1.6	expansion character matching any single character
*	1.6	expansion character matching any sequence of characters
[]	1.6	expansion sequence matching any single character from a set
~	1.6	used at the beginning of a filename to indicate home directories
{ }	4.2	used to specify groups of arguments with common parts

Quotation metacharacters

\	1.7	prevents meta-meaning of following single character
'	1.7	prevents meta-meaning of a group of characters
"	4.3	like ', but allows variable and command expansion

Input/output metacharacters

<	1.3	indicates redirected input
>	1.5	indicates redirected output

Expansion/substitution metacharacters

\$	3.4	indicates variable substitution
!	2.3	indicates history substitution
:	3.6	precedes substitution modifiers
↑	2.3	used in special forms of history substitution
`	4.3	indicates command substitution

Other metacharacters

#	1.3,3.6	begins scratch file names; indicates shell comments
-	1.2	prefixes option (flag) arguments to commands

Glossary

This glossary lists the most important terms introduced in the introduction to the shell and gives references to sections of the shell document for further information about them. References of the form 'pr (1)' indicate that the command *pr* is in the UNIX programmers manual in section 1. You can get an online copy of its manual page by doing

```
man 1 pr
```

References of the form (2.5) indicate that more information can be found in section 2.5 of this manual.

- . Your current directory has the name '.' as well as the name printed by the command *pwd*. The current directory '.' is usually the first component of the search path contained in the variable *path*, thus commands which are in '.' are found first (2.2). The character '.' is also used in separating components of filenames (1.6). The character '.' at the beginning of a component of a path-name is treated specially and not matched by the filename expansion metacharacters '?', '*', and '[' ']' pairs (1.6).
- .. Each directory has a file '..' in it which is a reference to its *parent* directory. After changing into the directory with *chdir*, i.e.

```
chdir paper
```

you can return to the parent directory by doing

```
chdir ..
```

The current directory is printed by *pwd* (2.6).
- a.out Compilers which create executable images create them, by default, in the file 'a.out', for historical reasons (2.3).
- alias An *alias* specifies a shorter or different name for a UNIX command, or a transformation on a command to be performed in the shell. The shell has a command *alias* which establishes aliases and can print their current values. The command *unalias* is used to remove aliases (2.6).
- argument Commands in UNIX receive a list of argument words. Thus the command

```
echo a b c
```

consists of a command name 'echo' and three argument words 'a', 'b' and 'c' (1.1).
- argv The list of arguments to a command written in the shell language (a shell script or shell procedure) is stored in a variable called *argv* within the shell. This name is taken from the conventional name in the C programming language (3.4).
- background Commands started without waiting for them to complete are called *background* commands (1.5).
- bin A directory containing binaries of programs and shell scripts to be executed is typically called a 'bin' directory. The standard system 'bin' directories are '/bin' containing the most heavily used commands and '/usr/bin' which contains most other user programs. Programs developed at UC Berkeley live in '/usr/ucb', while locally written programs live in '/usr/local'. Games are kept in the directory '/usr/games'. You can place binaries in any

- directory. If you wish to execute them often, the name of the directories should be a component of the variable *path*.
- break** *Break* is a built-in command used to exit from loops within the control structure of the shell (3.6).
- builtin** A command executed directly by the shell is called a *builtin* command. Most commands in UNIX are not built into the shell, but rather exist as files in 'bin' directories. These commands are accessible because the directories in which they reside are named in the *path* variable.
- case** A *case* command is used as a label in a *switch* statement in the shells control structure, similar to that of the language C. Details are given in the shells documentation 'csh(1)' (3.7).
- cat** The *cat* program catenates a list of specified files on the standard output. It is usually used to look at the contents of a single file on the terminal, to 'cat a file' (1.8, 2.3).
- cd** The *cd* command is used to change the working directory. With no arguments, *cd* changes your working directory to be your *home* directory (2.3) (2.6).
- chdir** The *chdir* command is a synonym for *cd*. *Cd* is usually used because it is easier to type.
- chsh** The *chsh* command is used to change the shell which you use on UNIX. By default, you use an different version of the shell which resides in '/bin/sh'. You can change your shell to '/bin/csh' by doing
- ```
chsh your-login-name /bin/csh
```
- Thus I would do
- ```
chsh bill /bin/csh
```
- It is only necessary to do this once. The next time you log in to UNIX after doing this command, you will be using *csh* rather than the shell in '/bin/sh' (1.9).
- cmp** *Cmp* is a program which compares files. It is usually used on binary files, or to see if two files are identical (3.6). For comparing text files the program *diff*, described in 'diff (1)' is used.
- command** A function performed by the system, either by the shell (a builtin command) or by a program residing in a file in a directory within the UNIX system is called a *command* (1.1).
- command substitution** The replacement of a command enclosed in "" characters by the text output by that command is called *command substitution* (3.6, 4.3).
- component** A part of a *pathname* between '/' characters is called a *component* of that pathname. A *variable* which has multiple strings as value is said to have several *components*, each string is a *component* of the variable.
- continue** A builtin command which causes execution of the enclosing *foreach* or *while* loop to cycle prematurely. Similar to the *continue* command in the programming language C (3.6).

- core dump When a program terminates abnormally, the system places an image of its current state in a file named 'core'. This 'core dump' can be examined with the system debuggers 'adb(1)' or 'sdb(1)' in order to determine what went wrong with the program (1.8). If the shell produces a message of the form:
- commandname: Illegal instruction -- Core dumped
- (where 'Illegal instruction' is only one of several possible messages) you should report this to the author of the program or a system administrator, saving the 'core' file.
- cp The *cp* (copy) program is used to copy the contents of one file into another file. It is one of the most commonly used UNIX commands (2.6).
- .cshrc The file *.cshrc* in your *home* directory is read by each shell as it begins execution. It is usually used to change the setting of the variable *path* and to set *alias* parameters which are to take effect globally (2.1).
- date The *date* command prints the current date and time (1.3).
- debugging *Debugging* is the process of correcting mistakes in programs and shell scripts. The shell has several options and variables which may be used to aid in shell debugging (4.4).
- default The label *default:* is used within shell *switch* statements, as it is in the C language to label the code to be executed if none of the *case* labels matches the value switched on (3.7).
- DELETE The DELETE or RUBOUT key on the terminal is used to generate an INTERRUPT signal in UNIX which stops the execution of most programs (2.6).
- detached A command run without waiting for it to complete is said to be detached (2.5).
- diagnostic An error message produced by a program is often referred to as a *diagnostic*. Most error messages are not written to the standard output, since that is often directed away from the terminal (1.3, 1.5). Error messages are instead written to the *diagnostic output* which may be directed away from the terminal, but usually is not. Thus diagnostics will usually appear on the terminal (2.5).
- directory A structure which contains files. At any time you are in one particular directory whose names can be printed by the command 'pwd'. The *chdir* command will change you to another directory, and make the files in that directory visible. The directory in which you are when you first login is your *home* directory (1.1, 1.6).
- echo The *echo* command prints its arguments (1.6, 2.6, 3.6, 3.10).
- else The *else* command is part of the 'if-then-else-endif' control command construct (3.6).
- EOF An *end-of-file* is generated by the terminal by a control-d, and whenever a command reads to the end of a file which it has been given as input. Commands receiving input from a *pipe* receive an end-of-file when the command sending them input completes. Most commands terminate when they receive an end-of-file. The shell has an option to ignore end-of-file from a terminal input which may help you keep from logging out accidentally by typing too many control-d's (1.1, 1.8, 3.8).

- escape** A character `\` used to prevent the special meaning of a metacharacter is said to *escape* the character from its special meaning. Thus
- ```
echo *
```
- will echo the character `*` while just

```
echo *
```

will echo the names of the file in the current directory. In this example, `\` *escapes* `*` (1.7). There is also a non-printing character called *escape*, usually labelled ESC or ALTMODE on terminal keyboards. Some older UNIX systems use this character to indicate that output is to be suspended. Most systems use control-s to stop the output and control-q to start it.

**/etc/passwd** This file contains information about the accounts currently on the system. It consists of a line for each account with fields separated by `:` characters (2.3). You can look at this file by saying

```
cat /etc/passwd
```

The commands *finger* and *grep* are often used to search for information in this file. See `'finger(1)'`, `'passwd(5)'` and `'grep(1)'` for more details.

**exit** The *exit* command is used to force termination of a shell script, and is built into the shell (3.9).

**exit status** A command which discovers a problem may reflect this back to the command (such as a shell) which invoked (executed) it. It does this by returning a non-zero number as its *exit status*, a status of zero being considered 'normal termination'. The *exit* command can be used to force a shell command script to give a non-zero exit status (3.5).

**expansion** The replacement of strings in the shell input which contain metacharacters by other strings is referred to as the process of *expansion*. Thus the replacement of the word `*` by a sorted list of files in the current directory is a 'filename expansion'. Similarly the replacement of the characters `!!` by the text of the last command is a 'history expansion'. Expansions are also referred to as *substitutions* (1.6, 3.4, 4.2).

**expressions** Expressions are used in the shell to control the conditional structures used in the writing of shell scripts and in calculating values for these scripts. The operators available in shell expressions are those of the language C (3.5).

**extension** Filenames often consist of a *root* name and an *extension* separated by the character `.`. By convention, groups of related files often share the same root name. Thus if `'prog.c'` were a C program, then the object file for this program would be stored in `'prog.o'`. Similarly a paper written with the `'-me'` nroff macro package might be stored in `'paper.me'` while a formatted version of this paper might be kept in `'paper.out'` and a list of spelling errors in `'paper.errs'` (1.6).

**filename** Each file in UNIX has a name consisting of up to 14 characters and not including the character '/' which is used in *pathname* building. Most file names do not begin with the character '.', and contain only letters and digits with perhaps a '.' separating the root portion of the filename from an extension (1.6).

**filename expansion**

Filename expansion uses the metacharacters '\*', '?', '[' and ']' to provide a convenient mechanism for naming files. Using filename expansion it is easy to name all the files in the current directory, or all files which have a common root name. Other filename expansion mechanisms use the metacharacter '~' and allow files in other users directories to be named easily (1.6, 4.2).

**flag** Many UNIX commands accept arguments which are not the names of files or other users but are used to modify the action of the commands. These are referred to as *flag* options, and by convention consists of one or more letters preceded by the character '-' (1.2). Thus the *ls* list file commands has an option '-s' to list the sizes of files. This is specified

```
ls -s
```

**foreach** The *foreach* command is used in shell scripts and at the terminal to specify repetition of a sequence of commands while the value of a certain shell variable ranges through a specified list (3.6, 4.1).

**goto** The shell has a command *goto* used in shell scripts to transfer control to a given label (3.7).

**grep** The *grep* command searches through a list of argument files for a specified string. Thus

```
grep bill /etc/passwd
```

will print each line in the file '/etc/passwd' which contains the string 'bill'. Actually, *grep* scans for *regular expressions* in the sense of the editors 'ed(1)' and 'ex(1)' (2.3). *Grep* stands for 'globally find regular expression and print.'

**head** The *head* command prints the first few lines of one or more files. If you have a bunch of files containing text which you are wondering about it is sometimes useful to run *head* with these files as arguments. This will usually show enough of what is in these files to let you decide which you are interested in (1.5).

**history** The *history* mechanism of the shell allows previous commands to be repeated, possibly after modification to correct typing mistakes or to change the meaning of the command. The shell has a *history list* where these commands are kept, and a *history* variable which controls how large this list is (1.7, 2.6).

**home directory** Each user has a home directory, which is given in your entry in the password file, */etc/passwd*. This is the directory which you are placed in when you first log in. The *cd* or *chdir* command with no arguments takes you back to this directory, whose name is recorded in the shell variable *home*. You can also access the home directories of other users in forming filenames using a file expansion notation and the character '~' (1.6).

- if** A conditional command within the shell, the *if* command is used in shell command scripts to make decisions about what course of action to take next (3.6).
- ignoreeof** Normally, your shell will exit, printing 'logout' if you type a control-d at a prompt of '% '. This is the way you usually log off the system. You can set the *ignoreeof* variable if you wish in your *.login* file and then use the command *logout* to logout. This is useful if you sometimes accidentally type too many control-d characters, logging yourself off (2.2, 2.6).
- input** Many commands on UNIX take information from the terminal or from files which they then act on. This information is called *input*. Commands normally read for input from their *standard input* which is, by default, the terminal. This standard input can be redirected from a file using a shell metanotation with the character '<'. Many commands will also read from a file specified as argument. Commands placed in pipelines will read from the output of the previous command in the pipeline. The leftmost command in a pipeline reads from the terminal if you neither redirect its input nor give it a file name to use as standard input. Special mechanisms exist for supplying input to commands in shell scripts (1.2, 1.6, 3.8).
- interrupt** An *interrupt* is a signal to a program that is generated by hitting the RUBOUT or DELETE key. It causes most programs to stop execution. Certain programs such as the shell and the editors handle an interrupt in special ways, usually by stopping what they are doing and prompting for another command. While the shell is executing another command and waiting for it to finish, the shell does not listen to interrupts. The shell often wakes up when you hit interrupt because many commands die when they receive an interrupt (1.8, 2.6, 3.9).
- kill** A program which terminates processes run without waiting for them to complete. (2.6)
- .login** The file *.login* in your *home* directory is read by the shell each time you log in to UNIX and the commands there are executed. There are a number of commands which are usefully placed here especially *tset* commands and *set* commands to the shell itself (2.1).
- logout** The *logout* command causes a login shell to exit. Normally, a login shell will exit when you hit control-d generating an end-of-file, but if you have set *ignoreeof* in you *.login* file then this will not work and you must use *logout* to log off the UNIX system (2.2).
- .logout** When you log off of UNIX the shell will execute commands from the file *.logout* in your *home* directory after it prints 'logout'.
- lpr** The command *lpr* is the line printer daemon. The standard input of *lpr* is spooled and printed on the UNIX line printer. You can also give *lpr* a list of filenames as arguments to be printed. It is most common to use *lpr* as the last component of a *pipeline* (2.3).
- ls** The *ls* list files command is one of the most commonly used UNIX commands. With no argument filenames it prints the names of the files in the current directory. It has a number of useful *flag* arguments, and can also be given the names of directories as arguments, in which case it lists the names of the files in these

- directories (1.2).
- mail The *mail* program is used to send and receive messages from other UNIX users (1.1, 2.2).
- make The *make* command is used to maintain one or more related files and to organize functions to be performed on these files. In many ways *make* is easier to use, and more helpful than shell command scripts (3.2).
- makefile The file containing commands for *make* is called 'makefile' (3.2).
- manual The 'manual' often referred to is the 'UNIX programmers manual.' It contains a number of sections and a description of each UNIX program. An online version of the manual is accessible through the *man* command. Its documentation can be obtained online via
- man man
- metacharacter Many characters which are neither letters nor digits have special meaning either to the shell or to UNIX. These characters are called *metacharacters*. If it is necessary to place these characters in arguments to commands without them having their special meaning then they must be *quoted*. An example of a metacharacter is the character '>' which is used to indicate placement of output into a file. For the purposes of the *history* mechanism, most unquoted metacharacters form separate words (1.4). The appendix to this user's manual lists the metacharacters in groups by their function.
- mkdir The *mkdir* command is used to create a new directory (2.6).
- modifier Substitutions with the history mechanism, keyed by the character '!' or of variables using the metacharacter '\$' are often subjected to modifications, indicated by placing the character ':' after the substitution and following this with the modifier itself. The *command substitution* mechanism can also be used to perform modification in a similar way, but this notation is less clear (3.6).
- noclobber The shell has a variable *noclobber* which may be set in the file *.login* to prevent accidental destruction of files by the '>' output redirection metasyntax of the shell (2.2, 2.5).
- nroff The standard text formatter on UNIX is the program *nroff*. Using *nroff* and one of the available *macro* packages for it, it is possible to have documents automatically formatted and to prepare them for phototypesetting using the typesetter program *troff* (3.2).
- onintr The *onintr* command is built into the shell and is used to control the action of a shell command script when an interrupt signal is received (3.9).
- output Many commands in UNIX result in some lines of text which are called their *output*. This output is usually placed on what is known as the *standard output* which is normally connected to the users terminal. The shell has a syntax using the metacharacter '>' for redirecting the standard output of a command to a file (1.3). Using the *pipe* mechanism and the metacharacter '|' it is also possible for the standard output of one command to become the standard input of another command (1.5). Certain commands such as the line printer daemon *lpr* do not place their results on the standard output but rather in more useful places such as on the line

printer (2.3). Similarly the *write* command places its output on another users terminal rather than its standard output (2.3). Commands also have a *diagnostic output* where they write their error messages. Normally these go to the terminal even if the standard output has been sent to a file or another command, but it is possible to direct error diagnostics along with standard output using a special metanotation (2.5).

**path** The shell has a variable *path* which gives the names of the directories in which it searches for the commands which it is given. It always checks first to see if the command it is given is built into the shell. If it is, then it need not search for the command as it can do it internally. If the command is not builtin, then the shell searches for a file with the name given in each of the directories in the *path* variable, left to right. Since the normal definition of the *path* variable is

```
path (. /usr/ucb /bin /usr/bin)
```

the shell normally looks in the current directory, and then in the standard system directories '/usr/ucb', '/bin' and '/usr/bin' for the named command (2.2). If the command cannot be found the shell will print an error diagnostic. Scripts of shell commands will be executed using another shell to interpret them if they have 'execute' bits set. This is normally true because a command of the form

```
chmod 755 script
```

was executed to turn these execute bits on (3.3). If you add new commands to a directory in the path, you should issue the command 'rehash' (2.2).

**pathname** A list of names, separated by '/' characters forms a *pathname*. Each *component*, between successive '/' characters, names a directory in which the next component file resides. Pathnames which begin with the character '/' are interpreted relative to the *root* directory in the filesystem. Other pathnames are interpreted relative to the current directory as reported by *pwd*. The last component of a pathname may name a directory, but usually names a file.

**pipeline** A group of commands which are connected together, the standard output of each connected to the standard input of the next is called a *pipeline*. The *pipe* mechanism used to connect these commands is indicated by the shell metacharacter '|' (1.5, 2.3).

**pr** The *pr* command is used to prepare listings of the contents of files with headers giving the name of the file and the date and time at which the file was last modified (2.3).

**printenv** The *printenv* command is used on version 7 UNIX systems to print the current setting of variables in the environment (2.6).

**process** A instance of a running program is called a process (2.6). The numbers used by *kill* and printed by *wait* are unique numbers generated for these processes by UNIX. They are useful in *kill* commands which can be used to stop background processes. (2.6)

- program** Usually synonymous with *command*; a binary file or shell command script which performs a useful function is often called a program.
- programmers manual** Also referred to as the *manual*. See the glossary entry for 'manual'.
- prompt** Many programs will print a prompt on the terminal when they expect input. Thus the editor 'ex(1)' will print a ':' when it expects input. The shell prompts for input with '%' and occasionally with '?' when reading commands from the terminal (1.1). The shell has a variable *prompt* which may be set to a different value to change the shells main prompt. This is mostly used when debugging the shell (2.6).
- ps** The *ps* command is used to show the processes you are currently running. Each process is shown with its unique process number, an indication of the terminal name it is attached to, and the amount of CPU time it has used so far. The command is identified by printing some of the words used when it was invoked (2.6). Shells, such as the *csh* you use to run the 'ps' command are not normally shown in the output.
- pwd** The *pwd* command prints the full pathname of the current (working) directory.
- quit** The *quit* signal, generated by a control-\ is used to terminate programs which are behaving unreasonably. It normally produces a core image file (1.8).
- quotation** The process by which metacharacters are prevented their special meaning, usually by using the character '"' in pairs, or by using the character '\ ' is referred to as *quotation* (1.4).
- redirection** The routing of input or output from or to a file is known as *redirection* of input or output (1.3).
- repeat** The *repeat* command iterates another command a specified number of times (2.6).
- RUBOUT** The RUBOUT or DELETE key generates an interrupt signal which is used to stop programs or to cause them to return and prompt for more input (2.6).
- scratch** Files whose names begin with a '#' are referred to as scratch files, since they are automatically removed by the system after a couple of days of non-use, or more frequently if disk space becomes tight (1.3).
- script** Sequences of shell commands placed in a file are called shell command scripts. It is often possible to perform simple tasks using these scripts without writing a program in a language such as C, by using the shell to selectively run other programs (3.2, 3.3, 3.10).
- set** The builtin *set* command is used to assign new values to shell variables and to show the values of the current variables. Many shell variables have special meaning to the shell itself. Thus by using the set command the behavior of the shell can be affected (2.1).

- setenv** On version 7 systems variables in the environment 'environ(5)' can be changed by using the *setenv* builtin command (2.6). The *printenv* command can be used to print the value of the variables in the environment.
- shell** A shell is a command language interpreter. It is possible to write and run your own shell, as shells are no different than any other programs as far as the system is concerned. This manual deals with the details of one particular shell, called *csk*.
- shell script** See *script* (3.2, 3.3, 3.10).
- sort** The *sort* program sorts a sequence of lines in ways that can be controlled by argument flags (1.5).
- source** The *source* command causes the shell to read commands from a specified file. It is most useful for reading files such as *.cshrc* after changing them (2.6).
- special character**  
See *metacharacters* and the appendix to this manual.
- standard** We refer often to the *standard input* and *standard output* of commands. See *input* and *output* (1.3, 3.8).
- status** A command normally returns a *status* when it finishes. By convention a *status* of zero indicates that the command succeeded. Commands may return non-zero status to indicate that some abnormal event has occurred. The shell variable *status* is set to the status returned by the last command. It is most useful in shell command scripts (3.5, 3.6).
- substitution** The shell implements a number of *substitutions* where sequences indicated by metacharacters are replaced by other sequences. Notable examples of this are history substitution keyed by the metacharacter '!' and variable substitution indicated by '\$'. We also refer to substitutions as *expansions* (3.4).
- switch** The *switch* command of the shell allows the shell to select one of a number of sequences of commands based on an argument string. It is similar to the *switch* statement in the language C (3.7).
- termination** When a command which is being executed finished we say it undergoes *termination* or *terminates*. Commands normally terminate when they read an end-of-file from their standard input. It is also possible to terminate commands by sending them an *interrupt* or *quit* signal (1.8). The *kill* program terminates specified command whose numbers are given (2.8).
- then** The *then* command is part of the shells 'if-then-else-endif' control construct used in command scripts (3.6).
- time** The *time* command can be used to measure the amount of CPU and real time consumed by a specified command (2.1, 2.6).
- troff** The *troff* program is used to typeset documents. See also *nroff* (3.2).
- tset** The *tset* program is used to set standard erase and kill characters and to tell the system what kind of terminal you are using. It is often invoked in a *.login* file (2.1).



- unalias**      The *unalias* command removes aliases (2.6).
- UNIX**            UNIX is an operating system on which *cs*h runs. UNIX provides facilities which allow *cs*h to invoke other programs such as editors and text formatters which you may wish to use.
- unset**          The *unset* command removes the definitions of shell variables (2.2, 2.6).
- variable expansion**  
See *variables* and *expansion* (2.2, 3.4).
- variables**      Variables in *cs*h hold one or more strings as value. The most common use of variables is in controlling the behavior of the shell. See *path*, *noclobber*, and *ignoreeof* for examples. Variables such as *argv* are also used in writing shell programs (shell command scripts) (2.2).
- verbose**        The *verbose* shell variable can be set to cause commands to be echoed after they are history expanded. This is often useful in debugging shell scripts. The *verbose* variable is set by the shells *-v* command line option (3.10).
- wait**            The builtin command *wait* causes the shell to pause, and not prompt, until all commands run in the background have terminated (2.6).
- while**          The *while* builtin control construct is used in shell command scripts (3.7).
- word**            A sequence of characters which forms an argument to a command is called a *word*. Many characters which are neither letters, digits, '-', '.', or '/' form words all by themselves even if they are not surrounded by blanks. Any sequence of character may be made into a word by surrounding it with "" characters except for the characters "" and '!' which require special treatment (1.1, 1.6). This process of placing special characters in words without their special meaning is called *quoting*.
- working directory**  
At an given time you are in one particular directory, called your working directory. This directories name is printed by the *pwd* command and the files listed by *ls* are the ones in this directory. You can change working directories using *chdir*.
- write**          The *write* command is used to communicate with other users who are logged in to UNIX (2.3).











# UNIX Programming — Second Edition

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

## *ABSTRACT*

This paper is an introduction to programming on the UNIX† system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

November 12, 1978

---

†UNIX is a Trademark of Bell Laboratories.

# UNIX Programming — Second Edition

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

## 2. BASICS

### 2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv) /* echo arguments */
int argc;
char *argv[];
{
 int i;

 for (i = 1; i < argc; i++)
 printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters: each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

### 2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the `<` convention: if `prog` uses `getchar`,



then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using `>`: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main() /* ccstrip: strip non-graphic characters */
{
 int c;
 while ((c = getchar()) != EOF)
 if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
 putchar(c);
 exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (`/usr/include/stdio.h`) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the

program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

### 3. THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

#### 3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in *x.c* and *y.c* and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function *fopen*. *fopen* takes an external name (like *x.c* or *y.c*), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including *stdio.h* is a structure definition called *FILE*. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that *fp* is a pointer to a *FILE*, and *fopen* returns a pointer to a *FILE*. (*FILE* is a type name, like *int*, not a structure tag.

The actual call to *fopen* in a program is

```
fp = fopen(name, mode);
```

The first argument of *fopen* is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("*r*"), write ("*w*"), or append ("*a*").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, *fopen* will return the null pointer value *NULL* (which is defined as zero in *stdio.h*).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which *getc* and *putc* are the simplest. *getc* returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in *c* the next character from the file referred to by *fp*; it returns *EOF* when it reaches end of file. *putc* is the inverse of *getc*:

```
putc(c, fp)
```

puts the character *c* on the file *fp* and returns *c*. *getc* and *putc* return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called *stdin*, *stdout*, and *stderr*. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. *stdin*, *stdout* and *stderr* are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[];
{
 int c, i, inword;
 FILE *fp, *fopen();
 long linect, wordct, charct;
 long tlinect = 0, twordct = 0, tcharct = 0;

 i = 1;
 fp = stdin;
 do {
 if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
 fprintf(stderr, "wc: can't open %s\n", argv[i]);
 continue;
 }
 linect = wordct = charct = inword = 0;
 while ((c = getc(fp)) != EOF) {
 charct++;
 if (c == '\n')
 linect++;
 if (c == ' ' || c == '\t' || c == '\n')
 inword = 0;
 else if (inword == 0) {
 inword = 1;
 wordct++;
 }
 }
 printf("%7ld %7ld %7ld", linect, wordct, charct);
 printf(argc > 1 ? " %s\n" : "\n", argv[i]);
 fclose(fp);
 tlinect += linect;
 twordct += wordct;
 tcharct += charct;
 } while (++i < argc);
 if (argc > 2)
 printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
 exit(0);
}
```

The function *fprintf* is identical to *printf*. save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

### 3.2. Error Handling — `stderr` and `Exit`

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. `wc` writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

### 3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns EOF at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`: at most `size-1` characters are copied; it returns NULL at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` "pushes back" the character `c` onto the input stream `fp`: a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

## 4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

### 4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

#### 4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);
```

```
n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512 /* best size for PDP-11 UNIX */

main() /* copy input to output */
{
 char buf[BUFSIZE];
 int n;

 while ((n = read(0, buf, BUFSIZE)) > 0)
 write(1, buf, n);
 exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```
#define CMASK 0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
 char c;

 return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

*c* must be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```
#define CMASK 0377 /* for making char's > 0 */
#define BUFSIZE 512

getchar() /* buffered version */
{
 static char buf[BUFSIZE];
 static char *bufp = buf;
 static int n = 0;

 if (n == 0) { /* buffer is empty */
 n = read(0, buf, BUFSIZE);
 bufp = buf;
 }
 return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

#### 4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rmode);
```

As with `fopen`, the name argument is a character string corresponding to the external file name. The access mode argument is different, however: `rmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility `cp`, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv) /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
 int f1, f2, n;
 char buf[BUFSIZE];

 if (argc != 3)
 error("Usage: cp from to", NULL);
 if ((f1 = open(argv[1], 0)) == -1)
 error("cp: can't open %s", argv[1]);
 if ((f2 = creat(argv[2], PMODE)) == -1)
 error("cp: can't create %s", argv[2]);

 while ((n = read(f1, buf, BUFSIZE)) > 0)
 if (write(f2, buf, n) != n)
 error("cp: write error", NULL);
 exit(0);
}

error(s1, s2) /* print error message and die */
char *s1, *s2;
{
 printf(s1, s2);
 printf("\n");
 exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

#### 4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
 lseek(fd, pos, 0); /* get to pos */
 return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given `offset` by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

#### 4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of `-1`. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed



because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

## 5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

### 5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
 system("date");
 /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember that `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

### 5.2. Low-Level Process Creation — `exec1` and `execv`

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `exec1`. To print the date as the last action of a running program, use

```
exec1("/bin/date", "date", NULL);
```

The first argument to `exec1` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `exec1` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `exec1` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
exec1("/bin/date", "date", NULL);
exec1("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `exec1` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

### 5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
 execl("/bin/sh", "sh", "-c", cmd, NULL); /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the command and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns `-1`).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
 execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status: it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to

return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `exec1`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

#### 5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned: the actual usage is like this:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
 /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the pipe with a `pipe` system call; it then `forks` to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `exec1`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ? (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int mode;
{
 int p[2];

 if (pipe(p) < 0)
 return(NULL);
 if ((popen_pid = fork()) == 0) {
 close(tst(p[WRITE], p[READ]));
 close(tst(0, 1));
 dup(tst(p[READ], p[WRITE]));
 close(tst(p[READ], p[WRITE]));
 execl("/bin/sh", "sh", "-c", cmd, 0);
 _exit(1); /* disaster has occurred if we get here */
 }
 if (popen_pid == -1)
 return(NULL);
 close(tst(p[READ], p[WRITE]));
 return(tst(p[WRITE], p[READ]));
}
```

The sequence of `closes` in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated: performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd) /* close pipe fd */
int fd;
{
 register r, (*hstat)(), (*istat)(), (*qstat)();
 int status;
 extern int popen_pid;

 close(fd);
 istat = signal(SIGINT, SIG_IGN);
 qstat = signal(SIGQUIT, SIG_IGN);
 hstat = signal(SIGHUP, SIG_IGN);
 while ((r = wait(&status)) != popen_pid && r != -1);
 if (r == -1)
 status = -1;
 signal(SIGINT, istat);
 signal(SIGQUIT, qstat);
 signal(SIGHUP, hstat);
 return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

## 6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address of either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to

allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
 int onintr();

 if (signal(SIGINT, SIG_IGN) != SIG_IGN)
 signal(SIGINT, onintr);

 /* Process ... */

 exit(0);
}

onintr()
{
 unlink(tempfile);
 exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like `interrupt` are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
 int (*istat)(), onintr();

 istat = signal(SIGINT, SIG_IGN); /* save original status */
 setjmp(sjbuf); /* save current stack position */
 if (istat != SIG_IGN)
 signal(SIGINT, onintr);

 /* main processing loop */
}
```

```
onintr()
{
 printf("\nInterrupt\n");
 longjmp(sjbuf); /* return to saved state */
}
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object: it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary: most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
 if (intflag)
 /* EOF caused by interrupt */
 else
 /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like `!` in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
 execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s) /* run command string s */
char *s;
{
 int status, pid, w;
 register int (*istat)(), (*qstat)();

 if ((pid = fork()) == 0) {
 execl("/bin/sh", "sh", "-c", s, 0);
 _exit(127);
 }
 istat = signal(SIGINT, SIG_IGN);
 qstat = signal(SIGQUIT, SIG_IGN);
 while ((w = wait(&status)) != pid && w != -1)
 ;
 if (w == -1)
 status = -1;
 signal(SIGINT, istat);
 signal(SIGQUIT, qstat);
 return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

## References

- [1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978.
- [2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.
- [3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.



## Appendix — The Standard I/O Library

*D. M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

### 1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore `_` to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file  
`stdout` The name of the standard output file  
`stderr` The name of the standard error file  
`EOF` is actually `-1`, and is the value returned by the read routines on end-of-file or error.  
`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error  
`FILE` expands to `struct _iobuf` and is a useful shorthand when declaring pointers to streams.  
`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.  
`getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `fileno`  
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

### 2. Calls

```
FILE *fopen(filename, type) char *filename, *type;
opens the file and, if needed, allocates a buffer for it. filename is a character string specifying the name. type is a character string (not a single character). It may be "r", "w", or "a" to indicate intent to read, write, or append. The value returned is a file pointer. If it is NULL the attempt to open failed.
FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;
```

The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

`int getc(ioptr) FILE *ioptr;`

returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer `EOF` is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

`int fgetc(ioptr) FILE *ioptr;`

acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`

`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but `EOF` is returned on error.

`fputc(c, ioptr) FILE *ioptr;`

acts like `putc` but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`

The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`

Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`

terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(ioptr) FILE *ioptr;`

returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`

returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`

is identical to `getc(stdin)`.

`putchar(c);`

is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`

reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or `NULL` if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`

writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`

The argument character *c* is pushed back on the input stream named by *ioptr*. Only one character may be pushed back.

```
printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...) char *s, *format;
```

`printf` writes on the standard output. `fprintf` writes on the named output stream. `sprintf` puts characters in the character array (string) named by *s*. The specifications are as described in section `printf(3)` of the *UNIX Programmer's Manual*.

```
scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;
```

`scanf` reads from the standard input. `fscanf` reads from the named input stream. `sscanf` reads from the character string supplied as *s*. `scanf` reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string *format*, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

`scanf` returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

```
fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
reads nitems of data beginning at ptr from file ioptr. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the fopen call.
```

```
fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
Like fread, but in the other direction.
```

```
rewind(ioptr) FILE *ioptr;
rewinds the stream named by ioptr. It is not very useful except on input, since a rewound output file is still open only for output.
```

```
system(string) char *string;
The string is executed by the shell as if typed at the terminal.
```

```
getw(ioptr) FILE *ioptr;
returns the next word from the input stream named by ioptr. EOF is returned on end-of-file or error, but since this a perfectly good integer feof and ferror should be used. A "word" is 16 bits on the PDP-11.
```

```
putw(w, ioptr) FILE *ioptr;
writes the integer w on the named output stream.
```

```
setbuf(ioptr, buf) FILE *ioptr; char *buf;
setbuf may be used after a stream has been opened but before I/O has started. If buf is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:
```

```
char buf[BUFSIZ];
```

```
fileno(ioptr) FILE *ioptr;
returns the integer file descriptor associated with the file.
```

```
fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
The location of the next byte in the stream named by ioptr is adjusted. offset is a long integer. If ptrname is 0, the offset is measured from the beginning of the file; if ptrname is 1, the offset is measured from the current read or write pointer; if ptrname is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When
```

this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the `pathname` must be 0).

`long ftell(FILE *ioptr);`

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`

allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`

allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`cfree(ptr) char *ptr;`

Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`isctrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

This document describes a product released by Bell Laboratories as part of Version 7 of UNIX. It has not yet been reviewed at Berkeley for possible errors or incompatibility with our UNIX system. If you find errors or omissions in this document, you can help us by sending your comments to Publications Editor, Computing Services, Evans Hall, University of California, Berkeley, 94720. 7/80

## UNIX Programming — Second Edition

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

### ABSTRACT

This paper is an introduction to programming on the UNIX<sup>†</sup> system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

November 12, 1978

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.



# UNIX Programming — Second Edition

*Brian W. Kernighan*

*Dennis M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

## 2. BASICS

### 2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv) /* echo arguments */
int argc;
char *argv[];
{
 int i;

 for (i = 1; i < argc; i++)
 printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

### 2.2. The "Standard Input" and "Standard Output"

The simplest input mechanism is to read the "standard input," which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the `<` convention: if `prog` uses `getchar`,

then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog`.

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the "standard output," which is also by default the terminal. The output can be captured on a file by using `>`: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog`.

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main() /* cstrip: strip non-graphic characters */
{
 int c;
 while ((c = getchar()) != EOF)
 if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
 putchar(c);
 exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (`/usr/include/stdio.h`) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | cstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the



program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

### 3. THE STANDARD I/O LIBRARY

The "Standard I/O Library" is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

#### 3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is `wc`, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag.)

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read ("`r`"), write ("`w`"), or append ("`a`").

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns EOF when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character *c* on the file *fp* and returns *c*. *getc* and *putc* return EOF on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called *stdin*, *stdout*, and *stderr*. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. *stdin*, *stdout* and *stderr* are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type FILE \* can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[];
{
 int c, i, inword;
 FILE *fp, *fopen();
 long linect, wordct, charct;
 long tlinect = 0, twordct = 0, tcharct = 0;

 i = 1;
 fp = stdin;
 do {
 if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
 fprintf(stderr, "wc: can't open %s\n", argv[i]);
 continue;
 }
 linect = wordct = charct = inword = 0;
 while ((c = getc(fp)) != EOF) {
 charct++;
 if (c == '\n')
 linect++;
 if (c == ' ' || c == '\t' || c == '\n')
 inword = 0;
 else if (inword == 0) {
 inword = 1;
 wordct++;
 }
 }
 printf("%7ld %7ld %7ld", linect, wordct, charct);
 printf(argc > 1 ? " %s\n" : "\n", argv[i]);
 fclose(fp);
 tlinect += linect;
 twordct += wordct;
 tcharct += charct;
 } while (++i < argc);
 if (argc > 2)
 printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
 exit(0);
}
```

The function *fprintf* is identical to *printf*, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

### 3.2. Error Handling — `stderr` and `Exit`

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. `wc` writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

### 3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns EOF at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns NULL at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` "pushes back" the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

## 4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

### 4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called "opening" the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the "shell") runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

#### 4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ("unbuffered"), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 512 /* best size for PDP-11 UNIX */

main() /* copy input to output */
{
 char buf[BUFSIZE];
 int n;

 while ((n = read(0, buf, BUFSIZE)) > 0)
 write(1, buf, n);
 exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some read will return a smaller number of bytes to be written by write; the next call to read after that will return zero.

It is instructive to see how read and write can be used to construct higher level routines like getchar, putchar, etc. For example, here is a version of getchar which does unbuffered input.

```
#define CMASK 0377 /* for making char's > 0 */

getchar() /* unbuffered single character input */
{
 char c;

 return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

*c* must be declared char, because read accepts a character pointer. The character being returned must be masked with 0377 to ensure that it is positive; otherwise sign extension may make it negative. (The constant 0377 is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of getchar does input in big chunks, and hands out the characters one at a time.

```
#define CMASK 0377 /* for making char's > 0 */
#define BUFSIZE 512

getchar() /* buffered version */
{
 static char buf[BUFSIZE];
 static char *bufp = buf;
 static int n = 0;

 if (n == 0) { /* buffer is empty */
 n = read(0, buf, BUFSIZE);
 bufp = buf;
 }
 return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

#### 4.3. Open, Creat, Close, Unlink :

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open and creat [sic].

open is rather like the fopen discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to open a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to create a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility `cp`, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv) /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
 int f1, f2, n;
 char buf[BUFSIZE];

 if (argc != 3)
 error("Usage: cp from to", NULL);
 if ((f1 = open(argv[1], 0)) == -1)
 error("cp: can't open %s", argv[1]);
 if ((f2 = creat(argv[2], PMODE)) == -1)
 error("cp: can't create %s", argv[2]);

 while ((n = read(f1, buf, BUFSIZE)) > 0)
 if (write(f2, buf, n) != n)
 error("cp: write error", NULL);
 exit(0);
}

error(s1, s2) /* print error message and die */
char *s1, *s2;
{
 printf(s1, s2);
 printf("\n");
 exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

#### 4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ("rewind"),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
 lseek(fd, pos, 0); /* get to pos */
 return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given `offset` by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

#### 4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of `-1`. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual*, so your program can, for example, determine if an attempt to open a file failed

because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and printed by your program.

## 5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

### 5.1. The "System" Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
 system("date");
 /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember that `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

### 5.2. Low-Level Process Creation — `execl` and `execv`

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can't be found or is not executable. If you don't know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don't know in advance how many arguments there are going to be. The call is



```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

### 5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl` or `execv`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork`:

```
proc_id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the "process id." In one of these processes (the "child"), `proc_id` is zero. In the other (the "parent"), `proc_id` is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
 execl("/bin/sh", "sh", "-c", cmd, NULL); /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork` makes two copies of the program. In the child, the value returned by `fork` is zero, so it calls `execl` which does the command and then dies. In the parent, `fork` returns non-zero so it skips the `execl`. (If there is any error, `fork` returns `-1`).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait`:

```
int status;

if (fork() == 0)
 execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl` or `fork`, or the possibility that there might be more than one child running simultaneously. (The `wait` returns the process id of the terminated child, if you want to check it against the value returned by `fork`.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in `status`). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The `status` returned by `wait` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to

return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork` nor the `exec` calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `exec1`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

#### 5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int fd[2];

stat = pipe(fd);
if (stat == -1)
 /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `forks` to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `exec1`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ 0
#define WRITE 1
#define tst(a, b) (mode == READ ? (b) : (a))
static int popen_pid;

popen(cmd, mode)
char *cmd;
int mode;
{
 int p[2];

 if (pipe(p) < 0)
 return(NULL);
 if ((popen_pid = fork()) == 0) {
 close(tst(p[WRITE], p[READ]));
 close(tst(0, 1));
 dup(tst(p[READ], p[WRITE]));
 close(tst(p[READ], p[WRITE]));
 execl("/bin/sh", "sh", "-c", cmd, 0);
 _exit(1); /* disaster has occurred if we get here */
 }
 if (popen_pid == -1)
 return(NULL);
 close(tst(p[READ], p[WRITE]));
 return(tst(p[WRITE], p[READ]));
}
```

The sequence of `closes` in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd) /* close pipe fd */
int fd;
{
 register r, (*hstat)(), (*istat)(), (*qstat)();
 int status;
 extern int popen_pid;

 close(fd);
 istat = signal(SIGINT, SIG_IGN);
 qstat = signal(SIGQUIT, SIG_IGN);
 hstat = signal(SIGHUP, SIG_IGN);
 while ((r = wait(&status)) != popen_pid && r != -1);
 if (r == -1)
 status = -1;
 signal(SIGINT, istat);
 signal(SIGQUIT, qstat);
 signal(SIGHUP, hstat);
 return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen_pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

## 6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
...
signal(SIGINT, SIG_IGN);
```

causes interrupts to be ignored. while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to

allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
 int onintr();

 if (signal(SIGINT, SIG_IGN) != SIG_IGN)
 signal(SIGINT, onintr);

 /* Process ... */

 exit(0);
}

onintr()
{
 unlink(tempfile);
 exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like `interrupt` are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

main()
{
 int (*istat)(), onintr();

 istat = signal(SIGINT, SIG_IGN); /* save original status */
 setjmp(sjbuf); /* save current stack position */
 if (istat != SIG_IGN)
 signal(SIGINT, onintr);

 /* main processing loop */
}
```

```
onintr()
(
 printf("\nInterrupt\n");
 longjmp(sjbuf); /* return to saved state */
)
```

The include file `setjmp.h` declares the type `jmp_buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that "execution resumes at the exact point it was interrupted," the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for "errors" which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
 if (intflag)
 /* EOF caused by interrupt */
 else
 /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like `!` in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
 execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s) /* run command string s */
char *s;
{
 int status, pid, w;
 register int (*istat)(), (*qstat)();

 if ((pid = fork()) == 0) {
 execl("/bin/sh", "sh", "-c", s, 0);
 _exit(127);
 }
 istat = signal(SIGINT, SIG_IGN);
 qstat = signal(SIGQUIT, SIG_IGN);
 while ((w = wait(&status)) != pid && w != -1)
 ;
 if (w == -1)
 status = -1;
 signal(SIGINT, istat);
 signal(SIGQUIT, qstat);
 return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define SIG_DFL (int (*)())0
#define SIG_IGN (int (*)())1
```

## References

- [1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978.
- [2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.
- [3] B. W. Kernighan, "UNIX for Beginners — Second Edition." Bell Laboratories, 1978.

## Appendix — The Standard I/O Library

*D. M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

### 1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore `_` to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file  
`stdout` The name of the standard output file  
`stderr` The name of the standard error file  
`EOF` is actually `-1`, and is the value returned by the read routines on end-of-file or error.  
`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error  
`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.  
`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.  
`getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `fileno`  
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

### 2. Calls

`FILE *fopen(filename, type) char *filename, *type;`  
opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

`FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;`



The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

`int getc(ioptr) FILE *ioptr;`

returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer `EOF` is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

`int fgetc(ioptr) FILE *ioptr;`

acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`

`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but `EOF` is returned on error.

`fputc(c, ioptr) FILE *ioptr;`

acts like `putc` but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`

The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`

Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`

terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(ioptr) FILE *ioptr;`

returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`

returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`

is identical to `getc(stdin)`.

`putchar(c);`

is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`

reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or `NULL` if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`

writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`

The argument character *c* is pushed back on the input stream named by *ioptr*. Only one character may be pushed back.

```
printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...) char *s, *format;
```

*printf* writes on the standard output. *fprintf* writes on the named output stream. *sprintf* puts characters in the character array (string) named by *s*. The specifications are as described in section *printf(3)* of the *UNIX Programmer's Manual*.

```
scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;
```

*scanf* reads from the standard input. *fscanf* reads from the named input stream. *sscanf* reads from the character string supplied as *s*. *scanf* reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string *format*, and a set of arguments, *each of which must be a pointer*, indicating where the converted input should be stored.

*scanf* returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, EOF is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

```
fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
reads nitems of data beginning at ptr from file ioptr. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the fopen call.
```

```
fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;
Like fread, but in the other direction.
```

```
rewind(ioptr) FILE *ioptr;
rewinds the stream named by ioptr. It is not very useful except on input, since a rewound output file is still open only for output.
```

```
system(string) char *string;
The string is executed by the shell as if typed at the terminal.
```

```
getw(ioptr) FILE *ioptr;
returns the next word from the input stream named by ioptr. EOF is returned on end-of-file or error, but since this a perfectly good integer feof and ferror should be used. A "word" is 16 bits on the PDP-11.
```

```
putw(w, ioptr) FILE *ioptr;
writes the integer w on the named output stream.
```

```
setbuf(ioptr, buf) FILE *ioptr; char *buf;
setbuf may be used after a stream has been opened but before I/O has started. If buf is NULL, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:
```

```
char buf[BUFSIZ];
```

```
fileno(ioptr) FILE *ioptr;
returns the integer file descriptor associated with the file.
```

```
fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
The location of the next byte in the stream named by ioptr is adjusted. offset is a long integer. If ptrname is 0, the offset is measured from the beginning of the file; if ptrname is 1, the offset is measured from the current read or write pointer; if ptrname is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When
```

this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the `ptrname` must be 0).

`long ftell(ioptr) FILE *ioptr;`

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`

The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`

allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`

allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`cfree(ptr) char *ptr;`

Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.







# **Yacc: Yet Another Compiler-Compiler**

*Stephen C. Johnson*

Bell Laboratories  
Murray Hill, New Jersey 07974

## *ABSTRACT*

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

July 31, 1978

# Yacc: Yet Another Compiler-Compiler

*Stephen C. Johnson*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C<sup>1</sup> and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month\_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month\_name*, *day*, and *year* are defined elsewhere. The comma “,” is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
```

...

```
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month\_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a



*month\_name* was seen; in this case, *month\_name* would be a token.

Literal characters such as “,” must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realivly easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be “slipped in” to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.<sup>2,3,4</sup> Yacc has been extensively used in numerous practical applications, including *lint*,<sup>5</sup> the Portable C Compiler,<sup>6</sup> and a system for typesetting mathematics.<sup>7</sup>

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

## 1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, (*grammar*)

rules, and programs. The sections are separated by double percent “%%” marks. (The percent “%” is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /\* . . . \*/ , as in C and PL/L.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot “.”, underscore “\_”, and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes “’” . As in C, the backslash “\” is an escape character within literals, and all the C escapes are recognized. Thus

```
\n' newline
\r' return
\' single quote “'”
\\ backslash “\”
\t' tab
\b' backspace
\f' form feed
\xxx' “xxx” in octal
```

For a number of technical reasons, the NUL character (‘\0’ or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar “|” can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to Yacc as

```
A : B C D
 | E F
 | G
 ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token name1 name2 ...
```

in the declarations section. (See Sections 3, 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the `%start` keyword:

```
%start symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker*. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

## 2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A : '(B)'
 { hello(1, "abc"); }
```

and

```
XXX : YYY ZZZ
 { printf("a message\n");
 flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "\$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "\$\$" to some value. For example, an action that does nothing but return the value 1 is

```
{ SS = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables \$1, \$2, . . . , which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A : B C D ;
```

for example, then \$2 has the value returned by C, and \$3 the value returned by D.

As a more concrete example, consider the rule

```
expr : '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr : '(' expr ')' { SS = $2 ; }
```

By default, the value of a rule is the value of the first element in it (\$1). Thus, grammar rules of the form

```
A : B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A : B
 C { SS = 1; }
 { x = $2; y = $3; }
 ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
SACT : /* empty */
 { SS = 1; }
 ;
A : B SACT C
 { x = $2; y = $3; }
 ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node(L, n1, n2)
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr : expr '+' expr
 { $$ = node('+', $1, $3); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks “%{” and “%}”. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in “yy”; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

### 3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yyval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the “# define” mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
 extern int yyval;
 int c;
 ...
 c = getchar();
 ...
 switch(c) {
 ...
 case '0':
 case '1':
 ...
 case '9':
 yyval = c-'0';
 return(DIGIT);
 ...
 }
 ...
}
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error

handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.<sup>8</sup> These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

#### 4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yyllex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

.           reduce 18

refers to *grammar rule* 18, while the action

IF        shift 34

refers to *state* 34.

Suppose the rule being reduced is

A        :       x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

A        goto 20

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yylval* is copied onto the value stack. The pseudo-variables \$1, \$2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme : sound place
 ;
sound : DING DONG
 ;
place : DELL
 ;
```

When Yacc is invoked with the `-v` option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:



```
state 0
 Saccept : _rhyme Send

 DING shift 3
 . error

 rhyme goto 1
 sound goto 2

state 1
 Saccept : rhyme_Send

 Send accept
 . error

state 2
 rhyme : sound_place

 DELL shift 5
 . error

 place goto 4

state 3
 sound : DING_DONG

 DONG shift 6
 . error

state 4
 rhyme : sound place_ (1)

 . reduce 1

state 5
 place : DELL_ (3)

 . reduce 3

state 6
 sound : DING DONG_ (2)

 . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The `_` character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

**DING DONG DELL**

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is

“shift 6”, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

sound : DING DONG

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

sound goto 2

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is “shift 5”, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by “Send” in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

### 5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr ‘-’ expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr - expr - expr

the rule allows this input to be structured as either

( expr - expr ) - expr

or as

expr - ( expr - expr )

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule: after applying the rule: the input is reduced to *expr*(the left side of the rule). The parser would then read the final part of the input:

— expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr — expr

it could defer the immediate application of the rule, and continue reading the input until it had seen

expr — expr — expr

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

expr — expr

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

expr — expr

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict*. It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict*. Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule*.

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat : IF '(' cond ')' stat
 | IF '(' cond ')' stat ELSE stat
 ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF (C1) IF (C2) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF (C1) {
 IF (C2) S1
}
ELSE S2
```

or

```
IF (C1) {
 IF (C2) S1
 ELSE S2
}
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF (C1) IF (C2) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF (C1) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF (C1) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF (C1) IF (C2) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF (C1) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things — there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF (C1) IF (C2) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (*-v*) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

```
stat : IF (cond) stat_ (18)
stat : IF (cond) stat_ELSE stat
```

```
ELSE shift 45
 reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF (cond) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF (cond) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by “.”, is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following “shift” commands refer to other states, while the numbers following “reduce” commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references<sup>2,3,4</sup> might be consulted; the services of a local guru might also be appropriate.

## 6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and

construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
 | expr '+' expr
 | expr '-' expr
 | expr '*' expr
 | expr '/' expr
 | NAME
 ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = (b = ((c*d) - e) - (f*g)))
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
 | expr '-' expr
 | expr '*' expr
 | expr '/' expr
 | '-' expr %prec '*'
 | NAME
 ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

## 7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is

legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat : error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat : error ';' ;
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input : error ^n' { printf("Reenter last line: "); } input
 { SS = S4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message: this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input : error ^n'
 { yyerrok;
 printf("Reenter last line: "); }
input : input
 { SS = S4; }
;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearn ;
```

in an action will have this effect. For example, suppose the action after error were to call some



sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by *yylex* would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat : error
 { resynch();
 yyerrok ;
 yyclearin ; }
 ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

### 8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a *-ly* argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
 return(yyparse());
}

and

include <stdio.h>

yyerror(s) char *s; {
 fprintf(stderr, "%s\n", s);
}
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

## 9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

### Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

- a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."
- b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
- c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.
- d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.
- e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

### Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list : item
 | list ',' item
 ;
```

and

```
seq : item
 | seq item
 ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq : item
 | item seq
 ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */
 | seq item
 ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

### Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
 int dflag;
}%
... other declarations ...

%%

prog : decls stats
 ;

decls : /* empty */
 { dflag = 1; }
 | decls declaration
 ;

stats : /* empty */
 { dflag = 0; }
 | stats statement
 ;

... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement*. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

### Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are

powerful stylistic reasons for preferring this, anyway.

### 10: Advanced Topics

This section discusses a number of advanced features of Yacc.

#### Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyperror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

#### Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```

sent : adj noun verb adj noun
 { look at the sentence . . . }
 ;

adj : THE { SS = THE; }
 | YOUNG { SS = YOUNG; }
 ...
 ;

noun : DOG { SS = DOG; }
 | CRONE { if(S0 == YOUNG){
 printf("what?\n");
 }
 SS = CRONE;
 }
 ;
 ...

```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

#### Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a *SS* or *Sn* construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*<sup>3</sup> will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
 body of union ...
}
```

This declares the Yacc value stack, and the external variables *yyval* and *yyval*, to have type equal to this union. If Yacc was invoked with the `-d` option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable `YYSTYPE` to represent this union. Thus, the header file might also have said:

```
typedef union {
 body of union ...
} YYSTYPE;
```

The header file must be included in the declarations section, by use of `%{` and `%}`.

Once `YYSTYPE` is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords `%token`, `%left`, `%right`, and `%nonassoc`, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, `%type`, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as `$0` — see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between `<` and `>`, immediately after the first `$`. An example of this usage is

```
rule : aaa { $<intval>$ = 3; } bbb
 { fun($<intval>2, $<other>0); }
 ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of `%type` will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of `$n` or `$$` to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

## **11: Acknowledgements**

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

### References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* 6(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* 18(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," *Comp. Sci. Tech. Rep. No. 65* (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* 18 pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," *Comp. Sci. Tech. Rep. No. 39*, Bell Laboratories, Murray Hill, New Jersey (October 1975).

### Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, -, \*, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
include <stdio.h>
include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* supplies precedence for unary minus */

%% /* beginning of rules section */

list : /* empty */
 | list stat '\n'
 | list error '\n'
 { yyerrok; }
 ;

stat : expr
 { printf("%d\n", S1); }
 | LETTER '=' expr
 { regs[S1] = S3; }
 ;

expr : '(' expr ')'
 { SS = S2; }
 | expr '+' expr
 { SS = S1 + S3; }
 | expr '-' expr
 { SS = S1 - S3; }
```



```
| expr '.' expr
| { SS = S1 * S3; }
| expr '/' expr
| { SS = S1 / S3; }
| expr '%' expr
| { SS = S1 % S3; }
| expr '&' expr
| { SS = S1 & S3; }
| expr '|' expr
| { SS = S1 | S3; }
| '-' expr %prec UMINUS
| { SS = - S2; }
| LETTER
| { SS = regs[S1]; }
| number
;

number: DIGIT
| { SS = S1; base = (S1==0) ? 8 : 10; }
| number DIGIT
| { SS = base * S1 + S2; }
;

%% /* start of programs */

yylex() { /* lexical analysis routine */
/* returns LETTER for a lower case letter, yylval = 0 through 25 */
/* return DIGIT for a digit, yylval = 0 through 9 */
/* all other characters are returned immediately */

int c;

while((c=getchar()) == ' ') /* skip blanks */

/* c is now nonblank */

if(islower(c)) {
 yyval = c - 'a';
 return (LETTER);
}
if(isdigit(c)) {
 yyval = c - '0';
 return(DIGIT);
}
return(c);
}
```

## Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C\_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C\_IDENTIFIER.

```
/* grammar for the input to Yacc */

/* basic entities */
%token IDENTIFIER /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier (but not literal) followed by colon */
%token NUMBER /* [0-9]+ */

/* reserved words: %type => TYPE, %left => LEFT, etc. */
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK /* the %% mark */
%token LCURL /* the %{ mark */
%token RCURL /* the %} mark */

/* ascii character literals stand for themselves */

%start spec

%%

spec : defs MARK rules tail
 ;

tail : MARK { In this action, eat up the rest of the file }
 | /* empty: the second MARK is optional */
 ;

defs : /* empty */
 | defs def
 ;

def : START IDENTIFIER
 | UNION { Copy union definition to output }
 | LCURL { Copy C code to output file } RCURL
 | ndefs rword tag nlist
 ;

rword : TOKEN
 | LEFT
 | RIGHT
```

```
NONASSOC
TYPE
;

tag : /* empty: union tag is optional */
 '<' IDENTIFIER '>'
;

nlist : nmno
 nlist nmno
 nlist ';' nmno
;

nmno : IDENTIFIER /* NOTE: literal illegal with %type */
 IDENTIFIER NUMBER /* NOTE: illegal with %type */
;

/* rules section */

rules : C IDENTIFIER rbody prec
 rules rule
;

rule : C IDENTIFIER rbody prec
 '|' rbody prec
;

rbody : /* empty */
 rbody IDENTIFIER
 rbody act
;

act : '{ { Copy action, translate $$, etc. } }'
;

prec : /* empty */
 PREC IDENTIFIER
 PREC IDENTIFIER act
 prec ';'
;
```

### Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations  $+$ ,  $-$ ,  $*$ ,  $/$ , unary  $-$ , and  $=$  (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

$$(x, y)$$

where  $x$  is less than or equal to  $y$ . There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double's*. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

$$2.5 + (3.5 - 4.)$$

and

$$2.5 + (3.5, 4.)$$

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

include <stdio.h>
include <ctype.h>

typedef struct interval {
 double lo, hi;
} INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start lines

%union {
 int ival;
 double dval;
 INTERVAL vval;
}

%token <ival> DREG VREG /* indices into dreg, vreg arrays */

%token <dval> CONST /* floating point constant */

%type <dval> dexp /* expression */

%type <vval> vexp /* interval expression */

 /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS /* precedence for unary minus */

%%

lines : /* empty */
 | lines line
 ;

line : dexp '\n'
 { printf("%15.8f\n", $1); }
 | vexp '\n'
 { printf("(%15.8f , %15.8f)\n", $1.lo, $1.hi); }
 | DREG '=' dexp '\n'
 { dreg[$1] = $3; }
 | VREG '=' vexp '\n'
```

```

error { vreg[S1] = S3; }
 { yyerrok; }
;

dexp : CONST
 DREG
 { SS = dreg[S1]; }
dexp '+' dexp
 { SS = S1 + S3; }
dexp '-' dexp
 { SS = S1 - S3; }
dexp '*' dexp
 { SS = S1 * S3; }
dexp '/' dexp
 { SS = S1 / S3; }
'-' dexp
 { %prec UMINUS
 SS = - S2; }
'(' dexp ')'
 { SS = S2; }
;

vexp : dexp
 { SS.hi = SS.lo == S1; }
 '(' dexp ',' dexp ')'
 {
 SS.lo = S2;
 SS.hi = S4;
 if(SS.lo > SS.hi){
 printf("interval out of order\n");
 YYERROR;
 }
 }
 VREG
 { SS = vreg[S1]; }
vexp '+' vexp
 { SS.hi = S1.hi + S3.hi;
 SS.lo = S1.lo + S3.lo; }
dexp '+' vexp
 { SS.hi = S1 + S3.hi;
 SS.lo = S1 + S3.lo; }
vexp '-' vexp
 { SS.hi = S1.hi - S3.lo;
 SS.lo = S1.lo - S3.hi; }
dexp '-' vexp
 { SS.hi = S1 - S3.lo;
 SS.lo = S1 - S3.hi; }
vexp '*' vexp
 { SS = vmul(S1.lo, S1.hi, S3); }
dexp '*' vexp
 { SS = vmul(S1, S1, S3); }
vexp '/' vexp
 { if(dcheck(S3)) YYERROR;
 SS = vdiv(S1.lo, S1.hi, S3); }

```

```
| dexp '/' vexp
| { if(dcheck($3)) YYERROR;
| $$ = vdiv($1, $1, $3); }
| '-' vexp %prec UMINUS
| { $$hi = -$2.lo; $$lo = -$2.hi; }
| '(' vexp ')'
| { $$ = $2; }
|
;

%%

define BSZ 50 /* buffer size for floating point numbers */

/* lexical analysis */

yylex(){
 register c;

 while((c=getchar()) == ' '){ /* skip over blanks */ }

 if(isupper(c)){
 yyval.ival = c - 'A';
 return(VREG);
 }

 if(islower(c)){
 yyval.ival = c - 'a';
 return(DREG);
 }

 if(isdigit(c) || c == '.'){
 /* gobble up digits, points, exponents */

 char buf[BSZ+1], *cp = buf;
 int dot = 0, exp = 0;

 for(; (cp-buf) < BSZ ; ++cp, c=getchar()){

 *cp = c;
 if(isdigit(c)) continue;
 if(c == '.'){
 if(dot++ || exp) return('.'); /* will cause syntax error */
 continue;
 }

 if(c == 'e'){
 if(exp++) return('e'); /* will cause syntax error */
 continue;
 }

 /* end of number */
 break;
 }

 *cp = '\0';
 if((cp-buf) >= BSZ) printf("constant too long: truncated\n");
 }
}
```

```
 else ungetc(c, stdin); /* push back last char read */
 yyival.dval = atof(buf);
 return(CONST);
 }
 return(c);
}

INTERVAL hilo(a, b, c, d) double a, b, c, d; {
 /* returns the smallest interval containing a, b, c, and d */
 /* used by *, / routines */
 INTERVAL v;

 if(a>b) { v.hi = a; v.lo = b; }
 else { v.hi = b; v.lo = a; }

 if(c>d) {
 if(c>v.hi) v.hi = c;
 if(d<v.lo) v.lo = d;
 }
 else {
 if(d>v.hi) v.hi = d;
 if(c<v.lo) v.lo = c;
 }
 return(v);
}

INTERVAL vmul(a, b, v) double a, b; INTERVAL v; {
 return(hilo(a*v.hi, a*v.lo, b*v.hi, b*v.lo));
}

dcheck(v) INTERVAL v; {
 if(v.hi >= 0. && v.lo <= 0.){
 printf("divisor interval contains 0.\n");
 return(1);
 }
 return(0);
}

INTERVAL vdiv(a, b, v) double a, b; INTERVAL v; {
 return(hilo(a/v.hi, a/v.lo, b/v.hi, b/v.lo));
}
```



#### Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes `""`.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or `_`, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.

3. Most places where `%` is legal, backslash `"\"` may be used. In particular, `\\` is the same as `%%`, `\left` the same as `%left`, etc.
4. There are a number of other synonyms:
  - `%<` is the same as `%left`
  - `%>` is the same as `%right`
  - `%binary` and `%2` are the same as `%nonassoc`
  - `%0` and `%term` are the same as `%token`
  - `%=` is the same as `%prec`

5. Actions may also have the form

```
={ ... }
```

and the curly braces can be dropped if the action is a single C statement.

6. C code between `%{` and `%}` used to be permitted at the head of the rules section, as well as in the declaration section.







## **SED — A Non-interactive Text Editor**

*Lee E. McMahon*

Bell Laboratories  
Murray Hill, New Jersey 07974

### **ABSTRACT**

*Sed* is a non-interactive context editor that runs on the UNIX<sup>†</sup> operating system. *Sed* is designed to be especially useful in three cases:

- 1) To edit files too large for comfortable interactive editing;
- 2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode.
- 3) To perform multiple 'global' editing functions efficiently in one pass through the input.

This memorandum constitutes a manual for users of *sed*.

August 15, 1978

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

# SED — A Non-interactive Text Editor

*Lee E. McMahon*

Bell Laboratories  
Murray Hill, New Jersey 07974

## Introduction

*Sed* is a non-interactive context editor designed to be especially useful in three cases:

- 1) To edit files too large for comfortable interactive editing;
- 2) To edit any size file when the sequence of editing commands is too complicated to be comfortably typed in interactive mode;
- 3) To perform multiple 'global' editing functions efficiently in one pass through the input.

Since only a few lines of the input reside in core at one time, and no temporary files are used, the effective size of file that can be edited is limited only by the requirement that the input and output fit simultaneously into available secondary storage.

Complicated editing scripts can be created separately and given to *sed* as a command file. For complex edits, this saves considerable typing, and its attendant errors. *Sed* running from a command file is much more efficient than any interactive editor known to the author, even if that editor can be driven by a pre-written script.

The principal loss of functions compared to an interactive editor are lack of relative addressing (because of the line-at-a-time operation), and lack of immediate verification that a command has done what was intended.

*Sed* is a lineal descendant of the UNIX editor, *ed*. Because of the differences between interactive and non-interactive operation, considerable changes have been made between *ed* and *sed*; even confirmed users of *ed* will frequently be surprised (and probably chagrined), if they rashly use *sed* without reading Sections 2 and 3 of this document. The most striking family resemblance between the two editors is in the class of patterns ('regular expressions') they recognize; the code for matching patterns is copied almost verbatim from the code for *ed*, and the description of regular expressions in Section 2 is copied almost verbatim from the UNIX Programmer's Manual[1]. (Both code and description were written by Dennis M. Ritchie.)

## 1. Overall Operation

*Sed* by default copies the standard input to the standard output, perhaps performing one or more editing commands on each line before writing it to the output. This behavior may be modified by flags on the command line; see Section 1.1 below.

The general format of an editing command is:

```
[address1.address2][function][arguments]
```

One or both addresses may be omitted; the format of addresses is given in Section 2. Any number of blanks or tabs may separate the addresses from the function. The function must be present; the available commands are discussed in Section 3. The arguments may be required or optional, according to which function is given; again, they are discussed in Section 3 under each individual function.

Tab characters and spaces at the beginning of lines are ignored.

### 1.1. Command-line Flags

Three flags are recognized on the command line:

- n: tells *sed* not to copy all lines, but only those specified by *p* functions or *p* flags after *s* functions (see Section 3.3);
- e: tells *sed* to take the next argument as an editing command;
- f: tells *sed* to take the next argument as a file name; the file should contain editing commands, one to a line.

### 1.2. Order of Application of Editing Commands

Before any editing is done (in fact, before any input file is even opened), all the editing commands are compiled into a form which will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file). The commands are compiled in the order in which they are encountered; this is generally the order in which they will be attempted at execution time. The commands are applied one at a time; the input to each command is the output of all preceding commands.

The default linear order of application of editing commands can be changed by the flow-of-control commands, *r* and *b* (see Section 3). Even when the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied command.

### 1.3. Pattern-space

The range of pattern matches is called the pattern space. Ordinarily, the pattern space is one line of the input text, but more than one line can be read into the pattern space by using the *N* command (Section 3.6.).

### 1.4. Examples

Examples are scattered throughout the text. Except where otherwise noted, the examples all assume the following input text:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

(In no case is the output of the *sed* commands to be considered an improvement on Coleridge.)

#### Example:

The command

```
2q
```

will quit after copying the first two lines of the input. The output will be:

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
```

## 2. ADDRESSES: Selecting lines for editing

Lines in the input file(s) to which editing commands are to be applied can be selected by addresses. Addresses may be either line numbers or context addresses.

The application of a group of commands can be controlled by one address (or address-pair) by grouping the commands with curly braces ('{ }')(Sec. 3.6.).

### 2.1. Line-number Addresses

A line number is a decimal integer. As each line is read from the input, a line-number counter is incremented; a line-number address matches (selects) the input line which causes the internal counter to equal the address line-number. The counter runs cumulatively through multiple input files; it is not reset when a new input file is opened.

As a special case, the character `$` matches the last line of the last input file.

### 2.2. Context Addresses

A context address is a pattern ('regular expression') enclosed in slashes ('/'). The regular expressions recognized by `sed` are constructed as follows:

- 1) An ordinary character (not one of those discussed below) is a regular expression, and matches that character.
- 2) A circumflex '^' at the beginning of a regular expression matches the null character at the beginning of a line.
- 3) A dollar-sign '\$' at the end of a regular expression matches the null character at the end of a line.
- 4) The characters '\n' match an imbedded newline character, but not the newline at the end of the pattern space.
- 5) A period '.' matches any character except the terminal newline of the pattern space.
- 6) A regular expression followed by an asterisk '\*' matches any number (including 0) of adjacent occurrences of the regular expression it follows.
- 7) A string of characters in square brackets '[' ]' matches any character in the string, and no others. If, however, the first character of the string is circumflex '^', the regular expression matches any character *except* the characters in the string and the terminal newline of the pattern space.
- 8) A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
- 9) A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side-effects which are described under the `s` command below and specification 10) immediately below.
- 10) The expression '\d' means the same string of characters matched by an expression enclosed in '\(' and '\)' earlier in the same pattern. Here *d* is a single digit; the string specified is that beginning with the *d*th occurrence of '\(' counting from the left. For example, the expression '\(.\*)\1' matches a line beginning with two repeated occurrences of the same string.
- 11) The null regular expression standing alone (e.g., '/') is equivalent to the last regular expression compiled.

To use one of the special characters (^ \$ . \* [ ] \ /) as a literal (to match an occurrence of itself in the input), precede the special character by a backslash '\.'

For a context address to 'match' the input requires that the whole pattern within the address match some portion of the pattern space.

### 2.3. Number of Addresses

The commands in the next section can have 0, 1, or 2 addresses. Under each command the maximum number of allowed addresses is given. For a command to have more addresses than the maximum allowed is considered an error.

If a command has no addresses, it is applied to every line in the input.

If a command has one address, it is applied to all lines which match that address.

If a command has two addresses, it is applied to the first line which matches the first address, and to all subsequent lines until (and including) the first subsequent line which matches the second address. Then an attempt is made on subsequent lines to again match the first address.



and the process is repeated.

Two addresses are separated by a comma.

**Examples:**

|             |                                          |
|-------------|------------------------------------------|
| /an/        | matches lines 1, 3, 4 in our sample text |
| /an.*an/    | matches line 1                           |
| /^an/       | matches no lines                         |
| /./         | matches all lines                        |
| /\./        | matches line 5                           |
| /r*an/      | matches lines 1,3, 4 (number = zero!)    |
| /(an\).*\1/ | matches line 1                           |

**3. FUNCTIONS**

All functions are named by a single character. In the following summary, the maximum number of allowable addresses is given enclosed in parentheses, then the single character function name, possible arguments enclosed in angles (< >), an expanded English translation of the single-character name, and finally a description of what each function does. The angles around the arguments are *not* part of the argument, and should not be typed in actual editing commands.

**3.1. Whole-line Oriented Functions**

(2)d -- delete lines

The *d* function deletes from the file (does not write to the output) all those lines matched by its address(es).

It also has the side effect that no further commands are attempted on the corpse of a deleted line; as soon as the *d* function is executed, a new line is read from the input, and the list of editing commands is re-started from the beginning on the new line.

(2)n -- next line

The *n* function reads the next line from the input, replacing the current line. The current line is written to the output if it should be. The list of editing commands is continued following the *n* command.

(1)a\

<text> -- append lines

The *a* function causes the argument <text> to be written to the output after the line matched by its address. The *a* command is inherently multi-line; *a* must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, the interior newlines must be hidden by a backslash character ('\') immediately preceding the new-line. The <text> argument is terminated by the first unhidden newline (the first one not immediately preceded by backslash).

Once an *a* function is successfully executed, <text> will be written to the output regardless of what later commands do to the line which triggered it. The triggering line may be deleted entirely; <text> will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. It does not cause any change in the line-number counter.

(1)i\

<text> -- insert lines

The *i* function behaves identically to the *a* function, except that <text> is written to the output *before* the matched line. All other comments about the *a* function apply to the *i* function as well.

(2)c\  
<text> -- change lines

The *c* function deletes the lines selected by its address(es), and replaces them with the lines in <text>. Like *a* and *i*, *c* must be followed by a newline hidden by a backslash; and interior new lines in <text> must be hidden by backslashes.

The *c* command may have two addresses, and therefore select a range of lines. If it does, all the lines in the range are deleted, but only one copy of <text> is written to the output, *not* one copy per line deleted. As with *a* and *i*, <text> is not scanned for address matches, and no editing commands are attempted on it. It does not change the line-number counter.

After a line has been deleted by a *c* function, no further commands are attempted on the corpse.

If text is appended after a line by *a* or *r* functions, and the line is subsequently changed, the text inserted by the *c* function will be placed *before* the text of the *a* or *r* functions. (The *r* function is described in Section 3.4.)

*Note:* Within the text put in the output by these functions, leading blanks and tabs will disappear, as always in *sed* commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash; the backslash will not appear in the output.

**Example:**

The list of editing commands:

```
n
a\
XXXX
d
```

applied to our standard input, produces:

```
In Xanadu did Kubhla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

In this particular case, the same effect would be produced by either of the two following command lists:

```
n n
i\ c\
XXXX XXXX
d
```

**3.2. Substitute Function**

One very important function changes parts of lines selected by a context search within the line.

(2)s<pattern><replacement><flags> -- substitute

The *s* function replaces *part* of a line (selected by <pattern>) with <replacement>. It can best be read:

Substitute for <pattern>, <replacement>

The `<pattern>` argument contains a pattern, exactly like the patterns in addresses (see 2.2 above). The only difference between `<pattern>` and a context address is that the context address must be delimited by slash (`/`) characters; `<pattern>` may be delimited by any character other than space or newline.

By default, only the first string matched by `<pattern>` is replaced, but see the `g` flag below.

The `<replacement>` argument begins immediately after the second delimiting character of `<pattern>`, and must be followed immediately by another instance of the delimiting character. (Thus there are exactly *three* instances of the delimiting character.)

The `<replacement>` is not a pattern, and the characters which are special in patterns do not have special meaning in `<replacement>`. Instead, other characters are special:

`&` is replaced by the string matched by `<pattern>`

`\d` (where *d* is a single digit) is replaced by the *d*th substring matched by parts of `<pattern>` enclosed in `\(` and `\)`. If nested substrings occur in `<pattern>`, the *d*th is determined by counting opening delimiters (`\(`).

As in patterns, special characters may be made literal by preceding them with backslash (`\`).

The `<flags>` argument may contain the following flags:

`g` -- substitute `<replacement>` for all (non-overlapping) instances of `<pattern>` in the line. After a successful substitution, the scan for the next instance of `<pattern>` begins just after the end of the inserted characters; characters put into the line from `<replacement>` are not rescanned.

`p` -- print the line if a successful replacement was done. The `p` flag causes the line to be written to the output if and only if a substitution was actually made by the `s` function. Notice that if several `s` functions, each followed by a `p` flag, successfully substitute in the same input line, multiple copies of the line will be written to the output: one for each successful substitution.

`w <filename>` -- write the line to a file if a successful replacement was done. The `w` flag causes lines which are actually substituted by the `s` function to be written to a file named by `<filename>`. If `<filename>` exists before `sed` is run, it is overwritten; if not, it is created.

A single space must separate `w` and `<filename>`.

The possibilities of multiple, somewhat different copies of one input line being written are the same as for `p`.

A maximum of 10 different file names may be mentioned after `w` flags and `w` functions (see below), combined.

**Examples:**

The following command, applied to our standard input,

`s/to/by/w changes`

produces, on the standard output:

In Xanadu did Kubhla Khan  
A stately pleasure dome decree:  
Where Alph, the sacred river, ran  
Through caverns measureless by man  
Down by a sunless sea.

and, on the file 'changes':

Through caverns measureless by man  
Down by a sunless sea.

If the nocopy option is in effect, the command:

`s/[.,;?:]/*P&*/gp`

produces:

A stately pleasure dome decree\*P:\*  
Where Alph\*P,\* the sacred river\*P,\* ran  
Down to a sunless sea\*P.\*

Finally, to illustrate the effect of the *g* flag, the command:

`/X/s/an/AN/p`

produces (assuming nocopy mode):

In XANadu did Kubhla Khan

and the command:

`/X/s/an/AN/gp`

produces:

In XANadu did Kubhla KhAN

**3.3. Input-output Functions**

(2)p -- print

The print function writes the addressed lines to the standard output file. They are written at the time the *p* function is encountered, regardless of what succeeding editing commands may do to the lines.

(2)w <filename> -- write on <filename>

The write function writes the addressed lines to the file named by <filename>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write function is encountered for each line, regardless of what subsequent editing commands may do to them.

Exactly one space must separate the *w* and <filename>.

A maximum of ten different files may be mentioned in write functions and *w* flags after *s* functions, combined.

(1)r <filename> -- read the contents of a file

The read function reads the contents of <filename>, and appends them after the line matched by the address. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address. If *r* and *a* functions are executed on the same line, the text from the *a*

functions and the *r* functions is written to the output in the order that the functions are executed.

Exactly one space must separate the *r* and <filename>. If a file mentioned by a *r* function cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

NOTE: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in *w* functions or flags; that number is reduced by one if any *r* functions are present. (Only one read file is open at one time.)

### Examples

Assume that the file 'note1' has the following contents:

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

Then the following command:

```
/Kubla/r note1
```

produces:

In Xanadu did Kubla Khan

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan, and founder of the Mongol dynasty in China.

A stately pleasure dome decree:

Where Alph, the sacred river, ran

Through caverns measureless to man

Down to a sunless sea.

### 3.4. Multiple Input-line Functions

Three functions, all spelled with capital letters, deal specially with pattern spaces containing imbedded newlines; they are intended principally to provide pattern matches across lines in the input.

(2)N -- Next line

The next input line is appended to the current line in the pattern space; the two input lines are separated by an imbedded newline. Pattern matches may extend across the imbedded newline(s).

(2)D -- Delete first part of the pattern space

Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline was the terminal newline), read another line from the input. In any case, begin the list of editing commands again from its beginning.

(2)P -- Print first part of the pattern space

Print up to and including the first newline in the pattern space.

The *P* and *D* functions are equivalent to their lower-case counterparts if there are no imbedded newlines in the pattern space.

### 3.5. Hold and Get Functions

Four functions save and retrieve part of the input for possible later use.

(2)h -- hold pattern space

The *h* function copies the contents of the pattern space into a hold area (destroying the previous contents of the hold area).

(2)H -- Hold pattern space

The *H* function appends the contents of the pattern space to the contents of the hold area; the former and new contents are separated by a newline.

(2)g -- get contents of hold area

The *g* function copies the contents of the hold area into the pattern space (destroying the previous contents of the pattern space).

(2)G -- Get contents of hold area

The *G* function appends the contents of the hold area to the contents of the pattern space; the former and new contents are separated by a newline.

(2)x -- exchange

The exchange command interchanges the contents of the pattern space and the hold area.

#### Example

The commands

```
lh
ls/ did.*//
lx
G
s^n/ :/
```

applied to our standard example, produce:

```
In Xanadu did Kubla Khan :In Xanadu
A stately pleasure dome decree: :In Xanadu
Where Alph, the sacred river, ran :In Xanadu
Through caverns measureless to man :In Xanadu
Down to a sunless sea. :In Xanadu
```

### 3.6. Flow-of-Control Functions

These functions do no editing on the input lines, but control the application of functions to the lines selected by the address part.

(2)! -- Don't

The *Don't* command causes the next command (written on the same line), to be applied to all and only those input lines *not* selected by the address part.

(2){ -- Grouping

The grouping command '{' causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line.

The group of commands is terminated by a matching `'}'` standing on a line by itself.

Groups can be nested.

(0):<label> -- place a label

The label function marks a place in the list of editing commands which may be referred to by *b* and *r* functions. The <label> may be any sequence of eight or fewer characters; if two different colon functions have identical labels, a compile time diagnostic will be generated, and no execution attempted.

(2)b<label> -- branch to label

The branch function causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon function with the same <label> was encountered. If no colon function with the same label can be found after all the editing commands have been compiled, a compile time diagnostic is produced, and no execution is attempted.

A *b* function with no <label> is taken to be a branch to the end of the list of editing commands; whatever should be done with the current input line is done, and another input line is read; the list of editing commands is restarted from the beginning on the new line.

(2)t<label> -- test substitutions

The *t* function tests whether *any* successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing. The flag which indicates that a successful substitution has been executed is reset by:

- 1) reading a new input line, or
- 2) executing a *t* function.

### 3.7. Miscellaneous Functions

(1)= -- equals

The = function writes to the standard output the line number of the line matched by its address.

(1)q -- quit

The *q* function causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

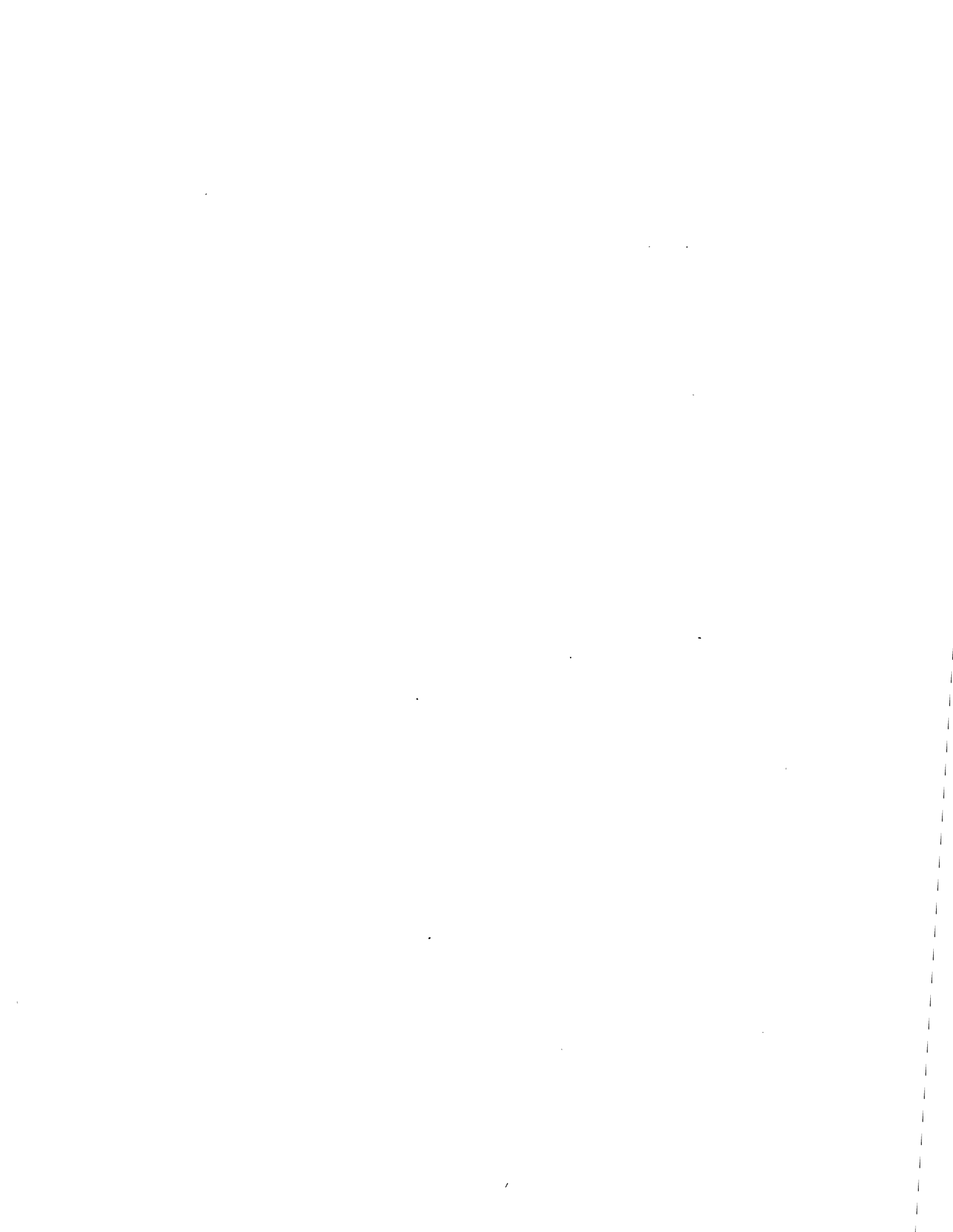
### Reference

- [1] Ken Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual*. Bell Laboratories, 1978.









# Awk — A Pattern Scanning and Processing Language (Second Edition)

*Alfred V. Aho*

*Brian W. Kernighan*

*Peter J. Weinberger*

Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

*Awk* is a programming language whose basic operation is to search a set of files for patterns, and to perform specified actions upon lines or fields of lines which contain instances of those patterns. *Awk* makes certain data selection and transformation operations easy to express; for example, the *awk* program

```
length > 72
```

prints all input lines whose length exceeds 72 characters; the program

```
NF % 2 == 0
```

prints all lines with an even number of fields; and the program

```
{ $1 = log($1); print }
```

replaces the first field of each line by its logarithm.

*Awk* patterns may include arbitrary boolean combinations of regular expressions and of relational operators on strings, numbers, fields, variables, and array elements. Actions may include the same pattern-matching constructions as in patterns, as well as arithmetic and string expressions and assignments, if-else, while, for statements, and multiple output streams.

This report contains a user's guide, a discussion of the design and implementation of *awk*, and some timing statistics.

September 1, 1978

# Awk — A Pattern Scanning and Processing Language (Second Edition)

*Alfred V. Aho*

*Brian W. Kernighan*

*Peter J. Weinberger*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 1. Introduction

*Awk* is a programming language designed to make many common information retrieval and text manipulation tasks easy to state and to perform.

The basic operation of *awk* is to scan a set of input lines in order, searching for lines which match any of a set of patterns which the user has specified. For each pattern, an action can be specified; this action will be performed on each line that matches the pattern.

Readers familiar with the UNIX† program *grep*<sup>1</sup> will recognize the approach, although in *awk* the patterns may be more general than in *grep*, and the actions allowed are more involved than merely printing the matching line. For example, the *awk* program

```
{print $3, $2}
```

prints the third and second columns of a table in that order. The program

```
$2 ~ /A|B|C/
```

prints all input lines with an A, B, or C in the second field. The program

```
$1 != prev { print; prev = $1 }
```

prints all lines in which the first field is different from the previous first field.

### 1.1. Usage

The command

```
awk program [files]
```

executes the *awk* commands in the string program on the set of named files, or on the standard input if there are no files. The statements can also be placed in a file *pfile*, and executed by the command

```
awk -f pfile [files]
```

### 1.2. Program Structure

An *awk* program is a sequence of statements of the form:

```
pattern { action }
pattern { action }
...
```

Each line of input is matched against each of the patterns in turn. For each pattern that matches, the associated action is executed. When all the patterns have been tested, the next line is fetched and the matching starts over.

Either the pattern or the action may be left out, but not both. If there is no action for a pattern, the matching line is simply copied to the output. (Thus a line which matches several patterns can be printed several times.) If there is no pattern for an action, then the action is performed for every input line. A line which matches no pattern is ignored.

Since patterns and actions are both optional, actions must be enclosed in braces to distinguish them from patterns.

### 1.3. Records and Fields

*Awk* input is divided into "records" terminated by a record separator. The default record separator is a newline, so by default *awk* processes its input a line at a time. The number of the current record is available in a variable named *NR*.

Each input record is considered to be divided into "fields." Fields are normally separated by white space — blanks or tabs — but the input field separator may be changed, as described below. Fields are referred to as *S1*, *S2*, and so forth, where *S1* is the first field, and *S0* is the whole input record itself. Fields may

†UNIX is a Trademark of Bell Laboratories.

be assigned to. The number of fields in the current record is available in a variable named `NF`.

The variables `FS` and `RS` refer to the input field and record separators; they may be changed at any time to any single character. The optional command-line argument `-Fc` may also be used to set `FS` to the character `c`.

If the record separator is empty, an empty input line is taken as the record separator, and blanks, tabs and newlines are treated as field separators.

The variable `FILENAME` contains the name of the current input file.

#### 1.4. Printing

An action may have no pattern, in which case the action is executed for all lines. The simplest action is to print some or all of a record; this is accomplished by the `awk` command `print`. The `awk` program

```
{ print }
```

prints each record, thus copying the input to the output intact. More useful is to print a field or fields from each record. For instance,

```
print $2, $1
```

prints the first two fields in reverse order. Items separated by a comma in the print statement will be separated by the current output field separator when output. Items not separated by commas will be concatenated, so

```
print $1 $2
```

runs the first and second fields together.

The predefined variables `NF` and `NR` can be used; for example

```
{ print NR, NF, $0 }
```

prints each record preceded by the record number and the number of fields.

Output may be diverted to multiple files; the program

```
{ print $1 >"foo1"; print $2 >"foo2" }
```

writes the first field, `$1`, on the file `foo1`, and the second field on file `foo2`. The `>>` notation can also be used:

```
print $1 >>"foo"
```

appends the output to the file `foo`. (In each case, the output files are created if necessary.) The file name can be a variable or a field as well as a constant; for example,

```
print $1 >$2
```

uses the contents of field 2 as a file name.

Naturally there is a limit on the number of output files; currently it is 10.

Similarly, output can be piped into another process (on UNIX only); for instance,

```
print | "mail bwk"
```

mails the output to `bwk`.

The variables `OFS` and `ORS` may be used to change the current output field separator and output record separator. The output record separator is appended to the output of the print statement.

`awk` also provides the `printf` statement for output formatting:

```
printf format expr, expr, ...
```

formats the expressions in the list according to the specification in `format` and prints them. For example,

```
printf "%8.2f %10ld\n", $1, $2
```

prints `$1` as a floating point number 8 digits wide, with two after the decimal point, and `$2` as a 10-digit long decimal number, followed by a newline. No output separators are produced automatically; you must add them yourself, as in this example. The version of `printf` is identical to that used with `C`.<sup>2</sup>

## 2. Patterns

A pattern in front of an action acts as a selector that determines whether the action is to be executed. A variety of expressions may be used as patterns: regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.

### 2.1. BEGIN and END

The special pattern `BEGIN` matches the beginning of the input, before the first record is read. The pattern `END` matches the end of the input, after the last record has been processed. `BEGIN` and `END` thus provide a way to gain control before and after processing, for initialization and wrapup.

As an example, the field separator can be set to a colon by

```
BEGIN { FS = ":" }
... rest of program ...
```

Or the input lines may be counted by

```
END { print NR }
```

If `BEGIN` is present, it must be the first pattern; `END` must be the last if used.

## 2.2. Regular Expressions

The simplest regular expression is a literal string of characters enclosed in slashes, like

```
/smith/
```

This is actually a complete *awk* program which will print all lines which contain any occurrence of the name "smith". If a line contains "smith" as part of a larger word, it will also be printed, as in

```
blacksmithing
```

*Awk* regular expressions include the regular expression forms found in the UNIX text editor *ed* and *grep* (without back-referencing). In addition, *awk* allows parentheses for grouping, | for alternatives, + for "one or more", and ? for "zero or one", all as in *lex*. Character classes may be abbreviated: [a-zA-Z0-9] is the set of all letters and digits. As an example, the *awk* program

```
/[Aa]ho|[Ww]einberger|[Kk]ernighan/
```

will print all lines which contain any of the names "Aho," "Weinberger" or "Kernighan," whether capitalized or not.

Regular expressions (with the extensions listed above) must be enclosed in slashes, just as in *ed* and *sed*. Within a regular expression, blanks and the regular expression metacharacters are significant. To turn off the magic meaning of one of the regular expression characters, precede it with a backslash. An example is the pattern

```
/\./
```

which matches any string of characters enclosed in slashes.

One can also specify that any field or variable matches a regular expression (or does not match it) with the operators ~ and !~. The program

```
$1 ~ /[jJ]ohn/
```

prints all lines where the first field matches "john" or "John." Notice that this will also match "Johnson", "St. Johnsbury", and so on. To restrict it to exactly [jJ]ohn, use

```
$1 ~ /^[jJ]ohn$/
```

The caret ^ refers to the beginning of a line or field; the dollar sign \$ refers to the end.

## 2.3. Relational Expressions

An *awk* pattern can be a relational expression involving the usual relational operators <, <=, ==, !=, >=, and >. An example is

```
$2 > $1 + 100
```

which selects lines where the second field is at least 100 greater than the first field. Similarly,

```
NF % 2 == 0
```

prints lines with an even number of fields.

In relational tests, if neither operand is numeric, a string comparison is made; otherwise it is numeric. Thus,

```
$1 >= "s"
```

selects lines that begin with an s, t, u, etc. In the absence of any other information, fields are treated as strings, so the program

```
$1 > $2
```

will perform a string comparison.

## 2.4. Combinations of Patterns

A pattern can be any boolean combination of patterns, using the operators || (or), && (and), and ! (not). For example,

```
$1 >= "s" && $1 < "t" && $1 != "smith"
```

selects lines where the first field begins with "s", but is not "smith". && and || guarantee that their operands will be evaluated from left to right; evaluation stops as soon as the truth or falsehood is determined.

## 2.5. Pattern Ranges

The "pattern" that selects an action may also consist of two patterns separated by a comma, as in

```
pat1, pat2 { ... }
```

In this case, the action is performed for each line between an occurrence of pat1 and the next occurrence of pat2 (inclusive). For example,

```
/start/, /stop/
```

prints all lines between start and stop, while

```
NR == 100, NR == 200 { ... }
```

does the action for lines 100 through 200 of the input.

## 3. Actions

An *awk* action is a sequence of action statements terminated by newlines or semicolons. These action statements can be used to do a variety of bookkeeping and string manipulating tasks.

### 3.1. Built-in Functions

*Awk* provides a "length" function to compute the length of a string of characters. This program prints each record, preceded by its length:

```
{print length, $0}
```

`length` by itself is a "pseudo-variable" which yields the length of the current record; `length(argument)` is a function which yields the length of its argument, as in the equivalent

```
{print length($0), $0}
```

The argument may be any expression.

*Awk* also provides the arithmetic functions `sqrt`, `log`, `exp`, and `int`, for square root, base *e* logarithm, exponential, and integer part of their respective arguments.

The name of one of these built-in functions, without argument or parentheses, stands for the value of the function on the whole record. The program

```
length < 10 || length > 20
```

prints lines whose length is less than 10 or greater than 20.

The function `substr(s, m, n)` produces the substring of `s` that begins at position `m` (origin 1) and is at most `n` characters long. If `n` is omitted, the substring goes to the end of `s`. The function `index(s1, s2)` returns the position where the string `s2` occurs in `s1`, or zero if it does not.

The function `sprintf(f, e1, e2, ...)` produces the value of the expressions `e1`, `e2`, etc., in the `printf` format specified by `f`. Thus, for example,

```
x = sprintf("%8.2f %10ld", $1, $2)
```

sets `x` to the string produced by formatting the values of `$1` and `$2`.

### 3.2. Variables, Expressions, and Assignments

*Awk* variables take on numeric (floating point) or string values according to context. For example, in

```
x = 1
```

`x` is clearly a number, while in

```
x = "smith"
```

it is clearly a string. Strings are converted to numbers and vice versa whenever context demands it. For instance,

```
x = "3" + "4"
```

assigns 7 to `x`. Strings which cannot be inter-

preted as numbers in a numerical context will generally have numeric value zero, but it is unwise to count on this behavior.

By default, variables (other than built-ins) are initialized to the null string, which has numerical value zero; this eliminates the need for most `BEGIN` sections. For example, the sums of the first two fields can be computed by

```
{ s1 += $1; s2 += $2 }
END { print s1, s2 }
```

Arithmetic is done internally in floating point. The arithmetic operators are `+`, `-`, `*`, `/`, and `%` (mod). The C increment `++` and decrement `--` operators are also available, and so are the assignment operators `+=`, `-=`, `*=`, `/=`, and `%=`. These operators may all be used in expressions.

### 3.3. Field Variables

Fields in *awk* share essentially all of the properties of variables — they may be used in arithmetic or string operations, and may be assigned to. Thus one can replace the first field with a sequence number like this:

```
{ $1 = NR; print }
```

or accumulate two fields into a third, like this:

```
{ $1 = $2 + $3; print $0 }
```

or assign a string to a field:

```
{ if ($3 > 1000)
 $3 = "too big"
 print
}
```

which replaces the third field by "too big" when it is, and in any case prints the record.

Field references may be numerical expressions, as in

```
{ print $i, $(i+1), $(i+n) }
```

Whether a field is deemed numeric or string depends on context; in ambiguous cases like

```
if ($1 == $2) ...
```

fields are treated as strings.

Each input line is split into fields automatically as necessary. It is also possible to split any variable or string into fields:

```
n = split(s, array, sep)
```

splits the the string `s` into `array[1]`, ..., `array[n]`. The number of elements found is returned. If the `sep` argument is provided, it is used as the field separator; otherwise `FS` is used as the separator.

### 3.4. String Concatenation

Strings may be concatenated. For example  
`length($1 $2 $3)`

returns the length of the first three fields. Or in a print statement,

```
print $1 " is " $2
```

prints the two fields separated by " is ". Variables and numeric expressions may also appear in concatenations.

### 3.5. Arrays

Array elements are not declared; they spring into existence by being mentioned. Subscripts may have any non-null value, including non-numeric strings. As an example of a conventional numeric subscript, the statement

```
x[NR] = $0
```

assigns the current input record to the NR-th element of the array x. In fact, it is possible in principle (though perhaps slow) to process the entire input in a random order with the *awk* program

```
{ x[NR] = $0 }
END { ... program ... }
```

The first action merely records each input line in the array x.

Array elements may be named by non-numeric values, which gives *awk* a capability rather like the associative memory of Snobol tables. Suppose the input contains fields with values like apple, orange, etc. Then the program

```
/apple/ { x["apple"]++ }
/orange/ { x["orange"]++ }
END { print x["apple"], x["orange"] }
```

increments counts for the named array elements, and prints them at the end of the input.

### 3.6. Flow-of-Control Statements

*awk* provides the basic flow-of-control statements *if-else*, *while*, *for*, and statement grouping with braces, as in C. We showed the *if* statement in section 3.3 without describing it. The condition in parentheses is evaluated; if it is true, the statement following the *if* is done. The *else* part is optional.

The *while* statement is exactly like that of C. For example, to print all input fields one per line,

```
i = 1
while (i <= NF) {
 print $i
 ++i
}
```

The *for* statement is also exactly that of C:

```
for (i = 1; i <= NF; i++)
 print $i
```

does the same job as the *while* statement above.

There is an alternate form of the *for* statement which is suited for accessing the elements of an associative array:

```
for (i in array)
 statement
```

does *statement* with *i* set in turn to each element of array. The elements are accessed in an apparently random order. Chaos will ensue if *i* is altered, or if any new elements are accessed during the loop.

The expression in the condition part of an *if*, *while* or *for* can include relational operators like *<*, *<=*, *>*, *>=*, *==* ("is equal to"), and *!=* ("not equal to"); regular expression matches with the match operators *~* and *!~*; the logical operators *||*, *&&*, and *!*; and of course parentheses for grouping.

The *break* statement causes an immediate exit from an enclosing *while* or *for*; the *continue* statement causes the next iteration to begin.

The statement *next* causes *awk* to skip immediately to the next record and begin scanning the patterns from the top. The statement *exit* causes the program to behave as if the end of the input had occurred.

Comments may be placed in *awk* programs: they begin with the character *#* and end with the end of the line, as in

```
print x, y # this is a comment
```

## 4. Design

The UNIX system already provides several programs that operate by passing input through a selection mechanism. *Grep*, the first and simplest, merely prints all lines which match a single specified pattern. *Egrep* provides more general patterns, i.e., regular expressions in full generality; *Jgrep* searches for a set of keywords with a particularly fast algorithm. *Sed*<sup>1</sup> provides most of the editing facilities of the editor *ed*, applied to a stream of input. None of these programs provides numeric capabilities, logical relations, or variables.



*Lex*<sup>3</sup> provides general regular expression recognition capabilities, and, by serving as a C program generator, is essentially open-ended in its capabilities. The use of *lex*, however, requires a knowledge of C programming, and a *lex* program must be compiled and loaded before use, which discourages its use for one-shot applications.

*Awk* is an attempt to fill in another part of the matrix of possibilities. It provides general regular expression capabilities and an implicit input/output loop. But it also provides convenient numeric processing, variables, more general selection, and control flow in the actions. It does not require compilation or a knowledge of C. Finally, *awk* provides a convenient way to access fields within lines; it is unique in this respect.

*Awk* also tries to integrate strings and numbers completely, by treating all quantities as both string and numeric, deciding which representation is appropriate as late as possible. In most cases the user can simply ignore the differences.

Most of the effort in developing *awk* went into deciding what *awk* should or should not do (for instance, it doesn't do string substitution) and what the syntax should be (no explicit operator for concatenation) rather than on writing or debugging the code. We have tried to make the syntax powerful but easy to use and well adapted to scanning files. For example, the absence of declarations and implicit initializations, while probably a bad idea for a general-purpose programming language, is desirable in a language that is meant to be used for tiny programs that may even be composed on the command line.

In practice, *awk* usage seems to fall into two broad categories. One is what might be called "report generation" — processing an input to extract counts, sums, sub-totals, etc. This also includes the writing of trivial data validation programs, such as verifying that a field contains only numeric information or that certain delimiters are properly balanced. The combination of textual and numeric processing is invaluable here.

A second area of use is as a data transformer, converting data from the form produced by one program into that expected by another. The simplest examples merely select fields, perhaps with rearrangements.

## 5. Implementation

The actual implementation of *awk* uses the language development tools available on the UNIX operating system. The grammar is specified with *yacc*<sup>4</sup>; the lexical analysis is done by *lex*; the regular expression recognizers are deterministic finite automata constructed directly from the expressions. An *awk* program is translated into a parse tree which is then directly executed by a simple interpreter.

*Awk* was designed for ease of use rather than processing speed; the delayed evaluation of variable types and the necessity to break input into fields makes high speed difficult to achieve in any case. Nonetheless, the program has not proven to be unworkably slow.

Table I below shows the execution (user + system) time on a PDP-11/70 of the UNIX programs *wc*, *grep*, *egrep*, *fgrep*, *sed*, *lex*, and *awk* on the following simple tasks:

1. count the number of lines.
2. print all lines containing "doug".
3. print all lines containing "doug", "ken" or "dmr".
4. print the third field of each line.
5. print the third and second fields of each line, in that order.
6. append all lines containing "doug", "ken", and "dmr" to files "jdoug", "jken", and "jdmr", respectively.
7. print each line prefixed by "line-number: ".
8. sum the fourth column of a table.

The program *wc* merely counts words, lines and characters in its input; we have already mentioned the others. In all cases the input was a file containing 10,000 lines as created by the command *ls -l*; each line has the form

```
-rw-rw-rw- 1 ava 123 Oct 15 17:05 xxx
```

The total length of this input is 452,960 characters. Times for *lex* do not include compile or load.

As might be expected, *awk* is not as fast as the specialized tools *wc*, *sed*, or the programs in the *grep* family, but is faster than the more general tool *lex*. In all cases, the tasks were about as easy to express as *awk* programs as programs in these other languages; tasks involving fields were considerably easier to express as *awk* programs. Some of the test programs are shown in *awk*, *sed* and *lex*.

## References

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*, Bell Laboratories (May 1975). Sixth Edition
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
3. M. E. Lesk, "Lex - A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson, "Yacc - Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

| Program      | Task |       |       |      |      |       |      |      |
|--------------|------|-------|-------|------|------|-------|------|------|
|              | 1    | 2     | 3     | 4    | 5    | 6     | 7    | 8    |
| <i>wc</i>    | 8.6  |       |       |      |      |       |      |      |
| <i>grep</i>  | 11.7 | 13.1  |       |      |      |       |      |      |
| <i>egrep</i> | 6.2  | 11.5  | 11.6  |      |      |       |      |      |
| <i>fgrep</i> | 7.7  | 13.8  | 16.1  |      |      |       |      |      |
| <i>sed</i>   | 10.2 | 11.6  | 15.8  | 29.0 | 30.5 | 16.1  |      |      |
| <i>lex</i>   | 65.1 | 150.1 | 144.2 | 67.7 | 70.3 | 104.0 | 81.7 | 92.8 |
| <i>awk</i>   | 15.0 | 25.6  | 29.9  | 33.3 | 38.9 | 46.4  | 71.4 | 31.1 |

Table I. Execution Times of Programs. (Times are in sec.)

The programs for some of these jobs are shown below. The *lex* programs are generally too long to show.

AWK:

1. END {print NR}
2. /doug/
3. /ken|doug|dmr/
4. {print \$3}
5. {print \$3, \$2}
6. /ken/ {print >"jken"}  
/doug/ {print >"jdoug"}  
/dmr/ {print >"jdmr"}
7. {print NR ": " \$0}
8. {sum = sum + \$4}  
END {print sum}

SED:

1. \$=
2. /doug/p
3. /doug/p  
/doug/d  
/ken/p  
/ken/d  
/dmr/p  
/dmr/d
4. /[ ]• [ ]•[ ]• [ ]•\([ ]•\)\ .•/s/\1/p
5. /[ ]• [ ]•\([ ]•\)\ [ ]•\([ ]•\)\ .•/s/\2 \1/p
6. /ken/w jken  
/doug/w jdoug  
/dmr/w jdmr

LEX:

1. %{  
int i;  
%}  
%%  
\n i++;  
.  
%;  
yywrap() {  
printf("%d\n", i);  
}
2. %%  
".doug.\*\$ printf("%s\n", yytext);  
.  
;\n ;







# Lex - A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

*Bell Laboratories*

*Murray Hill, New Jersey 07974*

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can be used to generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

## Table of Contents

|                               |    |
|-------------------------------|----|
| 1. Introduction.              | 1  |
| 2. Lex Source.                | 3  |
| 3. Lex Regular Expressions.   | 3  |
| 4. Lex Actions.               | 5  |
| 5. Ambiguous Source Rules.    | 7  |
| 6. Lex Source Definitions.    | 8  |
| 7. Usage.                     | 8  |
| 8. Lex and Yacc.              | 9  |
| 9. Examples.                  | 10 |
| 10. Left Context Sensitivity. | 11 |
| 11. Character Set.            | 12 |
| 12. Summary of Source Format. | 12 |
| 13. Caveats and Bugs.         | 13 |
| 14. Acknowledgments.          | 13 |
| 15. References.               | 13 |

## 1 Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file asso-

ciates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to

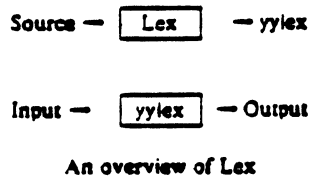


Figure 1

write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present there are only two host languages, C[1] and Fortran (in the form of the Ratfor language[2]). Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called *source* in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called *input* in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
%%
[\t] + $;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule.

This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates "one or more ..."; and the \$ indicates "end of line," as in QED. No action is specified, so the program generated by Lex (*yylex*) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[\t] + $;
[\t] + printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc [3]. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 2. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex. Yacc users will realize that the name *yylex* is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source [4]. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time

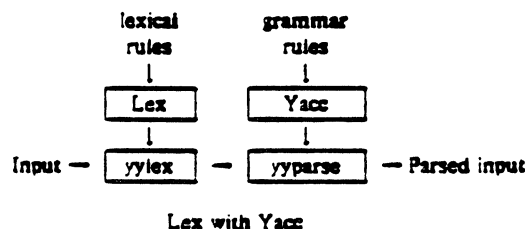


Figure 2



taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of re-scanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the *actions* to be performed as each regular expression is found) are gathered as cases of a switch (in C) or branches of a computed GOTO (in Ratfor). The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, Lex will recognize *ab* and leave the input pointer just before *cd*. . . Such backup is more costly than the processing of simpler languages.

## 2 Lex Source.

The general format of Lex source is:

```
(definitions)
%%
(rules)
%%
(user subroutines)
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the *rules* represent the user's control decisions; they are a table, in which the left column contains *regular expressions* (see section 3) and the right column contains *actions*, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

```
integer printf("found keyword INT");
```

to look for the string *integer* in the input stream and print the message "found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function *printf* is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in

braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

```
colour printf("color");
mechanise printf("mechanize");
petrol printf("gas");
```

would be a start. These rules are not quite enough, since the word *petroleum* would become *gaseum*, a way of dealing with this will be described later.

## 3 Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string *integer* wherever it appears and the expression

```
a57D
```

looks for the string *a57D*.

*Operators.* The operator characters are

```
" \ [] ^ - ? . * + | () $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

```
xyz'++"
```

matches the string *xyz++* when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above

expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

**Character classes.** Classes of characters can be specified using the operator pair []. The construction *[ab]* matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^ . The - character indicates ranges. For example,

`[a-z0-9<>_]`

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

`[-+0-9]`

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

`[^abc]`

matches all characters except a, b, or c, including all special or control characters; or

`[^a-zA-Z]`

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

**Arbitrary character.** To match almost any character, the operator character

is the class of all characters except newline. Escaping into octal is possible although non-portable:

`[\40-176]`

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

**Optional expressions.** The operator ? indicates an optional element of an expression. Thus

`ab?c`

matches either *ac* or *abc*.

**Repeated expressions.** Repetitions of classes are indicated by the operators \* and +.

`a*`

is any number of consecutive *a* characters, including zero; while

`a+`

is one or more instances of *a*. For example,

`[a-z]+`

is all strings of lower case letters. And

`[A-Za-z][A-Za-z0-9]*`

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

**Alternation and Grouping.** The operator | indicates alternation:

`(ab|cd)`

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary on the outside level;

`ab|cd`

would have sufficed. Parentheses can be used for more complex expressions:

`(ab|cd+)?(ef)*`

matches such strings as *abefef*, *efefef*, *cdef*, or *cddd*; but not *abc*, *abcd*, or *abcdef*.

**Context sensitivity.** Lex will recognize a small amount of surrounding context. The two simplest operators for this are ^ and \$. If the first character of an expression is ^, the expression will only be matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of ^, complementation of character classes, since that only applies within the [] operators. If the very last character is \$ the expression will only be matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the /operator character, which indicates trailing context. The expression

`ab/cd`

matches the string *ab*, but only if followed by *cd*. Thus

```
ab$
```

is the same as

```
ab/\n
```

Left context is handled in Lex by *start conditions* as explained in section 10. If a rule is only to be executed when the Lex automaton interpreter is in start condition *x*, the rule should be prefixed by

```
<x>
```

using the angle bracket operator characters. If we considered "being at the beginning of a line" to be start condition *ONE*, then the  $\wedge$  operator would be equivalent to

```
<ONE>
```

Start conditions are explained more fully later.

*Repetitions and Definitions.* The operators  $\{ \}$  specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression. The definitions are given in the first part of the Lex input, before the rules. In contrast,

```
a{1,5}
```

looks for 1 to 5 occurrences of *a*.

Finally, initial  $\%$  is special, being the separator for Lex source segments.

#### 4 Lex Actions.

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When Lex is being used with Yacc, this is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement,  $;$  as an action causes this result. A frequent rule is

```
[\t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is the action character  $|$ , which indicates that the action for this rule is the action for the next rule. The previous example could also have been written

```
" "
"\t"
"\n"
```

with the same result, although in different style. The quotes around  $\backslash n$  and  $\backslash t$  are not required.

In more complex actions, the user will often want to know the actual text that matched some expression like  $[a-z]^+$ . Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

```
[a-z]+ printf("%s", yytext);
```

will print the string in *yytext*. The C function *printf* accepts a format argument and data to be printed; in this case, the format is "print string" ( $\%$  indicating data conversion, and *s* indicating string type), and the data are the characters in *yytext*. So this just places the matched string on the output. This action is so common that it may be written as ECHO:

```
[a-z]+ ECHO;
```

is the same as the above. Since the default action is just to print the characters found, one might ask why give a rule, like this one, which merely specifies the default action? Such rules are often required to avoid matching some other rule which is not desired. For example, if there is a rule which matches *read* it will normally match the instances of *read* contained in *bread* or *readjust*, to avoid this, a rule of the form  $[a-z]^+$  is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence Lex also provides a count *yylen* of the number of characters matched. To count both the number of words and the number of characters in words in the input, the user might write

```
[a-zA-Z]+ (words++; chars += yylen);
```

which accumulates in *chars* the number of characters in the words recognized. The last character in the string matched can be accessed by

```
yytext[yylen-1]
```

in C or

```
yytext(yylen)
```

in Ratfor.

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore()* can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in *yytext*. Second, *yyless(n)* may be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the / operator, but in a different form.

*Example:* Consider a language which defines a string as a set of characters between quotation (") marks, and provides that to include a " in a string it must be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write

```
\["*"- {
 if (yytext[yyteng-1] == "\\")
 yymore();
 else
 ... normal user processing
}
```

which will, when faced with a string such as "abc\def" first match the five characters "abc\"; then the call to *yymore()* will cause the next part of the string, "def", to be tacked on the end. Note that the final quote terminating the string should be picked up in the code labeled "normal processing".

The function *yyless()* might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of "--a". Suppose it is desired to treat this as "-- a" but print a message. A rule might be

```
--[a-zA-Z] {
 printf("Operator (-- ambiguous\n");
 yyless(yyteng-1);
 ... action for -- ...
}
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as "--". Alternatively it might be desired to treat this as "= -a". To do this, just return the minus sign as well as the letter to the input:

```
--[a-zA-Z] {
 printf("Operator (-- ambiguous\n");
 yyless(yyteng-2);
 ... action for = - ...
}
```

will perform the other interpretation. Note that the expressions for the two cases might more easily be written

```
--/[A-Za-z]
```

in the first case and

```
--/[A-Za-z]
```

in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of "--3", however, makes

```
--/["\n]
```

a still better rule.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

- 1) *input()* which returns the next input character;
- 2) *output(c)* which writes the character *c* on the output; and
- 3) *unput(c)* pushes the character *c* back onto the input stream to be read later by *input()*.

By default these routines are provided as macro definitions, but the user can override them and supply private versions. There is another important routine in Ratfor, named *lexshf*, which is described below under "Character Set". These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input* must mean end of file; and the relationship between *unput* and *input* must be retained or the Lex lookahead will not work. Lex does not look ahead at all if it does not have to, but every rule ending in + \* ? or \$ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by Lex. The standard Lex library imposes a 100 character limit on backup.

Another Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If *yywrap* returns a 1, Lex continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default *yywrap* always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through *yywrap*. In fact, unless a private version of *input()* is supplied a file containing nulls cannot be handled, since a value of 0 returned by *input* is taken to be end-of-file.

In Ratfor all of the standard I/O library routines, *input*,

*output*, *unput*, *yywrap*, and *lexshf*, are defined as integer functions. This requires *input* and *yywrap* to be called with arguments. One dummy argument is supplied and ignored.

### 5 Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

- 1) The longest match is preferred.
- 2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

```
integer keyword action ...;
[a-z]+ identifier action ...;
```

to be given in that order. If the input is *integers*, it is taken as an identifier, because *[a-z]+* matches 8 characters while *integer* matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. *int*) will not match the expression *integer* and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like *.\** dangerous. For example,

```
'.'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

```
'first' quoted string here, 'second' here
```

the above expression will match

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule is of the form

```
'[^\n]*'
```

which, on the above input, will stop after *'first'*. The consequences of errors like this are mitigated by the fact that the *.* operator will not match newline. Thus expressions like *.\** stop on the current line. Don't try to defeat this with expressions like *[^\n]+* or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Note that Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some

Lex rules to do this might be

```
she s++;
he h++;
\n |
. ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that *.* does not include newline. Since *she* includes *he*, Lex will normally *not* recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means "go do the next alternative." It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*.

```
she {s++; REJECT;}
he {h++; REJECT;}
\n |
. ;
```

these rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that *she* includes *he* but not vice versa, and omit the REJECT action on *he*, in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

```
a[bc]+ { ... ; REJECT;}
a[cd]+ { ... ; REJECT;};
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word *the* is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* to be incremented, the appropriate source is

```
%%
[a-z][a-z] {digram[yytext[0]][yytext[1]]++; REJECT;}
\n ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

## 6 Lex Source Definitions.

Remember the format of the Lex source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far only the rules have been described. The user needs additional options, though, to define variables for use in his program and for use by Lex. These can go either in the definitions section or in the rules section.

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program. There are three classes of such things.

- 1) Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %% , it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule.

As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

- 2) Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.
- 3) Anything after the third %% delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first %% delimiter. Any line in this section not contained between %{ and %} , and beginning in column 1, is assumed to define Lex substitution strings. The format of such lines is

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field, for example, might abbreviate rules to recognize numbers:

```
D [0-9]
E [TEde|[-+]?{D}+
%%
{D}+ printf("integer");
{D}+"."{D}-({E})? |
{D}+"."{D}+({E})? |
{D}+{E}
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a Fortran expression such as *35.EQ.L*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"EQ printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs. These possibilities are discussed below under "Summary of Source Format," section 12.

## 7 Usage.

There are two steps in compiling a Lex source program. First, the Lex source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of Lex subroutines. The generated program is on a file named *lex.yy.c* for a C host language source and *lex.yy.r* for a Ratfor host environment. There are two I/O libraries, one for C defined in terms of the C standard library [6], and the other defined in terms of Ratfor. To indicate that a Lex source file is intended to be used with the Ratfor host language, make the first line of the file %R.

The C programs generated by Lex are slightly different on OS/370, because the OS compiler is less powerful than the UNIX or GCOS compilers, and does less at compile time. C programs generated on GCOS and UNIX are the same. The C host language is default, but may be explicitly requested by making the first line of the source file %C.

The Ratfor generated by Lex is the same on all systems, but can not be compiled directly on TSO. See below for instructions. The Ratfor I/O library, however, varies slightly because the different Fortrans disagree on the method of indicating end-of-input and the name of the library routine for logical AND. The Ratfor I/O library, dependent on Fortran character I/O, is quite slow. In particular it reads all input lines as 80A1 format; this will truncate any longer line, discarding your data, and pads any shorter line with blanks. The library version of *input* removes the padding (including any trailing blanks from the original input) before processing. Each source

file using a Ratfor host should begin with the "%R" command.

**UNIX.** The libraries are accessed by the loader flags *-llc* for C and *-llr* for Ratfor; the C name may be abbreviated to *-ll*. So an appropriate set of commands is

C Host	Ratfor Host
lex source	lex source
cc lex.yy.c -ll -lS	rc -2 lex.yy.r -llr

The resulting program is placed on the usual file *a.out* for later execution. To use Lex with Yacc see below. Although the default Lex I/O routines use the C standard library, the Lex automata themselves do not do so; if private versions of *input*, *output* and *unput* are given, the library can be avoided. Note the "-2" option in the Ratfor compile command; this requests the larger version of the compiler, a useful precaution.

**GCOS.** The Lex commands on GCOS are stored in the "." library. The appropriate command sequences are:

C Host	Ratfor Host
./lex source	./lex source
/cc lex.yy.c ./lexclib h=	/rc a= lex.yy.r ./lexrlib h=

The resulting program is placed on the usual file *.program* for later execution (as indicated by the "h=" option); it may be copied to a permanent file if desired. Note the "a=" option in the Ratfor compile command; this indicates that the Fortran compiler is to run in ASCII mode.

**TSO.** Lex is just barely available on TSO. Restrictions imposed by the compilers which must be used with its output make it rather inconvenient. To use the C version, type

```
exec 'dot.lex.clist(lex)' 'sourcename'
exec 'dot.lex.clist(cload)' 'libraryname membername'
```

The first command analyzes the source file and writes a C program on file *lex.yy.text*. The second command runs this file through the C compiler and links it with the Lex C library (stored on 'hr289.lcl.load') placing the object program in your file *libraryname.LOAD(membername)* as a completely linked load module. The compiling command uses a special version of the C compiler command on TSO which provides an unusually large intermediate assembler file to compensate for the unusual bulk of C-compiled Lex programs on the OS system. Even so, almost any Lex source program is too big to compile, and must be split.

The same Lex command will compile Ratfor Lex programs, leaving a file *lex.yy.rat* instead of *lex.yy.text* in your directory. The Ratfor program must be edited, however, to compensate for peculiarities of IBM Ratfor. A command sequence to do this, and then compile and load, is available. The full commands are:

```
exec 'dot.lex.clist(lex)' 'sourcename'
```

```
exec 'dot.lex.clist(rload)' 'libraryname membername'
```

with the same overall effect as the C language commands. However, the Ratfor commands will run in a 150K byte partition, while the C commands require 250K bytes to operate.

The steps involved in processing the generated Ratfor program are:

- a. Edit the Ratfor program.
  1. Remove all tabs.
  2. Change all lower case letters to upper case letters.
  3. Convert the file to an 80-column card image file.
- b. Process the Ratfor through the Ratfor preprocessor to get Fortran code.
- c. Compile the Fortran.
- d. Load with the libraries 'hr289.lrl.load' and 'sys1.fortlib'.

The final load module will only read input in 80-character fixed length records. **Warning:** Work is in progress on the IBM C compiler, and Lex and its availability on the IBM 370 are subject to change without notice.

## 8 Lex and Yacc.

If you want to use Lex with Yacc, note that what Lex writes is a program named *yylex()*, the name required by Yacc for its analyzer. Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call *yylex()*. In this case each Lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named "good" and the lexical rules to be named "better" the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll -lS
```

The Yacc library (-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser. The generations of Lex and Yacc programs can be done in either order.

## 9 Examples.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable Lex source program

```

%%
[0-9]+
 int k;
 {
 scanf(-1, yytext, "%d", &k);
 if (k%7 == 0)
 printf("%d", k+3);
 else
 printf("%d", k);
 }

```

to do just that. The rule `[0-9]+` recognizes strings of digits; `scanf` converts the digits to binary and stores the result in `k`. The operator `%` (remainder) is used to check whether `k` is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as `49.63` or `X7`. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```

%%
-?[0-9]+
 int k;
 {
 scanf(-1, yytext, "%d", &k);
 printf("%d", k%7 == 0 ? k+3 : k);
 }
-?[0-9.]+
 ECHO;
[A-Za-z][A-Za-z0-9]+
 ECHO;

```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The *if-else* has been replaced by a C conditional expression to save space; the form `a?b:c` means "if `a` then `b` else `c`".

For an example of statistics gathering, here is a program which histograms the lengths of words, where a word is defined as a string of letters.

```

int lengs[100];
%%
[a-z]+
 lengs[yytext]++;
\n
%%
yywrap()
{
 int i;
 printf("Length No. words\n");
 for(i=0; i<100; i++)
 if (lengs[i] > 0)
 printf("%5d%10d\n", i, lengs[i]);
 return(1);
}

```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement `return(1)`; indicates that Lex is to perform wrapup. If `yywrap` returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a `yywrap` that

never returns true causes an infinite loop.

As a larger example, here are some parts of a program written by N. L. Schryer to convert double precision Fortran to single precision Fortran. Because Fortran does not distinguish upper and lower case letters, this routine begins by defining a set of classes including both cases of each letter:

```

a [aA]
b [bB]
c [cC]
...
z [zZ]

```

An additional class recognizes white space:

```

W [\t]

```

The first rule changes "double precision" to "real", or "DOUBLE PRECISION" to "REAL".

```

(d)(o){u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
 printf(yytext[0] == 'd' ? "real" : "REAL");
}

```

Care is taken throughout this program to preserve the case (upper or lower) of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```

" "[0] ECHO;

```

In the regular expression, the quotes surround the blanks. It is interpreted as "beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of `"`. There follow some rules to change double precision constants to ordinary floating constants.

```

[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ |
[0-9]+{W}"{W}{d}{W}[+-]?{W}[0-9]+ |
"{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+ {
 /* convert constants */
 for(p=yytext; *p != 0; p++)
 {
 if (*p == 'd' | *p == 'D')
 *p = + 'e' - 'd';
 ECHO;
 }
}

```

After the floating point constant is recognized, it is scanned by the `for` loop to find the letter `d` or `D`. The program then adds `'e'-'d'`, which converts it to the next letter of the alphabet. The modified constant, now single-precision, is written out again. There follow a series of names which must be respelled to remove their initial `d`. By using the array `yytext` the same action suffices for all the names (only a sample of a rather long list is given here).



```

{d}{s}{i}{n} |
{d}{c}{o}{s} |
{d}{s}{q}{r}{t} |
{d}{a}{t}{a}{n} |
...
{d}{f}{l}{o}{a}{t} printf("%s",yytext+1);

```

Another list of names must have initial *d* changed to initial *a*:

```

{d}{l}{o}{g} |
{d}{l}{o}{g}10 |
{d}{m}{i}{n}1 |
{d}{m}{a}{x}1 |
 {
yytext[0] = + 'a' - 'd';
ECHO;
}

```

And one routine must have initial *d* changed to initial *r*:

```
{d}l{m}{a}{c}{h} {yytext[0] = + 'r' - 'd';
```

To avoid such names as *dsinx* being detected as instances of *dsin*, some final rules pick up longer words as identifiers and copy some surviving characters:

```

[A-Za-z][A-Za-z0-9]* |
[0-9]+ |
\n |
 {
 ECHO;
 }

```

Note that this program is not complete; it does not deal with the spacing problems in Fortran or with the use of keywords as identifiers.

## 10 Left Context Sensitivity.

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The *^* operator, for example, is a prior context operator, recognizing immediately preceding left context just as *\$* recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, the use of *start conditions* on rules, and the possibility of making multiple lexical analyzers all run together. In each case, there are rules which recognize the need to change the environment in which the following input text

is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line which began with the letter *a*, changing *magic* to *second* on every line which began with the letter *b*, and changing *magic* to *third* on every line which began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```

int flag;
%%
^a {flag = 'a'; ECHO;}
^b {flag = 'b'; ECHO;}
^c {flag = 'c'; ECHO;}
\n {flag = 0; ECHO;}
magic {
 switch (flag)
 {
 case 'a': printf("first"); break;
 case 'b': printf("second"); break;
 case 'c': printf("third"); break;
 default: ECHO; break;
 }
}

```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%Start name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with the *<>* brackets:

```
<name1>expression
```

is a rule which is only recognized when Lex is in the start condition *name1*. To enter a start condition, execute the action statement

```
BEGIN name1;
```

which changes the start condition to *name1*. To resume the normal state,

BEGIN 0;

resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions:

<name1,name2,name3>

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can be written:

```
%START AA BB CC
%%
^a {ECHO; BEGIN AA;}
^b {ECHO; BEGIN BB;}
^c {ECHO; BEGIN CC;}
\n {ECHO; BEGIN 0;}
<AA>magic printf("first");
<BB>magic printf("second");
<CC>magic printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

### 11 Character Set.

The programs generated by Lex handle character I/O only through the routines *input*, *output*, and *unput*. Thus the character representation provided in these routines is accepted by Lex and employed to return values in *yytext*. For internal use a character is represented as a small integer which, if the standard library is used, has a value equal to the integer value of the bit pattern representing the character on the host computer. In C, the I/O routines are assumed to deal directly in this representation. In Ratfor, it is anticipated that many users will prefer left-adjusted rather than right-adjusted characters; thus the routine *lexshf* is called to change the representation delivered by *input* into a right-adjusted integer. If the user changes the I/O library, the routine *lexshf* should also be changed to a compatible version. The Ratfor library I/O system is arranged to represent the letter *a* as in the Fortran value *1Ha* while in C the letter *a* is represented as the character constant 'a'. If this interpretation is changed, by providing I/O routines which translate the characters, Lex must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only "%T". The table contains lines of the form

{integer} {character string}

which indicate the value associated with each character. Thus the next example maps the lower and upper case letters together into the integers 1 through 26, newline into 27, + and - into 28 and 29, and the digits into 30 through 39. Note the escape for newline. If a table is supplied, every character that is to appear either in the

%T	
1	Aa
2	Bb
...	
26	Zz
27	\n
28	+
29	-
30	0
31	1
...	
39	9
%T	

Sample character table.

rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a bigger number than the size of the hardware character set.

It is not likely that C users will wish to use the character table feature; but for Fortran portability it may be essential.

Although the contents of the Lex Ratfor library routines for input and output run almost unmodified on UNIX, GCOS, and OS/370, they are not really machine independent, and would not work with CDC or Burroughs Fortran compilers. The user is of course welcome to replace *input*, *output*, *unput* and *lexshf* but to replace them by completely portable Fortran routines is likely to cause a substantial decrease in the speed of Lex Ratfor programs. A simple way to produce portable routines would be to leave *input* and *output* as routines that read with 80A1 format, but replace *lexshf* by a table lookup routine.

### 12 Summary of Source Format.

The general form of a Lex source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of

- 1) Definitions, in the form "name space translation".
- 2) Included code, in the form "space code".
- 3) Included code, in the form

```
%{
code
}%
```

- 4) Start conditions, given in the form

%S name1 name2 ...

- 5) Character set tables, in the form

```
%T
number space character-string
...
%T
```

- 6) A language specifier, which must also precede any rules or included code, in the form "%C" for C or "%R" for Ratfor.
- 7) Changes to internal array sizes, in the form

%x *nnn*

where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form "expression action" where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x\$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	m through n occurrences of x

### 13 Caveats and Bugs.

There are pathological expressions which produce exponential growth of the tables when converted to deterministic machines; fortunately, they are rare.

REJECT does not rescan the input; instead it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, the user must not have used *unput* to change the characters forthcoming from the input stream. This is the only restriction on the user's ability to manipulate the not-yet-processed input.

TSO Lex is an older version. Among the non-supported features are REJECT, start conditions, or variable length trailing context. And any significant Lex source is too big for the IBM C compiler when translated.

### 14 Acknowledgments.

As should be obvious from the above, the outside of Lex is patterned on Yacc and the inside on Aho's string matching routines. Therefore, both S. C. Johnson and A. V. Aho are really originators of much of Lex, as well as debuggers of it. Many thanks are due to both.

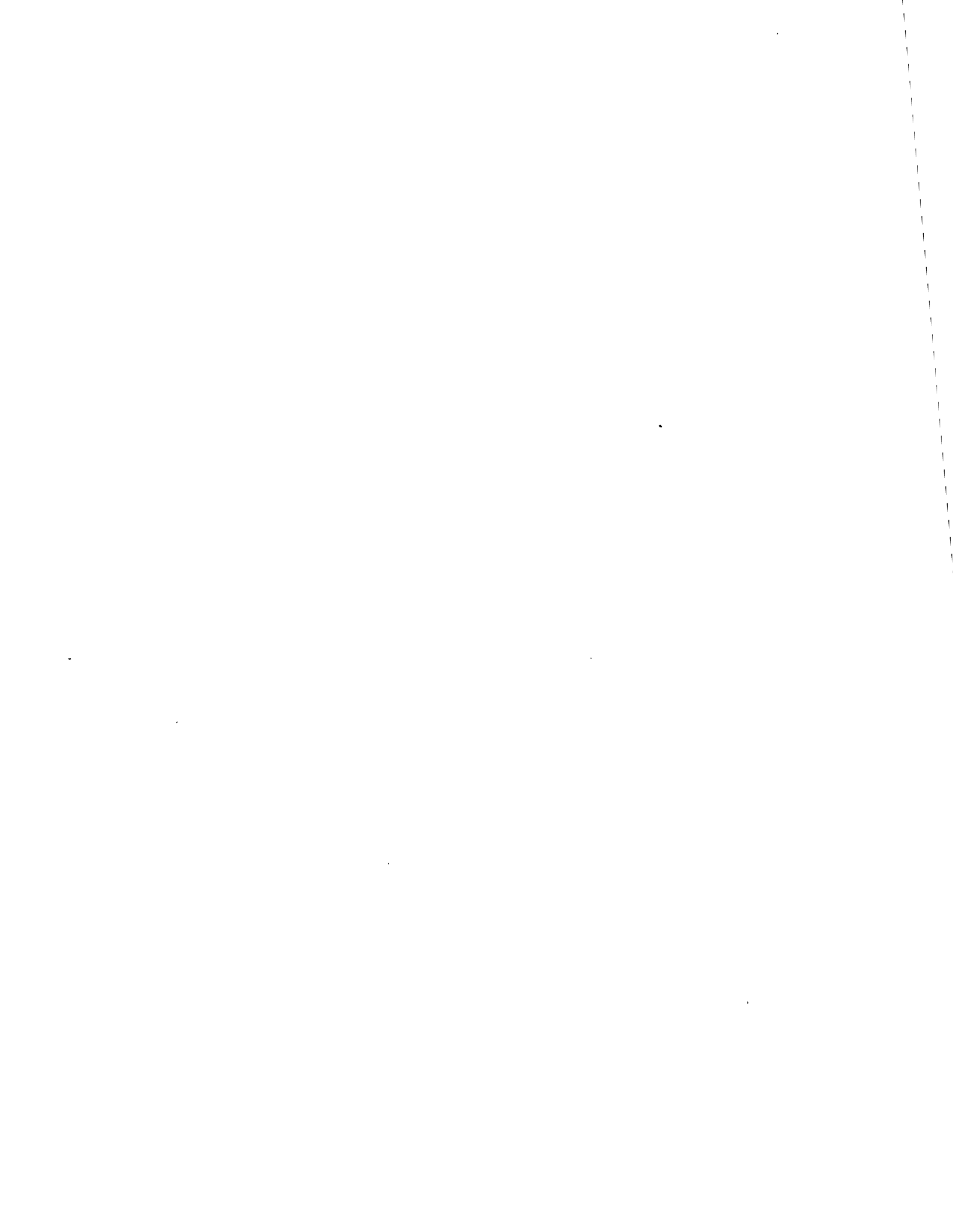
The code of the current version of Lex was designed, written, and debugged by Eric Schmidt.

### 15 References.

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, N. J. (1978).
2. B. W. Kernighan, *Ratfor: A Preprocessor for a Rational Fortran*, Software - Practice and Experience, 5, pp. 395-496 (1975).
3. S. C. Johnson, *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
4. A. V. Aho and M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Comm. ACM 18, 333-340 (1975).
5. B. W. Kernighan, D. M. Ritchie and K. L. Thompson, *QED Text Editor*, Computing Science Technical Report No. 5, 1972, Bell Laboratories, Murray Hill, NJ 07974.
6. D. M. Ritchie, private communication. See also M. E. Lesk, *The Portable C Library*, Computing Science Technical Report No. 31, Bell Laboratories, Murray Hill, NJ 07974.







## DC — An Interactive Desk Calculator

*Robert Morris*

*Lorinda Cherry*

Bell Laboratories  
Murray Hill, New Jersey 07974

### ABSTRACT

DC is an interactive desk calculator program implemented on the UNIX† time-sharing system to do arbitrary-precision integer arithmetic. It has provision for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated is limited only by available core storage. On typical implementations of UNIX, the size of numbers that can be handled varies from several hundred digits on the smallest systems to several thousand on the largest.

November 15, 1978

---

†UNIX is a Trademark of Bell Laboratories.

## DC — An Interactive Desk Calculator

*Robert Morris*

*Lorinda Cherry*

Bell Laboratories  
Murray Hill, New Jersey 07974

DC is an arbitrary precision arithmetic package implemented on the UNIX<sup>†</sup> time-sharing system in the form of an interactive desk calculator. It works like a stacking calculator using reverse Polish notation. Ordinarily DC operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained.

A language called BC [1] has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles output which is interpreted by DC. Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into DC are put on a push-down stack. DC commands work by taking the top number or two off the stack, performing the desired operation, and pushing the result on the stack. If an argument is given, input is taken from that file until its end, then from the standard input.

### SYNOPTIC DESCRIPTION

Here we describe the DC commands that are intended for use by people. The additional commands that are intended to be invoked by compiled output are described in the detailed description.

Any number of commands are permitted on a line. Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

The following constructions are recognized:

#### number

The value of the number is pushed onto the main stack. A number is an unbroken string of the digits 0-9 and the capital letters A-F which are treated as digits with values 10-15 respectively. The number may be preceded by an underscore to input a negative number. Numbers may contain decimal points.

+ - \* % ^

The top two values on the stack are added (+), subtracted (-), multiplied (\*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. The result of a division is an integer truncated toward zero. See the detailed description below for the treatment of numbers with decimal points. An exponent must not have any digits after the decimal point.

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.



**sx**

The top of the main stack is popped and stored into a register named *x*, where *x* may be any character. If the *s* is capitalized, *x* is treated as a stack and the value is pushed onto it. Any character, even blank or new-line, is a valid register name.

**lx**

The value in register *x* is pushed onto the stack. The register *x* is not altered. If the *l* is capitalized, register *x* is treated as a stack and its top value is popped onto the main stack.

All registers start with empty value which is treated as a zero by the command *l* and is treated as an error by the command *L*.

**d**

The top value on the stack is duplicated.

**p**

The top value on the stack is printed. The top value remains unchanged.

**f**

All values on the stack and in registers are printed.

**x**

treats the top element of the stack as a character string, removes it from the stack, and executes it as a string of DC commands.

**[ ... ]**

puts the bracketed character string onto the top of the stack.

**q**

exits the program. If executing a string, the recursion level is popped by two. If *q* is capitalized, the top value on the stack is popped and the string execution level is popped by that value.

**<x >x =x !<x !>x !=x**

The top two elements of the stack are popped and compared. Register *x* is executed if they obey the stated relation. Exclamation point is negation.

**v**

replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer. For the treatment of numbers with decimal points, see the detailed description below.

**!**

interprets the rest of the line as a UNIX command. Control returns to DC when the UNIX command terminates.

**c**

All values on the stack are popped; the stack becomes empty.

i

The top value on the stack is popped and used as the number radix for further input. If i is capitalized, the value of the input base is pushed onto the stack. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

o

The top value on the stack is popped and used as the number radix for further output. If o is capitalized, the value of the output base is pushed onto the stack.

k

The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100. If k is capitalized, the value of the scale factor is pushed onto the stack.

z

The value of the stack level is pushed onto the stack.

?

A line of input is taken from the input source (usually the console) and executed.

## DETAILED DESCRIPTION

### Internal Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string. For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0-99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100's complement notation, which is analogous to two's complement notation for binary numbers. The high order digit of a negative number is always -1 and all other digits are in the range 0-99. The digit preceding the high order -1 digit is never a 99. The representation of -157 is 43,98,-1. We shall call this the canonical form of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when that is convenient, as it sometimes is.

An additional byte is stored with each number beyond the high order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is 1,*3* where the scale has been italicized to emphasize the fact that it is not the high order digit. The value of this extra byte is called the **scale factor** of the number.

### The Allocator

DC uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is done through the allocator. Associated with each string in the allocator is a four-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and DC is done via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers. Requests for strings are made by size. The size of the string actually supplied is the next higher power of 2. When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size. Left-over strings are put on the free list. If there are no larger strings, the allocator tries to coalesce smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long. This is an implementation of the 'buddy system' of allocation described in [2].

Failing to find a string of the proper length after coalescing, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in DC. If at any time in the process of trying to allocate a string, the allocator runs out of headers, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward-spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls. The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end-of-string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

### **Internal Arithmetic**

All arithmetic operations are done on integers. The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine. For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called *scale* plays a part in the results of most arithmetic operations. *scale* is the bound on the number of decimal places retained in arithmetic computations. *scale* may be set to the number on the top of the stack truncated to an integer with the *k* command. *K* may be used to push the value of *scale* on the stack. *scale* must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations will include the exact effect of *scale* on the computations.

### **Addition and Subtraction**

The scales of the two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale. The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

Finally, the addition is performed digit by digit from the low order end of the number. The carries are propagated in the usual way. The resulting number is brought into canonical form, which may require stripping of leading zeros, or for negative numbers replacing the high-order configuration 99, -1 by the digit -1. In any case, digits which are not in the range 0-99 must be brought into that range, propagating any carries or borrows that result.

### **Multiplication**

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly mimics the hand method of multiplying. The first number is multiplied by each digit of the second number, beginning with its low order digit. The intermediate products are accumulated into a partial sum which becomes the final product. The product is put into the canonical form and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register scale and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

### **Division**

The scales are removed from the two operands. Zeros are appended or digits removed from the dividend to make the scale of the result of the integer division equal to the internal quantity scale. The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. It may turn out be one unit too low, but if it is, the next trial quotient will be larger than 99 and this will be adjusted at the end of the process. The trial digit is multiplied by the divisor and the result subtracted from the dividend and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor. At the end, the digits of the quotient are put into the canonical form, with propagation of carry as needed. The sign is set from the sign of the operands.

### **Remainder**

The division routine is called and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process. Since division truncates toward zero, remainders have the same sign as the dividend. The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

### **Square Root**

The scale is stripped from the operand. Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity scale and the scale of the operand.

The method used to compute  $\text{sqrt}(y)$  is Newton's method with successive approximations by the rule

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{y}{x_n} \right)$$

The initial guess is found by taking the integer square root of the top two digits.

### **Exponentiation**

Only exponents with zero scale factor are handled. If the exponent is zero, then the result is 1. If the exponent is negative, then it is made positive and the base is divided into one. The scale of the base is removed.

The integer exponent is viewed as a binary number. The base is repeatedly squared and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent. Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

### Input Conversion and Base

Numbers are converted to the internal representation as they are read in. The scale stored with a number is simply the number of fractional digits input. Negative numbers are indicated by preceding the number with a `_`. The hexadecimal digits A–F correspond to the numbers 10–15 regardless of input base. The `i` command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input. The input base is initialized to 10 but may, for example be changed to 8 or 16 to do octal or hexadecimal to decimal conversions. The command `I` will push the value of the input base on the stack.

### Output Commands

The command `p` causes the top of the stack to be printed. It does not remove the top of the stack. All of the stack and internal registers can be output by typing the command `f`. The `o` command can be used to change the output base. This command uses the top of the stack, truncated to an integer as the base for all further output. The output base is initialized to 10. It will work correctly for any base. The command `O` pushes the value of the output base on the stack.

### Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output; they have no effect on arithmetic computations. Large numbers are output with 70 characters per line; a `\` indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 can be used for decimal-octal or decimal-hexadecimal conversions.

### Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands `s` and `l`. The command `sx` pops the top of the stack and stores the result in register `x`. `x` can be any character. `lx` puts the contents of register `x` on the top of the stack. The `l` command has no effect on the contents of register `x`. The `s` command, however, is destructive.

### Stack Commands

The command `c` clears the stack. The command `d` pushes a duplicate of the number on the top of the stack on the stack. The command `z` pushes the stack size on the stack. The command `X` replaces the number on the top of the stack with its scale factor. The command `Z` replaces the top of the stack with its length.

### Subroutine Definitions and Calls

Enclosing a string in `||` pushes the ascii string on the stack. The `q` command quits or in executing a string, pops the recursion levels by two.

### Internal Registers – Programming DC

The load and store commands together with `||` to store strings, `x` to execute and the testing commands `'<'`, `'>'`, `'='`, `'!<'`, `'!>'`, `'!='` can be used to program DC. The `x` command assumes the top of the stack is an string of DC commands and executes it. The testing commands compare the top two elements on the stack and if the relation holds, execute the register that follows the relation. For example, to print the numbers 0-9,

```
[lip1+ si li10>a]sa
0si lax
```

### Push-Down Registers and Arrays

These commands were designed for use by a compiler, not by people. They involve push-down registers and arrays. In addition to the stack that commands work on, DC can be thought of as having individual stacks for each register. These registers are operated on by the commands S and L. Sx pushes the top value of the main stack onto the stack for the register x. Lx pops the stack for register x and puts the result on the main stack. The commands s and l also work on registers but not as push-down stacks. l doesn't effect the top of the register stack, and s destroys what was there before.

The commands to work on arrays are : and ;. ;x pops the stack and uses this value as an index into the array x. The next element on the stack is stored at this index in x. An index must be greater than or equal to 0 and less than 2048. ;x is the command to load the main stack from the array x. The value on the top of the stack is the index into the array x of the value to be loaded.

### Miscellaneous Commands

The command ! interprets the rest of the line as a UNIX command and passes it to UNIX to execute. One other compiler command is Q. This command uses the top of the stack as the number of levels of recursion to skip.

### DESIGN CHOICES

The real reason for the use of a dynamic storage allocator was that a general purpose program could be (and in fact has been) used for a variety of other tasks. The allocator has some value for input and for compiling (i.e. the bracket [...] commands) where it cannot be known in advance how long a string will be. The result was that at a modest cost in execution time, all considerations of string allocation and sizes of strings were removed from the remainder of the program and debugging was made easier. The allocation method used wastes approximately 25% of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5% in space, debugging was made a great deal easier and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all DC commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases was to provide an understandable means of proceeding after a change of base or scale when numbers had already been entered. An earlier implementation which had global notions of scale and base did not work out well. If the value of scale were to be interpreted in the current input or output base, then a change of base or scale in the midst of a computation would cause great confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations. The value of scale is not used for any essential purpose by any part of the program and it is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

The design rationale for the choices for the scales of the results of arithmetic were that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give him the result 5.017 without requiring him to unnecessarily specify his rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands and it seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user

asked for them by specifying a value for **scale**. Square root can be handled in just the same way as multiplication. The operation of division gives arbitrarily many decimal places and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement; no digits are thrown away.

#### **References**

- [1] L. L. Cherry, R. Morris, *BC — An Arbitrary Precision Desk-Calculator Language*.
- [2] K. C. Knowlton, *A Fast Storage Allocator*, Comm. ACM 8, pp. 623-625 (Oct. 1965).









## BC – An Arbitrary Precision Desk-Calculator Language

*Lorinda Cherry*

*Robert Morris*

Bell Laboratories  
Murray Hill, New Jersey 07974

### *ABSTRACT*

BC is a language and a compiler for doing arbitrary precision arithmetic on the PDP-11 under the UNIX<sup>†</sup> time-sharing system. The output of the compiler is interpreted and executed by a collection of routines which can input, output, and do arithmetic on indefinitely large integers and on scaled fixed-point numbers.

These routines are themselves based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

Two five hundred-digit numbers can be multiplied to give a thousand digit result in about ten seconds.

A small collection of library functions is also available, including sin, cos, arctan, log, exponential, and Bessel functions of integer order.

Some of the uses of this compiler are

- to do computation with large integers,
- to do computation accurate to many decimal places,
- conversion of numbers from one base to another base.

November 12, 1978

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

## BC – An Arbitrary Precision Desk-Calculator Language

*Lorinda Cherry*

*Robert Morris*

Bell Laboratories  
Murray Hill, New Jersey 07974

### Introduction

BC is a language and a compiler for doing arbitrary precision arithmetic on the UNIX<sup>†</sup> time-sharing system [1]. The compiler was written to make conveniently available a collection of routines (called DC [5]) which are capable of doing arithmetic on integers of arbitrary size. The compiler is by no means intended to provide a complete programming language. It is a minimal language facility.

There is a scaling provision that permits the use of decimal point notation. Provision is made for input and output in bases other than decimal. Numbers can be converted from decimal to octal by simply setting the output base to equal 8.

The actual limit on the number of digits that can be handled depends on the amount of storage available on the machine. Manipulation of numbers with many hundreds of digits is possible even on the smallest versions of UNIX.

The syntax of BC has been deliberately selected to agree substantially with the C language [2]. Those who are familiar with C will find few surprises in this language.

### Simple Computations with Integers

The simplest kind of statement is an arithmetic expression on a line by itself. For instance, if you type in the line:

```
142857 + 285714
```

the program responds immediately with the line

```
428571
```

The operators  $-$ ,  $*$ ,  $/$ ,  $\%$ , and  $^$  can also be used; they indicate subtraction, multiplication, division, remaindering, and exponentiation, respectively. Division of integers produces an integer result truncated toward zero. Division by zero produces an error comment.

Any term in an expression may be prefixed by a minus sign to indicate that it is to be negated (the 'unary' minus sign). The expression

```
7 + -3
```

is interpreted to mean that  $-3$  is to be added to 7.

More complex expressions with several operators and with parentheses are interpreted just as in Fortran, with  $^$  having the greatest binding power, then  $*$  and  $\%$  and  $/$ , and finally  $+$  and  $-$ . Contents of parentheses are evaluated before material outside the parentheses. Exponentiations are performed from right to left and the other operators from left to right. The two expressions

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

$a^b \cdot c$  and  $a^{(b \cdot c)}$

are equivalent, as are the two expressions

$a \cdot b \cdot c$  and  $(a \cdot b) \cdot c$

BC shares with Fortran and C the undesirable convention that

$a/b \cdot c$  is equivalent to  $(a/b) \cdot c$

Internal storage registers to hold numbers have single lower-case letter names. The value of an expression can be assigned to a register in the usual way. The statement

```
x = x + 3
```

has the effect of increasing by three the value of the contents of the register named x. When, as in this case, the outermost operator is an =, the assignment is performed but the result is not printed. Only 26 of these named storage registers are available.

There is a built-in square root function whose result is truncated to an integer (but see scaling below). The lines

```
x = sqrt(191)
x
```

produce the printed result

```
13
```

### Bases

There are special internal quantities, called 'ibase' and 'obase'. The contents of 'ibase', initially set to 10, determines the base used for interpreting numbers read in. For example, the lines

```
ibase = 8
11
```

will produce the output line

```
9
```

and you are all set up to do octal to decimal conversions. Beware, however of trying to change the input base back to decimal by typing

```
ibase = 10
```

Because the number 10 is interpreted as octal, this statement will have no effect. For those who deal in hexadecimal notation, the characters A–F are permitted in numbers (no matter what base is in effect) and are interpreted as digits having values 10–15 respectively. The statement

```
ibase = A
```

will change you back to decimal input base no matter what the current input base is. Negative and large positive input bases are permitted but useless. No mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The contents of 'obase', initially set to 10, are used as the base for output numbers. The lines

```
obase = 16
1000
```

will produce the output line

### 3E8

which is to be interpreted as a 3-digit hexadecimal number. Very large output bases are permitted, and they are sometimes useful. For example, large numbers can be output in groups of five digits by setting 'obase' to 100000. Strange (i.e. 1, 0, or negative) output bases are handled appropriately.

Very large numbers are split across lines with 70 characters per line. Lines which are continued end with \. Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Non-decimal output conversion of a one hundred digit number takes about three seconds.

It is best to remember that 'ibase' and 'obase' have no effect whatever on the course of internal computation or on the evaluation of expressions, but only affect input and output conversion, respectively.

### Scaling

A third special internal quantity called 'scale' is used to determine the scale of calculated quantities. Numbers may have up to 99 decimal digits after the decimal point. This fractional part is retained in further computations. We refer to the number of digits after the decimal point of a number as its scale.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules. For addition and subtraction, the scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result. For multiplications, the scale of the result is never less than the maximum of the two scales of the operands, never more than the sum of the scales of the operands and, subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity 'scale'. The scale of a quotient is the contents of the internal quantity 'scale'. The scale of a remainder is the sum of the scales of the quotient and the divisor. The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer. The scale of a square root is set to the maximum of the scale of the argument and the contents of 'scale'.

All of the internal operations are actually carried out in terms of integers, with digits being discarded when necessary. In every case where digits are discarded, truncation and not rounding is performed.

The contents of 'scale' must be no greater than 99 and no less than 0. It is initially set to 0. In case you need more than 99 fraction digits, you may arrange your own scaling.

The internal quantities 'scale', 'ibase', and 'obase' can be used in expressions just like other variables. The line

```
scale = scale + 1
```

increases the value of 'scale' by one, and the line

```
scale
```

causes the current value of 'scale' to be printed.

The value of 'scale' retains its meaning as a number of decimal digits to be retained in internal computation even when 'ibase' or 'obase' are not equal to 10. The internal computations (which are still conducted in decimal, regardless of the bases) are performed to the specified number of decimal digits, never hexadecimal or octal or any other kind of digits.

### Functions

The name of a function is a single lower-case letter. Function names are permitted to collide with simple variable names. Twenty-six different defined functions are permitted in addition to the twenty-six variable names. The line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements, which make up the body of the function, ending with a right brace `}`. Return of control from a function occurs when a return statement is executed or when the end of the function is reached. The return statement can take either of the two forms

```
return
return(x)
```

In the first case, the value of the function is 0, and in the second, the value of the expression in parentheses.

Variables used in the function can be declared as automatic by a statement of the form

```
auto x,y,z
```

There can be only one 'auto' statement in a function and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return. The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected. The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. An example of a function definition is

```
define a(x,y){
 auto z
 z = x*y
 return(z)
}
```

The value of this function, when called, will be the product of its two arguments.

A function is called by the appearance of its name followed by a string of arguments enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using parentheses with nothing between them: `b()`.

If the function *a* above has been defined, then the line

```
a(7,3.14)
```

would cause the result 21.98 to be printed and the line

```
x = a(a(3,4),5)
```

would cause the value of *x* to become 60.

### Subscripted Variables

A single lower-case letter variable name followed by an expression in brackets is called a subscripted variable (an array element). The variable name is called the array name and the expression in brackets is called the subscript. Only one-dimensional arrays are permitted. The names of arrays are permitted to collide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to zero and less than or equal to 2047.

Subscripted variables may be freely used in expressions, in function calls, and in return statements.

An array name may be used as an argument to a function, or may be declared as automatic in a function definition by the use of empty brackets:

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function, and thrown away on exit from the function. Array names which refer to whole arrays cannot be used in any other contexts.

### Control Statements

The 'if', the 'while', and the 'for' statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a statement or a compound statement consisting of a collection of statements enclosed in braces. They are written in the following way

```
if(relation) statement
while(relation) statement
for(expression1; relation; expression2) statement
```

or

```
if(relation) {statements}
while(relation) {statements}
for(expression1; relation; expression2) {statements}
```

A relation in one of the control statements is an expression of the form

$x > y$

where two expressions are related by one of the six relational operators  $<$ ,  $>$ ,  $< =$ ,  $> =$ ,  $= =$ , or  $! =$ . The relation  $= =$  stands for 'equal to' and  $! =$  stands for 'not equal to'. The meaning of the remaining relational operators is clear.

BEWARE of using  $=$  instead of  $= =$  in a relational. Unfortunately, both of them are legal, so you will not get a diagnostic message, but  $=$  really will not do a comparison.

The 'if' statement causes execution of its range if and only if the relation is true. Then control passes to the next statement in sequence.

The 'while' statement causes execution of its range repeatedly as long as the relation is true. The relation is tested before each execution of its range and if the relation is false, control passes to the next statement beyond the range of the while.

The 'for' statement begins by executing 'expression1'. Then the relation is tested and, if true, the statements in the range of the 'for' are executed. Then 'expression2' is executed. The relation is tested, and so on. The typical use of the 'for' statement is for a controlled iteration, as in the statement

```
for(i=1; i<=10; i=i+1) i
```

which will print the integers from 1 to 10. Here are some examples of the use of the control statements.

```
define f(n){
auto i, x
x=1
for(i=1; i<=n; i=i+1) x=x*i
return(x)
}
```

The line

$f(a)$



will print  $a$  factorial if  $a$  is a positive integer. Here is the definition of a function which will compute values of the binomial coefficient ( $m$  and  $n$  are assumed to be positive integers).

```
define b(n,m){
 auto x, j
 x = 1
 for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
 return(x)
}
```

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
 auto a, b, c, d, n
 a = 1
 b = 1
 c = 1
 d = 0
 n = 1
 while(1==1){
 a = a*x
 b = b*n
 c = c + a/b
 n = n + 1
 if(c==d) return(c)
 d = c
 }
}
```

### Some Details

There are some language features that every user should know about even if he will not use them.

Normally statements are typed one to a line. It is also permissible to type several statements on a line separated by semicolons.

If an assignment statement is parenthesized, it then has a value and it can be used anywhere that an expression can. For example, the line

```
(x=y+17)
```

not only makes the indicated assignment, but also prints the resulting value.

Here is an example of a use of the value of an assignment statement even when it is not parenthesized.

```
x = a[i=i+1]
```

causes a value to be assigned to  $x$  and also increments  $i$  before it is used as a subscript.

The following constructs work in BC in exactly the same manner as they do in the C language. Consult the appendix or the C manuals [2] for their exact workings.

<code>x=y=z</code> is the same as	<code>x=(y=z)</code>
<code>x =+ y</code>	<code>x = x+y</code>
<code>x =- y</code>	<code>x = x-y</code>
<code>x =* y</code>	<code>x = x*y</code>
<code>x =/ y</code>	<code>x = x/y</code>
<code>x =% y</code>	<code>x = x%y</code>
<code>x =^ y</code>	<code>x = x^y</code>
<code>x++</code>	<code>(x=x+1)-1</code>
<code>x--</code>	<code>(x=x-1)+1</code>
<code>++x</code>	<code>x = x+1</code>
<code>--x</code>	<code>x = x-1</code>

Even if you don't intend to use the constructs, if you type one inadvertently, something correct but unexpected may happen.

**WARNING!** In some of these constructions, spaces are significant. There is a real difference between `x=-y` and `x= -y`. The first replaces `x` by `x-y` and the second by `-y`.

### Three Important Things

1. To exit a BC program, type 'quit'.
2. There is a comment convention identical to that of C and of PL/I. Comments begin with '/' and end with '\*/'.
3. There is a library of math functions which may be obtained by typing at command level

```
bc -l
```

This command will load a set of library functions which, at the time of writing, consists of sine (named 's'), cosine ('c'), arctangent ('a'), natural logarithm ('l'), exponential ('e') and Bessel functions of integer order ('j(n,x)'). Doubtless more functions will be added in time. The library sets the scale to 20. You can reset it to something else if you like. The design of these mathematical library routines is discussed elsewhere [3].

If you type

```
bc file ...
```

BC will read and execute the named file or files before accepting commands from the keyboard. In this way, you may load your favorite programs and function definitions.

### Acknowledgement

The compiler is written in YACC [4]; its original version was written by S. C. Johnson.

### References

- [1] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual*. Bell Laboratories, 1978.
- [2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice-Hall, 1978.
- [3] R. Morris, *A Library of Reference Standard Mathematical Subroutines*, Bell Laboratories internal memorandum, 1975.
- [4] S. C. Johnson, *YACC - Yet Another Compiler-Compiler*. Bell Laboratories Computing Science Technical Report #32, 1978.
- [5] R. Morris and L. L. Cherry, *DC - An Interactive Desk Calculator*.

## Appendix

### 1. Notation

In the following pages syntactic categories are in *italics*; literals are in **bold**; material in brackets [] is optional.

### 2. Tokens

Tokens consist of keywords, identifiers, constants, operators, and separators. Token separators may be blanks, tabs or comments. Newline characters or semicolons separate statements.

#### 2.1. Comments

Comments are introduced by the characters /\* and terminated by \*/.

#### 2.2. Identifiers

There are three kinds of identifiers — ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets, possibly enclosing an expression describing a subscript. Arrays are singly dimensioned and may contain up to 2048 elements. Indexing begins at zero so an array may be indexed from 0 to 2047. Subscripts are truncated to integers. Function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict: a program can have a variable named **x**, an array named **x** and a function named **x**, all of which are separate and distinct.

#### 2.3. Keywords

The following are reserved keywords:

**ibase** **if**  
**obase** **break**  
**scale** **define**  
**sqrt** **auto**  
**length** **return**  
**while** **quit**  
**for**

#### 2.4. Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits **A–F** are also recognized as digits with values 10–15, respectively.

### 3. Expressions

The value of an expression is printed unless the main operator is an assignment. Precedence is the same as the order of presentation here, with highest appearing first. Left or right associativity, where applicable, is discussed with each operator.

### 3.1. Primitive expressions

#### 3.1.1. Named expressions

Named expressions are places where values are stored. Simply stated, named expressions are legal on the left side of an assignment. The value of a named expression is the value stored in the place named.

##### 3.1.1.1. *identifiers*

Simple identifiers are named expressions. They have an initial value of zero.

##### 3.1.1.2. *array-name [expression]*

Array elements are named expressions. They have an initial value of zero.

##### 3.1.1.3. *scale, ibase and obase*

The internal registers *scale*, *ibase* and *obase* are all named expressions. *scale* is the number of digits after the decimal point to be retained in arithmetic operations. *scale* has an initial value of zero. *ibase* and *obase* are the input and output number radix respectively. Both *ibase* and *obase* have initial values of 10.

#### 3.1.2. Function calls

##### 3.1.2.1. *function-name ([expression [, expression ... ]])*

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a return statement, the value of the function is the value of the expression in the parentheses of the return statement or is zero if no expression is provided or if there is no return statement.

##### 3.1.2.2. *sqrt (expression)*

The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of *scale*, whichever is larger.

##### 3.1.2.3. *length (expression)*

The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

##### 3.1.2.4. *scale (expression)*

The result is the scale of the expression. The scale of the result is zero.

#### 3.1.3. Constants

Constants are primitive expressions.

#### 3.1.4. Parentheses

An expression surrounded by parentheses is a primitive expression. The parentheses are used to alter the normal precedence.

### 3.2. Unary operators

The unary operators bind right to left.

#### 3.2.1. $-$ *expression*

The result is the negative of the expression.

#### 3.2.2. $++$ *named-expression*

The named expression is incremented by one. The result is the value of the named expression after incrementing.

#### 3.2.3. $--$ *named-expression*

The named expression is decremented by one. The result is the value of the named expression after decrementing.

#### 3.2.4. *named-expression* $++$

The named expression is incremented by one. The result is the value of the named expression before incrementing.

#### 3.2.5. *named-expression* $--$

The named expression is decremented by one. The result is the value of the named expression before decrementing.

### 3.3. Exponentiation operator

The exponentiation operator binds right to left.

#### 3.3.1. *expression* $^$ *expression*

The result is the first expression raised to the power of the second expression. The second expression must be an integer. If  $a$  is the scale of the left expression and  $b$  is the absolute value of the right expression, then the scale of the result is:

$$\min(a \times b, \text{scale}, a)$$

### 3.4. Multiplicative operators

The operators  $*$ ,  $/$ ,  $\%$  bind left to right.

#### 3.4.1. *expression* $*$ *expression*

The result is the product of the two expressions. If  $a$  and  $b$  are the scales of the two expressions, then the scale of the result is:

$$\min(a + b, \text{scale}, a, b)$$

#### 3.4.2. *expression* $/$ *expression*

The result is the quotient of the two expressions. The scale of the result is the value of **scale**.

#### 3.4.3. *expression* $\%$ *expression*

The  $\%$  operator produces the remainder of the division of the two expressions. More precisely,  $a\%b$  is  $a - a/b * b$ .

The scale of the result is the sum of the scale of the divisor and the value of **scale**

### 3.5. Additive operators

The additive operators bind left to right.

#### 3.5.1. *expression + expression*

The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

#### 3.5.2. *expression - expression*

The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

### 3.6. assignment operators

The assignment operators bind right to left.

#### 3.6.1. *named-expression = expression*

This expression results in assigning the value of the expression on the right to the named expression on the left.

#### 3.6.2. *named-expression = + expression*

#### 3.6.3. *named-expression = - expression*

#### 3.6.4. *named-expression = \* expression*

#### 3.6.5. *named-expression = / expression*

#### 3.6.6. *named-expression = % expression*

#### 3.6.7. *named-expression = ^ expression*

The result of the above expressions is equivalent to "named expression = named expression OP expression", where OP is the operator after the = sign.

### 4. Relations

Unlike all other operators, the relational operators are only valid as the object of an **if**, **while**, or inside a **for** statement.

#### 4.1. *expression < expression*

#### 4.2. *expression > expression*

#### 4.3. *expression < = expression*

#### 4.4. *expression > = expression*

#### 4.5. *expression == expression*

#### 4.6. *expression != expression*

## 5. Storage classes

There are only two storage classes in BC, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as **auto** are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in BC do not work in exactly the same way as in either C or PL/I. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

## 6. Statements

Statements must be separated by semicolon or newline. Except where altered by control statements, execution is sequential.

### 6.1. Expression statements

When a statement is an expression, unless the main operator is an assignment, the value of the expression is printed, followed by a newline character.

### 6.2. Compound statements

Statements may be grouped together and used when one statement is expected by surrounding them with { }.

### 6.3. Quoted string statements

"any string"

This statement prints the string inside the quotes.

### 6.4. If statements

*if (relation) statement*

The substatement is executed if the relation is true.

### 6.5. While statements

*while (relation) statement*

The statement is executed while the relation is true. The test occurs before each execution of the statement.

### 6.6. For statements

*for (expression; relation; expression) statement*

The for statement is the same as

```
first-expression
while (relation) {
 statement
 last-expression
}
```

All three expressions must be present.

### 6.7. Break statements

**break**

**break** causes termination of a **for** or **while** statement.

### 6.8. Auto statements

**auto** *identifier* [ *,identifier* ]

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The **auto** statement must be the first statement in a function definition.

### 6.9. Define statements

**define**( [*parameter* [ *,parameter...* ] ] ) {  
    *statements* }

The **define** statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

### 6.10. Return statements

**return**

**return**( *expression* )

The **return** statement causes termination of a function, popping of its auto variables, and specifies the result of the function. The first form is equivalent to **return(0)**. The result of the function is the result of the expression in parentheses.

### 6.11. Quit

The **quit** statement stops execution of a BC program and returns control to UNIX when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.







# UNIX† Assembler Reference Manual

*Dennis M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 0. Introduction

This document describes the usage and input syntax of the UNIX PDP-11 assembler *as*. The details of the PDP-11 are not described.

The input syntax of the UNIX assembler is generally similar to that of the DEC assembler PAL-11R, although its internal workings and output format are unrelated. It may be useful to read the publication DEC-11-ASDB-D, which describes PAL-11R, although naturally one must use care in assuming that its rules apply to *as*.

*As* is a rather ordinary assembler without macro capabilities. It produces an output file that contains relocation information and a complete symbol table; thus the output is acceptable to the UNIX link-editor *ld*, which may be used to combine the outputs of several assembler runs and to obtain object programs from libraries. The output format has been designed so that if a program contains no unresolved references to external symbols, it is executable without further processing.

## 1. Usage

*as* is used as follows:

```
as [-u] [-o output] file, ...
```

If the optional “-u” argument is given, all undefined symbols in the current assembly will be made undefined-external. See the `.globl` directive below.

The other arguments name files which are concatenated and assembled. Thus programs may be written in several pieces and assembled together.

The output of the assembler is by default placed on the file *a.out* in the current directory; the “-o” flag causes the output to be placed on the named file. If there were no unresolved external references, and no errors detected, the output file is marked executable; otherwise, if it is produced at all, it is made non-executable.

## 2. Lexical conventions

Assembler tokens include identifiers (alternatively, “symbols” or “names”), temporary symbols, constants, and operators.

### 2.1 Identifiers

An identifier consists of a sequence of alphanumeric characters (including period “.”, underscore “\_”, and tilde “~” as alphanumeric) of which the first may not be numeric. Only the first eight characters are significant. When a name begins with a tilde, the tilde is discarded and that occurrence of the identifier generates a unique entry in the symbol table which can match no other occurrence of the identifier. This feature is used by the C compiler to place

---

† UNIX is a Trademark of Bell Laboratories.

names of local variables in the output symbol table without having to worry about making them unique.

## 2.2 Temporary symbols

A temporary symbol consists of a digit followed by "f" or "b". Temporary symbols are discussed fully in §5.1.

## 2.3 Constants

An octal constant consists of a sequence of digits; "8" and "9" are taken to have octal value 10 and 11. The constant is truncated to 16 bits and interpreted in two's complement notation.

A decimal constant consists of a sequence of digits terminated by a decimal point ".". The magnitude of the constant should be representable in 15 bits; i.e., be less than 32,768.

A single-character constant consists of a single quote "'" followed by an ASCII character not a new-line. Certain dual-character escape sequences are acceptable in place of the ASCII character to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the given character in the least significant byte of the word and is null-padded on the left.

A double-character constant consists of a double quote "\"" followed by a pair of ASCII characters not including new-line. Certain dual-character escape sequences are acceptable in place of either of the ASCII characters to represent new-line and other non-graphics (see *String statements*, §5.5). The constant's value has the code for the first given character in the least significant byte and that for the second character in the most significant byte.

## 2.4 Operators

There are several single- and double-character operators; see §6.

## 2.5 Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens (except character constants). A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

## 2.6 Comments

The character "/" introduces a comment, which extends through the end of the line on which it appears. Comments are ignored by the assembler.

## 3. Segments

Assembled code and data fall into three segments: the text segment, the data segment, and the bss segment. The text segment is the one in which the assembler begins, and it is the one into which instructions are typically placed. The UNIX system will, if desired, enforce the purity of the text segment of programs by trapping write operations into it. Object programs produced by the assembler must be processed by the link-editor *ld* (using its "-n" flag) if the text segment is to be write-protected. A single copy of the text segment is shared among all processes executing such a program.

The data segment is available for placing data or instructions which will be modified during execution. Anything which may go in the text segment may be put into the data segment. In programs with write-protected, sharable text segments, data segment contains the initialized but variable parts of a program. If the text segment is not pure, the data segment begins immediately after the text segment; if the text segment is pure, the data segment begins at the lowest 8K byte boundary after the text segment.

The bss segment may not contain any explicitly initialized code or data. The length of the

bss segment (like that of text or data) is determined by the high-water mark of the location counter within it. The bss segment is actually an extension of the data segment and begins immediately after it. At the start of execution of a program, the bss segment is set to 0. Typically the bss segment is set up by statements exemplified by

```
lab: . = .+10
```

The advantage in using the bss segment for storage that starts off empty is that the initialization information need not be stored in the output file. See also *Location counter* and *Assignment statements* below.

#### 4. The location counter

One special symbol, “.”, is the location counter. Its value at any time is the offset within the appropriate segment of the start of the statement in which it appears. The location counter may be assigned to, with the restriction that the current segment may not change; furthermore, the value of “.” may not decrease. If the effect of the assignment is to increase the value of “.”, the required number of null bytes are generated (but see *Segments* above).

#### 5. Statements

A source program is composed of a sequence of *statements*. Statements are separated either by new-lines or by semicolons. There are five kinds of statements: null statements, expression statements, assignment statements, string statements, and keyword statements.

Any kind of statement may be preceded by one or more labels.

##### 5.1 Labels

There are two kinds of label: name labels and numeric labels. A name label consists of a name followed by a colon (:). The effect of a name label is to assign the current value and type of the location counter “.” to the name. An error is indicated in pass 1 if the name is already defined; an error is indicated in pass 2 if the “.” value assigned changes the definition of the label.

A numeric label consists of a digit 0 to 9 followed by a colon (:). Such a label serves to define temporary symbols of the form “nb” and “nf”, where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of “.” to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form “nf” refer to the first numeric label “n:” forward from the reference; “nb” symbols refer to the first “n:” label backward from the reference. This sort of temporary label was introduced by Knuth [*The Art of Computer Programming, Vol 1: Fundamental Algorithms*]. Such labels tend to conserve both the symbol table space of the assembler and the inventive powers of the programmer.

##### 5.2 Null statements

A null statement is an empty statement (which may, however, have labels). A null statement is ignored by the assembler. Common examples of null statements are empty lines or lines containing only a label.

##### 5.3 Expression statements

An expression statement consists of an arithmetic expression not beginning with a keyword. The assembler computes its (16-bit) value and places it in the output stream, together with the appropriate relocation bits.

#### 5.4 Assignment statements

An assignment statement consists of an identifier, an equals sign (=), and an expression. The value and type of the expression are assigned to the identifier. It is not required that the type or value be the same in pass 2 as in pass 1, nor is it an error to redefine any symbol by assignment.

Any external attribute of the expression is lost across an assignment. This means that it is not possible to declare a global symbol by assigning to it, and that it is impossible to define a symbol to be offset from a non-locally defined global symbol.

As mentioned, it is permissible to assign to the location counter ".". It is required, however, that the type of the expression assigned be of the same type as ".", and it is forbidden to decrease the value of ".". In practice, the most common assignment to "." has the form ". = . + n" for some number n: this has the effect of generating n null bytes.

#### 5.5 String statements

A string statement generates a sequence of bytes containing ASCII characters. A string statement consists of a left string quote "<" followed by a sequence of ASCII characters not including newline, followed by a right string quote ">". Any of the ASCII characters may be replaced by a two-character escape sequence to represent certain non-graphic characters, as follows:

\n	NL	(012)
\s	SP	(040)
\t	HT	(011)
\e	EOT	(004)
\0	NUL	(000)
\r	CR	(015)
\a	ACK	(006)
\p	PFX	(033)
\\	\	
\>	>	

The last two are included so that the escape character and the right string quote may be represented. The same escape sequences may also be used within single- and double-character constants (see §2.3 above).

#### 5.6 Keyword statements

Keyword statements are numerically the most common type, since most machine instructions are of this sort. A keyword statement begins with one of the many predefined keywords of the assembler; the syntax of the remainder depends on the keyword. All the keywords are listed below with the syntax they require.

### 6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, temporary symbols, operators, and brackets. Each expression has a type.

All operators in expressions are fundamentally binary in nature; if an operand is missing on the left, a 0 of absolute type is assumed. Arithmetic is two's complement and has 16 bits of precision. All operators have equal precedence, and expressions are evaluated strictly left to right except for the effect of brackets.

## 6.1 Expression operators

The operators are:

- (blank) when there is no operand between operands, the effect is exactly the same as if a “+” had appeared.
- + addition
- subtraction
- multiplication
- \ / division (note that plain “/” starts a comment)
- & bitwise and
- | bitwise or
- \> logical right shift
- \< logical left shift
- % modulo
- !  $a!b$  is  $a$  or (not  $b$ ); i.e., the or of the first operand and the one’s complement of the second; most common use is as a unary.
- ^ result has the value of first operand and the type of the second; most often used to define new machine instructions with syntax identical to existing instructions.

Expressions may be grouped by use of square brackets “[ ]”. (Round parentheses are reserved for address modes.)

## 6.2 Types

The assembler deals with a number of types of expressions. Most types are attached to keywords and used to select the routine which treats that keyword. The types likely to be met explicitly are:

### undefined

Upon first encounter, each symbol is undefined. It may become undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1, it is not (except that certain keywords require operands which are not undefined).

### undefined external

A symbol which is declared `.globl` but not defined in the current assembly is an undefined external. If such a symbol is declared, the link editor `ld` must be used to load the assembler’s output with another routine that defines the undefined reference.

**absolute** An absolute symbol is defined ultimately from a constant. Its value is unaffected by any possible future applications of the link-editor to the output file.

**text** The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor’s output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of “.” is text 0.

**data** The value of a data symbol is measured with respect to the origin of the data segment of a program. Like text symbols, the value of a data symbol may change during a subsequent link-editor run since previously loaded programs may have data segments. After the first `.data` statement, the value of “.” is data 0.

**bss** The value of a bss symbol is measured from the beginning of the bss segment of a program. Like text and data symbols, the value of a bss symbol may change during a subsequent link-editor run, since previously loaded programs may have bss segments. After the first `.bss` statement, the value of “.” is bss 0.

**external** absolute, text, data, or bss

symbols declared `.globl` but defined within an assembly as `absolute`, `text`, `data`, or `bss` symbols may be used exactly as if they were not declared `.globl`; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

**register**

The symbols

```
r0 ... r5
fr0 ... fr5
sp
pc
```

are predefined as register symbols. Either they or symbols defined from them must be used to refer to the six general-purpose, six floating-point, and the 2 special-purpose machine registers. The behavior of the floating register names is identical to that of the corresponding general register names: the former are provided as a mnemonic aid.

**other types**

Each keyword known to the assembler has a type which is used to select the routine which processes the associated keyword statement. The behavior of such symbols when not used as keywords is the same as if they were absolute.

### 6.3 Type propagation in expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. The rules involved are complex to state but were intended to be sensible and predictable. For purposes of expression evaluation the important types are

```
undefined
absolute
text
data
bss
undefined external
other
```

The combination rules are then: If one of the operands is undefined, the result is undefined. If both operands are absolute, the result is absolute. If an absolute is combined with one of the "other types" mentioned above, or with a register expression, the result has the register or other type. As a consequence, one can refer to `r3` as "`r0+3`". If two operands of "other type" are combined, the result has the numerically larger type. An "other type" combined with an explicitly discussed type other than absolute acts like an absolute.

Further rules applying to particular operators are:

- + If one operand is text-, data-, or bss-segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.
- If the first operand is a relocatable text-, data-, or bss-segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.
- ^ This operator follows no other rule than that the result has the value of the first operand and the type of the second.



others

It is illegal to apply these operators to any but absolute symbols.

## 7. Pseudo-operations

The keywords listed below introduce statements that generate data in unusual forms or influence the later operations of the assembler. The metanotation

[ stuff ] ...

means that 0 or more instances of the given stuff may appear. Also, boldface tokens are literals, italic words are substitutable.

### 7.1 *.byte expression [ , expression ] ...*

The *expressions* in the comma-separated list are truncated to 8 bits and assembled in successive bytes. The expressions must be absolute. This statement and the string statement above are the only ones that assemble data one byte at a time.

### 7.2 *.even*

If the location counter “.” is odd, it is advanced by one so the next statement will be assembled at a word boundary.

### 7.3 *.if expression*

The *expression* must be absolute and defined in pass 1. If its value is nonzero, the *.if* is ignored; if zero, the statements between the *.if* and the matching *.endif* (below) are ignored. *.if* may be nested. The effect of *.if* cannot extend beyond the end of the input file in which it appears. (The statements are not totally ignored, in the following sense: *.ifs* and *.endifs* are scanned for, and moreover all names are entered in the symbol table. Thus names occurring only inside an *.if* will show up as undefined if the symbol table is listed.)

### 7.4 *.endif*

This statement marks the end of a conditionally-assembled section of code. See *.if* above.

### 7.5 *.globl name [ , name ] ...*

This statement makes the *names* external. If they are otherwise defined (by assignment or appearance as a label) they act within the assembly exactly as if the *.globl* statement were not given; however, the link editor *ld* may be used to combine this routine with other routines that refer these symbols.

Conversely, if the given symbols are not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbols. As discussed in §1, it is possible to force the assembler to make all otherwise undefined symbols external.

### 7.6 *.text*

### 7.7 *.data*

### 7.8 *.bss*

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment respectively. Assembly starts in the text segment. It is forbidden to assemble any code or data into the bss segment, but symbols may be defined and “.” moved about by assignment.

### 7.9 .comm name , expression

Provided the *name* is not defined elsewhere, this statement is equivalent to

```
.globl name
name = expression ` name
```

That is, the type of *name* is "undefined external", and its value is *expression*. In fact the *name* behaves in the current assembly just like an undefined external. However, the link-editor *ld* has been special-cased so that all external symbols which are not otherwise defined, and which have a non-zero value, are defined to lie in the bss segment, and enough space is left after the symbol to hold *expression* bytes. All symbols which become defined in this way are located before all the explicitly defined bss-segment locations.

## 8. Machine instructions

Because of the rather complicated instruction and addressing structure of the PDP-11, the syntax of machine instruction statements is varied. Although the following sections give the syntax in detail, the machine handbooks should be consulted on the semantics.

### 8.1 Sources and Destinations

The syntax of general source and destination addresses is the same. Each must have one of the following forms, where *reg* is a register symbol, and *expr* is any sort of expression:

syntax	words	mode
<i>reg</i>	0	00+ <i>reg</i>
( <i>reg</i> ) +	0	20+ <i>reg</i>
- ( <i>reg</i> )	0	40+ <i>reg</i>
<i>expr</i> ( <i>reg</i> )	1	60+ <i>reg</i>
( <i>reg</i> )	0	10+ <i>reg</i>
* <i>reg</i>	0	10+ <i>reg</i>
* ( <i>reg</i> ) +	0	30+ <i>reg</i>
* - ( <i>reg</i> )	0	50+ <i>reg</i>
* ( <i>reg</i> )	1	70+ <i>reg</i>
* <i>expr</i> ( <i>reg</i> )	1	70+ <i>reg</i>
<i>expr</i>	1	67
<i>\$expr</i>	1	27
* <i>expr</i>	1	77
* <i>\$expr</i>	1	37

The *words* column gives the number of address words generated; the *mode* column gives the octal address-mode number. The syntax of the address forms is identical to that in DEC assemblers, except that "\*" has been substituted for "@" and "\$" for "#"; the UNIX typing conventions make "@" and "#" rather inconvenient.

Notice that mode "\*reg" is identical to "(reg)"; that "(reg)" generates an index word (namely, 0); and that addresses consisting of an unadorned expression are assembled as pc-relative references independent of the type of the expression. To force a non-relative reference, the form "\$expr" can be used, but notice that further indirection is impossible.

### 8.3 Simple machine instructions

The following instructions are defined as absolute symbols:

clc  
clv  
clz  
cin  
sec  
sev  
sez  
sen

They therefore require no special syntax. The PDP-11 hardware allows more than one of the "clear" class, or alternatively more than one of the "set" class to be or-ed together; this may be expressed as follows:

clc | clv

#### 8.4 Branch

The following instructions take an expression as operand. The expression must lie in the same segment as the reference, cannot be undefined-external, and its value cannot differ from the current location of "." by more than 254 bytes:

<b>br</b>	<b>bls</b>	
<b>bne</b>	<b>bvc</b>	
<b>beq</b>	<b>bvs</b>	
<b>bge</b>	<b>bhis</b>	
<b>blt</b>	<b>bec</b>	(= bcc)
<b>bgt</b>	<b>bcc</b>	
<b>ble</b>	<b>blo</b>	
<b>bpl</b>	<b>bcs</b>	
<b>bmi</b>	<b>bes</b>	(= bcs)
<b>bhi</b>		

**bes** ("branch on error set") and **bec** ("branch on error clear") are intended to test the error bit returned by system calls (which is the c-bit).

#### 8.5 Extended branch instructions

The following symbols are followed by an expression representing an address in the same segment as ".". If the target address is close enough, a branch-type instruction is generated; if the address is too far away, a **jmp** will be used.

<b>jbr</b>	<b>jlos</b>
<b>jne</b>	<b>jvc</b>
<b>jeq</b>	<b>jvs</b>
<b>jge</b>	<b>jhis</b>
<b>jlt</b>	<b>jec</b>
<b>jgt</b>	<b>jcc</b>
<b>jle</b>	<b>jlo</b>
<b>jpl</b>	<b>jcs</b>
<b>jmi</b>	<b>jes</b>
<b>jhi</b>	

**jbr** turns into a plain **jmp** if its target is too remote; the others (whose names are constructed by replacing the "b" in the branch instruction's name by "j") turn into the converse branch over a **jmp** to the target address.

### 8.6 Single operand instructions

The following symbols are names of single-operand machine instructions. The form of address expected is discussed in §8.1 above.

clr	sbc
clrb	ror
com	rorb
comb	rol
inc	rolb
incb	asr
dec	asrb
decb	asl
neg	asib
negb	jmp
adc	swab
adcb	tst
sbc	tstb

### 8.7 Double operand instructions

The following instructions take a general source and destination (§8.1), separated by a comma, as operands.

mov  
movb  
cmp  
cmpb  
bit  
bitb  
bic  
bicb  
bis  
bisb  
add  
sub

### 8.8 Miscellaneous instructions

The following instructions have more specialized syntax. Here *reg* is a register name, *src* and *dst* a general source or destination (§8.1), and *expr* is an expression:

jsr	<i>reg, dst</i>	
rts	<i>reg</i>	
sys	<i>expr</i>	
ash	<i>src, reg</i>	(or, als)
ashc	<i>src, reg</i>	(or, als)
mul	<i>src, reg</i>	(or, mpy)
div	<i>src, reg</i>	(or, dvd)
xor	<i>reg, dst</i>	
sxt	<i>dst</i>	
mark	<i>expr</i>	
sob	<i>reg, expr</i>	

*sys* is another name for the *trap* instruction. It is used to code system calls. Its operand is required to be expressible in 6 bits. The expression in *mark* must be expressible in six bits, and the expression in *sob* must be in the same segment as “.”, must not be external-undefined, must be less than “.”, and must be within 510 bytes of “..”.

## 8.9 Floating-point unit instructions

The following floating-point operations are defined, with syntax as indicated:

<b>cfcc</b>		
<b>setf</b>		
<b>setd</b>		
<b>seti</b>		
<b>setl</b>		
<b>clrf</b>	<i>fdst</i>	
<b>negf</b>	<i>fdst</i>	
<b>absf</b>	<i>fdst</i>	
<b>tstf</b>	<i>fsrc</i>	
<b>movf</b>	<i>fsrc, freg</i>	(= ldf)
<b>movf</b>	<i>freg, fdst</i>	(= stf)
<b>movif</b>	<i>src, freg</i>	(= ldcif)
<b>movfi</b>	<i>freg, dst</i>	(= stcfi)
<b>movof</b>	<i>fsrc, freg</i>	(= ldcdf)
<b>movfo</b>	<i>freg, fdst</i>	(= stcfd)
<b>movie</b>	<i>src, freg</i>	(= ldexp)
<b>movei</b>	<i>freg, dst</i>	(= stexp)
<b>addf</b>	<i>fsrc, freg</i>	
<b>subf</b>	<i>fsrc, freg</i>	
<b>mulf</b>	<i>fsrc, freg</i>	
<b>divf</b>	<i>fsrc, freg</i>	
<b>cmpf</b>	<i>fsrc, freg</i>	
<b>modf</b>	<i>fsrc, freg</i>	
<b>ldfps</b>	<i>src</i>	
<b>stfps</b>	<i>dst</i>	
<b>stst</b>	<i>dst</i>	

*fsrc*, *fdst*, and *freg* mean floating-point source, destination, and register respectively. Their syntax is identical to that for their non-floating counterparts, but note that only floating registers 0-3 can be a *freg*.

The names of several of the operations have been changed to bring out an analogy with certain fixed-point instructions. The only strange case is **movf**, which turns into either **stf** or **ldf** depending respectively on whether its first operand is or is not a register. Warning: **ldf** sets the floating condition codes, **stf** does not.

## 9. Other symbols

### 9.1 ..

The symbol “..” is the *relocation counter*. Just before each assembled word is placed in the output stream, the current value of this symbol is added to the word if the word refers to a text, data or bss segment location. If the output word is a pc-relative address word that refers to an absolute location, the value of “..” is subtracted.

Thus the value of “..” can be taken to mean the starting memory location of the program. The initial value of “..” is 0.

The value of “..” may be changed by assignment. Such a course of action is sometimes necessary, but the consequences should be carefully thought out. It is particularly ticklish to change “..” midway in an assembly or to do so in a program which will be treated by the loader, which has its own notions of “..”.

## 9.2 System calls

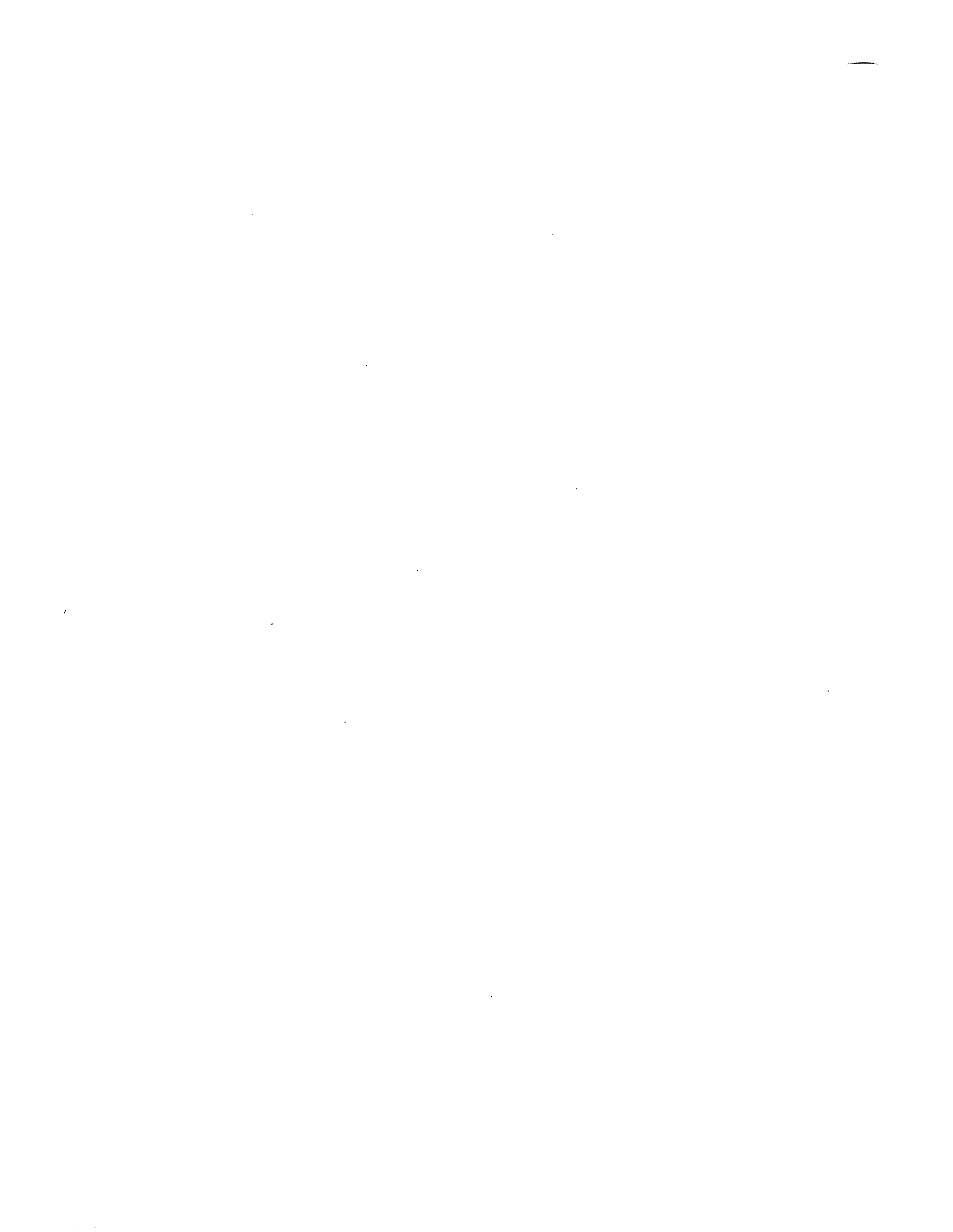
System call names are not predefined. They may be found in the file */usr/include/sys.s*

## 10. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

- ) parentheses error
- ] parentheses error
- > string not terminated properly
- indirection ( \* ) used illegally
- . illegal assignment to “ . ”
- A error in address
- B branch address is odd or too remote
- E error in expression
- F error in local (“f” or “b”) type symbol
- G garbage (unknown) character
- I end of file inside an .if
- M multiply defined symbol as label
- O word quantity assembled at odd address
- P phase error— “ . ” different in pass 1 and 2
- R relocation error
- U undefined symbol
- X syntax error







## Summary of Differences Between UniSoft and Motorola Macro Assemblers

The reference document used for this comparison is the *M68000 Family Resident Structured Assembler Reference Manual* [Document# *M68KMASM(D7)*] available from UniSoft Systems or from Motorola Literature, Motorola, Inc., P.O. Box 20924, Phoenix, AZ 85036, (602) 994-6561.

The extensions, modifications and unimplemented features of the Motorola Macro Assembler by the UniSoft Macro Assembler are listed by each chapter and section of the above-mentioned document. An extension or modification is denoted by a bullet (●), an unimplemented feature by a square (□).

### Chapter 1. General Information

#### 1.2 Introduction

- The assembler does not include the following features:
  - absolute code generation
  - symbol table listing
  - structured syntax
  - cross-reference table

#### 1.4 M68000 Resident Structured Assembler

- Object modules produced by the assembler are compatible with the UniPlus<sup>+</sup> linker/"loader" 'ld' .
- The assembler does not implement structured syntax.

#### 1.5 Relocation and Linkage

- UNIX-style program sections provide the basis of the relocation and linking scheme. Refer to the AS Assembler Reference Guide in the UniPlus<sup>+</sup> User's Manual, Volume II.
- There is no ORG directive.

#### 1.6 Linker Restrictions

- Disregard references to Pascal.
- There is no XREF or XDEF directive.

#### 1.8 Related Publications

- The user should be familiar with the following UniSoft publications:
  - UniPlus<sup>+</sup> User's Manual — Volume I: ASM(1)
  - UniPlus<sup>+</sup> User's Manual — Volume II: AS Assembler Reference Guide, "C" Interface Notes for 68000 UNIX
- The user does not need to be familiar with the following publications:
  - EXORmacs Development System Operation Manual (M68KMACS)
  - M68000 Family Linkage Editor User's Manual (M68KLINK)
  - M68000 Family Resident Pascal User's Manual (M68KPASC)
  - VERSA dos Messages Reference Manual (M68KVMSG)
  - VERSA dos System Facilities Reference Manual (M68KVSF)

## Chapter 2. Source Program Coding

### 2.2 Comments

- A comment may be inserted in an additional way:
  - c. Following the label field, where an asterisk (\*) is the first non-blank character after the label.  
Example: COMMENT: \* THIS COMMENT FOLLOWS THE LABEL FIELD.

### 2.4 Source Line Format

- The assembler will prefix each line with a sequential number only if the **-n** option is used. Refer to the ASM(1) entry in the UniPlus<sup>+</sup> User's Manual, Volume I.

#### 2.4.1 Label Field

- There is no IDNT directive.

#### 2.4.2 Operation Field

- If the data size code is input but is not permitted, it is ignored.

### 2.5 Instruction Mnemonics

- The instruction operations can be in uppercase, lowercase, or a combination of both.

#### 2.5.10 DBcc Instruction

- A "dbra" alternate exists for the "dbf" operation.

#### 2.5.11 Load/Store Multiple Registers

- An immediate operand can be used instead of a register set description.
- The immediate operand is used directly as the extension word.

#### 2.6.1 Symbols

- Symbols recognized by the assembler consist of at least one but not more than 49 valid characters, all of which are significant. The loader will not properly handle symbols of 50 or more characters.
- The first character of the symbol must be an uppercase or lowercase letter (a-z, A-Z), a period (.), or an underscore (\_). Each remaining character may be an uppercase or lowercase letter (a-z, A-Z), a digit (0-9), a dollar sign (\$), a period (.), an underscore (\_), or an "at" sign (@).
- Uppercase and lowercase characters are distinct.
- Hexadecimal numbers are also specified by using a zero and the letter "x" (0x) as the first digit.  
Example: 0x3A4
- Octal numbers are also specified by using a zero (0) as the first digit.  
Example: 052
- One or more ASCII characters enclosed by quote marks (") also constitutes an ASCII string.  
Example: DC.L "ABCD"
- ASCII strings are denoted in the same way (i.e., surrounded by apostrophes or quotes) except that C style escapes can be used. The valid escapes are (\b, \t, \n, \f, \f, \\, \xxx, \', \"). The "" form will also escape a ' and the "" form will also escape a ".
- ASCII strings are left-justified and padded with zeros, if necessary. The left justification is to the size specified by the opcode size modifier that is byte, word or long.

#### 2.6.3 User-Defined Labels

- There is no SECTION or ORG directive.

#### 2.6.4 Expressions

- There are no complex relocatable expressions.
- There is no XREF directive.

### 2.7 Registers

- The mnemonics can be in uppercase, lowercase, or a combination of both.

## 2.9 Addressing Modes

- There are no complex relocatable expressions.
- ASM does not require data addresses to be even when referenced by a word or long instruction.

### Table 2-1

- There are no complex relocatable expressions.

### Table 2-2

- There are no complex relocatable expressions.

### Table 2-3 & 2-4a-e

- There is no ORG, SECTION or XREF directive.
- There is no PCS or PCO option.

## Chapter 3. Assembler Directives

### 3. Introduction

- All of the AS assembler directives have been implemented. Refer to the AS Assembler Reference Guide in the UniPlus<sup>+</sup> User's Manual, Volume II.

#### 3.2.1 ORG

- ORG is not implemented.

#### 3.2.2 SECTION

- SECTION is not implemented.

#### 3.2.3 END

- The END directive does not allow a start address.

#### 3.2.4 OFFSET

- Code-producing directives may appear but no code is produced.
- The OFFSET assembler directive is identical to the ".abs" directive.

#### 3.2.5 MASK2

- MASK2 is not implemented.

#### 3.2.6 INCLUDE

- INCLUDE is not implemented.

#### 3.3.3 REG

- REG is not implemented.

#### 3.4.4 COMLINE

- COMLINE is not implemented.

#### 3.5.2.3 FORMAT

- Formats the operand and comment fields as one unit.

#### 3.5.2.8 TTL

- A title string consists of up to as many characters that will fit on a line.

#### 3.5.2.10 OPT

- The following options are not implemented:

[NO]A	[NO]PCO	D
[NO]PCS	P=680[01]0	CRE

### 3.7.1 IDNT

- IDNT is not implemented.

### 3.7.2 XDEF

- XDEF is not implemented because XREF is not implemented.

### 3.7.3 XREF

- XREF is not implemented because SECTION is not implemented.

Table 3-1

- The assembler does not include the following directives:

ORG	MASK2	IDNT
INCLUDE	REG	XREF
SECTION	COMLINE	XDEF

## Chapter 4. Invoking the Assembler

### 4.1 Command Line Format

- Refer to the ASM(1) entry in the UniPlus<sup>+</sup> User's Manual, Volume I.

#### 4.1.1 Symbol Table Size Option

- This option is not implemented.
- The symbol table size is dynamically increased.

#### 4.1.2 Microprocessor Type Option

- Refer to the ASM(1) entry in the UniPlus<sup>+</sup> User's Manual, Volume I.

### 4.2 Assembler Output

- Assembler output is an assembly listing and an object program file. A symbol table can be generated by running the standard UniPlus<sup>+</sup> program 'nm' on the object file.
- Listing is different (see Table 4-1 below).

### 4.3 Assembler Runtime Errors

- Most errors will not produce object code, extension words or zeros.
- One should never attempt to run or patch an object module produced when errors occurred in assembly.

Table 4-1

Columns	Contents
1-6	Location counter value
8-11	Operation word
13-16	First extension word
18-21	Second extension word; subsequent extension words occur on the next line
26-N	Source line
<i>NOTE:</i>	if the -n option is specified, a five character field containing the line number is prefixed to the line

## Chapter 5. Macro Operations and Conditional Assembly

### 5.2.1 Macro Definition

- Nested macro definition is permitted.

#### 5.2.4 Labels Within Macros

- Each macro call has a unique value of \@ associated with it. All references to \@ within that expansion of the macro are replaced by the same value.

#### 5.3.2 Example of Macro and Conditional Assembly Usage

- Implementation of the example in the manual will demonstrate the difference in expansion of \@.

#### 5.2.7 Implementation of Macro Definition.

- No check is made for missing parameter references.

### **Chapter 6. Structured Control Statements**

- Chapter 6 is not implemented.

### **Appendix C**

- The listing formats are different (see Table 4-1 above).

### **Appendix D**

- Not applicable.

### **Appendix E**

- Error messages are used instead of error code numbers.



# AS Assembler Reference Guide

James L. Gula† and Thomas J. Teixeira, March 1980

Revised by UniSoft Systems, 27 March 1982.

## INTRODUCTION

This document describes the syntax and usage of the AS assembler for the Motorola 68000 microprocessor. The basic format of AS is loosely based on the Digital Equipment Corp Macro-11 assembler described in DEC's publication DEC-11-0MACA-A-D but also contains elements of the UNIX\* PDP 11 *as (1)* assembler. The instruction mnemonics and effective address format are derived from a Motorola publication on the 68000, the *MACSS MC68000 Design Specification Instruction Set Processor* dated June 30, 1979.

Sections 1-3 of this document describe the general form of AS programs, section 4 describes the instruction mnemonics and addressing modes, section 5 describes the pseudo-ops supported by the assembler and section 6 describes the error codes generated. For instructions on how to operate the assembler from UniPlus+, readers should consult the UniPlus+ *man* entry on AS.

## 1. NOTATION

The notation used in this document is a modified BNF similar to that used in the MULTICS PL/I Language Manual. The operators of the BNF in order of decreasing precedence are:

Repetition	X...	Denotes one or more occurrences of X.
Juxtaposition	XY	Denotes an occurrence of X followed by an occurrence of Y.
Alternation	X Y	Denotes an occurrence of X or Y but not both.

Brackets and braces define the order of expression interpretation. Brackets also indicate that the syntax described by the subexpression they enclose is optional. That is,

[X]	denotes zero or one occurrence of X.
{X Y}Z	denotes an X or a Y, followed by a Z.

† This document is actually an edited version of another document, the MICAL MANUAL for the INTEL 8080 written by Mike Patrick just as AS is an edit of the MICAL assembler also written by Mike. Thus the reader should consider the features of this work to be Mike's and the bugs to be mine.

\* UNIX is a trademark of Bell Laboratories.

UniPlus+ is a trademark of UniSoft Systems.

## 2. SOURCE PROGRAM FORMAT

An AS program consists of a series of statements, each of which occupies exactly one line, i.e. a sequence of characters followed by the <newline> character. Neither multiple statements on a single line nor continuation lines are allowed, and the maximum line length is 255 characters.

The format of an AS assembly language statement is:

[ <label field> ] <op-code> [ <operand field> ] [ | <comment> ]

There are three exceptions to this rule:

- 1) Blank lines are permitted.
- 2) A statement may contain only a label field. The label defined in this field has the same value as if it were defined in the label field of the next statement in the program. For example, the two statements

```
foo:
movw r1,r2
```

are equivalent to the single statement

```
foo: movw d1,d2
```

- 3) A line may consist of only the comment field. For example, the two statements below are allowed:

```
| This is a comment field.
| So is this.
```

In general, blanks or tabs are allowed anywhere in a statement. For example, multiple blanks are allowed in the operand field to separate symbols from operators. Blanks are meaningful only when they occur in a character string (for instance, as the operand of an .ASCII pseudo-op) or in a character constant. At least one blank or tab must appear between the op-code and the operand field of a statement.

### 2.1. Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the assembler's symbol table. The value of the label may be either absolute or relocatable, depending on whether the location counter value is currently absolute or relocatable. In the latter case, the absolute value of the symbol is assigned when the program is linked via *ld*.

A label is a symbolic means of referring to a specific location within a program. If present, a label *always* occurs first in a statement and *must* be terminated by a colon. A maximum of 10 labels may be defined by a single source statement. The collection of label definitions in a statement is called the 'label field'.

The format of a <label field> is:

<symbol> : [ <symbol> : ]...

*Examples:*



start:	
foo: bar:	Multiple symbols
7S:	A local symbol, defined below

## 2.2. Op-code Field

The opcode field of an assembly language statement identifies the statement as either a machine instruction, or an assembler directive. One or more blanks (or tabs) must separate the opcode field from the operand field in a statement. No blanks are necessary between the label and opcode fields, but they are recommended to improve readability of the program.

A machine instruction is indicated by an instruction mnemonic. The assembly language statement is intended to produce a single executable machine instruction. The operation of each instruction is described in the manufacturer's user manual. Some conventions used in AS for instruction mnemonics are described in section 4 and a complete list of the instructions is presented in the appendix.

An assembler directive, or pseudo-op, performs some function during the assembly process. It does not produce any executable code, but it may assign space in a program for data. AS assembly directives are patterned after those of DEC's MACRO-11 assembler and are described in detail in section 5.

No distinction between upper and lower case is made for instruction mnemonics and assembler directive names. That is,

```
add a,b
.word 0xAA00
```

is equivalent to

```
ADD a,b
.Word 0xAA00
```

## 2.3. Operand Field

A distinction is made between *<operand field>* and *<operand>* in AS. Several machine instructions and assembler directives require two or more arguments, and each of these is referred to as an 'operand'. In general, an operand field consists of zero or more operands, and **in all cases, operands are separated by a comma**. In other words, the format for an *<operand field>* is:

[ *<operand>* [, *<operand>*] ... ]

The format of the operand field for machine instruction statements is the same for all instructions, and is described in section 4. The format of the operand field for assembler directives depends on the directive itself, and is included in the directive's description in section 5 of this manual.

## 2.4. Comment Field

The comment character in AS is the vertical bar, (|); *not* the semicolon, (;). Use of the semicolon as a comment character results in an 'Invalid Operator' error.

The comment field consists of all characters on a source line following and including the comment character. These characters are ignored by the assembler. Any character may appear in the comment field, with the obvious exception of the *<newline>* character, which starts a new line.

### 3. SYMBOLS AND EXPRESSIONS

This section describes the various components of AS expressions: symbols, numbers, terms, and, expressions.

#### 3.1. SYMBOLS

A symbol consists of 1 to 31 characters, with the following restrictions:

1. Valid characters include A-Z, a-z, 0-9, period (.), underscore (\_), and dollar sign (\$).
2. The first character must not be numeric.

All 31 characters are significant and are checked in comparisons with other symbols. Upper and lower cases are distinct, ('Foo' and 'foo' are separate symbols). The period ('.') and dollar sign ('\$') characters are valid symbol characters, but they are reserved for system software symbols (for example, pseudo-ops) and should not appear in user-defined symbols.

A symbol is said to be declared when the assembler recognizes it as a symbol of the program. A symbol is said to be 'defined' when a value is associated with it. With the exception of symbols declared by a .GLOBL directive, all symbols are defined when they are declared. A label symbol (which represents an address of the program) may not be redefined; all other symbols are allowed to receive a new value.

There are several ways to declare a symbol:

1. As the *label* of a statement (see section 2.1)
2. In a *direct assignment* statement
3. As an *external* symbol via the .GLOBL directive
4. As a *common* symbol via the .COMM directive
5. As a *local* symbol

##### 3.1.1. DIRECT ASSIGNMENT STATEMENTS

A direct assignment statement assigns the value of an arbitrary expression to a specified symbol. The format of a direct assignment statement is:

`<symbol> = [<symbol> =]... <expression>`

Examples of valid direct assignments are:

```
vect_size = 4
vectora = 0xFFFE
vectorb = vectora-vect_size
CRLF = 0x0D0A
```

More than one symbol may be assigned in a single statement. For example:

```
abc = cde = fgh = 5
```

assigns the value 5 to the three symbols 'abc', 'cde' and 'fgh'. This statement is equivalent to the three statements:

```
abc = 5
cde = 5
fgh = 5
```

Up to 10 symbols may be defined in this manner in a single statement.

Any symbol defined by direct assignment may be redefined later in the program, in which case its value is the result of the last such statement. A local symbol may be defined by direct assignment, though this doesn't make much sense. A label or register

symbol (see section 3.1.2) may *not* be redefined.

If the *<expression>* is absolute, then the symbol is also absolute, and may be treated as a constant (section 3.4) in subsequent expressions. If the *<expression>* is relocatable, however, then the *<symbol>* is also relocatable, and it is considered to be declared the same program section as the expression. See section 3.7 for an explanation of absolute and relocatable expressions.

If the *expression* contains an external symbol, then the symbol defined by the = statement is also considered external. For example:

<b>.globl X</b>	<b>X is declared as external symbol</b>
<b>foo = X</b>	<b>foo becomes an external symbol</b>

assigns the value of X (zero if it is undefined) to foo and makes foo an external symbol. External symbols may be defined by direct assignment.

### 3.1.2. EXTERNAL SYMBOLS

A program may be assembled in separate modules, and then linked together to form a single program (see *ld(1)*). External symbols are defined in each of these separate modules. A symbol which is declared (given a value) in one module may be referenced in another module by declaring the symbol to be external in *both* modules. There are two forms of external symbols: those defined with the *.GLOBL* and those defined with the *.COMM* directive.

External symbols are declared with the *.GLOBL* assembler directive. The format is:

**.GLOBL** *<symbol>* [, *<symbol>*] ...

For example, the following statements declare the array *TABLE* and the routine *SRCH* to be external symbols:

	<b>.globl TABLE, SRCH</b>	
<b>TABLE:</b>	<b>.word</b>	<b>0,0,0,0,0</b>
<b>SRCH:</b>	<b>movw TABLE,r0</b>	
	<b>etc...</b>	

External symbols are only *declared* to the assembler. They must be *defined* (i.e. given a value) in some other statement by one of the methods mentioned above. They need not be defined in the current program; in this case they are flagged as 'undefined' in the symbol table. If they are undefined, they are considered to have a value of zero in expressions.

The other form of external symbol is defined with the *.COMM* directive. These statement reserve storage in the *bss* section similar to FORTRAN common areas. The format of the statement is:

**.COMM** *<name>* *<constant expression>*

which causes AS to declare the *name* as a *common* symbol with a value equal to the expression. For the rest of the assembly this symbol is treated as though it were an undefined global. AS does **not** *allocate storage for common* symbols; this task is left to the loader. The loader computes the maximum size of for each *common* symbol which may appear in several load modules, allocates storage for it in the final *bss* section and resolves linkages.

### 3.1.3. LOCAL SYMBOLS

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of local symbols reduces the possibility of multiply-defined symbols in a program, and separates entry point symbols from local references, such as the top of a loop. Local symbols cannot be referenced by other object modules.

Local symbols are of the form *n*\$ where *n* is any integer. Valid local symbols include:

**1\$**  
**27\$**  
**394\$**

A local symbol is defined and referenced only within a single 'local symbol block' (lsb). A new local symbol block is entered when either 1) a label is declared; or 2) a new program section is entered. There is no conflict between local symbols with the same name which appear in different local symbol blocks.

### 3.2. ASSEMBLY LOCATION COUNTER

The assembly location counter is the period character, '.'; hence its name 'dot'. When used in the operand field of any statement, dot represents the address of the first byte of the statement. Even in assembly directives, it represents the address of the start of the directive. A dot appearing as the third argument in a .BYTE instruction would have the value of the address where the first byte was loaded; it is not updated 'during' the pseudo-op.

For example,

**Ralph: movl .,a0 | load value of Ralph into a0**

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generated code. However, the location where the code is stored may be changed by a direct assignment altering the location counter:

**. = <expression>**

This <expression> must not contain any forward references, and must not change from one pass to another. Storage area may also be reserved by advancing dot. For example, if the current value of dot is 1000, the direct assignment statement:

**Table: . = . + 0x100**

would reserve 100 hex bytes of storage, with the address of the first byte as the value of Table. The next instruction would be stored at address 1100.

### 3.3. PROGRAM SECTIONS

As in UNIX, programs to AS are divided into three sections: *text*, *data* and *bss*. The normal interpretation of these sections is: instruction space, initialized data space and uninitialized data space, respectively.

These sections are equivalent as far as AS is concerned with the exception that no instructions or data are output for the *bss* section although its size is computed and its symbol values are output.

In the first pass of the assembly, AS maintains a separate location counter for each section, thus for code like:

```

 .text
foo: movw d1,d2
 .data
blah: .long 27
 .text
bar: addw d2,d1
 .data
bletch: .byte 4

```

in the output, *foo* immediately precedes *bar* and *blah* immediately precedes *bletch*. At the end of the first pass, AS rearranges all the addresses so that the sections are output in the following order: *text*, *data* and *bss*. The resulting output file is an executable image file with all addresses correctly resolved, with the exception of undefined *.GLOBL's* and *.COMM's*. For more information on the format of the output file, consult the UNIX *man* entry on *a.out* files.

### 3.4. CONSTANTS

All constants are considered absolute quantities when appearing in an expression.

#### 3.4.1. NUMERIC CONSTANTS

Any 'symbol' beginning with a digit is assumed to be a number, and is interpreted as decimal. Numbers may be represented in other radices by prefixing them with the appropriate indicators.

The prefixes to indicate number bases other than base ten are:

<i>Radix</i>	<i>Prefix</i>	<i>Example</i>
octal	0	017 equals 15 base 10
hex	0x	0xA1 equals 161 base 10

Letters in hex constants may be upper or lower case; for example, 0xaa = 0xAa = 0xaa = 176. Illegal digits for a particular radix generate an error (for example, 018).

#### 3.4.2. STRINGS

A string is a sequence of ASCII characters, enclosed in quote signs (").

Within a string, the following escape sequences are valid:

<i>X</i>	<i>Value of X</i>
<code>\b</code>	< backspace >, octal 010
<code>\t</code>	< tab >, hex 09
<code>\n</code>	< line-feed >, octal 012
<code>\f</code>	< form-feed >, octal 014
<code>\r</code>	< return >, octal 015
<code>\\</code>	< backslash >, octal 134
<code>\nnn</code>	octal <i>nnn</i>

### 3.5. OPERATORS

#### 3.5.1. UNARY OPERATORS

There are two unary operators in AS:

<i>Operator</i>	<i>Function</i>
-	unary minus.
'	logical negation.

#### 3.5.2. BINARY OPERATORS

Binary operators in AS include:

<i>Operator</i>	<i>Description</i>
+	Addition; for example, "3+4" evaluates to 7.
-	Subtraction; for example, "3-4" evaluates to -1, or 0xFFFF
*	Multiplication; for example, "4*3" evaluates to 12.

Each operator is assumed to work on a 32 bit number. If the value of a particular term occupies only 8 bits, the sign bit is extended into the high byte.

#### 3.6. TERMS

A term is a component of an expression. A term may be one of the following:

- a. A number, as defined in section 3.3, whose 32-bit value is used.
- b. A symbol, as defined in section 3.1.
- c. An expression or term enclosed in square brackets ([ ]). Any quantity enclosed in square brackets is evaluated before the rest of the expression. This can be used to alter the normal left-to-right evaluation of expressions (for example, differentiating between  $a*b+c$  and  $a*(b+c)$ ) or to apply a unary operator to an entire expression (for example,  $-[a*b+c]$ ).
- d. A term preceded by a unary operator. For example, both 'foo' and '~foo' may be considered to be terms. Multiple unary operators are allowed; for example, '--A' has the same value as 'A'.

#### 3.7. EXPRESSIONS

Expression are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value. If the instruction calls for only one byte, (for example, .BYTE), then the low-order byte is used.

Expressions are evaluated left to right with no operator precedence. Thus '1+2\*3' evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied.

A missing expression or term is interpreted as having a value of zero. In this case, an 'Invalid expression' error is generated. An 'Invalid Operator' error means that a valid end\_of\_line character or binary operator was not detected after the assembler processed a term. In particular, this error is generated if an expression contains a symbol with an illegal character, or if an incorrect comment character was used.

Any expression, when evaluated, is either absolute, relocatable, or external:

- a. An expression is absolute if its value is fixed. An expression whose terms are constants, or symbols whose values are constants via a direct assignment statement, is absolute. A relocatable expression minus a relocatable term,

where both items belong to the same program section is also absolute.

- b. An expression is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked, or loaded into core. All labels of a program defined in relocatable sections are relocatable terms, and any expression which contains them must only **add or subtract constants to their value**. For example, assume the symbol 'foo' was defined in a relocatable section of the program. Then the following demonstrates the use of relocatable expressions:

foo	<i>Relocatable</i>
foo+5	<i>Relocatable</i>
foo-'A	<i>Relocatable</i>
foo*2	<i>Not relocatable</i>
2-foo	<i>Not relocatable, since the expression cannot be linked by adding foo's offset to it.</i>
foo-bar	<i>Absolute, since the offsets added to foo and bar cancel each other out.</i>

- c. An expression is external (or global) if it contains an external symbol not defined in the current program. The same restrictions on expressions containing relocatable symbols apply to expressions containing external symbols. Exception: the expression 'foo-bar' where both foo and bar are external symbols is not allowed.

#### 4. INSTRUCTIONS AND ADDRESSING MODES

This section describes the conventions used in AS to specify instruction mnemonics and addressing modes.

##### 4.1. INSTRUCTION MNEMONICS

The instruction mnemonics used by AS are described in the previously mentioned Motorola manual with a few variations. Most of the 68000 instructions can apply to byte, word or long operands, so in AS the normal instruction mnemonic is suffixed with *b*, *w*, or *l* to indicate which length operand was intended. For example, there are three mnemonics for the *or* instruction: **orb**, **orw** and **orl**. Op-codes for instructions with unusual opcodes may have additional suffixes, thus in addition to the normal *add* variations, there also exist: **addqb**, **addqw** and **addql** for the *add quick* instruction.

Branch instructions come in two flavors, byte and word. In AS, the byte version is specified by appending the suffix *s* to the basic mnemonic as in **beq** and **beqs**.

In addition to the instructions which explicitly specify the instruction length, AS supports extended branch instructions, whose names are generally constructed by replacing the **b** with **j**. If the operand of the extended branch instruction is a simple address in the current segment, and the offset to that address is sufficiently small, AS will automatically generate the corresponding short branch instruction. If the offset is too large for a short branch, but small enough for a branch, then the corresponding branch instruction is generated. If the operand references an external address or is complex, then the extended branch instruction is implemented either by a **jmp** or **jsr** (for **jra** or **jbsr**), or by a conditional branch (with the sense of the conditional inverted) around a **jmp** for the extended conditional branches. In this context, a complex

address is either an address which specifies other than normal mode addressing, or relocatable expressions containing more than one relocatable symbol. i.e. if  $a$ ,  $b$  and  $c$  are symbols in the current segment, then the expression  $a+b-c$  is relocatable, but not simple.

Note that AS is not optimal for extended branch instructions whose operand addresses the next instruction. The optimal code is no instruction at all, but AS currently retains insufficient information to make this optimization. The difficulty is that if AS decides to just eliminate the instruction, the address of the next instruction will be the same as the address of the (nonexistent) extended branch instruction. This instruction will then look like a branch to the current location, which would require an instruction to be generated. The code that AS actually generates for this case is a `nop` (recall that an offset of zero in a branch instruction indicates a long offset). Although this problem may arise in compiler code generators, it can easily be handled by a peephole optimizer.

The algorithm used by AS for deciding how to implement extended branch instructions is described in "Assembling Code for Machines with Span-Dependent Instruction," by Thomas G. Szymanski in *Communications of the ACM*, Volume 21, Number 4, pp. 300-308, April 1978.

Consult the Appendix for a complete list of the instruction op-codes.

#### 4.2. ADDRESSING MODES

The following table describes the addressing modes recognized by AS. In this table  $an$  refers to an address register,  $dn$  refers to a data register,  $Ri$  to either a data or an address register,  $d$  to a displacement, which, in AS is a constant expression, and  $xxx$  to a constant expression. Certain instructions, particularly `move`, accept a variety of special registers including `sp`, the stack pointer which is equivalent to `a7`, `sr`, the status register, `cc`, the condition codes of the status register, `usp`, the user mode stack pointer, and `pc`, the program counter.

Mode	Notation	Example
Register	$an, dn, sp, pc, cc, sr, usp$	<code>movw a3, d2</code>
Register Deferred	$an@$	<code>movw a3@, d2</code>
Postincrement	$an@+$	<code>movw a3@+, d2</code>
Predecrement	$an@-$	<code>movw a3@-, d2</code>
Displacement	$an@(d)$	<code>movw a3@(24), d2</code>
Word Index	$an@(d, Ri:W)$	<code>movw a3@(16, d2:W), d3</code>
Long Index	$an@(d, Ri:L)$	<code>movw a3@(16, d2:L), d3</code>
Absolute Short	$xxx.W$	<code>movw 14.W, d2</code>
Absolute Long	$xxx.L$	<code>movw 14.L, d2</code>
PC Displacement	$pc@(d)$	<code>movw pc@(20), d3</code>
PC Word Index	$pc@(d, Ri:W)$	<code>movw pc@(14, d2:W), d3</code>
PC Long Index	$pc@(d, Ri:L)$	<code>movw pc@(14, d2:L), d3</code>
Normal	$foo$	<code>movw foo, d3</code>
Immediate	$\#xxx$	<code>movw \#27+3, d3</code>

Normal mode actually assembles as absolute long, although the value of the constant will be flagged as relocatable to the loader. The notation for these modes derived from the Motorola notation with the exception of the colon in index mode rather than period.

The Motorola manual presents different opcodes for instructions that use the effective address as data rather than the contents of the effective address such as `adda` for `add address`. AS does not make this distinction because it can determine the type of



the operand from its form. Thus an instruction of the form:

```
foo: .word 0
 ...
 addl #foo,a0
```

will assemble to the *add address* instruction because *foo* is known to be an address.

The 68000 tends to be very restrictive in that most instructions accept only a limited subset of the address modes above. For example, the *add address* instruction does not accept a data register as a destination. AS tries to check all these restrictions and will generate the *illegal operand* error code for instructions that do not satisfy the address mode restrictions.

## 5. ASSEMBLER DIRECTIVES

The following pseudo-ops are available in AS:

.ascii .asciz	stores character strings
.byte .word .long .space	stores bytes/words/longs
.text .data .bss	Text csect Data csect Bss csect
.globl .comm	declares external symbols
.even	forces location counter to next word boundary

### 5.1. .ASCII .ASCIZ

The *.ASCII* directive translates character strings into their 7-bit ASCII (represented as 8-bit bytes) equivalents for use in the source program. The format of the *.ASCII* directive is as follows:

```
.ASCII "< character string>"
```

where

< character string> contains any character valid in a character constant (section 3.3.2). Obviously, a newline must not appear within the character string. A newline can be represented by the escape sequence '\n'.

The examples following illustrate the use of the `.ASCII` statement:

<i>ASCII Code Generated:</i>	<i>Statement:</i>
150 145 154 154 157 040 164 150 145 162 145	<code>.ascii "hello there"</code>
127 141 162 156 151 156 147 055 007 007 040 012	<code>.ascii "Warning-\007\007 \n"</code>
141 142 143 144 145 146 147	<code>.ascii "abcdefg"</code>

The `.ASCIZ` directive is equivalent to the `.ASCII` directive with a zero byte automatically inserted as the final character of the string. Thus when a list or text string is to be printed, a search for the null character can terminate the string.

## 5.2. `.BYTE` `.WORD` `.LONG` `.SPACE`

The `.BYTE`, `.WORD`, `.LONG` and `.SPACE` directives are used to reserve bytes and words and to initialize them with certain values.

The format of the various forms of data generation statements is:

```
[< label>:] .BYTE [< expression >] [, < expression >] ...
[< label>:] .WORD [< expression >] [, < expression >] ...
[< label>:] .LONG [< expression >] [, < expression >] ...
[< label>:] .SPACE [< expression >] [, < expression >] ...
```

initializes the value of the byte to be the low-order byte of the corresponding expression. Note that multiple expressions must be separated by commas. A blank expression is interpreted as zero, and no error is generated. For example,

```
.byte 7,0,1,2 reserves 4 bytes with values 7, 0, 1, and 2 respectively.
.byte ,,,, reserves five bytes, each with a value of zero.
.byte reserves a single byte, with a value of zero.
```

The syntax and semantics for `.WORD` is identical, except that 16-bit words are reserved and initialized and `.LONG` uses 32-bit quantities.

The `.SPACE` directive reserves the indicated number of bytes and fills them with zero.

## 5.3. `.TEXT` `.DATA` `.BSS`

These statements change the 'program section' where assembled code will be loaded.

## 5.4. `.GLOBL` `.COMM`

See section 3.1.2, above.

## 5.5. `.EVEN`

This directive advances the location counter if its current value is odd. This is useful for forcing storage allocation like `.WORD` directives to be on word boundaries.

## 6. ERROR CODES

If an assembly listing is produced, the error code for each statement appears on the line before the code. A message of the form:

```
error (<line_no>): <error_code>
```

is printed out on the terminal. The letter 'W' appearing after the error code signifies a warning.

The following error codes, and their probable cause, appear below:

### Invalid Character

An invalid character for a character constant or character string was encountered.

### Multiply defined symbol

A symbol appears twice as a label, or an attempt to redefine a label using an = statement.

### Symbol storage exceeded

No more room is left in the symbol table. Assemble portions of the program separately, then bind them together.

### Symbol length exceeded

A symbol of more than 31 characters was encountered.

### Undefined symbol

A symbol not declared by one of the methods mentioned above in 'Symbols' was encountered. This happens when an invalid instruction mnemonic is used. This also occurs when an invalid or non-printing character occurs in the statement.

### Invalid Constant

An invalid digit was encountered in a number.

### Invalid Term

The expression evaluator could not find a valid term: symbol, constant or [*expression*]. An invalid prefix to a number or a bad symbol name in an operand generates this message.

### Invalid Operator

Check the operand field for a bad operator.

### Non-relocatable expression

If an expression contains a relocatable symbol (e.g. label) then the only operations that can be applied to it are the addition of absolute expressions or the subtraction of another relocatable symbol (which produces an absolute result).

### Invalid operand type

The <type> field of an operand is either not defined for the machine, or represents an addressing mode incompatible with the current instruction.

### Invalid operand

This is a catch-all error. It appears notably when an attempt is made to assign an undefined value to dot during pass 1.

### Invalid symbol

If the first token on the source line is not a valid symbol (or the beginning of a comment), this is generated. This might happen if you try an implied .WORD.

### Invalid assignment

An attempt was made to redefine a label with an = statement.

### Too many labels

More than 10 labels and/or <symbol> = 's appeared on a single statement.

**Invalid op-code**

An op-code mnemonic was not recognized by the assembler.

**Invalid string**

An invalid string for .ASCII or .ASCIZ was encountered. Make sure string is enclosed in double quotes.

**Wrong number of operands**

This is usually a warning. Check the manufacturer's assembly manual for the correct number of operands for the current instruction.

**Line too long**

A statement with more than 255 characters before the newline was encountered.

**Invalid register expression**

Any expression inside parentheses should be absolute and have the value of a register code (<register symbols> are such an expression. This may occur if you use parentheses for anything other than the <register> portion of an operand.

**Offset too large**

The instruction is a relative addressing instruction and the displacement between this instruction and the label specified is too large for the address field of the instruction.

**Odd address**

The previous instruction required an odd number of bytes and this instruction requires word alignment. This error can only follow a .ASCII or a .byte pseudo-operation.

APPENDIX - AS OPCODES

Double Operand Instructions	
addb addw      add addl	
andb andw      and andl	
cmpb cmpw      compare cmpl	
eorb eorw      exclusive or eorl	
movb movmw     move movw movl	
orb orw       inclusive or orl	
subb subw      subtract subl	

Single Operand Instructions	
clrb clrw      clear an operand crl	
nbcd negb negw      negate binary negl	
negxb negxw     negate binary with extend negxl	
notb notw      logical complement notl	
st sf shi sls scc scs sne seq svc svs spl smi sge slt sgt sle	set all ones set all zeros set high set lower or same set carry clear set carry set set not equal set equal set no overflow set on overflow set plus set minus set greater or equal set less than set greater than set less than or equal
tas	test operand then set
tstb tstw tstl	test operand

Branch Instructions	
bcc bccs	branch carry clear
bcs bcss	branch on carry
beq beqs	branch on equal
bge bges	branch greater or equal
bgt bgts	branch greater than
bhi bhis	branch higher
ble bles	branch less than or equal
bis blss	branch lower or same
blt blts	branch less than
bmi bmis	branch minus
bne bnes	branch not equal
bpl bpls	branch positive
bra bras	branch
bsr bsrs	subroutine branch
bvc bvcs	branch no overflow
bvs bvss	branch on overflow

Extended Branch Instructions	
jcc	jump/branch carry clear
jcs	jump/branch on carry
jeq	jump/branch on equal
jge	jump/branch greater or equal
jgt	jump/branch greater than
jhi	jump/branch higher
jle	jump/branch less than or equal
jls	jump/branch lower or same
jlt	jump/branch less than
jmi	jump/branch minus
jne	jump/branch not equal
jpl	jump/branch positive
jra	jump/branch
jbsr	jump/branch to subroutine
jvc	jump/branch no overflow
jvs	jump/branch on overflow

Test Conditions, Decrement and Branch	
dbcc	Decrement and Branch on Carry Clear
dbcs	Decrement and Branch on Carry Set
dbeq	Decrement and Branch on Equal
dbf	Decrement and Branch on False
dbge	Decrement and Branch on Greater Than or Equal
dbgt	Decrement and Branch on Greater Than
dbhi	Decrement and Branch on High
dble	Decrement and Branch on Less Than or Equal
dbls	Decrement and Branch on Low or Same
dblt	Decrement and Branch on Less Than
dbmi	Decrement and Branch on Minus
dbne	Decrement and Branch on Not Equal
dbpl	Decrement and Branch on Plus
dbra	Decrement and Branch Always (same as dbf)
dbt	Decrement and Branch on True
dbvc	Decrement and Branch on Overflow Clear
dbvs	Decrement and Branch on Overflow Set

Shift Instructions	
aslb	arithmetic shift left
aslw	
asll	
asrb	arithmetic shift right
asrw	
asrl	
lslb	logical shift left
lslw	
lsl	
lsrb	logical shift right
lsrw	
lsrl	
rolb	rotate left
rolw	
roll	
rorb	rotate right
rorw	
rorl	
roxb	rotate left with extend
roxlw	
roxll	
roxr	rotate right with extend
roxrw	
roxrl	

Miscellaneous Classes	
abcd	add decimal with extend
addqb	add quick
addqw	
addql	
addxb	add extended
addxw	
addxl	
bchg	test a bit and change
bclr	test a bit and clear
bset	test a bit and set
btst	test a bit
cmpmb	compare memory
cmpmw	
cmpml	
chk	check register against bounds
dcnt	decrement count and branch nz
divs	signed divide
divu	unsigned divide
exg	exchange registers
extw	sign extend
extl	
jmp	jump
jsr	jump to subroutine
lea	load effective address
link	link and allocate

Miscellaneous Classes, continued	
moveml movemw	move multiple registers
movepl movepw	move peripheral
moveq	move quick
mulb	signed multiply
mulu	unsigned multiply
nop	no operation
pea	push effective address
reset	reset machine
rte	return from exception
rtr	return and restore codes
rts	return from subroutine
sbcd	subtract decimal with extend
stop	halt machine
subqb subqw subql	subtract quick
subxb subxw subxl	subtract extended
swap	swap register halves
trap	trap
trapv	trap on overflow
unlk	unlink



Miscellaneous Classes, continued	
movebl	
movew	move multiple registers
movepl	
movepw	move peripheral
moveq	move quick
muls	signed multiply
mulu	unsigned multiply
nop	no operation
pea	push effective address
reset	reset machine
rte	return from exception
rtr	return and restore codes
rts	return from subroutine
sbcd	subtract decimal with extend
stop	halt machine
subqb	
subqw	subtract quick
subql	
subxb	
subxw	subtract extended
subxl	
swap	swap register halves
trap	trap
trapv	trap on overflow
unlk	unlink







"C" INTERFACE NOTES FOR 68000 UNIX.

UNISOFT Corporation

Second Edition, 19 May 1982.

These notes describe the way in which the UNISOFT 68000 "C" programming language represents data in storage, and how that data is passed between functions and subroutines. Also described is the environment of a function, and the calling mechanism for functions.

The information in these notes is intended for programmers who must have detailed knowledge of the interface mechanisms in order to match "C" code with the assembler. It is also intended for those who wish to write new system functions or mathematical functions.

1. GENERAL INFORMATION

When a "C" program is compiled and assembled, the various parts of the program are split out into three parts. These are:

1. The executable code of the program. This is known as the text in UNIX terminology.
2. The initialized data area. This contains literal constants, character strings, and so on.
3. The uninitialized or ".BSS" data areas.

These three data areas are called:

```
.text
.data
.bss
```

These three parts of the program appear in the above order. The compiler/assembler combination produces the first two. The loader actually generates the .bss area at load time.

The .bss area is cleared to zero (0) by the system at load time. This is a feature of the system, and can be relied upon.

During execution of a program, the stack area contains indeterminate data. In other words, its previous contents (if any) cannot be relied upon.

## 2. DATA REPRESENTATIONS

In general, all data elements of whatever size, are stored such that their least significant bit is in the highest addressed byte, and their most significant bit is in the lowest addressed byte. The list below describes the representation of data.

### char

Values of type char occupy 8 bits. Such values can be aligned on any byte boundary.

### short

Values of type short occupy 16 bits. Values of type short are aligned on word (16-bit) address boundaries.

### long

Values of type long occupy 32 bits. A long value is the same as an int value in 68000 "C". Values of this type are aligned on word (16-bit) boundaries.

### float

Values of type float occupy 32 bits. All float values are automatically converted to type double to do arithmetic operations. Values of type float are aligned on word (16-bit) boundaries. A float value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 23 bit mantissa.

### double

Values of type double occupy 64 bits. Values of this type are aligned on word (16-bit) boundaries. A double value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 55 bit mantissa.

### pointers

All pointers are represented as long (32-bit) values. Pointers are aligned on word (16-bit) boundaries.

### arrays

The base address of an array value is always aligned on a word (16-bit) address boundary.

Elements of an array are stored contiguously, one after the other. Elements of multi-dimensional arrays are stored in row-major order. That is, the last dimension of an array varies the fastest.

When a multi-dimensional array is declared, it is possible to omit the size specification for the last dimension. In such a case, what is

allocated is actually an array of pointers to the elements of the last dimension.

### Structures and Unions

Within structures and unions, it is possible to obtain unfilled holes of size char, due to the compiler rounding addresses up to 16-bit boundaries to accomodate word-aligned elements.

This situation can best be demonstrated by an example. Consider the following structure:

```
struct {
 int x; /* This is a 32-bit element */
 char y; /* Takes up a single byte */
 short z; /* Aligned to a 16-bit boundary */
};
```

The total number of bytes declared above is seven: four for the int, one for the char, and two for the short.

In reality, the 'z' field which is a short, will be aligned on a 16-bit boundary by the "C" compiler. In this case, the compiler inserts a hole after the char element, 'y', to align the short element, 'z'. The net effect of these machinations is a structure that behaves like this:

```
struct {
 int x; /* This is a 32-bit element */
 char y; /* Takes up a single byte */
 char dummy; /* Fills the structure */
 short z; /* Aligned to a 16-bit boundary */
};
```

The "C" compiler never reorders any parts of a structure.

Similar considerations apply to arrays of structures or unions. Each element of an array (other than an array of char) begins on a 16-bit boundary.

For a detailed treatment of data storage, consult "The 'C' Programming Language, by Kernighan and Ritchie".



### 3. CALLING MECHANISMS AND PARAMETER PASSING MECHANISMS.

This part of the "C" interface notes describes the interfaces between functions.

#### 3.1. Parameter Passing in C.

The "C" programming language is unique in that it really has only functions. The effect of a subroutine is achieved simply by having a function which does not return a value.

The other unique part of "C" is that parameters to functions are always passed by value. The "C" programming language has no concept of declaring parameters to be passed by reference, as there is in languages such as Pascal. In order to pass a parameter by reference in a "C" program, the programmer must explicitly pass the address of the parameter. The called function must be aware that it is receiving an address instead of a value, and the appropriate code must be present to handle that case.

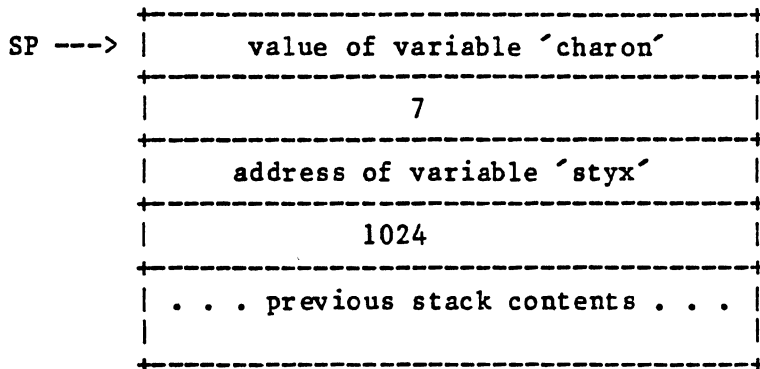
When a function is called, its parameters (if any) are evaluated, and are then pushed onto the stack in reverse order. All parameters are pushed onto the stack as 32-bit longs. If a parameter is shorter than 32 bits, it is expanded to a 32-bit value, with sign-extension if necessary.

The calling procedure is responsible for popping the parameters off the stack.

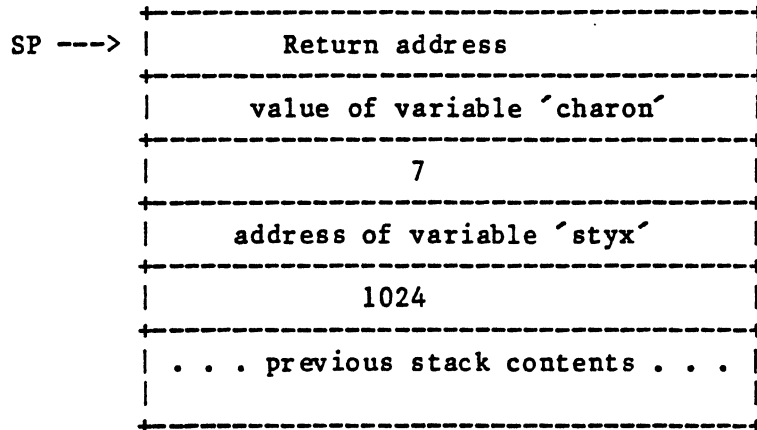
Consider a "C" function call like this:

```
ferry (charon, 7, &styx, 1<<10);
```

After evaluation, but just before the call, the stack looks like this;



Functions are called by issuing either a 'bsr' instruction, or a addressing range or not, and whether the "C" optimizer was used. The and then branches to the indicated function. So, after the call, on entry to the function, the stack looks like this:



In each function, register A6 is used as a stack frame base. The stack location referenced by A6 contains the return address.

### 3.2. Setting up the Stack in the Called Routine.

Upon entry into the function, the prolog code is executed. The prolog code is responsible for allocating enough space on the stack for the local variables, plus enough space to save any registers that this function uses.

Then the prolog code ensures that there is enough stack space available for the function.

If there is not enough space, the system grows the stack to allot more space. The prolog code looks like this:

```
link a6,#-.F1
tstb sp@(-page_size)
moveml #.S1,a6@(-.F1)
```

The '.F1' constant is the size of the stack frame for the local variables, plus four bytes for each register variable.

The 'page\_size' constant is an implementation dependent constant. It is used to probe the stack region at some place below the current stack top. If the probe generates a trap, the system grows the stack by an amount sufficient to include the probe address.

Lastly, the '.S1' constant is a mask to determine which registers need

to be saved on the stack for this particular function. This is of course dependent on the register variables that the programmer declared for that particular routine.

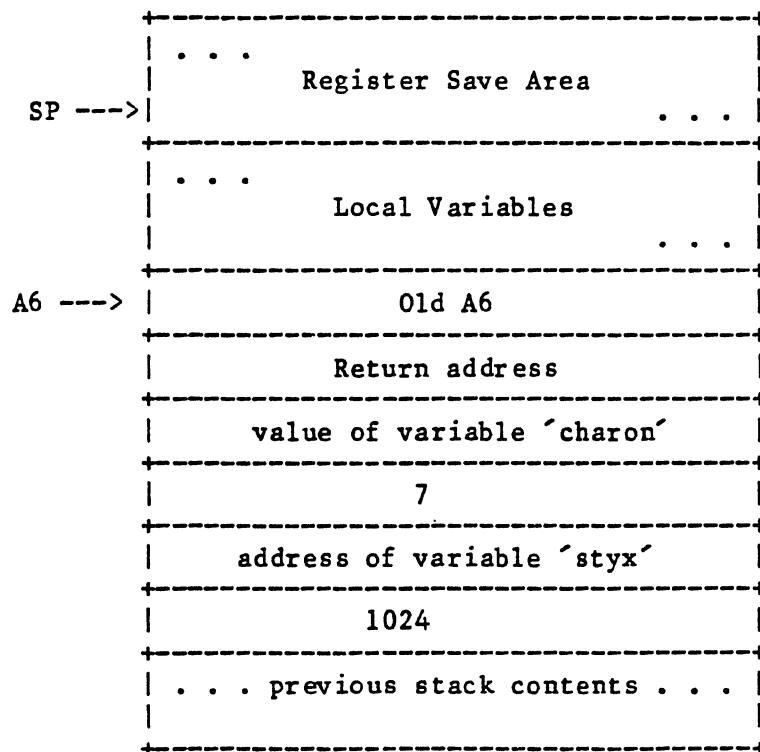
### 3.3. Allocation of Local Variables and Registers.

A total of nine registers are available for register variables. Five of these are the 68000 data (D) registers, and four are the 68000 address (A) registers. The available A registers are A2 through A5. The available D registers are D3 through D7.

Any register variable declared as a pointer variable is always allocated to an address register. Non pointer variables are assigned to data registers. Register variables are allocated to registers in the order in which they are declared in the "C" source program, starting at the high end (A5 or D7) of the appropriate type of register.

If there are more register variables of either kind than there are registers to accomodate them, the remaining variables are allocated on the stack as local variables, just as if the register attribute had never been given in the declaration.

Upon completion of the prolog code, the stack then looks like this:



### 3.4. Returning From a Function or Subroutine.

Upon reaching a 'return' statement, either explicit or implicit, the function executes the epilog code.

If the function has a return value, generated from a

```
return(expression);
```

statement, the value of the expression (which is synonymous with the value of the function) is placed in register D0.

The epilog code is then executed to effect a return from the function:

```
moveml a6@(-.F1),#.S1
unlk a6
rts
```

The 'moveml' instruction restores any registers which were saved during the prolog. The stack frame base pointer in A6 is then put back to the point where A6 once again points to the return address. The function is then exited via the 'rts' instruction, which pops the stack to the state it was in prior to the original call, and then returns to the function that called it.

#### 4. SYSTEM CALLS

This section describes the way in which the C compiler handles system calls.

##### 4.1. Interface to System Calls

The C compiler generates code for system calls in the following way:

1. The system call number is placed in register D0,
2. The first parameter is placed in register A0; the second parameter goes in register D1; the third parameter is placed in register A1; and the fourth parameter is placed in register D2,
3. A 'TRAP #0' instruction is executed.

The C compiler sometimes generates code which uses register D2, so if your code uses D2, you must save it before executing the system call.

On return from the system call, errors are signalled by the carry flag being set.

##### 4.2. System Call Descriptions:

Number	Name	Description
0		Not assigned.
1	exit	Terminate a process.
2	fork	Fork a new process.
3	read	Read from a file.
4	write	Write to a file.
5	open	Open a file for reading or writing.
6	close	Close a file.
7	wait	Wait for a process to terminate.
8	creat	Create a new file.
9	link	Link to a file.
10	unlink	Remove a directory entry.
11	exec	Execute a file.
12	chdir	Change current working directory.
13	time	Get date and time.
14	mknod	Make a directory or a special file.
15	chmod	Change mode of a file.

16	chown	Change owner and group of a file.
17	break	Change memory allocation.
18	stat	Get status of a file.
19	seek	Position in a file.
20	getpid	Get process identification.
21	mount	Mount a file system.
22	umount	Unmount a file system.
23	setuid	Set user ID.
24	getuid	Get user ID.
25	stime	Set time.
26	ptrace	Process trace.
27	alarm	Schedule signal after specified time.
28	fstat	Get file status.
29	pause	Stop until signal.
30	utime	Set file times.
31	stty	Control device.
32	gtty	Control device.
33	access	Determine accessibility of a file.
34	nice	Set program priority.
35	ftime	Get date and time.
36	sync	Update the super block.
37	kill	Send signal to kill a process.
38		Not assigned.
39		Not assigned.
40	tell	Obsolete.
41	dup	Duplicate an open file descriptor.
42	pipe	Create an interprocess channel.
43	times	Get process times.
44	prof	Make execution time profile.
45	locking	File concurrency control.
46	setgid	Set group ID.
47	getgid	Get group ID.
48	sig	Catch or ignore signals.
49		Not assigned.
50		Not assigned.

51	acct	Turn accounting on or off.
52	phys	Map user virtual memory to physical memory.
53	lock	Lock a process in primary memory.
54	ioctl	Control device.
55		Not assigned.
56	mpxchan	Not implemented.
57		Not assigned.
58		Not assigned.
59	exece	Execute a file.
60	umask	Set file creation mode mask.
61		Not assigned.
62		Not assigned.
63		Not assigned.

## 5. OPTIMIZATIONS.

This section describes some of the ways in which the programmer can optimize the use of the "C" language.

The "C" compiler can be run such as to optimize the code it generates, to make that code both compact and fast. Using a "C" command line as follows:

```
cc -O file
```

generates optimized code. The option for optimized code generation is an upper-case 'O'.

If a "C" program contains a 'do' loop of the form:

```
register short x;
x = 10;
do {
 statement
} while (--x != -1);
```

Such a loop is optimized to use the 'dbra' instruction, resulting in faster execution.

### 5.1. Use Of Register Variables

The decision as to whether to use declare a variable in a register depends on the number of times that variable is referenced in the function.

If a variable is used more than twice in a function, it can be declared as a register variable.

If a variable is used less than twice in a function, it is not useful to declare it as a register variable, because the amount of time spent saving and restoring that register is more than the time saved in using a register instead of a location on the stack.



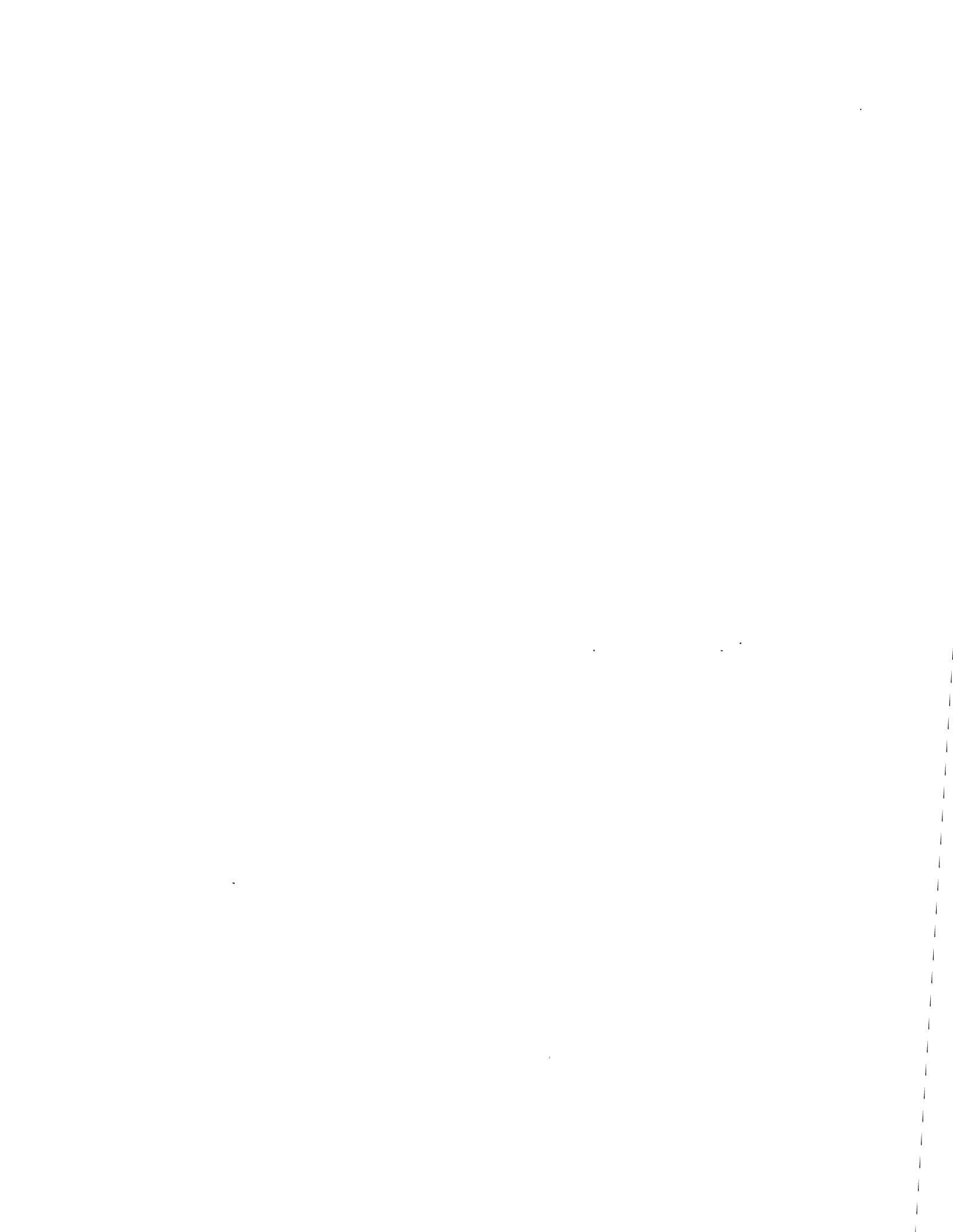
6. REFERENCES.

The C Programming Language.

Brian W. Kernighan and Dennis M. Ritchie.  
Published by Prentice-Hall.

AS Assembler Reference Guide.

UNISOFT Corporation. 27 March 1982.







# The C Programming Language — Reference Manual

Dennis M. Ritchie

Bell Laboratories, Murray Hill, New Jersey

This manual is reprinted, with minor changes, from *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978.

## 1. Introduction

This manual describes the C language on the DEC PDP-11, the DEC VAX-11, the Honeywell 6000, the IBM System/370, and the Interdata 8/32. Where differences exist, it concentrates on the PDP-11, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware: the various compilers are generally quite compatible.

## 2. Lexical conventions

There are six classes of tokens: identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

### 2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`. Comments do not nest.

### 2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore `_` counts as a letter. Upper and lower case letters are different. No more than the first eight characters are significant, although more may be used. External identifiers, which are used by various assemblers and loaders, are more restricted:

DEC PDP-11	7 characters, 2 cases
DEC VAX-11	8 characters, 2 cases
Honeywell 6000	6 characters, 1 case
IBM 360/370	7 characters, 1 case
Interdata 8/32	8 characters, 2 cases

### 2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<code>int</code>	<code>extern</code>	<code>else</code>
<code>char</code>	<code>register</code>	<code>for</code>
<code>float</code>	<code>typedef</code>	<code>do</code>
<code>double</code>	<code>static</code>	<code>while</code>
<code>struct</code>	<code>goto</code>	<code>switch</code>
<code>union</code>	<code>return</code>	<code>case</code>
<code>long</code>	<code>sizeof</code>	<code>default</code>
<code>short</code>	<code>break</code>	<code>entry</code>
<code>unsigned</code>	<code>continue</code>	
<code>auto</code>	<code>if</code>	

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some

† UNIX is a Trademark of Bell Laboratories.

implementations also reserve the words `fortran` and `asm`.

## 2.4 Constants

There are several kinds of constants, as listed below. Hardware characteristics which affect sizes are summarized in §2.6.

### 2.4.1 Integer constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero), decimal otherwise. The digits 8 and 9 have octal value 10 and 11 respectively. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. A decimal constant whose value exceeds the largest signed machine integer is taken to be long; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be long.

### 2.4.2 Explicit long constants

A decimal, octal, or hexadecimal integer constant immediately followed by l (letter ell) or L is a long constant. As discussed below, on some machines integer and long values may be considered identical.

### 2.4.3 Character constants

A character constant is a character enclosed in single quotes, as in 'x'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain non-graphic characters, the single quote ' and the backslash \, may be represented according to the following table of escape sequences:

newline	NL (LF)	\n
horizontal tab	HT	\t
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	<i>ddd</i>	<i>\ddd</i>

The escape `\ddd` consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is `\0` (not followed by a digit), which indicates the character NUL. If the character following a backslash is not one of those specified, the backslash is ignored.

### 2.4.4 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

## 2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of characters" and storage class `static` (see §4 below) and is initialized with the given characters. All strings, even when written identically, are distinct. The compiler places a null byte `\0` at the end of each string so that programs which scan the string can find its end. In a string, the double quote character " must be preceded by a \; in addition, the same escapes as described for character constants may be used. Finally, a \ and an immediately following newline are ignored.

## 2.6 Hardware characteristics

The following table summarizes certain hardware properties which vary from machine to machine. Although these affect program portability, in practice they are less of a problem than might be thought *a priori*.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
<b>char</b>	8 bits	9 bits	8 bits	8 bits
<b>int</b>	16	36	32	32
<b>short</b>	16	36	16	16
<b>long</b>	32	36	32	32
<b>float</b>	32	36	32	32
<b>double</b>	64	72	64	64
<b>range</b>	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 76}$	$\pm 10^{\pm 76}$

The VAX-11 is identical to the PDP-11 except that integers have 32 bits.

### 3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript "opt," so that

( *expression*<sub>opt</sub> )

indicates an optional expression enclosed in braces. The syntax is summarized in §18.

### 4. What's in a name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (§9.2), and are discarded upon exit from the block; static variables are local to a block, but retain their values upon reentry to a block even after control has left the block; external variables exist and retain their values throughout the execution of the entire program, and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (**char**) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a character variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers, or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture; the other sizes are provided to meet special needs.

Unsigned integers, declared **unsigned**, obey the laws of arithmetic modulo  $2^n$  where  $n$  is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (**float**) and double-precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types **char** and **int** of all sizes will collectively be called *integral* types. **float** and **double** will collectively be called *floating* types.

Besides the fundamental arithmetic types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays* of objects of most types;
- functions* which return objects of a given type;
- pointers* to objects of a given type;
- structures* containing a sequence of objects of various types;
- unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

## 5. Objects and lvalues

An *object* is a manipulatable region of storage; an *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if *E* is an expression of pointer type, then *\*E* is an lvalue expression referring to the object to which *E* points. The name "lvalue" comes from the assignment expression *E1 = E2* in which the left operand *E1* must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

## 6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions. §6.6 summarizes the conversions demanded by most ordinary operators; it will be supplemented as required by the discussion of each operator.

### 6.1 Characters and integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer always involves sign extension; integers are signed quantities. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated by this manual, only the PDP-11 sign-extends. On the PDP-11, character variables range in value from -128 to 127; the characters of the ASCII alphabet are all positive. A character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value -1.

When a longer integer is converted to a shorter or to a `char`, it is truncated on the left; excess bits are simply discarded.

### 6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

### 6.3 Floating and integral

Conversions of floating values to integral type tend to be rather machine-dependent; in particular the direction of truncation of negative numbers varies from machine to machine. The result is undefined if the value will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of precision occurs if the destination lacks sufficient bits.

### 6.4 Pointers and integers

An integer or long integer may be added to or subtracted from a pointer; in such a case the first is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

### 6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo  $2^{\text{wordsize}}$ ). In a 2's complement representation, this conversion is conceptual and there is no actual change in the bit pattern.

When an unsigned integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

### 6.6 Arithmetic conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

First, any operands of type `char` or `short` are converted to `int`, and any of type `float` are converted to `double`.



Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.

Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.

Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.

Otherwise, both operands must be `int`, and that is the type of the result.

## 7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of `+` (§7.4) are those expressions defined in §§7.1-7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in the grammar of §18.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects. The order in which side effects take place is unspecified. Expressions involving a commutative and associative operator (`*`, `+`, `&`, `!`, `^`) may be rearranged arbitrarily, even in the presence of parentheses; to force a particular order of evaluation an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is machine-dependent. All existing implementations of C ignore integer overflows; treatment of division by 0, and all floating-point exceptions, varies between machines, and is usually adjustable by a library function.

### 7.1 Primary expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

*primary-expression:*

*identifier*  
*constant*  
*string*  
*( expression )*  
*primary-expression [ expression ]*  
*primary-expression ( expression-list<sub>opt</sub> )*  
*primary-lvalue . identifier*  
*primary-expression -> identifier*

*expression-list:*

*expression*  
*expression-list , expression*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of ...", however, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is "pointer to ...". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning ...", when used except in the function-name position of a call, is converted to "pointer to function returning ...".

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int`; floating constants are `double`.

A string is a primary expression. Its type is originally "array of `char`"; but following the same rule given above for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see §8.6.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is `int`, and the type of the result is "...". The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1, 7.2, and 7.4 on identifiers, `*`, and `+` respectively; §14.3 below summarizes the implications.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` or `short` are converted to `int`; and as usual, array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see §7.2, 8.7.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. On the other hand, it is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ.

Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be an lvalue naming a structure or a union, and the identifier must name a member of the structure or union. The result is an lvalue referring to the named member of the structure or union.

A primary expression followed by an arrow (built from a `-` and a `>`) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points.

Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in §8.5. The rules given here for the use of structures and unions are not enforced strictly, in order to allow an escape from the typing mechanism. See §14.1.

## 7.2 Unary operators

Expressions with unary operators group right-to-left.

```

unary-expression:
 * expression
 & lvalue
 - expression
 ! expression
 ~ expression
 ++ lvalue
 -- lvalue
 lvalue ++
 lvalue --
 (type-name) expression
 sizeof expression
 sizeof (type-name)

```

The unary `*` operator means *indirection*: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...", the type of the result is "...".

The result of the unary `&` operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary `-` operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from  $2^n$ , where  $n$  is the number of bits in an `int`. There is no unary `+` operator.

The result of the logical negation operator `!` is 1 if the value of its operand is 0, 0 if the value of its operand is non-zero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The `~` operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix `++` is incremented. The value is the new value of the operand, but is not an lvalue. The expression `++x` is equivalent to `x+=1`. See the discussions of addition (§7.4) and assignment operators (§7.14) for information on conversions.

The lvalue operand of prefix `--` is decremented analogously to the prefix `++` operator.

When postfix `++` is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix `++` operator. The type of the result is the same as the type of the lvalue expression.

When postfix `--` is applied to an lvalue the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix `--` operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in §8.7.

The `sizeof` operator yields the size, in bytes, of its operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size, in bytes, of an object of the indicated type.

The construction `sizeof(type)` is taken to be a unit, so the expression `sizeof(type)-2` is the same as `(sizeof(type))-2`.

### 7.3 Multiplicative operators

The multiplicative operators `*`, `/`, and `%` group left-to-right. The usual arithmetic conversions are performed.

*multiplicative-expression:*  
`expression * expression`  
`expression / expression`  
`expression % expression`

The binary `*` operator indicates multiplication. The `*` operator is associative and expressions with several multiplications at the same level may be rearranged by the compiler.

The binary `/` operator indicates division. When positive integers are divided truncation is toward 0, but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that  $(a/b)*b + a\%b$  is equal to  $a$  (if  $b$  is not 0).

The binary `%` operator yields the remainder from the division of the first expression by the second. The usual arithmetic conversions are performed. The operands must not be `float`.

### 7.4 Additive operators

The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

*additive-expression:*  
`expression + expression`  
`expression - expression`

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer, and which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array.

No further type combinations are allowed for pointers.

The `+` operator is associative and expressions with several additions at the same level may be rearranged by the compiler.

The result of the `-` operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions as for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same

array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

### 7.5 Shift operators

The shift operators `<<` and `>>` group left-to-right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative, or greater than or equal to the length of the object in bits.

*shift-expression:*  
`expression << expression`  
`expression >> expression`

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits; vacated bits are 0-filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0-fill) if `E1` is unsigned; otherwise it may be (and is, on the PDP-11) arithmetic (fill by a copy of the sign bit).

### 7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; `a<b<c` does not mean what it seems to.

*relational-expression:*  
`expression < expression`  
`expression > expression`  
`expression <= expression`  
`expression >= expression`

The operators `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

### 7.7 Equality operators

*equality-expression:*  
`expression == expression`  
`expression != expression`

The `==` (equal to) and the `!=` (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value).

A pointer may be compared to an integer, but the result is machine dependent unless the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object, and will appear to be equal to 0; in conventional usage, such a pointer is considered to be null.

### 7.8 Bitwise AND operator

*and-expression:*  
`expression & expression`

The `&` operator is associative and expressions involving `&` may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral operands.

### 7.9 Bitwise exclusive OR operator

*exclusive-or-expression:*  
`expression ^ expression`

The `^` operator is associative and expressions involving `^` may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

### 7.10 Bitwise inclusive OR operator

*inclusive-or-expression:*  
*expression | expression*

The `|` operator is associative and expressions involving `|` may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

### 7.11 Logical AND operator

*logical-and-expression:*  
*expression && expression*

The `&&` operator groups left-to-right. It returns 1 if both its operands are non-zero, 0 otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

### 7.12 Logical OR operator

*logical-or-expression:*  
*expression || expression*

The `||` operator groups left-to-right. It returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike `|`, `||` guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

### 7.13 Conditional operator

*conditional-expression:*  
*expression ? expression : expression*

Conditional expressions group right-to-left. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type; otherwise, if both are pointers of the same type, the result has the common type; otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

### 7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

*assignment-expression:*  
*lvalue = expression*  
*lvalue += expression*  
*lvalue -= expression*  
*lvalue \*= expression*  
*lvalue /= expression*  
*lvalue %= expression*  
*lvalue >>= expression*  
*lvalue <<= expression*  
*lvalue &= expression*  
*lvalue ^= expression*  
*lvalue |= expression*

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left

preparatory to the assignment.

The behavior of an expression of the form  $E1 \text{ op} = E2$  may be inferred by taking it as equivalent to  $E1 = E1 \text{ op} (E2)$ ; however,  $E1$  is evaluated only once. In  $+=$  and  $-=$ , the left operand may be a pointer, in which case the (integral) right operand is converted as explained in §7.4; all right operands and all non-pointer left operands must have arithmetic type.

The compilers currently allow a pointer to be assigned to an integer, an integer to a pointer, and a pointer to a pointer of another type. The assignment is a pure copy operation, with no conversion. This usage is nonportable, and may produce pointers which cause addressing exceptions when used. However, it is guaranteed that assignment of the constant 0 to a pointer will produce a null pointer distinguishable from a pointer to any object.

### 7.15 Comma operator

*comma-expression:*  
*expression , expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. In contexts where comma is given a special meaning, for example in a list of actual arguments to functions (§7.1) and lists of initializers (§8.6), the comma operator as described in this section can only appear in parentheses; for example,

$f(a, (t=3, t+2), c)$

has three arguments, the second of which has the value 5.

## 8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

*declaration:*  
*decl-specifiers declarator-list<sub>opt</sub> ;*

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

*decl-specifiers:*  
*type-specifier decl-specifiers<sub>opt</sub>*  
*sc-specifier decl-specifiers<sub>opt</sub>*

The list must be self-consistent in a way described below.

### 8.1 Storage class specifiers

The sc-specifiers are:

*sc-specifier:*  
**auto**  
**static**  
**extern**  
**register**  
**typedef**

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience; it is discussed in §8.8. The meanings of the various storage classes were discussed in §4.

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case there must be an external definition (§10) for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations are effective. Moreover, only variables of certain types will be stored in registers: on the PDP-11, they are **int**, **char**, or **pointer**. One other restriction applies to register variables: the address-of operator **&** cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most one *sc-specifier* may be given in a declaration. If the *sc-specifier* is missing from a declaration, it is taken to be *auto* inside a function, *extern* outside. Exception: functions are never automatic.

## 8.2 Type specifiers

The type-specifiers are

```
type-specifier:
char
short
int
long
unsigned
float
double
struct-or-union-specifier
typedef-name
```

The words *long*, *short*, and *unsigned* may be thought of as adjectives; the following combinations are acceptable.

```
short int
long int
unsigned int
long float
```

The meaning of the last is the same as *double*. Otherwise, at most one type-specifier may be given in a declaration. If the type-specifier is missing from a declaration, it is taken to be *int*.

Specifiers for structures and unions are discussed in §8.5; declarations with *typedef* names are discussed in §8.8.

## 8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

```
declarator-list:
init-declarator
init-declarator , declarator-list
```

```
init-declarator:
declarator initializeropt
```

Initializers are discussed in §8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

```
declarator:
identifier
(declarator)
* declarator
declarator ()
declarator [constant-expressionopt]
```

The grouping is the same as in expressions.

## 8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier: it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

**T D1**

where **T** is a type-specifier (like `int`, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type "... **T**," where the "..." is empty if **D1** is just a plain identifier (so that the type of **x** in "`int x`" is just `int`). Then if **D1** has the form

**\*D**

the type of the contained identifier is "... pointer to **T**."

If **D1** has the form

**D()**

then the contained identifier has the type "... function returning **T**."

If **D1** has the form

**D[constant-expression]**

or

**D[]**

then the contained identifier has type "... array of **T**." In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is `int`. (Constant expressions are defined precisely in §15.) When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant-expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures, unions or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure or union may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer `i`, a pointer `ip` to an integer, a function `f` returning an integer, a function `fip` returning a pointer to an integer, and a pointer `pfi` to a function which returns an integer. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the (pointer) result to yield an integer. In the declarator `(*pfi)()`, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank  $3 \times 5 \times 7$ . In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression. The first three have type "array," the last has type `int`.

### 8.5 Structure and union declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.



*struct-or-union-specifier:*  
*struct-or-union ( struct-decl-list )*  
*struct-or-union identifier ( struct-decl-list )*  
*struct-or-union identifier*

*struct-or-union:*  
*struct*  
*union*

The struct-decl-list is a sequence of declarations for the members of the structure or union:

*struct-decl-list:*  
*struct-declaration*  
*struct-declaration struct-decl-list*

*struct-declaration:*  
*type-specifier struct-declarator-list ;*

*struct-declarator-list:*  
*struct-declarator*  
*struct-declarator , struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field*; its length is set off from the field name by a colon.

*struct-declarator:*  
*declarator*  
*declarator : constant-expression*  
*: constant-expression*

Within a structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. Fields are assigned right-to-left on the PDP-11, left-to-right on other machines.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, an unnamed field with a width of 0 specifies alignment of the next field at a word boundary. The "next field" presumably is a field, not an ordinary structure member, because in the latter case the alignment would have been automatic.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values. In all implementations, there are no arrays of fields, and the address-of operator `&` may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

*struct identifier ( struct-decl-list )*  
*union identifier ( struct-decl-list )*

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

*struct identifier*  
*union identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The names of members and tags may be the same as ordinary variables. However, names of tags and members must be mutually distinct.

Two structures may share a common initial sequence of members; that is, the same member may appear in two different structures if it has the same type in both and if all previous members are the same in both. (Actually, the compiler checks only that a name in two different structures has the same type and offset in both, but if preceding members differ the construction is nonportable.)

A simple example of a structure declaration is

```
struct tnode {
 char tword[20];
 int count;
 struct tnode *left;
 struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the *count* field of the structure to which *sp* points;

```
s.left
```

refers to the left subtree pointer of the structure *s*; and

```
s.right->tword[0]
```

refers to the first character of the *tword* member of the right subtree of *s*.

## 8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by =, and consists of an expression or a list of values nested in braces.

*initializer:*

- = *expression*
- = { *initializer-list* }
- = { *initializer-list* , }

*initializer-list:*

- expression*
- initializer-list* , *initializer-list*
- { *initializer-list* }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in §15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants, and previously declared variables and functions.

Static and external variables which are not initialized are guaranteed to start off as 0; automatic and register variables which are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array) then the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate, written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with 0's. It is not permitted to initialize unions or automatic aggregates.

Braces may be elided as follows. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a `char` array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a 1-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] = (
 { 1, 3, 5 },
 { 2, 4, 6 },
 { 3, 5, 7 },
);
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with 0. Precisely the same effect could have been achieved by

```
float y[4][3] = (
 1, 3, 5, 2, 4, 6, 3, 5, 7
);
```

The initializer for `y` begins with a left brace, but that for `y[0]` does not, therefore 3 elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float y[4][3] = (
 { 1 }, { 2 }, { 3 }, { 4 }
);
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

### 8.7 Type names

In two contexts (to specify type conversions explicitly by means of a cast, and as an argument of `sizeof`) it is desired to supply the name of a data type. This is accomplished using a "type name," which in essence is a declaration for an object of that type which omits the name of the object.

*type-name:*

*type-specifier abstract-declarator*

*abstract-declarator:*

*empty*

*( abstract-declarator )*

*\* abstract-declarator*

*abstract-declarator ( )*

*abstract-declarator [ constant-expression<sub>opt</sub> ]*

To avoid ambiguity, in the construction

*( abstract-declarator )*

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
```

name respectively the types "integer," "pointer to integer," "array of 3 pointers to integers," "pointer to an array of 3 integers," "function returning pointer to integer," and "pointer to function returning an integer."

### 8.8 Typedef

Declarations whose "storage class" is `typedef` do not define storage, but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

*typedef-name:*  
*identifier*

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein become syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in §8.4. For example, after

```
typedef int MILES, *KCLICKSP;
typedef struct (double re, im;) complex;
```

the constructions

```
MILES distance;
extern KCLICKSP metricp;
complex z, *zp;
```

are all legal declarations: the type of `distance` is `int`, that of `metricp` is "pointer to `int`," and that of `z` is the specified structure. `zp` is a pointer to such a structure.

`typedef` does not introduce brand new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

## 9. Statements

Except as indicated, statements are executed in sequence.

### 9.1 Expression statement

Most statements are expression statements, which have the form

*expression* ;

Usually expression statements are assignments or function calls.

### 9.2 Compound statement, or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

*compound-statement:*  
( *declaration-list*<sub>opt</sub> *statement-list*<sub>opt</sub> )

*declaration-list:*  
*declaration*  
*declaration declaration-list*

*statement-list:*  
*statement*  
*statement statement-list*

If any of the identifiers in the `declaration-list` were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

### 9.3 Conditional statement

The two forms of the conditional statement are

```
if (expression) statement
if (expression) statement else statement
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered `else-less if`.

### 9.4 While statement

The while statement has the form

```
while (expression) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

### 9.5 Do statement

The do statement has the form

```
do statement while (expression) ;
```

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

### 9.6 For statement

The for statement has the form

```
for (expression-1opt ; expression-2opt ; expression-3opt) statement
```

This statement is equivalent to

```
expression-1 ;
while (expression-2) {
 statement
 expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression often specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing `expression-2` makes the implied `while` clause equivalent to `while (1)`; other missing expressions are simply dropped from the expansion above.

### 9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

```
switch (expression) statement
```

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

```
case constant-expression :
```

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

**default :**

When the **switch** statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a default prefix, control passes to the prefixed statement. If no case matches and if there is no default then none of the statements in the switch is executed.

**case** and **default** prefixes in themselves do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see **break**, §9.8.

Usually the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

### 9.8 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing **while**, **do**, **for**, or **switch** statement; control passes to the statement following the terminated statement.

### 9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...) { do (for (...) (

 contin: ; contin: ; contin: ;
)) while (...);)
```

a **continue** is equivalent to **goto contin**. (Following the **contin: ;** is a null statement, §9.13.)

### 9.10 Return statement

A function returns to its caller by means of the **return** statement, which has one of the forms

```
return ;
return expression ;
```

In the first case the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

### 9.11 Goto statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (§9.12) located in the current function.

### 9.12 Labeled statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been redeclared. See §11.

### 9.13 Null statement

The null statement has the form

;

A null statement is useful to carry a label just before the ) of a compound statement or to supply a null body to a looping statement such as while.

## 10. External definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (§8.2) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations, except that only at this level may the code for functions be given.

### 10.1 External function definitions

Function definitions have the form

*function-definition:*  
*decl-specifiers<sub>opt</sub> function-declarator function-body*

The only `sc`-specifiers allowed among the `decl-specifiers` are `extern` or `static`; see §11.2 for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

*function-declarator:*  
*declarator ( parameter-list<sub>opt</sub> )*

*parameter-list:*  
*identifier*  
*identifier , parameter-list*

The function-body has the form

*function-body:*  
*declaration-list compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
 int m;

 m = (a > b) ? a : b;
 return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; `{ ... }` is the block giving the code for the statement.

C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...". Finally, because structures, unions and functions cannot be passed to a function, it is useless to declare a formal parameter to be a structure, union or function (pointers to such objects are of course permitted).

## 10.2 External data definitions

An external data definition has the form

*data-definition:*  
*declaration*

The storage class of such data may be `extern` (which is the default) or `static`, but not `auto` or `register`.

## 11. Scope rules

A C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

### 11.1 Lexical scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of blocks persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

Because all references to the same external identifier refer to the same object (see §11.2) the compiler checks all declarations of the same external identifier for compatibility; in effect their scope is increased to the whole file in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (§8.5) that identifiers associated with ordinary variables on the one hand and those associated with structure and union members and tags on the other form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
 auto int distance;
 ...
}
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance†`.

### 11.2 Scope of externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

The appearance of the `extern` keyword in an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the declaration where storage is allocated.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

---

†It is agreed that the ice is thin here.



## 12. Compiler control lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

### 12.1 Token replacement

A compiler-control line of the form

```
#define identifier token-string
```

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. A line of the form

```
#define identifier(identifier , ... , identifier) token-string
```

where there is no space between the first identifier and the (, is a macro definition with arguments. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a ) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Text inside a string or a character constant is not subject to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100
```

```
int table[TABSIZE];
```

A control line of the form

```
#undef identifier
```

causes the identifier's preprocessor definition to be forgotten.

### 12.2 File inclusion

A compiler control line of the form

```
#include "filename"
```

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the original source file, and then in a sequence of standard places. Alternatively, a control line of the form

```
#include <filename>
```

searches only the standard places, and not the directory of the source file.

#include's may be nested.

### 12.3 Conditional compilation

A compiler control line of the form

```
#if constant-expression
```

checks whether the constant expression (see §15) evaluates to non-zero. A control line of the form

```
#ifdef identifier
```

checks whether the identifier is currently defined in the preprocessor: that is, whether it has been the subject of a #define control line. A control line of the form

```
#ifndef identifier
```

checks whether the identifier is currently undefined in the preprocessor.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
#else
```

and then by a control line

```
#endif
```

If the checked condition is true then any lines between `#else` and `#endif` are ignored. If the checked condition is false then any lines between the test and an `#else` or, lacking an `#else`, the `#endif`, are ignored.

These constructions may be nested.

#### 12.4 Line control

For the benefit of other preprocessors which generate C programs, a line of the form

```
#line constant identifier
```

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent the remembered file name does not change.

#### 13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack); if the type of an identifier is "function returning ...", it is implicitly declared to be `extern`.

In an expression, an identifier followed by `(` and not already declared is contextually declared to be "function returning `int`".

#### 14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

##### 14.1 Structures and unions

There are only two things that can be done with a structure or union: name one of its members (by means of the `.` operator); or take its address (by unary `&`). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

§7.1 says that in a direct or indirect structure reference (with `.` or `->`) the name on the right must be a member of the structure named or pointed to by the expression on the left. To allow an escape from the typing rules, this restriction is not firmly enforced by the compiler. In fact, any lvalue is allowed before `.`, and that lvalue is then assumed to have the form of the structure of which the name on the right is a member. Also, the expression before a `->` is required only to be a pointer or an integer. If a pointer, it is assumed to point to a structure of which the name on the right is a member. If an integer, it is taken to be the absolute address, in machine storage units, of the appropriate structure.

Such constructions are non-portable.

##### 14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of `g` might read

```
g(funcp)
int (*funcp)();
{
 ...
 (*funcp)();
 ...
}
```

Notice that  $f$  must be declared explicitly in the calling routine since its appearance in  $g(f)$  was not followed by  $($ .

### 14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator  $[]$  is interpreted in such a way that  $E1[E2]$  is identical to  $*(E1+(E2))$ . Because of the conversion rules which apply to  $+$ , if  $E1$  is an array and  $E2$  an integer, then  $E1[E2]$  refers to the  $E2$ -th member of  $E1$ . Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If  $E$  is an  $n$ -dimensional array of rank  $i \times j \times \dots \times k$ , then  $E$  appearing in an expression is converted to a pointer to an  $(n-1)$ -dimensional array with rank  $j \times \dots \times k$ . If the  $*$  operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to  $(n-1)$ -dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here  $x$  is a  $3 \times 5$  array of integers. When  $x$  appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression  $x[i]$ , which is equivalent to  $*(x+i)$ ,  $x$  is first converted to a pointer as described; then  $i$  is converted to the type of  $x$ , which involves multiplying  $i$  by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

### 14.4 Explicit pointer conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, §§7.2 and 8.7.

A pointer may be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine dependent. The mapping function is also machine dependent, but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

`alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`: then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and is measured in bytes. `chars` have no alignment requirements; everything else must have an even address.

On the Honeywell 6000, a pointer corresponds to a 36-bit integer; the word part is in the left 18 bits, and the two bits that select the character in a word just to their right. Thus `char` pointers are measured in units of  $2^{16}$  bytes; everything else is measured in units of  $2^{18}$  machine words. `double` quantities and aggregates containing them must lie on an even word address ( $0 \bmod 2^{19}$ ).

The IBM 370 and the Interdata 8/32 are similar. On both, addresses are measured in bytes; elementary objects must be aligned on a boundary equal to their length, so pointers to `short` must be  $0 \bmod 2$ , to `int` and `float`  $0 \bmod 4$ , and to `double`  $0 \bmod 8$ . Aggregates are aligned on the strictest boundary required by any of their constituents.

### 15. Constant expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

`+ - * / % & | ^ << >> == != < > <= >=`

or by the unary operators

`-`

or by the ternary operator

`?:`

Parentheses can be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external or static objects, and to external or static arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

### 16. Portability considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive, but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are a nuisance that must be carefully watched. Most of the others are only minor problems.

The number of `register` variables that can actually be placed in registers varies from machine to machine, as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. It is right to left on the PDP-11, and VAX-11, left to right on the others. The order in which side effects take place is also unspecified.

Since character constants are really objects of type `int`, multi-character character constants may be permitted. The specific implementation is very machine dependent, however, because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right-to-left on the PDP-11 and VAX-11 and left-to-right on other machines. These differences are invisible to isolated programs which do not indulge in type punning (for example, by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage), but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bit-fields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

### 17. Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form `=op` instead of `op=` for assignment operators. This leads to ambiguities, typified by

```
x=-1
```

which actually decrements `x` since the `=` and the `-` are adjacent, but which might easily be intended to assign `-1` to `x`.

The syntax of initializers has changed: previously, the equals sign that introduces an initializer was not present, so instead of

```
int x = 1;
```

one used

```
int x 1;
```

The change was made because the initialization

```
int f (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

## 18. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

### 18.1 Expressions

The basic expressions are:

*expression:*  
primary  
\* *expression*  
& *expression*  
- *expression*  
! *expression*  
~ *expression*  
++ *lvalue*  
-- *lvalue*  
*lvalue* ++  
*lvalue* --  
sizeof *expression*  
( *type-name* ) *expression*  
*expression* *binop* *expression*  
*expression* ? *expression* : *expression*  
*lvalue* *asgnop* *expression*  
*expression* , *expression*

*primary:*  
*identifier*  
*constant*  
*string*  
( *expression* )  
*primary* ( *expression-list*<sub>opt</sub> )  
*primary* [ *expression* ]  
*lvalue* . *identifier*  
*primary* -> *identifier*

*lvalue:*  
*identifier*  
*primary* [ *expression* ]  
*lvalue* . *identifier*  
*primary* -> *identifier*  
\* *expression*  
( *lvalue* )

The primary-expression operators

( ) [ ] . ->

have highest priority and group left-to-right. The unary operators

\* & - ! ~ ++ -- sizeof ( *type-name* )

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators group left-to-right; they have priority decreasing as indicated below. The conditional operator groups right to left.

*binop:*  
\* / %  
+ -  
>> <<  
< > <= >=  
== !=  
&  
^  
|  
&&  
||  
?:

Assignment operators all have the same priority, and all group right-to-left.

*asgnop:*  
= += -= \*= /= %= >>= <<= &= ^= |=

The comma operator has the lowest priority, and groups left-to-right.

## 18.2 Declarations

*declaration:*  
*decl-specifiers* *init-declarator-list*<sub>opt</sub> ;

*decl-specifiers:*  
*type-specifier* *decl-specifiers*<sub>opt</sub>  
*sc-specifier* *decl-specifiers*<sub>opt</sub>

*sc-specifier:*  
auto  
static  
extern  
register  
typedef

*type-specifier:*  
char  
short  
int  
long  
unsigned  
float  
double  
*struct-or-union-specifier*  
*typedef-name*

*init-declarator-list:*  
*init-declarator*  
*init-declarator* , *init-declarator-list*

*init-declarator:*  
*declarator* *initializer*<sub>opt</sub>

*declarator:*  
*identifier*  
( *declarator* )  
\* *declarator*  
*declarator* ( )  
*declarator* [ *constant-expression*<sub>opt</sub> ]

*struct-or-union-specifier:*

*struct* ( *struct-decl-list* )  
*struct* *identifier* ( *struct-decl-list* )  
*struct* *identifier*  
*union* ( *struct-decl-list* )  
*union* *identifier* ( *struct-decl-list* )  
*union* *identifier*

*struct-decl-list:*

*struct-declaration*  
*struct-declaration struct-decl-list*

*struct-declaration:*

*type-specifier struct-declarator-list ;*

*struct-declarator-list:*

*struct-declarator*  
*struct-declarator , struct-declarator-list*

*struct-declarator:*

*declarator*  
*declarator : constant-expression*  
*: constant-expression*

*initializer:*

*= expression*  
*= ( initializer-list )*  
*= ( initializer-list , )*

*initializer-list:*

*expression*  
*initializer-list , initializer-list*  
*( initializer-list )*

*type-name:*

*type-specifier abstract-declarator*

*abstract-declarator:*

*empty*  
*( abstract-declarator )*  
*\* abstract-declarator*  
*abstract-declarator ( )*  
*abstract-declarator [ constant-expression<sub>opt</sub> ]*

*typedef-name:*

*identifier*

### 18.3 Statements

*compound-statement:*

*( declaration-list<sub>opt</sub> statement-list<sub>opt</sub> )*

*declaration-list:*

*declaration*  
*declaration declaration-list*



*statement-list:*  
    *statement*  
    *statement statement-list*

*statement:*  
    *compound-statement*  
    *expression ;*  
    *if ( expression ) statement*  
    *if ( expression ) statement else statement*  
    *while ( expression ) statement*  
    *do statement while ( expression ) ;*  
    *for ( expression-1<sub>opt</sub> ; expression-2<sub>opt</sub> ; expression-3<sub>opt</sub> ) statement*  
    *switch ( expression ) statement*  
    *case constant-expression : statement*  
    *default : statement*  
    *break ;*  
    *continue ;*  
    *return ;*  
    *return expression ;*  
    *goto identifier ;*  
    *identifier : statement*  
    *i*

#### 18.4 External definitions

*program:*  
    *external-definition*  
    *external-definition program*

*external-definition:*  
    *function-definition*  
    *data-definition*

*function-definition:*  
    *type-specifier<sub>opt</sub> function-declarator function-body*

*function-declarator:*  
    *declarator ( parameter-list<sub>opt</sub> )*

*parameter-list:*  
    *identifier*  
    *identifier , parameter-list*

*function-body:*  
    *type-decl-list function-statement*

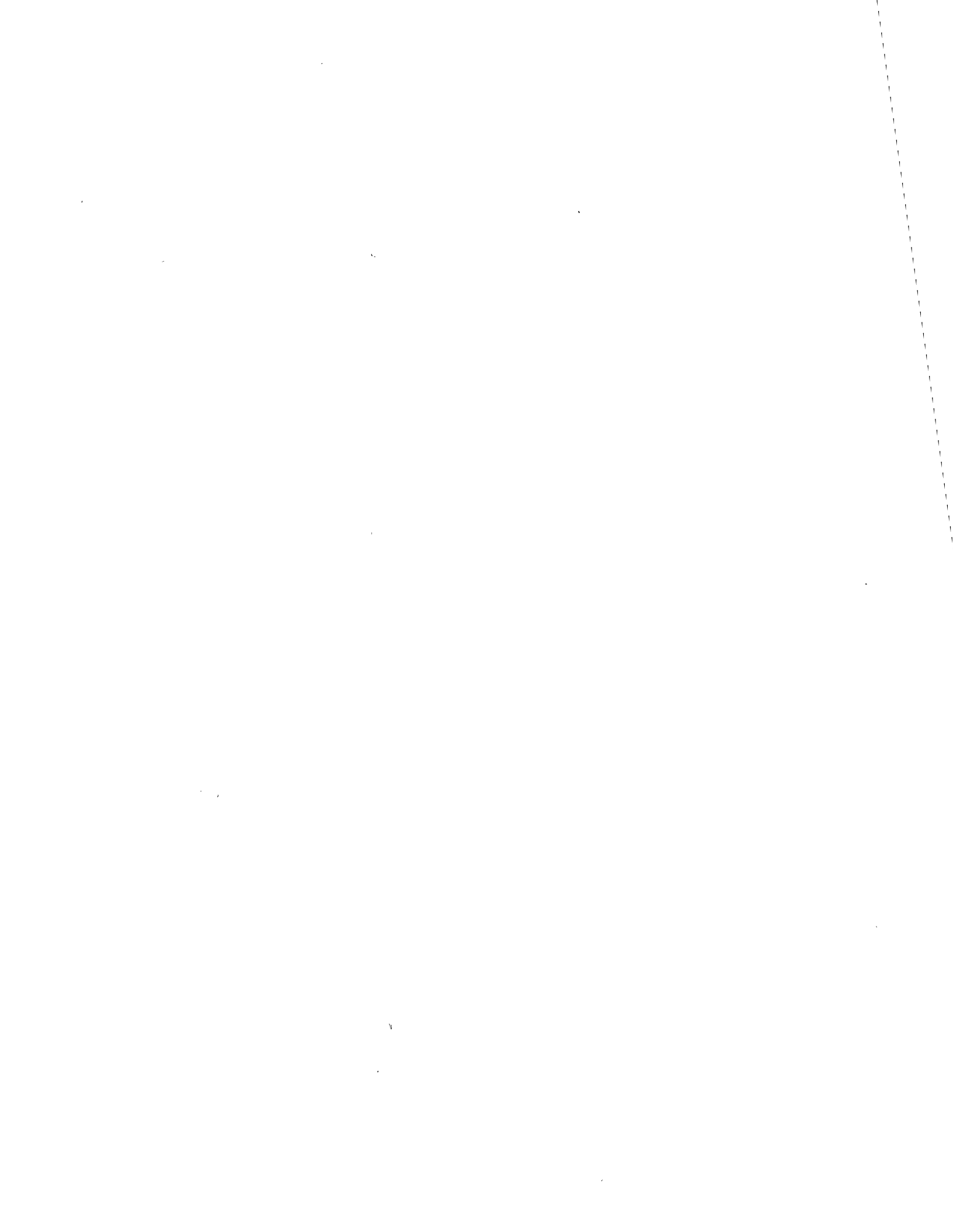
*function-statement:*  
    *{ declaration-list<sub>opt</sub> statement-list }*

*data-definition:*  
    *extern<sub>opt</sub> type-specifier<sub>opt</sub> init-declarator-list<sub>opt</sub> ;*  
    *static<sub>opt</sub> type-specifier<sub>opt</sub> init-declarator-list<sub>opt</sub> ;*

#### 18.5 Preprocessor

```
#define identifier token-string
#define identifier(identifier , ... , identifier) token-string
#undef identifier
#include "filename"
#include <filename>
#if constant-expression
#ifdef identifier
#ifdef identifier
#else
#endif
#endif
#endif
#line constant identifier
```





## Recent Changes to C

November 15, 1978

A few extensions have been made to the C language beyond what is described in the reference document ("The C Programming Language," Kernighan and Ritchie, Prentice-Hall, 1978).

### 1. Structure assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same. Other plausible operators, such as equality comparison, have not been implemented.

There is a subtle defect in the PDP-11 implementation of functions that return structures: if an interrupt occurs during the return sequence, and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of true interrupts, as in an operating system or a user program that makes significant use of signals; ordinary recursive calls are quite safe.

### 2. Enumeration type

There is a new data type analogous to the scalar types of Pascal. To the type-specifiers in the syntax on p. 193 of the C book add

*enum-specifier*

with syntax

```
enum-specifier:
enum { enum-list }
enum identifier { enum-list }
enum identifier
```

```
enum-list:
enumerator
enum-list , enumerator
```

```
enumerator:
identifier
identifier = constant-expression
```

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret, winedark };
...
enum color *cp, col;
```

makes `color` the enumeration-tag of a type describing various colors, and then declares `cp` as a pointer to an object of that type, and `col` as an object of that type.

The identifiers in the enum-list are declared as constants, and may appear wherever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types, and *lint* flags type mismatches. In the PDP-11 implementation all enumeration variables are treated as if they were `int`.

# Screen Updating and Cursor Movement Optimization: A Library Package

*Kenneth C. R. C. Arnold*

Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley  
Berkeley, California 94720

## ***ABSTRACT***

This document describes a package of C library functions which allow the user to:

- update a screen with reasonable optimization,
- get input from the terminal in a screen-oriented fashion, and
- independent from the above, move the cursor optimally from one point to another.

These routines all use the `/etc/termcap` database to describe the capabilities of the terminal.

### **Acknowledgements**

This package would not exist without the work of Bill Joy, who, in writing his editor, created the capability to generally describe terminals, wrote the routines which read this database, and, most importantly, those which implement optimal cursor movement, which routines I have simply lifted nearly intact. Doug Merritt and Kurt Shoens also were extremely important, as were both willing to waste time listening to me rant and rave. The help and/or support of Ken Abrams, Alan Char, Mark Horton, and Joe Kalash, was, and is, also greatly appreciated.

# Screen Package

## Contents

1 Overview .....	1
1.1 Terminology (or, Words You Can Say to Sound Brilliant) .....	1
1.2 Compiling Things .....	1
1.3 Screen Updating .....	1
1.4 Naming Conventions .....	2
2 Variables .....	2
3 Usage .....	3
3.1 Starting up .....	3
3.2 The Nitty-Gritty .....	3
3.2.1 Output .....	3
3.2.2 Input .....	4
3.2.3 Miscellaneous .....	4
3.3 Finishing up .....	4
4 Cursor Motion Optimization: Standing Alone .....	4
4.1 Terminal Information .....	4
4.2 Movement Optimizations, or, Getting Over Yonder .....	5
5 The Functions .....	6
5.1 Output Functions .....	6
5.2 Input Functions .....	9
5.3 Miscellaneous Functions .....	10
5.4 Details .....	13

## Appendices

<b>Appendix A</b> .....	14
1 Capabilities from termcap .....	14
1.1 Disclaimer .....	14
1.2 Overview .....	14
1.3 Variables Set By setterm() .....	14
1.4 Variables Set By gettmode() .....	15
<b>Appendix B</b> .....	16
1 The WINDOW structure .....	16
<b>Appendix C</b> .....	17
1 Examples .....	17
2 Screen Updating .....	17
2.1 Twinkle .....	17
2.2 Life .....	19
3 Motion optimization .....	22
3.1 Twinkle .....	22

## Screen Package

### 1. Overview

In making available the generalized terminal descriptions in `/etc/termcap`, much information was made available to the programmer, but little work was taken out of one's hands. The purpose of this package is to allow the C programmer to do the most common type of terminal dependent functions, those of movement optimization and optimal screen updating, without doing any of the dirty work, and (hopefully) with nearly as much ease as is necessary to simply print or read things.

The package is split into three parts: (1) Screen updating; (2) Screen updating with user input; and (3) Cursor motion optimization.

It is possible to use the motion optimization without using either of the other two, and screen updating and input can be done without any programmer knowledge of the motion optimization, or indeed the database itself.

#### 1.1. Terminology (or, Words You Can Say to Sound Brilliant)

In this document, the following terminology is kept to with reasonable consistency:

*window*: An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

*terminal*: Sometimes called *terminal screen*. The package's idea of what the terminal's screen currently looks like, i.e., what the user sees now. This is a special *screen*.

*screen*: This is a subset of windows which are as large as the terminal screen, i.e., they start at the upper left hand corner and encompass the lower right hand corner. One of these, *stdscr*, is automatically provided for the programmer.

#### 1.2. Compiling Things

In order to use the library, it is necessary to have certain types and variables defined. Therefore, the programmer must have a line:

```
#include <curses.h>
```

at the top of the program source. The header file `<curses.h>` needs to include `<sgtty.h>`, so the one should not do so oneself<sup>1</sup>. Also, compilations should have the following form:

```
cc [flags] file ... -lcurses -ltermlib
```

#### 1.3. Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named *WINDOW* is defined which describes a window image to the routines, including its starting position on the screen (the (y, x) co-ordinates of the upper left hand corner) and its size. One of these (called *curscr* for *current screen*) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for *standard screen*) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When one has a window which describes what some part the terminal should look like, the routine *refresh()* (or *wrefresh()* if the window is not *stdscr*) is called. *refresh()* makes the ter-

---

<sup>1</sup> The screen package also uses the Standard I/O library, so `<curses.h>` includes `<stdio.h>`. It is redundant (but harmless) for the programmer to do it, too.



## Screen Package

minal, in the area covered by the window, look like that window. Note, therefore, that changing something on a window *does not change the terminal*. Actual updates to the terminal screen are made only by calling *refresh()* or *wrefresh()*. This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

### 1.4. Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: *curscr*, which knows what the terminal looks like, and *stdscr*, which is what the programmer wants the terminal to look like next. The user should never really access *curscr* directly. Changes should be made to the appropriate screen, and then the routine *refresh()* (or *wrefresh()*) should be called.

Many functions are set up to deal with *stdscr* as a default screen. For example, to add a character to *stdscr*, one calls *addch()* with the desired character. If a different window is to be used, the routine *waddch()* (for window-specific *addch()*) is provided<sup>2</sup>. This convention of prepending function names with a "w" when they are to be applied to specific windows is consistent. The only routines which do *not* do this are those to which a window must always be specified.

In order to move the current (y, x) co-ordinates from one point to another, the routines *move()* and *wmove()* are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsiness, most I/O routines can be preceded by the prefix "mv" and the desired (y, x) co-ordinates then can be added to the arguments to the function. For example, the calls

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

Note that the window description pointer (*win*) comes before the added (y, x) co-ordinates. If such pointers are needed, they are always the first parameters passed.

## 2. Variables

Many variables which are used to describe the terminal environment are available to the programmer. They are:

type	name	description
WINDOW *	<i>curscr</i>	current version of the screen (terminal screen).
WINDOW *	<i>stdscr</i>	standard screen. Most updates are usually done here.
char *	<i>Def_term</i>	default terminal type if type cannot be determined

---

<sup>2</sup> Actually, *addch()* is really a "#define" macro with arguments, as are most of the "functions" which deal with *stdscr* as a default.

## Screen Package

bool	My_term	use the terminal specification in <i>Def_term</i> as terminal. irrelevant of real terminal type
char *	ttytype	full name of the current terminal.
int	LINES	number of lines on the terminal
int	COLS	number of columns on the terminal
int	ERR	error flag returned by routines on a fail.
int	OK	error flag returned by routines when things go right.

There are also several “#define” constants and types which are of general usefulness:

reg	storage class “register” (e.g., <i>reg int i</i> ;) )
bool	boolean type, actually a “char” (e.g., <i>bool doneit</i> ;) )
TRUE	boolean “true” flag (1).
FALSE	boolean “false” flag (0).

### 3. Usage

This is a description of how to actually use the screen package. In it, we assume all updating, reading, etc. is applied to *stdscr*. All instructions will work on any window, with changing the function name and parameters as mentioned above.

#### 3.1. Starting up

In order to use the screen package, the routines must know about terminal characteristics, and the space for *curscr* and *stdscr* must be allocated. These functions are performed by *initscr()*. Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, *initscr()* returns ERR. *initscr()* must *always* be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either *curscr* or *stdscr* are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like *nl()* and *crmode()* should be called after *initscr()*.

Now that the screen windows have been allocated, you can set them up for the run. If you want to, say, allow the window to scroll, use *scrollok()*. If you want the cursor to be left after the last change, use *leaveok()*. If this isn't done, *refresh()* will move the cursor to the window's current (y, x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions *newwin()* and *subwin()*. *delwin()* will allow you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables *LINES* and *COLS* to be what you want, and then call *initscr()*. This is best done before, but can be done either before or after, the first call to *initscr()*, as it will always delete any existing *stdscr* and/or *curscr* before creating new ones.

#### 3.2. The Nitty-Gritty

##### 3.2.1. Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are *addch()* and *move()*. *addch()* adds a character at the current (y, x) co-ordinates, returning ERR if it would cause the window to illegally scroll, i.e., printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. *move()* changes the current (y, x) co-ordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into *mvaddch()* to do both things in one fell swoop.

The other output functions, such as *addstr()* and *printw()*, all call *addch()* to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call *refresh()*. In order

## Screen Package

to optimize finding changes, *refresh()* assumes that any part of the window not changed since the last *refresh()* of that window has not been changed on the terminal, i.e., that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine *touchwin()* is provided to make it look like the entire window has been changed, thus making *refresh()* check the whole subsection of the terminal for changes.

If you call *wrefresh()* with *curscr*, it will make the screen look like *curscr* thinks it looks like. This is useful for implementing a command which would redraw the screen in case it got messed up.

### 3.2.2. Input

Input is essentially a mirror image of output. The complementary function to *addch()* is *getch()* which, if echo is set, will call *addch()* to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, *getch()* sets it to be cbreak, and then reads in the character.

### 3.2.3. Miscellaneous

All sorts of fun functions exists for maintaining and changing information about the windows. For the most part, the descriptions in section 5.4. should suffice.

### 3.3. Finishing up

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in *gettmode()* and *setterm()*, which are called by *initscr()*. In order to clean up after the routines, the routine *endwin()* is provided. It restores tty modes to what they were when *initscr()* was first called. Thus, anytime after the call to *initscr*, *endwin()* should be called before exiting.

## 4. Cursor Motion Optimization: Standing Alone

It is possible to use the cursor optimization functions of this screen package without the overhead and additional size of the screen updating functions. The screen updating functions are designed for uses where parts of the screen are changed, but the overall image remains the same. This includes such programs as *eye* and *vi*<sup>3</sup>. Certain other programs will find it difficult to use these functions in this manner without considerable unnecessary program overhead. For such applications, such as some "*crt hacks*"<sup>4</sup> and optimizing *cat(1)*-type programs, all that is needed is the motion optimizations. This, therefore, is a description of what some of what goes on at the lower levels of this screen package. The descriptions assume a certain amount of familiarity with programming problems and some finer points of C. None of it is terribly difficult, but you should be forewarned.

### 4.1. Terminal Information

In order to use a terminal's features to the best of a program's abilities, it must first know what they are<sup>5</sup>. The */etc/termcap* database describes these, but a certain amount of decoding is necessary, and there are, of course, both efficient and inefficient ways of reading them in. The algorithm that the uses is taken from *vi* and is hideously efficient. It reads them in a tight loop into a set of variables whose names are two uppercase letters with some mnemonic value. For

---

<sup>3</sup> *Eye* actually uses these functions, *vi* does not.

<sup>4</sup> Graphics programs designed to run on character-oriented terminals. I could name many, but they come and go, so the list would be quickly out of date. Recently, there have been programs such as *rocket* and *gun*.

<sup>5</sup> If this comes as any surprise to you, there's this tower in Paris they're thinking of junking that I can let you have for a song.

## Screen Package

example, *HO* is a string which moves the cursor to the "home" position<sup>6</sup>. As there are two types of variables involving ttys, there are two routines. The first, *gettmode()*, sets some variables based upon the tty modes accessed by *gty(2)* and *stty(2)*. The second, *setterm()*, a larger task by reading in the descriptions from the */etc/termcap* database. This is the way these routines are used by *initscr()*:

```
if (isatty(0)) {
 gettmode();
 if (sp=getenv("TERM"))
 setterm(sp);
}
else
 setterm(Def_term);
_puts(TI);
_puts(VS);
```

*isatty()* checks to see if file descriptor 0 is a terminal<sup>7</sup>. If it is, *gettmode()* sets the terminal description modes from a *gty(2)* *getenv()* is then called to get the name of the terminal, and that value (if there is one) is passed to *setterm()*, which reads in the variables from */etc/termcap* associated with that terminal. (*getenv()* returns a pointer to a string containing the name of the terminal, which we save in the character pointer *sp*.) If *isatty()* returns false, the default terminal *Def\_term* is used. The *TI* and *VS* sequences initialize the terminal (*\_puts()* is a macro which uses *tputs()* (see *termcap(3)*) to put out a string). It is these things which *endwin()* undoes.

### 4.2. Movement Optimizations, or, Getting Over Yonder

Now that we have all this useful information, it would be nice to do something with it<sup>8</sup>. The most difficult thing to do properly is motion optimization. When you consider how many different features various terminals have (tabs, backtabs, non-destructive space, home sequences, absolute tabs, .....), you can see that deciding how to get from here to there can be a decidedly non-trivial task. The editor *vi* uses many of these features, and the routines it uses to do this take up many pages of code. Fortunately, I was able to liberate them with the author's permission, and use them here.

After using *gettmode()* and *setterm()* to get the terminal descriptions, the function *mvcur()* deals with this task. Its usage is simple: you simply tell it where you are now and where you want to go. For example

```
mvcur(0, 0, LINES/2, COLS/2)
```

would move the cursor from the home position (0, 0) to the middle of the screen. If you wish to force absolute addressing, you can use the function *goto()* from the *termlib(7)* routines, or you can tell *mvcur()* that you are impossibly far away, like Cleveland. For example, to absolutely address the lower left hand corner of the screen from anywhere just claim that you are in the upper right hand corner:

```
mvcur(0, COLS-1, LINES-1, 0)
```

---

<sup>6</sup> These names are identical to those variables used in the */etc/termcap* database to describe each capability. See Appendix A for a complete list of those read, and *termcap(5)* for a full description.

<sup>7</sup> *isatty()* is defined in the default C library function routines. It does a *gty(2)* on the descriptor and checks the return value.

<sup>8</sup> Actually, it *can* be emotionally fulfilling just to get the information. This is usually only true, however, if you have the social life of a kumquat.

## Screen Package

### 5. The Functions

In the following definitions, “+” means that the “function” is really a “#define” macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as *addch()*, it will show up as its “w” counterpart. The arguments are given to show the order and type of each. Their names are not mandatory, just suggestive.

#### 5.1. Output Functions

**addch(ch) +**  
*char*        *ch*;

**waddch(win, ch)**  
*WINDOW* \**win*;  
*char*        *ch*;

Add the character *ch* on the window at the current (y, x) co-ordinates. If the character is a newline ('\n') the line will be cleared to the end, and the current (y, x) co-ordinates will be changed to the beginning of the next line if newline mapping is on, or to the next line at the same x co-ordinate if it is off. A return ('\r') will move to the beginning of the line on the window. Tabs ('\t') will be expanded into spaces in the normal tabstop positions of every eight characters. This returns ERR if it would cause the screen to scroll illegally.

**addstr(str) +**  
*char*        \**str*;

**waddstr(win, str)**  
*WINDOW* \**win*;  
*char*        \**str*;

Add the string pointed to by *str* on the window at the current (y, x) co-ordinates. This returns ERR if it would cause the screen to scroll illegally. In this case, it will put on as much as it can.

**box(win, vert, hor)**  
*WINDOW* \**win*;  
*char*        *vert, hor*;

Draws a box around the window using *vert* as the character for drawing the vertical sides, and *hor* for drawing the horizontal lines. If scrolling is not allowed, and the window encompasses the lower right-hand corner of the terminal, the corners are left blank to avoid a scroll.

**clear() +**

**wclear(win)**  
*WINDOW* \**win*;

Resets the entire window to blanks. If *win* is a screen, this sets the clear flag, which will cause a clear-screen sequence to be sent on the next *refresh()* call. This also moves the current (y, x) co-ordinates to (0, 0).

## Screen Package

**clearok**(scr, boolf) †

*WINDOW* \*scr;  
*bool* boolf;

Sets the clear flag for the screen *scr*. If *boolf* is TRUE, this will force a clear-screen to be printed on the next *refresh()*, or stop it from doing so if *boolf* is FALSE. This only works on screens, and, unlike *clear()*, does not alter the contents of the screen. If *scr* is *curscr*, the next *refresh()* call will cause a clear-screen, even if the window passed to *refresh()* is not a screen.

**clrtoobot**() †

**wclrtoobot**(win)  
*WINDOW* \*win;

Wipes the window clear from the current (y, x) co-ordinates to the bottom. This does not force a clear-screen sequence on the next refresh under any circumstances. This has no associated "mv" command.

**clrtoeol**() †

**wclrtoeol**(win)  
*WINDOW* \*win;

Wipes the window clear from the current (y, x) co-ordinates to the end of the line. This has no associated "mv" command.

**delch**()

**wdelch**(win)  
*WINDOW* \*win;

Delete the character at the current (y, x) co-ordinates. Each character after it on the line shifts to the left, and the last character becomes blank.

**deleteln**()

**wdeleteln**(win)  
*WINDOW* \*win;

Delete the current line. Every line below the current one will move up, and the bottom line will become blank. The current (y, x) co-ordinates will remain unchanged.

**erase**() †

**werase**(win)  
*WINDOW* \*win;

## Screen Package

Erases the window to blanks without setting the clear flag. This is analagous to *clear()*, except that it never causes a clear-screen sequence to be generated on a *refresh()*. This has no associated "mv" command.

### **insch(c)**

*char*        *c*;

### **winsch(win, c)**

*WINDOW*   *\*win*;

*char*        *c*;

Insert *c* at the current (y, x) co-ordinates. Each character after it shifts to the right, and the last character disappears.

### **insertln()**

### **winsertln(win)**

*WINDOW*   *\*win*;

Insert a line above the current one. Every line below the current line will be shifted down, and the bottom line will disappear. The current line will become blank, and the current (y, x) co-ordinates will remain unchanged.

### **move(y, x) †**

*int*         *y, x*;

### **wmove(win, y, x)**

*WINDOW*   *\*win*;

*int*         *y, x*;

Change the current (y, x) co-ordinates of the window to (y, x). This returns ERR if it would cause the screen to scroll illegally.

### **overlay(win1, win2)**

*WINDOW*   *\*win1, \*win2*;

Overlay *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done non-destructively, i.e., blanks on *win1* leave the contents of the space on *win2* untouched.

### **overwrite(win1, win2)**

*WINDOW*   *\*win1, \*win2*;

Overwrite *win1* on *win2*. The contents of *win1*, insofar as they fit, are placed on *win2* at their starting (y, x) co-ordinates. This is done destructively, i.e., blanks on *win1* become blank on *win2*.

### **printw(fmt, arg1, arg2, ...)**

*char*         *\*fmt*;

## Screen Package

**wprintw(win, fmt, arg1, arg2, ...)**

*WINDOW* \*win;

char \*fmt;

Performs a *printf()* on the window starting at the current (y, x) co-ordinates. It uses *addstr()* to add the string on the window. It is often advisable to use the field width options of *printf()* to avoid leaving things on the window from earlier calls. This returns ERR if it would cause the screen to scroll illegally.

**refresh() †**

**wrefresh(win)**

*WINDOW* \*win;

Synchronize the terminal screen with the desired window. If the window is not a screen, only that part covered by it is updated. This returns ERR if it would cause the screen to scroll illegally. In this case, it will update whatever it can without causing the scroll.

**standout() †**

**wstandout(win)**

*WINDOW* \*win;

**standend() †**

**wstandend(win)**

*WINDOW* \*win;

Start and stop putting characters onto *win* in standout mode. *standout()* causes any characters added to the window to be put in standout mode on the terminal (if it has that capability). *standend()* stops this. The sequences *SO* and *SE* (or *US* and *UE* if they are not defined) are used (see Appendix A).

### 5.2. Input Functions

**crmode() †**

**nocrmode() †**

Set or unset the terminal to/from cbreak mode.

**echo() †**

**noecho() †**

Sets the terminal to echo or not echo characters.



## Screen Package

**getch()** †

**wgetch(win)**  
*WINDOW* \*win;

Gets a character from the terminal and (if necessary) echos it on the window. This returns ERR if it would cause the screen to scroll illegally. Otherwise, the character gotten is returned. If *noecho* has been set, then the window is left unaltered. In order to retain control of the terminal, it is necessary to have one of *noecho*, *cbreak*, or *rawmode* set. If you do not set one, whatever routine you call to read characters will set *cbreak* for you, and then reset to the original mode when finished.

**getstr(str)** †  
*char* \*str;

**wgetstr(win, str)**  
*WINDOW* \*win;  
*char* \*str;

Get a string through the window and put it in the location pointed to by *str*, which is assumed to be large enough to handle it. It sets tty modes if necessary, and then calls *getch()* (or *wgetch(win)*) to get the characters needed to fill in the string until a newline or EOF is encountered. The newline stripped off the string. This returns ERR if it would cause the screen to scroll illegally.

**raw()** †

**noraw()** †

Set or unset the terminal to/from raw mode. On version 7 UNIX<sup>9</sup> this also turns of new-line mapping (see *nl()*).

**scanw(fmt, arg1, arg2, ...)**  
*char* \*fmt;

**wscanw(win, fmt, arg1, arg2, ...)**  
*WINDOW* \*win;  
*char* \*fmt;

Perform a *scanf()* through the window using *fmt*. It does this using consecutive *getch()*'s (or *wgetch(win)*'s). This returns ERR if it would cause the screen to scroll illegally.

### 5.3. Miscellaneous Functions

**delwin(win)**  
*WINDOW* \*win;

---

<sup>9</sup> UNIX is a trademark of Bell Laboratories.

## Screen Package

**delwin(win)**

*WINDOW \*win;*

Deletes the window from existence. All resources are freed for future use by `calloc(3)`. If a window has a `subwin()` allocated window inside of it, deleting the outer window the subwindow is not affected, even though this does invalidate it. Therefore, subwindows should be deleted before their outer windows are.

**endwin()**

Finish up window routines before exit. This restores the terminal to the state it was before `initscr()` (or `getmode()` and `setterm()`) was called. It should always be called before exiting. It does not exit. This is especially useful for resetting tty stats when trapping rubeouts via `signal(2)`.

**getyx(win, y, x) †**

*WINDOW \*win;*

*int y, x;*

Puts the current (y, x) co-ordinates of *win* in the variables *y* and *x*. Since it is a macro, not a function, you do not pass the address of *y* and *x*.

**inch() †**

**winch(win) †**

*WINDOW \*win;*

Returns the character at the current (y, x) co-ordinates on the given window. This does not make any changes to the window. This has no associated "mv" command.

**initscr()**

Initialize the screen routines. This must be called before any of the screen routines are used. It initializes the terminal-type data and such, and without it, none of the routines can operate. If standard input is not a tty, it sets the specifications to the terminal whose name is pointed to by `Def_term` (initially "dumb"). If the boolean `My_term` is true, `Def_term` is always used.

**leaveok(win, boolf) †**

*WINDOW \*win;*

*bool boolf;*

Sets the boolean flag for leaving the cursor after the last change. If *boolf* is TRUE, the cursor will be left after the last update on the terminal, and the current (y, x) co-ordinates for *win* will be changed accordingly. If it is FALSE, it will be moved to the current (y, x) co-ordinates. This flag (initially FALSE) retains its value until changed by the user.

**longname(termbuf, name)**

*char \*termbuf, \*name;*

## Screen Package

Fills in *name* with the long (full) name of the terminal described by the termcap entry in *termbuf*. It is generally of little use, but is nice for telling the user in a readable format what terminal we think he has. This is available in the global variable *ttytype*. *Termbuf* is usually set via the *term*lib routine *tgetent()*.

**mvwin(win, y, x)**  
*WINDOW* \*win;  
*int* y, x;

Move the home position of the window *win* from its current starting coordinates to (*y*, *x*). If that would put part or all of the window off the edge of the terminal screen, *mvwin()* returns ERR and does not change anything.

*WINDOW* \*  
**newwin(lines, cols, begin\_y, begin\_x)**  
*int* lines, cols, begin\_y, begin\_x;

Create a new window with *lines* lines and *cols* columns starting at position (*begin\_y*, *begin\_x*). If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin\_y*) or (*COLS* - *begin\_x*) respectively. Thus, to get a new window of dimensions *LINES* × *COLS*, use *newwin(0, 0, 0, 0)*.

**nl()** †

**nonl()** †

Set or unset the terminal to/from nl mode, i.e., start/stop the system from mapping <RETURN> to <LINE-FEED>. If the mapping is not done, *refresh()* can do more optimization, so it is recommended, but not required, to turn it off.

**scrollok(win, boolf)** †  
*WINDOW* \*win;  
*bool* boolf;

Set the scroll flag for the given window. If *boolf* is FALSE, scrolling is not allowed. This is its default setting.

**touchwin(win)**  
*WINDOW* \*win;

Make it appear that the every location on the window has been changed. This is usually only needed for refreshes with overlapping windows.

*WINDOW* \*  
**subwin(win, lines, cols, begin\_y, begin\_x)**  
*WINDOW* \*win;  
*int* lines, cols, begin\_y, begin\_x;

Create a new window with *lines* lines and *cols* columns starting at position (*begin\_y*, *begin\_x*) in the middle of the window *win*. This means that any change made to either window in the area covered by the subwindow will be made on both windows. *begin\_y*, *begin\_x* are specified relative to the overall screen, not the relative (0, 0) of *win*. If either *lines* or *cols* is 0 (zero), that dimension will be set to (*LINES* - *begin\_y*) or

## Screen Package

(*COLS* - *begin\_x*) respectively.

**unctrl(ch) †**

*char*        *ch*;

This is actually a debug function for the library, but it is of general usefulness. It returns a string which is a representation of *ch*. Control characters become their upper-case equivalents preceded by a "^". Other letters stay just as they are. To use *unctrl()*, you must have `#include <unctrl.h>` in your file.

### 5.4. Details

**gettmode()**

Get the tty stats. This is normally called by *initscr()*.

**mvcur(lasty, lastx, newy, newx)**

*int*        *lasty, lastx, newy, newx*;

Moves the terminal's cursor from (*lasty, lastx*) to (*newy, newx*) in an approximation of optimal fashion. This routine uses the functions borrowed from *ex* version 2.6. It is possible to use this optimization without the benefit of the screen routines. With the screen routines, this should not be called by the user. *move()* and *refresh()* should be used to move the cursor position, so that the routines know what's going on.

**scroll(win)**

*WINDOW*   *\*win*;

Scroll the window upward one line. This is normally not used by the user.

**savetty() †**

**resetty() †**

*savetty()* saves the current tty characteristic flags. *resetty()* restores them to what *savetty()* stored. These functions are performed automatically by *initscr()* and *endwin()*.

**setterm(name)**

*char*        *\*name*;

Set the terminal characteristics to be those of the terminal named *name*. This is normally called by *initscr()*.

**tstp()**

If the new tty(4) driver is in use, this function will save the current tty state and then put the process to sleep. When the process gets restarted, it restores the tty state and then calls *wrefresh(curscr)* to redraw the screen. *initscr()* sets the signal SIGTSTP to trap to this routine.

## Appendix A

### 1. Capabilities from termcap

#### 1.1. Disclaimer

The description of terminals is a difficult business, and we only attempt to summarize the capabilities here: for a full description see the paper describing termcap.

#### 1.2. Overview

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability, and has meaning for the capabilities which have a **P** at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be changed (specified by *PC*)). In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes which will shift several lines.) This is specified as, e.g., **12\***, before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say **P\***.

#### 1.3. Variables Set By `setterm()`

variables set by `setterm()`

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char *	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P	Back Tab
bool	CA		Cursor Addressable
char *	CD	P*	Clear to end of Display
char *	CE	P	Clear to End of line
char *	CL	P*	CLear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		DOWn line sequence
char *	ED		End Delete mode
bool	EO		can Erase Overstrikes with ` `
char *	EI		End Insert mode
char *	HO		HOMe cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing
char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmd mode
bool	MI		can Move in Insert mode
bool	NC		No Cr: \r sends \r\n then eats \n

## Appendix A

### variables set by *setterm()*

Type	Name	Pad	Description
char *	ND		Non-Destructive space
bool	OS		OverStrike works
char *	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SO		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reverse
char *	TA	P	TAb (not <code>^I</code> or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		UnderLining works even though !OS
char *	UP		Upline
char *	US		Underline Starting sequence <sup>10</sup>
char *	VB		Visible Bell
char *	VE		Visual End sequence
char *	VS		Visual Start sequence
bool	XN		a Newline gets eaten after wrap

Names starting with **X** are reserved for severely nauseous glitches

#### 1.4. Variables Set By *gettmode()*

##### variables set by *gettmode()*

type	name	description
bool	NONL	Term can't hack linefeeds doing a CR
bool	GT	Gtty indicates Tabs
bool	UPPERCASE	Terminal generates only uppercase letters

<sup>10</sup> US and UE, if they do not exist in the termcap entry, are copied from SO and SE in *setterm()*

## Appendix B

### 1.

#### The WINDOW structure

The WINDOW structure is defined as follows:

```
define WINDOW struct _win_st

struct _win_st {
 short _cury, _curx;
 short _maxy, _maxx;
 short _begy, _begx;
 short _flags;
 bool _clear;
 bool _leave;
 bool _scroll;
 char *_y;
 short *_firstch;
 short *_lastch;
};

define _SUBWIN 01
define _ENDLINE 02
define _FULLWIN 04
define _SCROLLWIN 010
define _STANDOUT 0200
```

`_cury` and `_curx` are the current (y, x) co-ordinates for the window. New characters added to the screen are added at this point. `_maxy` and `_maxx` are the maximum values allowed for (`_cury`, `_curx`). `_begy` and `_begx` are the starting (y, x) co-ordinates on the terminal for the window, i.e., the window's home. `_cury`, `_curx`, `_maxy`, and `_maxx` are measured relative to (`_begy`, `_begx`), not the terminal's home.

`_clear` tells if a clear-screen sequence is to be generated on the next `refresh()` call. This is only meaningful for screens. The initial clear-screen for the first `refresh()` call is generated by initially setting `clear` to be TRUE for `curscr`, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. `_leave` is TRUE if the current (y, x) co-ordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. `_scroll` is TRUE if scrolling is allowed.

`_y` is a pointer to an array of lines which describe the terminal. Thus:

`_y[i]`

is a pointer to the *n*th line, and

`_y[i][j]`.

is the *j*th character on the *n*th line.

`_flags` can have one or more values or'd into it. `_SUBWIN` means that the window is a subwindow, which indicates to `delwin()` that the space for the lines is not to be freed. `_ENDLINE` says that the end of the line for this window is also the end of a screen. `_FULLWIN` says that this window is a screen. `_SCROLLWIN` indicates that the last character of this screen is at the lower right-hand corner of the terminal; i.e., if a character was put there, the terminal would scroll. `_STANDOUT` says that all characters added to the screen are in stand-out mode.

---

<sup>11</sup> All variables not normally accessed directly by the user are named with an initial "\_" to avoid conflicts with the user's variables.

## Appendix C

### 1. Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

### 2. Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculational sections which are irrelevant to the example, and are therefore usually not included. It is hoped that the data structure definitions give enough of an idea to allow understanding of what the relevant portions do. The rest is left as an exercise to the reader, and will not be on the final.

#### 2.1. Twinkle

This is a moderately simple program which prints pretty patterns on the screen that might even hold your interest for 30 seconds or more. It switches between patterns of asterisks, putting them on one by one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, as is demonstrated below.

```
include <curses.h>
include <signal.h>

/*
 * the idea for this program was a product of the imagination of
 * Kurt Schoens. Not responsible for minds lost or stolen.
 */

define NCOLS 80
define NLINES 24
define MAXPATTERNS 4

struct locs {
 char y, x;
};

typedef struct locs LOCS;

LOCS Layout[NCOLS * NLINES]; /* current board layout */

int Pattern, /* current pattern number */
 Numstars; /* number of stars in pattern */

main() {

 char *getenv();
 int die();

 srand(getpid()); /* initialize random sequence */

 initscr();
 signal(SIGINT, die);
 noecho();
 nonl();
 leaveok(stdscr, TRUE);
 scrollok(stdscr, FALSE);
```



## Appendix C

```

 for (;;) {
 makeboard(); /* make the board setup */
 puton('.'); /* put on '.'s */
 puton(' '); /* cover up with ' 's */
 }

/*
 * On program exit, move the cursor to the lower left corner by
 * direct addressing, since current location is not guaranteed.
 * We lie and say we used to be at the upper right corner to guarantee
 * absolute addressing.
 */
die() {

 signal(SIGINT, SIG_IGN);
 mvcur(0, COLS-1, LINES-1, 0);
 endwin();
 exit(0);
}

/*
 * Make the current board setup. It picks a random pattern and
 * calls ison() to determine if the character is on that pattern
 * or not.
 */
makeboard() {

 reg int y, x;
 reg LOCS *lp;

 Pattern = rand() % MAXPATTERNS;
 lp = Layout;
 for (y = 0; y < NLINES; y++)
 for (x = 0; x < NCOLS; x++)
 if (ison(y, x)) {
 lp->y = y;
 lp++->x = x;
 }
 Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int y, x; {

 switch (Pattern) {
 case 0: /* alternating lines */
 return !(y & 01);
 }
}

```

## Appendix C

```
case 1: /* box */
 if (x >= LINES && y >= NCOLS)
 return FALSE;
 if (y < 3 || y >= NLINES - 3)
 return TRUE;
 return (x < 3 || x >= NCOLS - 3);
case 2: /* holy pattern! */
 return ((x + y) & 01);
case 3: /* bar across center */
 return (y >= 9 && y <= 15);
}
/* NOTREACHED */
}

puton(ch)
reg char ch: {

 reg LOCS *lp;
 reg int r;
 reg LOCS *end;
 LOCS temp;

 end = &Layout[Numstars];
 for (lp = Layout; lp < end; lp++) {
 r = rand() % Numstars;
 temp = *lp;
 *lp = Layout[r];
 Layout[r] = temp;
 }

 for (lp = Layout; lp < end; lp++) {
 mvaddch(lp->y, lp->x, ch);
 refresh();
 }
}
}
```

### 2.2. Life

This program plays the famous computer pattern game of life (Scientific American, May, 1974). The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

```
include <curses.h>
include <signal.h>

/*
 * Run a life game. This is a demonstration program for
 * the Screen Updating section of the -lcurses cursor package.
 */

struct lst_st { /* linked list element */
```

## Appendix C

```

 int y, x; /* (y, x) position of piece */
 struct lst_st *next, *last; /* doubly linked */
};

typedef struct lst_st LIST;

LIST *Head; /* head of linked list */

main(ac, av)
int ac;
char *av[], {

 int die();

 evalargs(ac, av); /* evaluate arguments */

 initscr(); /* initialize screen package */
 signal(SIGINT, die); /* set to restore tty stats */
 crmode(); /* set for char-by-char */
 noecho(); /* input */
 nonl(); /* for optimization */

 getstart(); /* get starting position */
 for (;;) {
 prboard(); /* print out current board */
 update(); /* update board position */
 }
}

/*
 * This is the routine which is called when rubout is hit.
 * It resets the tty stats to their original values. This
 * is the normal way of leaving the program.
 */
die() {

 signal(SIGINT, SIG_IGN); /* ignore rubouts */
 mvcur(0, COLS-1, LINES-1, 0); /* go to bottom of screen */
 endwin(); /* set terminal to initial state */
 exit(0);
}

/*
 * Get the starting position from the user. The keys u, i, o, j, l,
 * m, ,, and . are used for moving their relative directions from the
 * k key. Thus, u move diagonally up to the left, , moves directly down,
 * etc. x places a piece at the current position, " " takes it away.
 * The input can also be from a file. The list is built after the
 * board setup is ready.
 */
getstart() {

 reg char c;
 reg int x, y;

```

## Appendix C

```

box(stdscr, ↑, ' _'):
move(1, 1):
do {
 refresh();
 if ((c=getch()) == 'q')
 break;
 switch (c) {
 case 'u':
 case 'i':
 case 'o':
 case 'j':
 case 'l':
 case 'm':
 case ',':
 case '.':
 adjustx(c);
 break;
 case 'f':
 mvaddstr(0, 0, "File name: ");
 getstr(buf);
 readfile(buf);
 break;
 case 'x':
 addch('X');
 break;
 case ' ':
 addch(' ');
 break;
 }
}

if (Head != NULL)
 dellist(Head);
Head = malloc(sizeof (LIST));

/*
 * loop through the screen looking for 'x's, and add a list
 * element for each one
 */
for (y = 1; y < LINES - 1; y++)
 for (x = 1; x < COLS - 1; x++) {
 move(y, x);
 if (inch() == 'x')
 addlist(y, x);
 }
}

/*
 * Print out the current board position from the linked list
 */
prboard() {
 reg LIST *hp;

```

## Appendix C

```
erase(); /* clear out last position */
box(stdscr, ↑, ' '); /* box in the screen */

/*
 * go through the list adding each piece to the newly
 * blank board
 */
for (hp = Head; hp; hp = hp->next)
 mvaddch(hp->y, hp->x, 'X');

refresh();
}
```

### 3. Motion optimization

The following example shows how motion optimization is written on its own. Programs which flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

#### 3.1. Twinkle

The `twinkle` program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

 reg char *sp;
 char *getenv();
 int _putchar(), die();

 srand(getpid()); /* initialize random sequence */

 if (isatty(0)) {
 gettmode();
 if (sp=getenv("TERM"))
 setterm(sp);
 signal(SIGINT, die);
 }
 else {
 printf("Need a terminal on %d\n", _tty_ch);
 exit(1);
 }
 _puts(TI);
 _puts(VS);

 noecho();
 nonl();
 tputs(CL, NLINES, _putchar);
 for (;;) {
 makeboard(); /* make the board setup */
 puton('.'); /* put on '.'s */
 puton(' '); /* cover up with ' 's */
 }
}
```

## Appendix C

```

/*
 * putchar defined for tputs() (and _puts())
 */
_putchar(c)
reg char c: {

 putchar(c);
}

puton(ch)
char ch: {

 static int lasty, lastx;
 reg LOCS *lp;
 reg int r;
 reg LOCS *end;
 LOCS temp;

 end = &Layout[Numstars];
 for (lp = Layout; lp < end; lp++) {
 r = rand() % Numstars;
 temp = *lp;
 *lp = Layout[r];
 Layout[r] = temp;
 }

 for (lp = Layout; lp < end; lp++)
 /* prevent scrolling */
 if (!AM || (lp->y < NLINES - 1 || lp->x < NCOLS - 1)) {
 mvcur(lasty, lastx, lp->y, lp->x);
 putchar(ch);
 lasty = lp->y;
 if ((lastx = lp->x + 1) >= NCOLS)
 if (AM) {
 lastx = 0;
 lasty++;
 }
 else
 lastx = NCOLS - 1;
 }
 }
}

```







## **A Tutorial Introduction to ADB**

*J. F. Maranzano*

*S. R. Bourne*

**Bell Laboratories  
Murray Hill, New Jersey 07974**

### ***ABSTRACT***

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine "core" files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine "core" and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

May 5, 1977

---

†UNIX is a Trademark of Bell Laboratories.

# A Tutorial Introduction to ADB

*J. F. Maranzano*

*S. R. Bourne*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at "core" files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX† with the C language, and with References 1, 2 and 3.

## 2. A Quick Survey

### 2.1. Invocation

ADB is invoked as:

```
adb obfile corefile
```

where *obfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

```
adb a.out core
```

or more simply:

```
adb
```

where the defaults are *a.out* and *core* respectively. The filename minus (-) means ignore this argument as in:

```
adb - core
```

ADB has requests for examining locations in either file. The ? request examines the contents of *obfile*, the / request examines the *corefile*. The general form of these requests is:

```
address ? format
```

or

```
address / format
```

### 2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

```
0126?l
```

---

†UNIX is a Trademark of Bell Laboratories.

sets dot to octal 126 and prints the instruction at that address. The request:

**.,10/d**

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the ? or / requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, -, \*, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *\_name*; ADB will recognize both forms.

### 2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

<b>b</b>	<b>one byte in octal</b>
<b>c</b>	<b>one byte as a character</b>
<b>o</b>	<b>one word in octal</b>
<b>d</b>	<b>one word in decimal</b>
<b>f</b>	<b>two words in floating point</b>
<b>i</b>	<b>PDP 11 instruction</b>
<b>s</b>	<b>a null terminated character string</b>
<b>a</b>	<b>the value of dot</b>
<b>u</b>	<b>one word as unsigned integer</b>
<b>n</b>	<b>print a newline</b>
<b>r</b>	<b>print a blank space</b>
<b>^</b>	<b>backup dot</b>

(Format letters are also available for "long" values, for example, 'D' for long decimal, and 'F' for double floating point.) For other formats see the ADB manual.

### 2.4. General Request Meanings

The general form of a request is:

**address,count command modifier**

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

<b>Command</b>	<b>Meaning</b>
<b>?</b>	<b>Print contents from <i>a.out</i> file</b>
<b>/</b>	<b>Print contents from <i>core</i> file</b>
<b>=</b>	<b>Print value of "dot"</b>
<b>:</b>	<b>Breakpoint control</b>
<b>\$</b>	<b>Miscellaneous requests</b>
<b>;</b>	<b>Request separator</b>
<b>!</b>	<b>Escape to shell</b>

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request Sq or \$Q (or cntl-D) must be used to exit from ADB.

### 3. Debugging C Programs

#### 3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp*. Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

```
adb a.out core
```

The first debugging request:

```
$c
```

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.

The next request:

```
$C
```

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

```
$r
```

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

```
$e
```

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by ? whereas the map for *core* file is referenced by /. Furthermore, a good rule of thumb is to use ? for instructions and / for data when looking at programs. To print out information about the maps type:

```
$m
```

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp*. This is done by:

```
*charp/s
```

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

```
main.argc/d
```

prints the decimal *core* image value of the argument *argc* in the function *main*.

The request:

```
*main.argv,3/o
```

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main*. Note that these values are the addresses of the arguments to *main*. Therefore:

```
0177770/s
```

prints the ASCII value of the first argument. Another way to print this value would have been

```
*/s
```

The \* means ditto which remembers the last address typed, in this case *main.argv* ; the \* instructs ADB to use the address field of the *core* file as a pointer.

The request:

```
. = o
```

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

```
. - 10/d
```

### 3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f*, *g*, and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

```
adb
```

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

```
$c
```

will fill a page of backtrace references to *f*, *g*, and *h*. Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

```
,5SC
```

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

```
fcnt/d
```

prints the decimal value of the counter for the function *f*. Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h*, type:

```
h.x/d
```

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with **SC** or the occurrence of a variable in the most recent call of a function. It is possible with the **SC** request, however, to print the stack frame starting at some address as **addressSC**.

### 3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

```
adb a.out -
```

Breakpoints are set in the program as:

```
address:b [request]
```

The requests:

```
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
```

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as `symbol+4` so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

```
Sb
```

The display indicates a *count* field. A breakpoint is bypassed *count - 1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the *jsr* to the C save routine. We can display the instructions using the ADB request:

```
settab,5?ia
```

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

```
settab,5?i
```

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the *?* command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

```
:r
```

To delete a breakpoint, for instance the entry to the function *settab*, one types:

```
settab+4:d
```

To continue execution of the program from the breakpoint type:

```
:c
```

Once the program has stopped (in this case at the breakpoint for *fopen*), ADB requests can be used to display the contents of memory. For example:

```
SC
```

to display a stack trace, or:

**tabs,3/8o**

to print three lines of 8 locations each from the array called *tabs*. By this time (at location *fopen*) in the C program, *settab* has been called and should have set a one in every eighth location of *tabs*.

### 3.4. Advanced Breakpoint Usage

We continue execution of the program with:

**:c**

See Figure 6b. *getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

**ibuf+6/20c**

When we continue the program with:

**:c**

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

**tabpos+4:d**

If the program is continued with:

**:c**

it resumes normal execution after ADB prints the message

**a.out:running**

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

**:c**

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

**:c 0**

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

**settab+4:b settab,5?ia \***

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only

\* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these statements must be written as:

```
settab+4:b settab,5?ia;0
getc+4,3:b main.c?C;0
settab+4:b settab,5?ia; ptab/o;0
```

Note that ;0 will set dot to zero and stop at the breakpoint.

stop after the third occurrence by typing:

```
getc+4,3:b main.c?C *
```

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

```
settab+4:b ..5?ia
fopen+4:b
```

will print the last thing dot was set to (in the example *fopen+4*) not the current location (*settab+4*) at which the program is executing.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

```
settab+4:b settab,5?ia; ptab/o *
```

could be entered after typing the above requests.

Now the display of breakpoints:

```
Sb
```

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f*, *g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

```
adb ex3 -
```

Suppose we enter the following breakpoints:

```
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification *d*). Since the format is not changed, the *d* can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f,fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a*, *b* and *fi*.



Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b fcnt/d; f.a/^a="d; f.b/^b="d; f.fi/^fi="d
```

In this format the quoted string is printed literally and the *d* produces a decimal display of the variables. The results are shown in Figure 7.

### 3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

```
:r arg1 arg2 ... <infile >outfile
```

This request kills any existing program under test and starts the *a.out* afresh.

- The program being debugged can be single stepped by:

```
:s
```

If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

```
address:r
```

- The count field can be used to skip the first *n* breakpoints as:

```
,n:r
```

The request:

```
,n:c
```

may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

```
address:c
```

- The program being debugged runs as a separate process and can be killed by:

```
:k
```

## 4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as `cc pgm.c`. A 410 file is produced by a C compiler command of the form `cc -n pgm.c`, whereas a 411 file is produced by `cc -i pgm.c`. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

```
$m
```

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and `?*` accesses the data part of the *a.out* file. The `?*` request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution

of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the ?\* operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

$$b1 \leq A \leq e1 \Rightarrow \text{file address} = (A - b1) + f1$$

$$b2 \leq A \leq e2 \Rightarrow \text{file address} = (A - b2) + f2$$

A user can access locations by using the ADB defined variables. The \$v request prints the variables initialized by ADB:

b	base address of data segment
d	length of the data segment
s	length of the stack
t	length of the text
m	execution type (407,410,411)

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

< b

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

02000> b

that sets b to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

### 5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

#### 5.1. Formatted dump

The line:

< b, -1/4o4^8Cn

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

< b            The base address of the data segment.

<b,-1 Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4^8Cn** is broken down as follows:

**4o** Print 4 octal locations.  
**4^** Backup the current address 4 locations (to the original start of the field).  
**8C** Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.  
**n** Print a newline.

The request:

**<b,<d/4o4^8Cn**

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

**adb a.out core < dump**

to read in a script file, *dump*, of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$R
0$s
=3n"Data Segment"
<b,-1/8ona
```

The request **120\$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

**symbol + offset**

The request **4095\$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request **=** can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

**=3n"C Stack Backtrace"**

that spaces three lines and prints the literal string. The request **\$v** prints all non-zero ADB variables (see Figure 8). The request **0\$s** sets the maximum offset for symbol matches to zero

thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,-1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

### 5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir -
=n8t'inum'8t'Name'
0,-1? u8t14cn
```

In this example, the *u* prints the *inumber* as an unsigned decimal integer, the *8t* means that ADB will space to the next multiple of 8 on the output line, and the *14c* prints the 14 character file name.

### 5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section V) could be dumped with the following set of requests:

```
adb /dev/src -
02000> b
?m <b
<b,-1?"flags'8ton'links,uid,gid'8t3bn',size'8tbrdn'addr'8t8un'times'8t2Y2na
```

In this example the value of the base for the map was changed to 02000 (by saying *?m<b*) since that is the start of an *ilist* within a file system. An artifice (*brd* above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the *2Y* operator. Figure 12 shows portions of these requests as applied to a directory and file system.

### 5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

```
072 = odx
```

will print

```
072 58 #3a
```

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

```
'a' = co
```

prints

```
a 0141
```

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

## 6. Patching

Patching files with ADB is accomplished with the *write*, *w* or *W*, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate*, *l* or *L* request. In general, the request syntax for *l* and *w* are similar as follows:

```
?l value
```

The request *l* is used to match on two bytes, *L* is used for four bytes. The request *w* is used to write two bytes, whereas *W* writes four bytes. The *value* field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

```
adb -w file1 file2
```

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

```
adb -w ex7 -
?l 'Th'
?W 'The '
```

The request *?l* starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of *?* to write to the *a.out* file. The form *?\** would have been used for a 411 file.

More frequently the request will be typed as:

```
?l 'Th'; ?s
```

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out -
:s arg1 arg2
flag/w 1
:c
```

The *:s* request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments *arg1* and *arg2*. If there is a subprocess running ADB writes to it rather than to the file so the *w* request causes *flag* to be changed in the memory of the subprocess.

## 7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if *?* is used for text (instructions) and */* for data.

3. ADB cannot handle C register variables in the most recently activated function.

### 8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

### 9. References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
 int fides;
 int nleft;
 char *nextp;
 char buff[512];
}bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
 char cc;

 if(argc < 2) {
 printf("Input file missing\n");
 exit(8);
 }

 if((fcreat(argv[1],obuf) < 0){
 printf("%s : not found\n", argv[1]);
 exit(8);
 }
 charp = "T";
 printf("debug 1 %s\n",charp);
 while(cc= *charp++)
 putc(cc,obuf);
 fflush(obuf);
}
```





Figure 3: Multiple function C program for stack trace illustration

```
int fcnt,gcnt,hcnt;
h(x,y)
{
 int hi; register int hr;
 hi = x+1;
 hr = x-y+1;
 hcnt++;
 hj:
 f(hr,hi);
}

g(p,q)
{
 int gi; register int gr;
 gi = q-p;
 gr = q-p+1;
 gcnt++;
 gj:
 h(gr,gi);
}

f(a,b)
{
 int fi; register int fr;
 fi = a+2*b;
 fr = a+b;
 fcnt++;
 fj:
 g(fr,fi);
}

main()
{
 f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
Sc
^h(04452,04451)
^g(04453,011124)
^f(02,04451)
^h(04450,04447)
^g(04451,011120)
^f(02,04447)
^h(04446,04445)
^g(04447,011114)
^f(02,04445)
^h(04444,04443)
HIT DEL KEY
adb
.SSC
^h(04452,04451)
 x: 04452
 y: 04451
 hi: ?
^g(04453,011124)
 p: 04453
 q: 011124
 gi: 04451
 gr: ?
^f(02,04451)
 a: 02
 b: 04451
 fi: 011124
 fr: 04453
^h(04450,04447)
 x: 04450
 y: 04447
 hi: 04451
 hr: 02
^g(04451,011120)
 p: 04451
 q: 011120
 gi: 04447
 gr: 04450

fcnt/d
_fcnt: 1173
gcnt/d
_gcnt: 1173
hcnt/d
_hcnt: 1172
h.x/d
022004: 2346
Sq
```

Figure 5: C program to decode tabs

```
#define MAXLINE 80
#define YES 1
#define NO 0
#define TABSP 8

char input[] "data";
char ibuf[518];
int tabs[MAXLINE];

main()
{
 int col, *ptab;
 char c;

 ptab = tabs;
 settab(ptab); /*Set initial tab stops */
 col = 1;
 if(fopen(input,ibuf) < 0) {
 printf("%s : not found\n",input);
 exit(8);
 }
 while((c = getc(ibuf)) != -1) {
 switch(c) {
 case '\t': /* TAB */
 while(tabpos(col) != YES) {
 putchar(' '); /* put BLANK */
 col++;
 }
 break;
 case '\n': /*NEWLINE */
 putchar('\n');
 col = 1;
 break;
 default:
 putchar(c);
 col++;
 }
 }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
 if(col > MAXLINE)
 return(YES);
 else
 return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
 int i;
 for(i = 0; i <= MAXLINE; i++)
 (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out -
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$B
breakpoints
count bkpt command
1 ~tabpos+04
1 _getc+04
1 _fopen+04
1 ~settab+04
settab,5?ia
~settab: jsr r5, csv
~settab+04: tst -(sp)
~settab+06: clr 0177770(r5)
~settab+012: cmp $0120,0177770(r5)
~settab+020: bit ~settab+076
~settab+022:
settab,5?!
~settab: jsr r5, csv
 tst -(sp)
 clr 0177770(r5)
 cmp $0120,0177770(r5)
 bit ~settab+076

:r
a.out: running
breakpoint ~settab+04: tst -(sp)
settab+4:d
:e
a.out: running
breakpoint _fopen+04: mov 04(r5),nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
 col: 01
 c: 0
 ptab: 03500
tabs,3/8o
03500: 01 0 0 0 0 0 0 0
 01 0 0 0 0 0 0 0
 01 0 0 0 0 0 0 0
```

Figure 6b: ADB output for C program of Figure 5

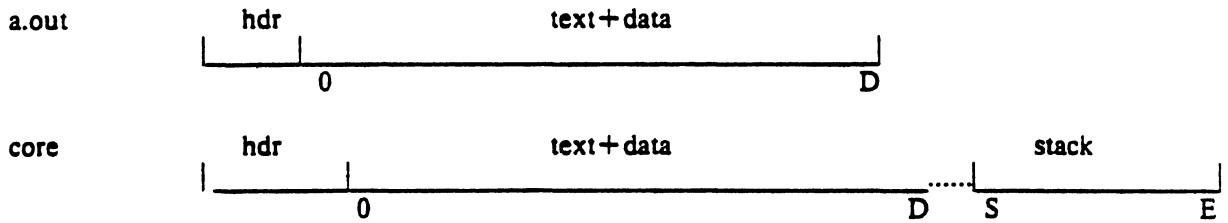
```
:c
a.out: running
breakpoint _getc+04: mov 04(r5),r1
ibuf+6/20c
__cleanu+0202: This is a test of
:c
a.out: running
breakpoint ~tabpos+04: cmp $0120,04(r5)
tabpos+4:d
settab+4:b settab,5?ia
settab+4:b settab,5?ia; 0
getc+4,3:b main.c?C; 0
settab+4:b settab,5?ia; ptab/o; 0
Sb
breakpoints
count bkpt command
1 ~tabpos+04
3 _getc+04 main.c?C:0
1 _fopen+04
1 ~settab+04 settab,5?ia;ptab?o:0
~settab: jsr r5,csv
~settab+04: bpt
~settab+06: clr 0177770(r5)
~settab+012: cmp $0120,0177770(r5)
~settab+020: blt ~settab+076
~settab+022:
0177766: 0177770
0177744: @`
T0177744: T
h0177744: h
i0177744: i
s0177744: s
```

Figure 7: ADB output for C program with breakpoints

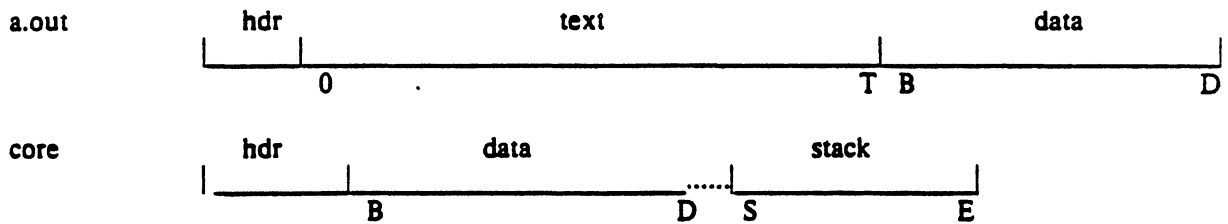
```
adb ex3 -
h+4:b hcnt/d: h.hi/: h.hr/
g+4:b gcnt/d: g.gi/: g.gr/
f+4:b fcnt/d: f.fi/: f.fr/
:r
ex3: running
_fcnt: 0
0177732: 214
symbol not found
f+4:b fcnt/d: f.a/: f.b/: f.fi/
g+4:b gcnt/d: g.p/: g.q/: g.gi/
h+4:b hcnt/d: h.x/: h.y/: h.hi/
:e
ex3: running
_fcnt: 0
0177746: 1
0177750: 1
0177732: 214
_gcnt: 0
0177726: 2
0177730: 3
0177712: 214
_hcnt: 0
0177706: 2
0177710: 1
0177672: 214
_fcnt: 1
0177666: 2
0177670: 3
0177652: 214
_gcnt: 1
0177646: 5
0177650: 8
0177632: 214
HIT DEL
f+4:b fcnt/d: f.a/^a = ^d: f.b/^b = ^d: f.fi/^fi = ^d
g+4:b gcnt/d: g.p/^p = ^d: g.q/^q = ^d: g.gi/^gi = ^d
h+4:b hcnt/d: h.x/^x = ^d: h.y/^h = ^d: h.hi/^hi = ^d
:r
ex3: running
_fcnt: 0
0177746: a = 1
0177750: b = 1
0177732: fi = 214
_gcnt: 0
0177726: p = 2
0177730: q = 3
0177712: gi = 214
_hcnt: 0
0177706: x = 2
0177710: y = 1
0177672: hi = 214
_fcnt: 1
0177666: a = 2
0177670: b = 3
0177652: fi = 214
HIT DEL
Sq
```

**Figure 8: ADB address maps**

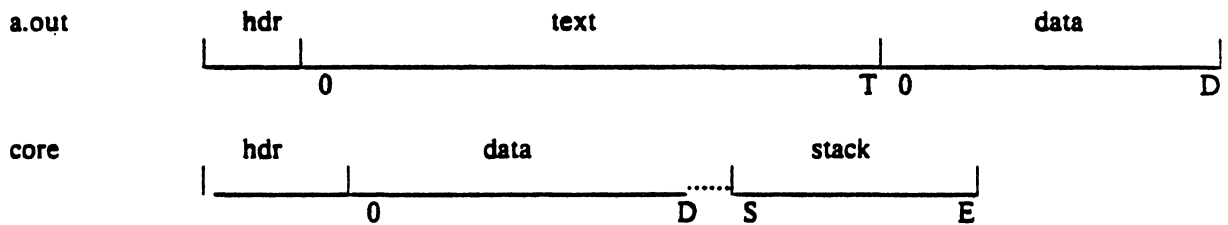
*407 files*



*410 files (shared text)*



*411 files (separated I and D space)*



The following *adb* variables are set.

		407	410	411
<b>b</b>	base of data	0	B	0
<b>d</b>	length of data	D	D-B	D
<b>s</b>	length of stack	S	S	S
<b>t</b>	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
Sm
text map `map407`
b1 = 0 e1 = 0256 f1 = 020
b2 = 0 e2 = 0256 f2 = 020
data map `core407`
b1 = 0 e1 = 0300 f1 = 02000
b2 = 0175400 e2 = 0200000 f2 = 02300
Sv
variables
d = 0300
m = 0407
s = 02400
Sq
```

```
adb map410 core410
Sm
text map `map410`
b1 = 0 e1 = 0200 f1 = 020
b2 = 020000 e2 = 020116 f2 = 0220
data map `core410`
b1 = 020000 e1 = 020200 f1 = 02000
b2 = 0175400 e2 = 0200000 f2 = 02200
Sv
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
Sq
```

```
adb map411 core411
Sm
text map `map411`
b1 = 0 e1 = 0200 f1 = 020
b2 = 0 e2 = 0116 f2 = 0220
data map `core411`
b1 = 0 e1 = 0200 f1 = 02000
b2 = 0175400 e2 = 0200000 f2 = 02200
Sv
variables
d = 0200
m = 0411
s = 02400
t = 0200
Sq
```



**Figure 10: Simple C program for illustrating formatting and patching**

```
char str1[] "This is a character string";
int one 1;
int number 456;
long lnum 1234;
float fpt 1.25;
char str2[] "This is the second character string";
main()
{
 one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

adb map410 core410

< b,-1/8ona

```

020000: 0 064124 071551 064440 020163 020141 064143 071141
_str1+016: 061541 062564 020162 072163 064562 063556 0 02
_number:
_number: 0710 0 02322040240 0 064124 071551 064440
_str2+06: 020163 064164 020145 062563 067543 062156 061440 060550
_str2+026: 060562 072143 071145 071440 071164 067151 0147 0
savr5+02: 0 0 0 0 0 0 0 0

```

< b,20/4o4^8Cn

```

020000: .0 064124 071551 064440 '@@ This i
 020163 020141 064143 071141 s a char
 061541 062564 020162 072163 acter st
 064562 063556 0 02 ring@@@b@'
_number: 0710 0 02322040240 H@a@'@'R@d @@
 0 064124 071551 064440 '@@ This i
 020163 064164 020145 062563 s the se
 067543 062156 061440 060550 cond cha
 060562 072143 071145 071440 racter s
 071164 067151 0147 0 tring@@@'
 0 0 0 0 '@@@@'@@@'@@@'
 0 0 0 0 '@@@@'@@@'@@@'

```

data address not found

< b,20/4o4^8t8cna

```

020000: 0 064124 071551 064440 This i
_str1+06: 020163 020141 064143 071141 s a char
_str1+016: 061541 062564 020162 072163 acter st
_str1+026: 064562 063556 0 02 ring
_number:
_number: 0710 0 02322040240 HR This i
_fpt+02: 0 064124 071551 064440
_str2+06: 020163 064164 020145 062563 s the se
_str2+016: 067543 062156 061440 060550 cond cha
_str2+026: 060562 072143 071145 071440 racter s
_str2+036: 071164 067151 0147 0 tring
savr5+02: 0 0 0 0
savr5+012: 0 0 0 0

```

data address not found

< b,10/2b8t^2cn

```

020000: 0 0
_str1: 0124 0150 Th
 0151 0163 is
 040 0151 i
 0163 040 s
 0141 040 a
 0143 0150 ch
 0141 0162 ar
 0141 0143 ac
 0164 0145 te

```

SQ

Figure 12: Directory and inode dumps

```
adb dir -
=nt"Inode"t"Name"
0,-1?ut14cn
```

```
0: Inode Name
 652 .
 82 ..
 5971 cap.c
 5323 cap
 0 pp
```

```
adb /dev/src -
```

```
02000> b
```

```
?m< b
```

```
new map '/dev/src'
```

```
b1 = 02000 e1 = 0100000000 f1 = 0
```

```
b2 = 0 e2 = 0 f2 = 0
```

```
$v
```

```
variables
```

```
b = 02000
```

```
<b,-1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
```

```
02000: flags 073145
 links,uid,gid 0163 0164 0141
 size 0162 10356
 addr 28770 8236 25956 27766 25455 8236 25956 25206
 times1976 Feb 5 08:34:56 1975 Dec 28 10:55:15
```

```
02040: flags 024555
 links,uid,gid 012 0163 0164
 size 0162 25461
 addr 8308 30050 8294 25130 15216 26890 29806 10784
 times1976 Aug 17 12:16:51 1976 Aug 17 12:16:51
```

```
02100: flags 05173
 links,uid,gid 011 0162 0145
 size 0147 29545
 addr 25972 8306 28265 8308 25642 15216 2314 25970
 times1977 Apr 2 08:58:01 1977 Feb 5 10:21:44
```

## ADB Summary

### Command Summary

- a) formatted printing
  - ? *format* print from *a.out* file according to *format*
  - / *format* print from *core* file according to *format*
  - = *format* print the value of *dot*
  - ?w *expr* write expression into *a.out* file
  - /w *expr* write expression into *core* file
  - ?l *expr* locate expression in *a.out* file
- b) breakpoint and program control
  - :b set breakpoint at *dot*
  - :c continue running program
  - :d delete breakpoint
  - :k kill the program being debugged
  - :r run *a.out* file under ADB control
  - :s single step
- c) miscellaneous printing
  - \$b print current breakpoints
  - \$c C stack trace
  - \$e external variables
  - \$f floating registers
  - \$m print ADB segment maps
  - \$q exit from ADB
  - \$r general registers
  - \$s set offset for symbol match
  - \$v print ADB variables
  - \$w set output line width
- d) calling the shell
  - ! call *shell* to read rest of line
- e) assignment to variables
  - > *name* assign dot to variable or register *name*

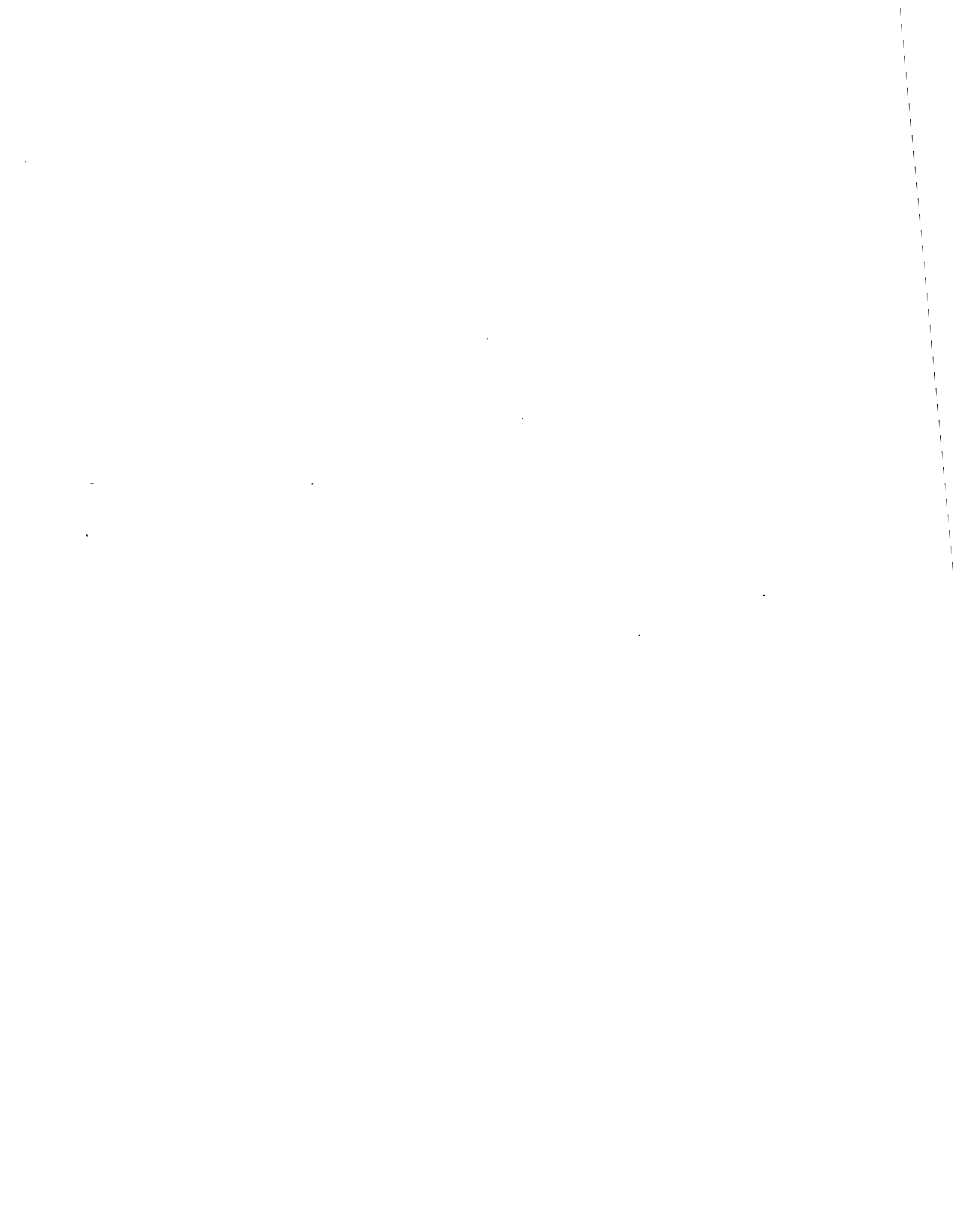
### Format Summary

- a the value of dot
- b one byte in octal
- c one byte as a character
- d one word in decimal
- f two words in floating point
- i PDP 11 instruction
- o one word in octal
- n print a newline
- r print a blank space
- s a null terminated character string
- $\pi$  move to next *n* space tab
- u one word as unsigned integer
- x hexadecimal
- Y date
- ^ backup dot
- "..." print string

### Expression Summary

- a) expression components
  - decimal integer e.g. 256
  - octal integer e.g. 0277
  - hexadecimal e.g. #ff
  - symbols e.g. flag \_main main.argc
  - variables e.g. < b
  - registers e.g. < pc < r0
  - (expression) expression grouping
- b) dyadic operators
  - + add
  - subtract
  - \* multiply
  - % integer division
  - & bitwise and
  - | bitwise or
  - # round up to the next multiple
- c) monadic operators
  - not
  - \* contents of location
  - integer negate





## Lint, a C Program Checker

S. C. Johnson

Bell Laboratories  
Murray Hill, New Jersey 07974

### ABSTRACT

*Lint* is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

*Lint* accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

July 26, 1978

# Lint, a C Program Checker

S. C. Johnson

Bell Laboratories  
Murray Hill, New Jersey 07974

## Introduction and Usage

Suppose there are two C<sup>1</sup> source files, *file1.c* and *file2.c*, which are ordinarily compiled and loaded together. Then the command

```
lint file1.c file2.c
```

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

```
lint -p file1.c file2.c
```

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the `-p` by `-h` will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying `-hp` gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

## A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous "halting problem," known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

*Lint* tries to give information with a high degree of relevance. Messages of the form "xxx might be a bug" are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

## Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These "errors of commission" rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!



*Lint* complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit `extern` statements but are never referenced; thus the statement

```
extern float sin();
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the `-x` flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The `-v` option is available to suppress the printing of complaints about unused arguments. When `-v` is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The `-u` flag may be used to suppress the spurious messages which might otherwise appear.

### Set/Used Information

*Lint* attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two `goto`'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

### Flow of Control

*Lint* attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following `goto`, `break`, `continue`, or `return` statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases `while( 1 )` and `for(;;)` as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

*Lint* has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to `exit` may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint*; a `break` statement that cannot be reached causes no message. Programs generated by *yacc*,<sup>2</sup> and especially *lex*,<sup>3</sup> may have literally hundreds of unreachable `break` statements. The `-O` flag in the C

compiler will often eliminate the resulting object code inefficiency. Thus, these unreachable statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the `-b` option.

### Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function "values" which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return(expr);
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

```
function name contains return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f (a) {
 if (a) return (3);
 g ();
}
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the "noise" messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in "working" programs: the desired function value just happened to have been computed in the function return register!

### Type Checking

*Lint* enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a return statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the `->` be a pointer to structure, the left operand of the `.` be a structure, and the right operand of

these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are **=**, initialization, **==**, **!=**, and function arguments and return values.

### Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The **-c** flag controls the printing of comments about casts. When **-c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

### Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from **-128** to **127**. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
...
if((c = getchar()) < 0) ...
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say "nonportable character comparison".

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value **3**, the problem disappears if the bitfield is declared to have type **unsigned**.

### Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **-a** flag.

### Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint*; the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The `-h` flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the `*` does nothing; this provokes the message "null effect" from *lint*. The program fragment

```
unsigned x ;
if(x < 0) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if(x > 0) ...
```

is equivalent to

```
if(x != 0)
```

which may not be the intended action. *Lint* will say "degenerate unsigned comparison" in these cases. If one says

```
if(1 != 0)
```

*lint* will report "constant in conditional context", since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if(x&077 == 0) ...
```

or

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the `-h` flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

### Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., `=+`, `=-`, ...) could cause ambiguous expressions, such as

```
a ==-1 ;
```

which could be taken as either

```
a == -1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (`+=`, `-=`, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned

operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties: for example,

```
int x (-1) ;
```

looks somewhat like the beginning of a function declaration:

```
int x (y) { . . .
```

and the compiler must read a fair ways past *x* in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

### Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message "possible pointer alignment problem" results from this situation whenever either the `-p` or `-h` flags are in effect.

### Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

*Lint* checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will draw the complaint:

```
warning: i evaluation order undefined
```

### Implementation

*Lint* consists of two programs and a driver. The first program is a version of the Portable C Compiler<sup>4,5</sup> which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file

which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

### Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the `-p` flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint -p* causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcdic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions ("left to right") on GCOS and IBM, and low to high ("right to left") on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

---

†UNIX is a Trademark of Bell Laboratories.

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed `unsigned`. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program, although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

### Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for "illegal" type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The `-v` flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

### Library Declaration Files

*Lint* accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

*Lint* library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the `-p` flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The `-n` flag can be used to suppress all library checking.

### Bugs, etc.

*Lint* was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the typedef is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

*Lint* shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are



pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

### References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
3. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson and D. M. Ritchie, "UNIX Time-Sharing System: Portability of C Programs and the UNIX System," *Bell Sys. Tech. J.* 57(6) pp. 2021-2048 (1978).
5. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).

**Appendix: Current Lint Options**

The command currently has the form

lint [-options ] files... library-descriptors...

The options are

- h** Perform heuristic checks
- p** Perform portability checks
- v** Don't report unused arguments
- u** Don't report unused or undefined externals
- b** Report unreachable **break** statements.
- x** Report unused external declarations
- a** Report assignments of **long** to **int** or shorter.
- c** Complain about questionable casts
- n** No library checking is done
- s** Same as **h** (for historical reasons)







## **Make — A Program for Maintaining Computer Programs**

*S. J. Feldman*

Bell Laboratories  
Murray Hill, New Jersey 07974

### **ABSTRACT**

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

*Make* also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978

# Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories  
Murray Hill, New Jersey 07974

## Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

`make`

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last "make". In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

*Make* is most useful for medium-sized programming projects: it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

## Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *IS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c*



and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog : x.o y.o z.o
 cc x.o y.o z.o -IS -o prog
x.o y.o : defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

*Make* operates using three sources of information: a user-supplied description file (as above), file names and "last-modified" times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three ".o" files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three ".c" files corresponding to the needed ".o" files, and uses built-in information on how to generate an object from a source file (*i.e.*, issue a "cc -c" command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog : x.o y.o z.o
 cc x.o y.o z.o -IS -o prog
x.o : x.c defs
 cc -c x.c
y.o : y.c defs
 cc -c y.c
z.o : z.c
 cc -c z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new ".o" files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry "save" might be included to copy a certain set of files, or an entry "cleanup"

might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

*Make* has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. `$$` is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: `$*`, `$@`, `$?`, and `$<`. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
 cc $(OBJECTS) $(LIBES) -o prog
...
```

The command

```
make
```

loads the three object files with the `lS` library. The command

```
make "LIBES = -ll -lS"
```

loads them with both the Lex ("`-ll`") and the Standard ("`-lS`") libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX† commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

### Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (`#`) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

---

†UNIX is a Trademark of Bell Laboratories.

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

```
target1 [target2 . . .] [:] [dependent1 . . .] [; commands] [# . . .]
[(tab) commands] [# . . .]
```

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters "\*" and "?" are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.
2. In the double-colon case, a command sequence may be associated with each dependency line: if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the "-i" flag has been specified on the *make* command line, if the fake target name ".IGNORE" appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process: the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. \$@ is set to the name of the file to be "made". \$? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), \$< is the name of the related file that caused the action, and \$\* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ".DEFAULT" are used. If there is no such name, *make* prints a message and stops.

### Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

```
make [flags] [macro definitions] [targets]
```

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

- i Ignore error codes returned by invoked commands. This mode is entered if the fake target name `IGNORE` appears in the description file.
- s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name `SILENT` appears in the description file.
- r Do not use the built-in rules.
- n No execute mode. Print commands, but do not execute them. Even lines beginning with an `@` sign are printed.
- t Touch the target files (causing them to be up to date) rather than issue the usual commands.
- q Question. The `make` command returns a zero or non-zero status code depending on whether the target file is or is not up to date.
- p Print out the complete set of macro definitions and target descriptions
- d Debug mode. Print out detailed information on files and times examined.
- f Description file name. The next argument is assumed to be the name of a description file. A file name of `-` denotes the standard input. If there are no `-f` arguments, the file named `makefile` or `Makefile` in the current directory is read. The contents of the description files override the built-in rules if they are present).

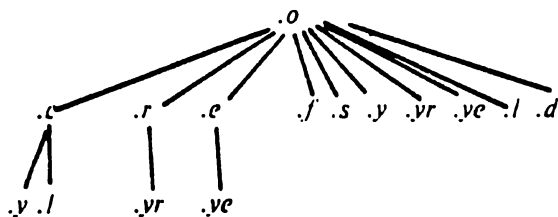
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is `made`.

### Implicit Rules

The `make` program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

<code>.o</code>	Object file
<code>.c</code>	C source file
<code>.e</code>	Efl source file
<code>.r</code>	Ratfor source file
<code>.f</code>	Fortran source file
<code>.s</code>	Assembler source file
<code>.y</code>	Yacc-C source grammar
<code>.yr</code>	Yacc-Ratfor source grammar
<code>.ye</code>	Yacc-Efl source grammar
<code>.l</code>	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the "newcc" command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=-O"
```

causes the optimizing C compiler to be used.

### Example

As an example of the use of *make*, we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
Description file for the Make command
P = und -3 | opr -r2 # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES = -IS
LINT = lint -p
CFLAGS = -O
make: $(OBJECTS)
 cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
 size make
$(OBJECTS): defs
gram.o: lex.c
cleanup:
 -rm *.o gram.c
 -du
install:
 @size make /usr/bin/make
 cp make /usr/bin/make ; rm make
print: $(FILES) # print recently changed files
 pr $? | SP
 touch print
test:
 make -dp | grep -v TIME > 1zap
 /usr/bin/make -dp | grep -v TIME > 2zap
 diff 1zap 2zap
 rm 1zap 2zap
lint : dosys.c doname.c files.c main.c misc.c version.c gram.c
 $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
 rm gram.c
arch:
 ar uv /sys/source/s2/make.a $(FILES)
```

*Make* usually prints out each command before issuing it. The following output results from typing the simple command

**make**

in a directory containing only the source and description file:

```
cc -c version.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o gram.o -IS -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits

results from the "size make" command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The "print" entry prints only the files that have been changed since the last "make print" command. A zero-length file *print* is maintained to keep track of the time of the printing; the \$? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr -sp"
 or
make print "P= cat >zap"
```

### Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a "#include "defs"" line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the "-n" option is very useful. The command

```
make -n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the "-t" (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag ("-d") causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

### Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

### References

1. S. C. Johnson, "Yacc - Yet Another Compiler-Compiler", Bell Laboratories Computing Science Technical Report #32, July 1978.
2. M. E. Lesk, "Lex - A Lexical Analyzer Generator", Computing Science Technical Report #39, October 1975.

## Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the "--r" flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ".SUFFIXES"; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ".r" file to a ".o" file is thus ".r.o". If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule ".r.o" is used. If a command is generated by using one of these suffixing rules, the macro \$\* is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro \$< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ".SUFFIXES" in his own description file; the dependents will be added to the usual list. A ".SUFFIXES" line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
 S(CC) S(CFLAGS) -c $<
.e.o .r.o .f.o :
 S(EC) S(RFLAGS) S(EFLAGS) S(FFLAGS) -c $<
.s.o :
 S(AS) -o $@ $<
.y.o :
 S(YACC) S(YFLAGS) $<
 S(CC) S(CFLAGS) -c y.tab.c
 rm y.tab.c
 mv y.tab.o $@
.y.c :
 S(YACC) S(YFLAGS) $<
 mv y.tab.c $@
```







# UNIX Implementation

*K. Thompson*

Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX<sup>†</sup> kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

## 1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

## 2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

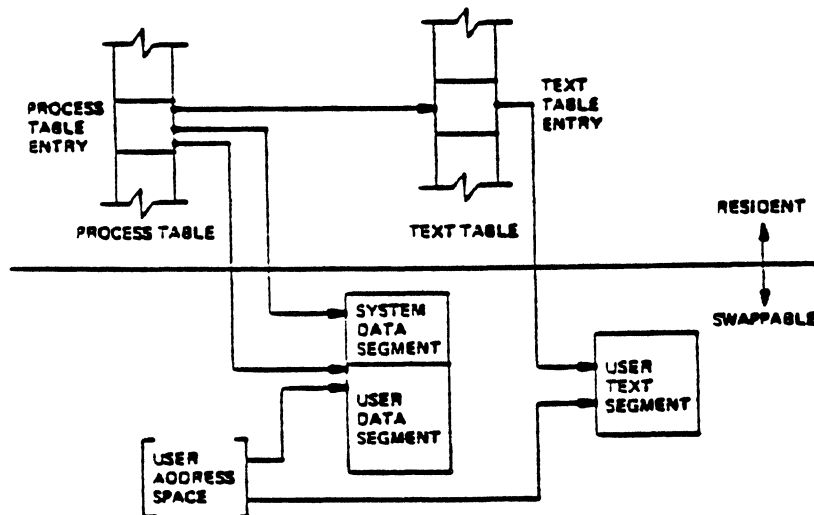


Fig. 1—Process control data structure.

### 2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **execs** the second program. This is analogous to a "goto." If a program wishes to regain control after **execing** a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.<sup>1</sup>

### 2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

### 2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,<sup>2</sup> in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.<sup>3</sup>

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runnable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

### 3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character

I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

### **3.1. Block I/O system**

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

### **3.2. Character I/O system**

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

### 3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

### 3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

### 3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.



#### 4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.5M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

#### 4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are open, create, read, write, seek, and close. The data structures maintained are shown in Fig. 2.

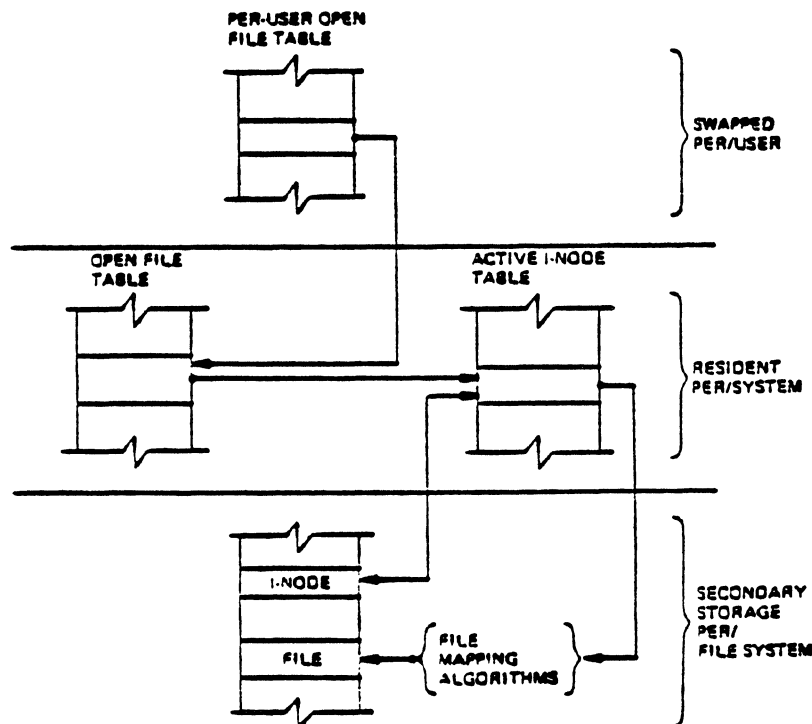


Fig. 2—File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer

and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe**. Implementation of **pipes** consists of implied **seeks** before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

#### 4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

#### 4.3. Other system functions

There are some other things that the system does for the user—a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.<sup>5</sup> Each user may have his own command language. Maintenance of such code is as easy as maintaining user

code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.<sup>6</sup>

#### References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices* 12(4) pp. 40-50 (April 1977).
2. E. W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).
3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* 57(6) pp. 1971-1990 (1978).
6. E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass. (1972).





# FSCK—The UNIX File System Check Program

*T. J. Kowalski*

Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

The UNIX† File System Check Program (*fscck*) is an interactive file system check and repair program. *fscck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. *fscck* is frequently able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fscck*. Both the program and the interaction between the program and the operator are described.

## 1. INTRODUCTION

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. No changes are made to any file system by *fscck* without prior operator approval.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of heuristically sound corrective actions used by *fscck* (the Coast Guard to the rescue) is presented.

## 2. UPDATE OF THE FILE SYSTEM

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens in the event of a permanent interruption in this sequence, it is important to understand the order in which the update requests were probably being honored. Knowing which pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the super-block, inodes, indirect blocks, data blocks (directories and files), and free-list blocks.

### 2.1 Super-Block

The super-block contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

The super-block of a mounted file system (the root file system is always mounted) is written to the file system whenever the file system is unmounted or a *sync* command is issued.

---

† UNIX is a Trademark of Bell Telephone Laboratories.

## 2.2 Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system upon closure<sup>1</sup> of the file associated with the inode.

## 2.3 Indirect Blocks

There are three types of indirect blocks: single-indirect, double-indirect and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each one of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released<sup>2</sup> by the operating system.

## 2.4 Data Blocks

A data block may contain file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system whenever they have been modified and released by the operating system.

## 2.5 First Free-List Block

The super-block contains the first free-list block. The free-list blocks are a list of all blocks that are not allocated to the super-block, inodes, indirect blocks, or data blocks. Each free-list block contains a count of the number of entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system whenever they have been modified and released by the operating system.

## 3. CORRUPTION OF THE FILE SYSTEM

A file system can become corrupted in a variety of ways. The most common of these ways are improper shutdown procedures and hardware failures.

### 3.1 Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to *sync* the system prior to halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

### 3.2 Hardware Failure

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

## 4. DETECTION AND CORRECTION OF CORRUPTION

A quiescent<sup>3</sup> file system may be checked for structural integrity by performing consistency

1. All in core blocks are also written to the file system upon issue of a *sync* system call.

2. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by *LINK* or a *sync* command is issued.

3. I.e., unmounted and not being written on.



checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. A quiescent state is important during the checking of a file system because of the multi-pass nature of the *fsck* program.

When an inconsistency is discovered *fsck* reports the inconsistency for the operator to choose a corrective action.

Discussed in this section are how to discover inconsistencies and possible corrective actions for the super-block, the inodes, the indirect blocks, the data blocks containing directory entries, and the free-list blocks. These corrective actions can be performed interactively by the *fsck* command under control of the operator.

#### 4.1 Super-Block

One of the most common corrupted items is the super-block. The super-block is prone to corruption because every change to the file system's blocks or inodes modifies the super-block.

The super-block and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a *sync* command.

The super-block can be checked for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.

**4.1.1 File-System Size and Inode-List Size.** The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The number of inodes must be less than 65,535. The file-system size and inode-list size are critical pieces of information to the *fsck* program. While there is no way to actually check these sizes, *fsck* can check for them being within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

**4.1.2 Free-Block List.** The free-block list starts in the super-block and continues through the free-list blocks of the file system. Each free-list block can be checked for a list count out of range, for block numbers out of range, and for blocks already allocated within the file system. A check is made to see that all the blocks in the file system were found.

The first free-block list is in the super-block. *Fsck* checks the list count for a value of less than zero or greater than fifty. It also checks each block number for a value of less than the first data block in the file system or greater than the last block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is non-zero, the next free-list block is read in and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the free-block list, then *fsck* may rebuild it, excluding all blocks in the list of allocated blocks.

**4.1.3 Free-Block Count.** The super-block contains a count of the total number of free blocks within the file system. *Fsck* compares this count to the number of blocks it found free within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-block count.

**4.1.4 Free-Inode Count.** The super-block contains a count of the total number of free inodes within the file system. *Fsck* compares this count to the number of inodes it found free within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-inode count.

#### 4.2 Inodes

An individual inode is not as likely to be corrupted as the super-block. However, because of the great number of active inodes, there is almost as likely a chance for corruption in the inode list as in the super-block.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

**4.2.1 Format and Type.** Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes may be one of four types: regular inode, directory inode, special block inode, and special character inode. If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states: unallocated, allocated, and neither unallocated nor allocated. This last state indicates an incorrectly formatted inode. An inode can get in this state if bad data is written into the inode list through, for example, a hardware failure. The only possible corrective action is for *fsck* is to clear the inode.

**4.2.2 Link Count.** Contained in each inode is a count of the total number of directory entries linked to the inode.

*Fsck* verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, calculating an actual link count for each inode.

If the stored link count is non-zero and the actual link count is zero, it means that no directory entry appears for the inode. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated.

If the stored link count is non-zero and the actual link count is zero, *fsck* may link the disconnected file to the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, *fsck* may replace the stored link count by the actual link count.

**4.2.3 Duplicate Blocks.** Contained in each inode is a list or pointers to lists (indirect blocks) of all the blocks claimed by the inode.

*Fsck* compares each block number claimed by an inode to a list of already allocated blocks. If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number. If there are any duplicate blocks, *fsck* will make a partial second pass of the inode list to find the inode of the duplicated block, because without examining the files associated with these inodes for correct content, there is not enough information available to decide which inode is corrupted and should be cleared. Most times, the inode with the earliest modify time is incorrect, and should be cleared.

This condition can occur by using a file system with blocks claimed by both the free-block list and by other parts of the file system.

If there is a large number of duplicate blocks in an inode, this may be due to an indirect block not being written to the file system.

*Fsck* will prompt the operator to clear both inodes.

**4.2.4 Bad Blocks.** Contained in each inode is a list or pointer to lists of all the blocks claimed by the inode.

*Fsck* checks each block number claimed by an inode for a value lower than that of the first data block, or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be due to an indirect block not being written to the file system.

*Fsck* will prompt the operator to clear both inodes.

**4.2.5 Size Checks.** Each inode contains a thirty-two bit (four-byte) size field. This size indicates the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of sixteen characters, or the number of blocks actually used not matching that indicated by the inode size.

A directory inode within the UNIX file system has the directory bit on in the inode mode word. The directory size must be a multiple of sixteen because a directory entry contains sixteen bytes of information (two bytes for the inode number and fourteen bytes for the file or directory name).

*Fsck* will warn of such directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by computing from the size field the number of blocks that should be associated with the inode and comparing it to the actual number of blocks claimed by the inode.

*Fsck* calculates the number of blocks that there should be in an inode by dividing the number of characters in a inode by the number of characters per block (512) and rounding up. *Fsck* adds one block for each indirect block associated with the inode. If the actual number of blocks does not match the computed number of blocks, *fsck* will warn of a possible file-size error. This is only a warning because UNIX does not fill in blocks in files created in random order.

### 4.3 Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns it.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, apply iteratively Sections 4.2.3 and 4.2.4 to each level of indirect blocks.

### 4.4 Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. *Fsck* does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories which are disconnected from the file system.

If a directory entry inode number points to an unallocated inode, then *fsck* may remove that directory entry. This condition probably occurred because the data blocks containing the directory entries were modified and written to the file system while the inode was not yet written out.

If a directory entry inode number is pointing beyond the end of the inode list, *fsck* may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ".." should be the second entry in the directory data block. Its value should be equal to the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory).

If the directory inode numbers are incorrect, *fsck* may replace them by the correct values.

*Fsck* checks the general connectivity of the file system. If directories are found not to be linked into the file system, *fsck* will link the directory back into the file system in the *lost+found* directory. This condition can be caused by inodes being written to the file system with the corresponding directory data blocks not being written to the file system.

#### 4.5 Free-List Blocks

Free-list blocks are owned by the super-block. Therefore, inconsistencies in free-list blocks directly affect the super-block.

Inconsistencies that can be checked are a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free-list blocks see Section 4.1.2.

#### ACKNOWLEDGEMENT

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS.

#### REFERENCES

- [1] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1905-29.
- [2] Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1* (January 1978).
- [3] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1931-46.

## Appendix—F<sub>s</sub>CK ERROR CONDITIONS

### 1. CONVENTIONS

*F<sub>s</sub>ck* is a multi-pass file system check program. Each file system pass invokes a different Phase of the *f<sub>s</sub>ck* program. After the initial setup, *f<sub>s</sub>ck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the free-block list (possibly rebuilding it), and performs some cleanup.

When an inconsistency is detected, *f<sub>s</sub>ck* reports the error condition to the operator. If a response is required, *f<sub>s</sub>ck* prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *f<sub>s</sub>ck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

### 2. INITIALIZATION

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file.

#### C option?

C is not a legal option to *f<sub>s</sub>ck*; legal options are *-y*, *-n*, *-s*, *-S*, and *-t*. *F<sub>s</sub>ck* terminates on this error condition. See the *f<sub>s</sub>ck(1M)* manual entry for further detail.

#### Bad *-t* option

The *-t* option is not followed by a file name. *F<sub>s</sub>ck* terminates on this error condition. See the *f<sub>s</sub>ck(1M)* manual entry for further detail.

#### Invalid *-s* argument, defaults assumed

The *-s* option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. *F<sub>s</sub>ck* assumes a default value of 400 blocks-per-cylinder and 9 blocks-to-skip. See the *f<sub>s</sub>ck(1M)* manual entry for more details.

#### Incompatible options: *-n* and *-s*

It is not possible to salvage the free-block list without modifying the file system. *F<sub>s</sub>ck* terminates on this error condition. See the *f<sub>s</sub>ck(1M)* manual entry for further detail.

#### Can't get memory

*F<sub>s</sub>ck*'s request for memory for its virtual memory tables failed. This should never happen. *F<sub>s</sub>ck* terminates on this error condition. See a guru.

#### Can't open checklist file: F

The default file system checklist file **F** (usually *letelchecklist*) can not be opened for reading. *F<sub>s</sub>ck* terminates on this error condition. Check access modes of **F**.

#### Can't stat root

*F<sub>s</sub>ck*'s request for statistics about the root directory *"/*" failed. This should never happen. *F<sub>s</sub>ck* terminates on this error condition. See a guru.

**Can't stat F**

*Fsck*'s request for statistics about the file system **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

**F is not a block or character device**

You have given *fsck* a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of **F**.

**Can't open F**

The file system **F** can not be opened for reading. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

**Size check: fsize X isize Y**

More blocks are used for the inode list **Y** than there are blocks in the file system **X**, or there are more than 65,535 inodes in the file system. It ignores this file system and continues checking the next file system given. See Section 4.1.1.

**Can't create F**

*Fsck*'s request to create a scratch file **F** failed. It ignores this file system and continues checking the next file system given. Check access modes of **F**.

**CAN NOT SEEK: BLK B (CONTINUE)**

*Fsck*'s request for moving to a specified block number **B** in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

- YES     attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".
- NO     terminate the program.

**CAN NOT READ: BLK B (CONTINUE)**

*Fsck*'s request for reading a specified block number **B** in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

- YES     attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".
- NO     terminate the program.

**CAN NOT WRITE: BLK B (CONTINUE)**

*FscK*'s request for writing a specified block number **B** in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

- YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fscK* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fscK* will terminate with the message "Fatal I/O error".
- NO terminate the program.

**3. PHASE 1: CHECK BLOCKS AND SIZES**

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

**UNKNOWN FILE TYPE I=I (CLEAR)**

The mode word of the inode **I** indicates that the inode is not a special character inode, special character inode, regular inode, or directory inode. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

- YES de-allocate inode **I** by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.
- NO ignore this error condition.

**LINK COUNT TABLE OVERFLOW (CONTINUE)**

An internal table for *fscK* containing allocated inodes with a link count of zero has no more room. Recompile *fscK* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

- YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fscK* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.
- NO terminate the program.

**B BAD I=I**

Inode **I** contains block number **B** with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the EXCESSIVE BAD BLKS error condition in Phase 1 if inode **I** has too many block numbers outside the file system range. This error condition will always invoke the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.4.

**EXCESSIVE BAD BLKS I=I (CONTINUE)**

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode I. See Section 4.2.4.

Possible responses to the CONTINUE prompt are:

- YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.
- NO terminate the program.

**B DUP I=I**

Inode I contains block number B which is already claimed by another inode. This error condition may invoke the EXCESSIVE DUP BLKS error condition in Phase 1 if inode I has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.3.

**EXCESSIVE DUP BLKS I=I (CONTINUE)**

There is more than a tolerable number (usually 10) of blocks claimed by other inodes. See Section 4.2.3.

Possible responses to the CONTINUE prompt are:

- YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.
- NO terminate the program.

**DUP TABLE OVERFLOW (CONTINUE)**

An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

- YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.
- NO terminate the program.

**POSSIBLE FILE SIZE ERROR I=I**

The inode I size does not match the actual number of blocks used by the inode. This is only a warning. See Section 4.2.5.

**DIRECTORY MISALIGNED I=I**

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning. See Section 4.2.5.

**PARTIALLY ALLOCATED INODE I=I (CLEAR)**

Inode I is neither allocated nor unallocated. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

- YES de-allocate inode I by zeroing its contents.
- NO ignore this error condition.



#### 4. PHASE 1B: RESCAN FOR MORE DUPS

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This section lists the error condition when the duplicate block is found.

##### **B DUP I=I**

Inode **I** contains block number **B** which is already claimed by another inode. This error condition will always invoke the BAD/DUP error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1. See Section 4.2.3.

#### 5. PHASE 2: CHECK PATH-NAMES

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

##### **ROOT INODE UNALLOCATED. TERMINATING.**

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate. See Section 4.2.1.

##### **ROOT INODE NOT DIRECTORY (FIX)**

The root inode (usually inode number 2) is not directory inode type. See Section 4.2.1.

Possible responses to the FIX prompt are:

- YES    replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.
- NO     terminate the program.

##### **DUPS/BAD IN ROOT INODE (CONTINUE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system. See Section 4.2.3 and 4.2.4.

Possible responses to the CONTINUE prompt are:

- YES    ignore the DUPS/BAD error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in a large number of other error conditions.
- NO     terminate the program.

##### **I OUT OF RANGE I=I NAME=F (REMOVE)**

A directory entry **F** has an inode number **I** which is greater than the end of the inode list. See Section 4.4.

Possible responses to the REMOVE prompt are:

- YES    the directory entry **F** is removed.
- NO     ignore this error condition.

**UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE)**

A directory entry **F** has an inode **I** without allocate mode bits. The owner **O**, mode **M**, size **S**, modify time **T**, and file name **F** are printed. See Section 4.4.

Possible responses to the REMOVE prompt are:

- YES the directory entry **F** is removed.
- NO ignore this error condition.

**DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry **F**, directory inode **I**. The owner **O**, mode **M**, size **S**, modify time **T**, and directory name **F** are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

- YES the directory entry **F** is removed.
- NO ignore this error condition.

**DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry **F**, inode **I**. The owner **O**, mode **M**, size **S**, modify time **T**, and file name **F** are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

- YES the directory entry **F** is removed.
- NO ignore this error condition.

**6. PHASE 3: CHECK CONNECTIVITY**

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

**UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)**

The directory inode **I** was not connected to a directory entry when the file system was traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of directory inode **I** are printed. See Section 4.4 and 4.2.2.

Possible responses to the RECONNECT prompt are:

- YES reconnect directory inode **I** to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode **I** to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.
- NO ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

**SORRY. NO lost+found DIRECTORY**

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck(1M)* manual entry for further detail.

**SORRY. NO SPACE IN *lost+found* DIRECTORY**

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck(1M)* manual entry for further detail.

**DIR I=I1 CONNECTED. PARENT WAS I=I2**

This is an advisory message indicating a directory inode I1 was successfully connected to the *lost+found* directory. The parent inode I2 of the directory inode I1 is replaced by the inode number of the *lost+found* directory. See Section 4.4 and 4.2.2.

**7. PHASE 4: CHECK REFERENCE COUNTS**

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free-inode counts.

**UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)**

Inode I was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.2.

Possible responses to the RECONNECT prompt are:

- YES      reconnect inode I to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode I to *lost+found*.
- NO      ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

**SORRY. NO *lost+found* DIRECTORY**

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check access modes of *lost+found*.

**SORRY. NO SPACE IN *lost+found* DIRECTORY**

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*.

**(CLEAR)**

The inode mentioned in the immediately previous error condition can not be reconnected. See Section 4.2.2.

Possible responses to the CLEAR prompt are:

- YES      de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.
- NO      ignore this error condition.

**LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)**

The link count for inode **I** which is a file, is **X** but should be **Y**. The owner **O**, mode **M**, size **S**, and modify time **T** are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

- YES    replace the link count of file inode **I** with **Y**.
- NO     ignore this error condition.

**LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)**

The link count for inode **I** which is a directory, is **X** but should be **Y**. The owner **O**, mode **M**, size **S**, and modify time **T** of directory inode **I** are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

- YES    replace the link count of directory inode **I** with **Y**.
- NO     ignore this error condition.

**LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)**

The link count for **F** inode **I** is **X** but should be **Y**. The name **F**, owner **O**, mode **M**, size **S**, and modify time **T** are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

- YES    replace the link count of inode **I** with **Y**.
- NO     ignore this error condition.

**UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)**

Inode **I** which is a file, was not connected to a directory entry when the file system was traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Section 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

- YES    de-allocate inode **I** by zeroing its contents.
- NO     ignore this error condition.

**UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)**

Inode **I** which is a directory, was not connected to a directory entry when the file system was traversed. The owner **O**, mode **M**, size **S**, and modify time **T** of inode **I** are printed. See Section 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

- YES    de-allocate inode **I** by zeroing its contents.
- NO     ignore this error condition.

**BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode I. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

- YES de-allocate inode I by zeroing its contents.
- NO ignore this error condition.

**BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)**

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode I. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

- YES de-allocate inode I by zeroing its contents.
- NO ignore this error condition.

**FREE INODE COUNT WRONG IN SUPERBLK (FIX)**

The actual count of the free inodes does not match the count in the super-block of the file system. See Section 4.1.4.

Possible responses to the FIX prompt are:

- YES replace the count in the super-block by the actual count.
- NO ignore this error condition.

**8. PHASE 5: CHECK FREE LIST**

This phase concerns itself with the free-block list. This section lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and the total free-block count incorrect.

**EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)**

The free-block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system. See Section 4.1.2 and 4.2.4.

Possible responses to the CONTINUE prompt are:

- YES ignore the rest of the free-block list and continue the execution of *fsck*. This error condition will always invoke the BAD BLKS IN FREE LIST error condition in Phase 5.
- NO terminate the program.

**EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE)**

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list. See Section 4.1.2 and 4.2.3.

Possible responses to the CONTINUE prompt are:

- YES ignore the rest of the free-block list and continue the execution of *fsck*. This error condition will always invoke the DUP BLKS IN FREE LIST error condition in Phase 5.
- NO terminate the program.

**BAD FREEBLK COUNT**

The count of free blocks in a free-list block is greater than 50 or less than zero. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

**X BAD BLKS IN FREE LIST**

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2 and 4.2.4.

**X DUP BLKS IN FREE LIST**

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2 and 4.2.3.

**X BLK(S) MISSING**

X blocks unused by the file system were not found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

**FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)**

The actual count of free blocks does not match the count in the super-block of the file system. See Section 4.1.3.

Possible responses to the FIX prompt are:

- YES replace the count in the super-block by the actual count.
- NO ignore this error condition.

**BAD FREE LIST (SALVAGE)**

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system. See Section 4.1.2, 4.2.3, and 4.2.4.

Possible responses to the SALVAGE prompt are:

- YES replace the actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.
- NO ignore this error condition.

## 9. PHASE 6: SALVAGE FREE LIST

This phase concerns itself with the free-block list reconstruction. This section lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

### Default free-block list spacing assumed

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 500. The default values of 9 blocks-to-skip and 400 blocks-per-cylinder are used. See the *fsck(1M)* manual entry for further detail.

## 10. CLEANUP

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

### X files Y blocks Z free

This is an advisory message indicating that the file system checked contained X files using Y blocks leaving Z blocks free in the file system.

### \*\*\*\*\* BOOT UNIX (NO SYNC!) \*\*\*\*\*

This is an advisory message indicating that a mounted file system or the root file system has been modified by *fsck*: If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

### \*\*\*\*\* FILE SYSTEM WAS MODIFIED \*\*\*\*\*

This is an advisory message indicating that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

*May 1979*

**INDEX OF MESSAGES**  
(Alphabetically within each section)

**INITIALIZATION**

Bad -t option	7
C option'	7
CAN NOT READ BLK B (CONTINUE)	8
CAN NOT SEEK: BLK B (CONTINUE)	8
CAN NOT WRITE: BLK B (CONTINUE)	9
Can't create F	8
Can't get memory	-
Can't open checklist file: F	7
Can't open F	8
Can't stat F	8
Can't stat root	7
F is not a block or character device	8
Incompatible options: -n and -s	7
Invalid -s argument, defaults assumed	7
Size check: fsize X isize Y	8

**PHASE 1: CHECK BLOCKS AND SIZES**

B BAD I=I	9
B DU P I=I	10
DIRECTORY MISALIGNED I=I	10
DUP TABLE OVERFLOW (CONTINUE)	10
EXCESSIVE BAD BLKS I=I (CONTINUE)	10
EXCESSIVE DUP BLKS I=I (CONTINUE)	10
LINK COUNT TABLE OVERFLOW (CONTINUE)	9
PARTIALLY ALLOCATED INODE I=I (CLEAR)	10
POSSIBLE FILE SIZE ERROR I=I	10
UNKNOWN FILE TYPE I=I (CLEAR)	9

**PHASE 1B: RESCAN FOR MORE DUPS**

B DU P I=I	11
------------	----

**PHASE 2: CHECK PATH-NAMES**

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)	12
DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)	12
DUPS/BAD IN ROOT INODE (CONTINUE)	11
I OUT OF RANGE I=I NAME=F (REMOVE)	11
ROOT INODE NOT DIRECTORY (FIX)	11
ROOT INODE UNALLOCATED. TERMINATING.	11
UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE)	12

**PHASE 3: CHECK CONNECTIVITY**

DIR I=I1 CONNECTED. PARENT WAS I=I2	13
SORRY. NO SPACE IN lost+found DIRECTORY	13
SORRY. NO lost+found DIRECTORY	12
UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)	12



**PHASE 4: CHECK REFERENCE COUNTS**

BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . . . 15  
 BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . . . 15  
 (CLEAR) . . . . . 13  
 FREE INODE COUNT WRONG IN SUPERBLK (FIX) . . . . . 15  
 LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST) . . . . . 14  
 LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST) . . . . . 14  
 LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST) . . . . . 14  
 SORRY. NO SPACE IN lost+found DIRECTORY . . . . . 13  
 SORRY. NO lost+found DIRECTORY . . . . . 13  
 UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . . . 14  
 UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR) . . . . . 14  
 UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT) . . . . . 13

**PHASE 5: CHECK FREE LIST**

BAD FREE LIST (SALVAGE) . . . . . 16  
 BAD FREEBLK COUNT . . . . . 16  
 EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE) . . . . . 15  
 EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE) . . . . . 16  
 FREE BLK COUNT WRONG IN SUPERBLOCK (FIX) . . . . . 16  
 X BAD BLKS IN FREE LIST . . . . . 16  
 X BLK(S) MISSING . . . . . 16  
 X DUP BLKS IN FREE LIST . . . . . 16

**PHASE 6: SALVAGE FREE LIST**

Default free-block list spacing assumed . . . . . 17

**CLEANUP**

\*\*\*\*\* BOOT UNIX (NO SYNC) \*\*\*\*\* . . . . . 17  
 \*\*\*\*\* FILE SYSTEM WAS MODIFIED \*\*\*\*\* . . . . . 17  
 X files Y blocks Z free . . . . . 17







# The UNIX I/O System

*Dennis M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX<sup>†</sup> I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

## Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECTape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

## Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u\_file* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after

---

<sup>†</sup>UNIX is a Trademark of Bell Laboratories.

the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat*; if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u\_base*, *u.u\_count*, and *u.u\_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

## Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: *open*, *close*, *read*, *write*, and *special-function* (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev*; if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a

flag which is non-zero if the file was open for writing in the process which performs the final *close*.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u\_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u\_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u\_segflg* is supplied that indicates, if *on*, that *u.u\_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u\_count* characters from the user's buffer to the device, decrementing *u.u\_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u\_count* goes to 0 or an error occurs, when it returns -1. *Cpass* takes care of interrogating *u.u\_segflg* and updating *u.u\_count*.

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B\_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer + offset*, *count* or *u.u\_base* is odd.

The device's *read* routine is called under conditions similar to *write*, except that *u.u\_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns -1 as the last character specified by *u.u\_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write*; the flag should be *B\_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: (*\*p*) (*dev, v*) where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u\_arg[0...2]*.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the device's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure

```
struct {
 int c_cc; /* character count */
 char *c_cf; /* first character */
 char *c_ci; /* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns -1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or -1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of

characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority*.

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than *PZERO* on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "u." should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup*, (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( )*, *spl5( )*, *spl6( )*, *spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(\*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

## The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf*. Each buffer header contains a pair of pointers (*b\_forw*, *b\_back*) which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers (*av\_forw*, *av\_back*) which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a



residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B\_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. It is logically similar to *bread*, but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *breise* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread*. There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

*Bwrite* is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *breise*, *bwrite*, *bawrite*, or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

**B\_READ** This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B\_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swab* which have a separate argument which indicates read or write.

**B\_DONE** This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if it that the returned buffer actually contains the data in the requested block.

- B\_ERROR** This bit may be set to 1 when *B\_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b\_error* byte of the buffer header may contain an error code if it is non-zero. If *b\_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b\_error*; the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.
- B\_BUSY** This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread*, which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.
- B\_PHYS** This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.
- B\_MAP** This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.
- B\_WANTED** This flag is used in conjunction with the *B\_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelease*) a *wakeup* is given for the block header whenever *B\_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.
- B\_AGE** This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.
- B\_ASYNC** This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.
- B\_DELWRIT** This bit is set by *bdwrite* before releasing the buffer. When *getblk*, while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

### Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B\_DONE* (and possibly the *B\_ERROR*) bits should be set. Then if the *B\_ASYNC* bit is set, *brelease* should be called; otherwise, *wakeup*. In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swab* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this

device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device (*b\_forw*, *b\_back*), and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list (*av\_forw*, *av\_back*) are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B\_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed (*B\_DONE* has been set).

### Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat*, the buffer pointer *bp*, the device number *dev*, and a read-write flag *rw* whose value is either *B\_READ* or *B\_WRITE*. *Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.







# REGENERATING SYSTEM SOFTWARE

*Charles B. Haley*

*Dennis. M. Ritchie*

*Bell Laboratories*

*Murray Hill, New Jersey 07974*

## Introduction

This document discusses how to assemble or compile various parts of the UNIX† system software. This may be necessary because a command or library is accidentally deleted or otherwise destroyed; also, it may be desirable to install a modified version of some command or library routine. A few commands depend to some degree on the current configuration of the system; thus in any new system modifications to some commands are advisable. Most of the likely modifications relate to the standard disk devices contained in the system. For example, the `df(1)` ('disk free') command has built into it the names of the standardly present disk storage drives (e.g. `"/dev/rf0"`, `"/dev/rp0"`). `Df(1)` takes an argument to indicate which disk to examine, but it is convenient if its default argument is adjusted to reflect the ordinarily present devices. The companion document 'Setting up UNIX' discusses which commands are likely to require changes.

## Where Commands and Subroutines Live

The source files for commands and subroutines reside in several subdirectories of the directory `/usr/src`. These subdirectories, and a general description of their contents, are

<code>cmd</code>	Source files for commands.
<code>libc/stdio</code>	Source files making up the 'standard i/o package'.
<code>libc/sys</code>	Source files for the C system call interfaces.
<code>libc/gen</code>	Source files for most of the remaining routines described in section 3 of the manual.
<code>libc/crt</code>	Source files making up the C runtime support package, as in call save-return and long arithmetic.
<code>libc/csu</code>	Source for the C startup routines.
<code>games</code>	Source for (some of) the games. No great care has been taken to try to make it obvious how to compile these; treat it as a game.
<code>libF77</code>	Source for the Fortran 77 runtime library, exclusive of IO.
<code>libI77</code>	Source for the Fortran 77 IO runtime routines.
<code>libdbm</code>	Source for the 'data-base manager' package <code>dbm</code> (3).
<code>libfpsim</code>	Source for the floating-point simulator routine.
<code>libm</code>	Source for the mathematical library.

---

†UNIX is a Trademark of Bell Laboratories.

**libplot**      Source for plotting routines.

### Commands

The regeneration of most commands is straightforward. The 'cmd' directory will contain either a source file for the command or a subdirectory containing the set of files that make up the command. If it is a single file the command

```
cd /usr/src/cmd
cmake cmd_name
```

suffices. (Cmd\_name is the name of the command you are playing with.) The result of the cmake command will be an executable version. If you type

```
cmake -cp cmd_name
```

the result will be copied to /bin (or perhaps /etc or other places if appropriate).

If the source files are in a subdirectory there will be a 'makefile' (see make(1)) to control the regeneration. After changing to the proper directory (cd(1)) you type one of the following:

- make all**      The program is compiled and loaded; the executable is left in the current directory.
- make cp**      The program is compiled and loaded, and the executable is installed. Everything is cleaned up afterwards; for example .o files are deleted.
- make cmp**     The program is compiled and loaded, and the executable is compared against the one in /bin.

Some of the makefiles have other options. Print (cat(1)) the ones you are interested in to find out.

### The Assembler

The assembler consists of two executable files: /bin/as and /lib/as2. The first is the 0-th pass: it reads the source program, converts it to an intermediate form in a temporary file '/tmp/atm0?', and estimates the final locations of symbols. It also makes two or three other temporary files which contain the ordinary symbol table, a table of temporary symbols (like 1:) and possibly an overflow intermediate file. The program /lib/as2 acts as an ordinary multiple pass assembler with input taken from the files produced by /bin/as.

The source files for /bin/as are named '/usr/src/cmd/as/as1?.s' (there are 9 of them); /lib/as2 is produced from the source files '/usr/src/cmd/as/as2?.s'; they likewise are 9 in number. Considerable care should be exercised in replacing either component of the assembler. Remember that if the assembler is lost, the only recourse is to replace it from some backup storage; a broken assembler cannot assemble itself.

### The C Compiler

The C compiler consists of seven routines: '/bin/cc', which calls the phases of the compiler proper, the compiler control line expander '/lib/cpp', the assembler ('as'), and the loader ('ld'). The phases of the C compiler are '/lib/c0', which is the first phase of the compiler; '/lib/c1', which is the second phase of the compiler; and '/lib/c2', which is the optional third phase optimizer. The loss of the C compiler is as serious as that of the assembler.

The source for /bin/cc resides in '/usr/src/cmd/cc.c'. Its loss alone (or that of c2) is not fatal. If needed, prog.c can be compiled by



```
/lib/cpp prog.c > temp0
/lib/c0 temp0 temp1 temp2
/lib/c1 temp1 temp2 temp3
as - temp3
ld -n /lib/crt0.o a.out -lc
```

The source for the compiler proper is in the directory `/usr/src/cmd/c`. The first phase (`/lib/c0`) is generated from the files `c00.c`, ..., `c05.c`, which must be compiled by the C compiler. There is also `c0.h`, a header file *included* by the C programs of the first phase. To make a new `/lib/c0` use

```
make c0
```

Before installing the new `c0`, it is prudent to save the old one someplace.

The second phase of C (`/lib/c1`) is generated from the source files `c10.c`, ..., `c13.c`, the include-file `c1.h`, and a set of object-code tables combined into `table.o`. To generate a new second phase use

```
make c1
```

It is likewise prudent to save `c1` before installing a new version. In fact in general it is wise to save the object files for the C compiler so that if disaster strikes C can be reconstituted without a working version of the compiler.

In a similar manner, the third phase of the C compiler (`/lib/c2`) is made up from the files `c20.c` and `c21.c` together with `c2.h`. Its loss is not critical since it is completely optional.

The set of tables mentioned above is generated from the file `table.s`. This `.s` file is not in fact assembler source; it must be converted by use of the `cvopt` program, whose source and object are located in the C directory. Normally this is taken care of by `make(1)`. You might want to look at the makefile to see what it does.

## UNIX

The source and object programs for UNIX are kept in four subdirectories of `/usr/sys`. In the subdirectory `h` there are several files ending in `.h`; these are header files which are picked up (via `#include ...`) as required by each system module. The subdirectory `dev` consists mostly of the device drivers together with a few other things. The subdirectory `sys` is the rest of the system. There are files of the form `LIBx` in the directories `sys` and `dev`. These are archives (`ar(1)`) which contain the object versions of the routines in the directory.

Subdirectory `conf` contains the files which control device configuration of the system. `L.s` specifies the contents of the interrupt vectors; `c.c` contains the tables which relate device numbers to handler routines. A third file, `mch.s`, contains all the machine-language code in the system. A fourth file, `mch0.s`, is generated by `mkconf(1)` and contains flags indicating what sort of tape drive is available for taking crash dumps.

There are two ways to recreate the system. Use

```
cd /usr/sys/conf
make unix
```

if the libraries `/usr/sys/dev/LIB2` and `/usr/sys/sys/LIB1`, and also `c.o` and `l.o`, are correct. Use

```
cd /usr/sys/conf
make all
```

to recompile everything and recreate the libraries from scratch. This is needed, for example, when a header included in several source files is changed. See 'Setting Up UNIX' for other information about configuration and such.

When the make is done, the new system is present in the current directory as 'unix'. It should be tested before destroying the currently running '/unix', this is best done by doing something like

```
mv /unix /ounix
mv unix /unix
```

If the new system doesn't work, you can still boot 'ounix' and come up (see boot(8)). When you have satisfied yourself that the new system works, remove /ounix.

To install a new device driver, compile it and put it into its library. The best way to put it into the library is to use the command

```
ar uv LIB2 x.o
```

where x is the routine you just compiled. (All the device drivers distributed with the system are already in the library.)

Next, the device's interrupt vector must be entered in l.s. This is probably already done by the routine mkconf(1), but if the device is esoteric or nonstandard you will have to massage l.s by hand. This involves placing a pointer to a callout routine and the device's priority level in the vector. Use some other device (like the console) as a guide. Notice that the entries in l.s must be in order as the assembler does not permit moving the location counter '.' backwards. The assembler also does not permit assignation of an absolute number to '.', which is the reason for the '. = ZERO+100' subterfuge. If a constant smaller than 16(10) is added to the priority level, this number will be available as the first argument of the interrupt routine. This stratagem is used when several similar devices share the same interrupt routine (as dll's).

If you have to massage l.s, be sure to add the code to actually transfer to the interrupt routine. Again use the console as a guide. The apparent strangeness of this code is due to running the kernel in separate I&D space. The *call* routine saves registers as required and prepares a C-style call on the actual interrupt routine named after the 'jmp' instruction. When the routine returns, *call* restores the registers and performs an rti instruction. As an aside, note that external names in C programs have an underscore ('\_') prepended to them.

The second step which must be performed to add a device unknown to mkconf is to add it to the configuration table /usr/sys/conf/c.c. This file contains two subtables, one for block-type devices, and one for character-type devices. Block devices include disks, DECtape, and magtape. All other devices are character devices. A line in each of these tables gives all the information the system needs to know about the device handler; the ordinal position of the line in the table implies its major device number, starting at 0.

There are four subentries per line in the block device table, which give its open routine, close routine, strategy routine, and device table. The open and close routines may be nonexistent, in which case the name 'nulldev' is given; this routine merely returns. The strategy routine is called to do any I/O, and the device table contains status information for the device.

For character devices, each line in the table specifies a routine for open, close, read, and write, and one which sets and returns device-specific status (used, for example, for stty and gty on typewriters). If there is no open or close routine, 'nulldev' may be given; if there is no read, write, or status routine, 'nodev' may be given. Nodev sets an error flag and returns.

The final step which must be taken to install a device is to make a special file for it. This is done by mknod(1), to which you must specify the device class (block or character), major device number (relative line in the configuration table) and minor device number (which is made available to the driver at appropriate times).

The documents 'Setting up Unix' and 'The Unix IO system' may aid in comprehending these steps.

### The Library libc.a

The library /lib/libc.a is where most of the subroutines described in sections 2 and 3 of the manual are kept. This library can be remade using the following commands:

```
cd /usr/src/libc
sh compall
sh mklib
mv libc.a /lib/libc.a
```

If single routines need to be recompiled and replaced, use

```
cc -c -O x.c
ar vr /lib/libc.a x.o
rm x.o
```

The above can also be used to put new items into the library. See ar(1), lorder(1), and tsort(1).

The routines in /usr/src/cmd/libc/csu (C start up) are not in libc.a. These are separately assembled and put into /lib. The commands to do this are

```
cd /usr/src/libc/csu
as - x.s
mv a.out /lib/x
```

where x is the routine you want.

### Other Libraries

Likewise, the directories containing the source for the other libraries have files compall (that recompiles everything) and mklib (that recreates the library).

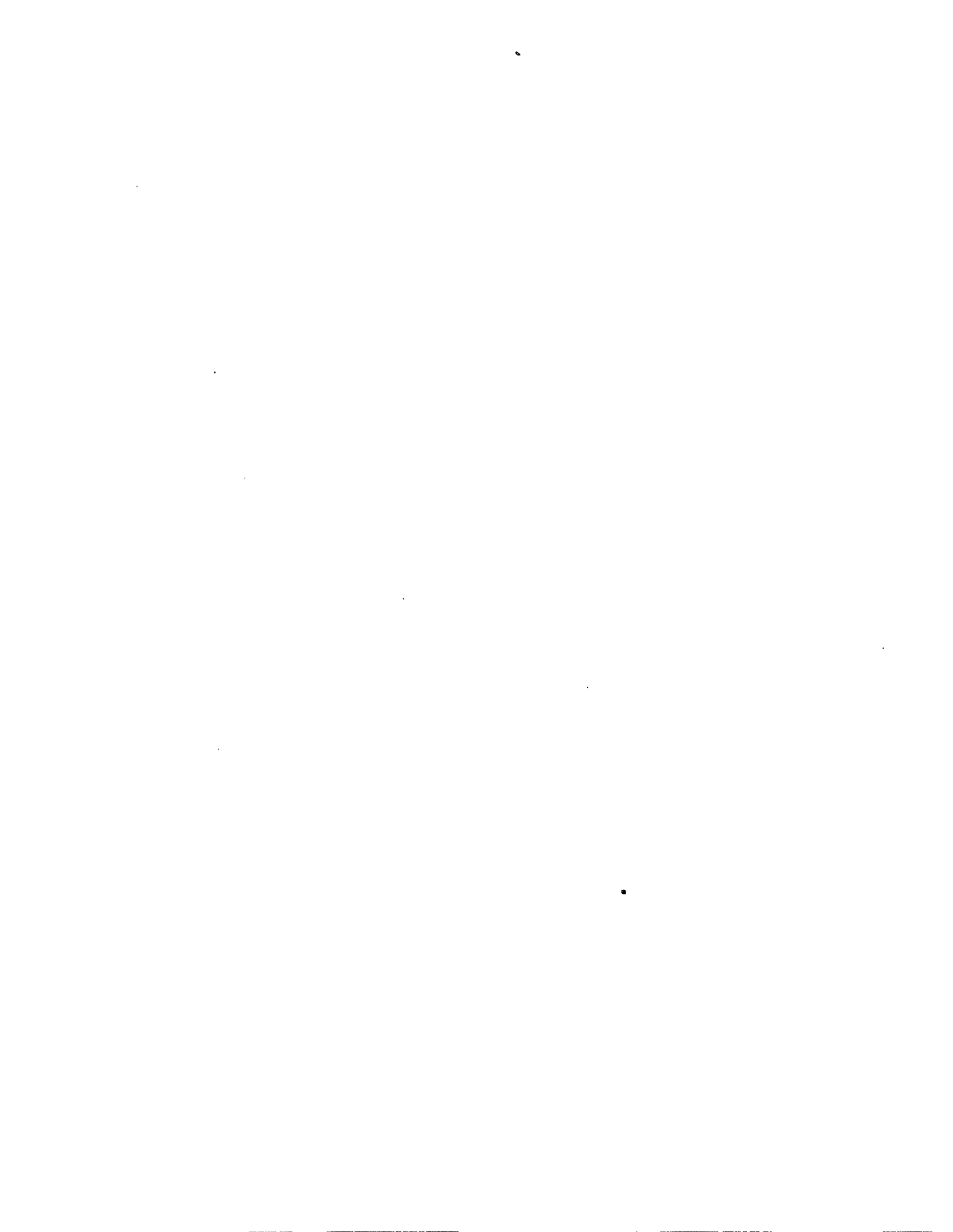
### System Tuning

There are several tunable parameters in the system. These set the size of various tables and limits. They are found in the file /usr/sys/h/param.h as manifests ('#define's). Their values are rather generous in the system as distributed. Our typical maximum number of users is about 20, but there are many daemon processes.

When any parameter is changed, it is prudent to recompile the entire system, as discussed above. A brief discussion of each follows:

- NBUF** This sets the size of the disk buffer cache. Each buffer is 512 bytes. This number should be around 25 plus N MOUNT, or as big as can be if the above number of buffers cause the system to not fit in memory.
- NFILE** This sets the maximum number of open files. An entry is made in this table every time a file is 'opened' (see open(2), creat(2)). Processes share these table entries across forks (fork(2)). This number should be about the same size as NINODE below. (It can be a bit smaller.)
- N MOUNT** This indicates the maximum number of mounted file systems. Make it big enough that you don't run out at inconvenient times.
- MAXMEM** This sets an administrative limit on the amount of memory a process may have. It is set automatically if the amount of physical memory is small, and thus should not need to be changed.
- MAXUPRC** This sets the maximum number of processes that any one user can be running at any one time. This should be set just large enough that people can get work done but not so large that a user can hog all the processes available (usually by accident!).

- NPROC** This sets the maximum number of processes that can be active. It depends on the demand pattern of the typical user; we seem to need about 8 times the number of terminals.
- NINODE** This sets the size of the inode table. There is one entry in the inode table for every open device, current working directory, sticky text segment, open file, and mounted device. Note that if two users have a file open there is still only one entry in the inode table. A reasonable rule of thumb for the size of this table is
- $NPROC + NMOUNT + (\text{number of terminals})$
- SSIZE** The initial size of a process stack. This may be made bigger if commonly run processes have large data areas on the stack.
- SINCR** The size of the stack growth increment.
- NOFILE** This sets the maximum number of files that any one process can have open. 20 is plenty.
- CANBSIZ** This is the size of the typewriter canonicalization buffer. It is in this buffer that erase and kill processing is done. Thus this is the maximum size of an input typewriter line. 256 is usually plenty.
- CMAPSIZ** The number of fragments that memory can be broken into. This should be big enough that it never runs out. The theoretical maximum is twice the number of processes, but this is a vast overestimate in practice. 50 seems enough.
- SMAPSIZ** Same as CMAPSIZ except for secondary (swap) memory.
- NCALL** This is the size of the callout table. Callouts are entered in this table when some sort of internal system timing must be done, as in carriage return delays for terminals. The number must be big enough to handle all such requests.
- NTEXT** The maximum number of simultaneously executing pure programs. This should be big enough so as to not run out of space under heavy load. A reasonable rule of thumb is about
- $(\text{number of terminals}) + (\text{number of sticky programs})$
- NCLIST** The number of clist segments. A clist segment is 6 characters. NCLIST should be big enough so that the list doesn't become exhausted when the machine is busy. The characters that have arrived from a terminal and are waiting to be given to a process live here. Thus enough space should be left so that every terminal can have at least one average line pending (about 30 or 40 characters).
- TIMEZONE** The number of minutes westward from Greenwich. See 'Setting Up UNIX'.
- DSTFLAG** See 'Setting Up UNIX' section on time conversion.
- MSGBUFS** The maximum number of characters of system error messages saved. This is used as a circular buffer.
- NCARGS** The maximum number of characters in an exec(2) arglist. This number controls how many arguments can be passed into a process. 5120 is practically infinite.
- HZ** Set to the frequency of the system clock (e.g., 50 for a 50 Hz. clock).





# A Tour through the UNIX† C Compiler

*D. M. Ritchie*

Bell Laboratories  
Murray Hill, New Jersey 07974

## The Intermediate Language

Communication between the two phases of the compiler proper is carried out by means of a pair of intermediate files. These files are treated as having identical structure, although the second file contains only the code generated for strings. It is convenient to write strings out separately to reduce the need for multiple location counters in a later assembly phase.

The intermediate language is not machine-independent; its structure in a number of ways reflects the fact that C was originally a one-pass compiler chopped in two to reduce the maximum memory requirement. In fact, only the latest version of the compiler has a complete intermediate language at all. Until recently, the first phase of the compiler generated assembly code for those constructions it could deal with, and passed expression parse trees, in absolute binary form, to the second phase for code generation. Now, at least, all inter-phase information is passed in a describable form, and there are no absolute pointers involved, so the coupling between the phases is not so strong.

The areas in which the machine (and system) dependencies are most noticeable are

1. Storage allocation for automatic variables and arguments has already been performed, and nodes for such variables refer to them by offset from a display pointer. Type conversion (for example, from integer to pointer) has already occurred using the assumption of byte addressing and 2-byte words.
2. Data representations suitable to the PDP-11 are assumed; in particular, floating point constants are passed as four words in the machine representation.

As it happens, each intermediate file is represented as a sequence of binary numbers without any explicit demarcations. It consists of a sequence of conceptual lines, each headed by an operator, and possibly containing various operands. The operators are small numbers; to assist in recognizing failure in synchronization, the high-order byte of each operator word is always the octal number 376. Operands are either 16-bit binary numbers or strings of characters representing names. Each name is terminated by a null character. There is no alignment requirement for numerical operands and so there is no padding after a name string.

The binary representation was chosen to avoid the necessity of converting to and from character form and to minimize the size of the files. It would be very easy to make each operator-operand 'line' in the file be a genuine, printable line, with the numbers in octal or decimal; this in fact was the representation originally used.

The operators fall naturally into two classes: those which represent part of an expression, and all others. Expressions are transmitted in a reverse-Polish notation; as they are being read, a tree is built which is isomorphic to the tree constructed in the first phase. Expressions are passed as a whole, with no non-expression operators intervening. The reader maintains a stack; each leaf of the expression tree (name, constant) is pushed on the stack; each unary operator replaces the top of the stack by a node whose operand is the old top-of-stack; each binary

---

†UNIX is a Trademark of Bell Laboratories.

operator replaces the top pair on the stack with a single entry. When the expression is complete there is exactly one item on the stack. Following each expression is a special operator which passes the unique previous expression to the 'optimizer' described below and then to the code generator.

Here is the list of operators not themselves part of expressions.

#### **EOF**

marks the end of an input file.

#### **BDATA** *flag data ...*

specifies a sequence of bytes to be assembled as static data. It is followed by pairs of words; the first member of the pair is non-zero to indicate that the data continue; a zero flag is not followed by data and terminates the operator. The data bytes occupy the low-order part of a word.

#### **WDATA** *flag data ...*

specifies a sequence of words to be assembled as static data; it is identical to the BDATA operator except that entire words, not just bytes, are passed.

#### **PROG**

means that subsequent information is to be compiled as program text.

#### **DATA**

means that subsequent information is to be compiled as static data.

#### **BSS**

means that subsequent information is to be compiled as uninitialized static data.

#### **SYMDEF** *name*

means that the symbol *name* is an external name defined in the current program. It is produced for each external data or function definition.

#### **CSPACE** *name size*

indicates that the name refers to a data area whose size is the specified number of bytes. It is produced for external data definitions without explicit initialization.

#### **SSPACE** *size*

indicates that *size* bytes should be set aside for data storage. It is used to pad out short initializations of external data and to reserve space for static (internal) data. It will be preceded by an appropriate label.

#### **EVEN**

is produced after each external data definition whose size is not an integral number of words. It is not produced after strings except when they initialize a character array.

#### **NLABEL** *name*

is produced just before a BDATA or WDATA initializing external data, and serves as a label for the data.



**RLABEL** *name*

is produced just before each function definition, and labels its entry point.

**SNAME** *name number*

is produced at the start of each function for each static variable or label declared therein. Subsequent uses of the variable will be in terms of the given number. The code generator uses this only to produce a debugging symbol table.

**ANAME** *name number*

Likewise, each automatic variable's name and stack offset is specified by this operator. Arguments count as automatics.

**RNAME** *name number*

Each register variable is similarly named, with its register number.

**SAVE** *number*

produces a register-save sequence at the start of each function, just after its label (RLABEL).

**SETREG** *number*

is used to indicate the number of registers used for register variables. It actually gives the register number of the lowest free register; it is redundant because the RNAME operators could be counted instead.

**PROFIL**

is produced before the save sequence for functions when the profile option is turned on. It produces code to count the number of times the function is called.

**SWIT** *deflab line label value ...*

is produced for switches. When control flows into it, the value being switched on is in the register forced by RFORCE (below). The switch statement occurred on the indicated line of the source, and the label number of the default location is *deflab*. Then the operator is followed by a sequence of label-number and value pairs; the list is terminated by a 0 label.

**LABEL** *number*

generates an internal label. It is referred to elsewhere using the given number.

**BRANCH** *number*

indicates an unconditional transfer to the internal label number given.

**RETRN**

produces the return sequence for a function. It occurs only once, at the end of each function.

**EXPR** *line*

causes the expression just preceding to be compiled. The argument is the line number in the source where the expression occurred.

**NAME** *class type name*

**NAME** *class type number*

indicates a name occurring in an expression. The first form is used when the name is external; the second when the name is automatic, static, or a register. Then the number indicates the stack offset, the label number, or the register number as appropriate. Class and type encoding is described elsewhere.

**CON** *type value*

transmits an integer constant. This and the next two operators occur as part of expressions.

**FCON** *type 4-word-value*

transmits a floating constant as four words in PDP-11 notation.

**SFCON** *type value*

transmits a floating-point constant whose value is correctly represented by its high-order word in PDP-11 notation.

**NULL**

indicates a null argument list of a function call in an expression; call is a binary operator whose second operand is the argument list.

**CBRANCH** *label cond*

produces a conditional branch. It is an expression operator, and will be followed by an EXPR. The branch to the label number takes place if the expression's truth value is the same as that of *cond*. That is, if *cond* = 1 and the expression evaluates to true, the branch is taken.

**binary-operator** *type*

There are binary operators corresponding to each such source-language operator; the type of the result of each is passed as well. Some perhaps-unexpected ones are: COMMA, which is a right-associative operator designed to simplify right-to-left evaluation of function arguments; prefix and postfix ++ and --, whose second operand is the increment amount, as a CON; QUEST and COLON, to express the conditional expression as 'a?(b:c)'; and a sequence of special operators for expressing relations between pointers, in case pointer comparison is different from integer comparison (e.g. unsigned).

**unary-operator** *type*

There are also numerous unary operators. These include ITOF, FTOI, FTOL, LTOF, ITOL, LTOI which convert among floating, long, and integer; JUMP which branches indirectly through a label expression; INIT, which compiles the value of a constant expression used as an initializer; RFORCE, which is used before a return sequence or a switch to place a value in an agreed-upon register.

### Expression Optimization

Each expression tree, as it is read in, is subjected to a fairly comprehensive analysis. This is performed by the *optim* routine and a number of subroutines; the major things done are

1. Modifications and simplifications of the tree so its value may be computed more efficiently and conveniently by the code generator.
2. Marking each interior node with an estimate of the number of registers required to evaluate it. This register count is needed to guide the code generation algorithm.

One thing that is definitely not done is discovery or exploitation of common subexpressions, nor is this done anywhere in the compiler.

The basic organization is simple: a depth-first scan of the tree. *Optim* does nothing for leaf nodes (except for automatics; see below), and calls *unoptim* to handle unary operators. For binary operators, it calls itself to process the operands, then treats each operator separately. One important case is commutative and associative operators, which are handled by *acommute*.

Here is a brief catalog of the transformations carried out by *optim* itself. It is not intended to be complete. Some of the transformations are machine-dependent, although they may well be useful on machines other than the PDP-11.

1. As indicated in the discussion of *unoptim* below, the optimizer can create a node type corresponding to the location addressed by a register plus a constant offset. Since this is precisely the implementation of automatic variables and arguments, where the register is fixed by convention, such variables are changed to the new form to simplify later processing.
2. Associative and commutative operators are processed by the special routine *acommute*.
3. After processing by *acommute*, the bitwise & operator is turned into a new *andn* operator; 'a & b' becomes 'a *andn* b'. This is done because the PDP-11 provides no *and* operator, but only *andn*. A similar transformation takes place for '=' & '.
4. Relationals are turned around so the more complicated expression is on the left. (So that '2 > f(x)' becomes 'f(x) < 2'). This improves code generation since the algorithm prefers to have the right operand require fewer registers than the left.
5. An expression minus a constant is turned into the expression plus the negative constant, and the *acommute* routine is called to take advantage of the properties of addition.
6. Operators with constant operands are evaluated.
7. Right shifts (unless by 1) are turned into left shifts with a negated right operand, since the PDP-11 lacks a general right-shift operator.
8. A number of special cases are simplified, such as division or multiplication by 1, and shifts by 0.

The *unoptim* routine performs the same sort of processing for unary operators.

1. '\*&x' and '&\*x' are simplified to 'x'.
2. If *r* is a register and *c* is a constant or the address of a static or external variable, the expressions '\* (r+c)' and '\*r' are turned into a special kind of name node which expresses the name itself and the offset. This simplifies subsequent processing because such constructions can appear as the the address of a PDP-11 instruction.
3. When the unary '&' operator is applied to a name node of the special kind just discussed, it is reworked to make the addition explicit again; this is done because the PDP-11 has no 'load address' instruction.
4. Constructions like '\*r++' and '\*--r' where *r* is a register are discovered and marked as being implementable using the PDP-11 auto-increment and -decrement modes.
5. If '!' is applied to a relational, the '!' is discarded and the sense of the relational is reversed.
6. Special cases involving reflexive use of negation and complementation are discovered.

7. Operations applying to constants are evaluated.

The *acommutate* routine, called for associative and commutative operators, discovers clusters of the same operator at the top levels of the current tree, and arranges them in a list: for 'a + ((b+c) + (d+f))' the list would be 'a,b,c,d,e,f'. After each subtree is optimized, the list is sorted in decreasing difficulty of computation; as mentioned above, the code generation algorithm works best when left operands are the difficult ones. The 'degree of difficulty' computed is actually finer than the mere number of registers required; a constant is considered simpler than the address of a static or external, which is simpler than reference to a variable. This makes it easy to fold all the constants together, and also to merge together the sum of a constant and the address of a static or external (since in such nodes there is space for an 'offset' value). There are also special cases, like multiplication by 1 and addition of 0.

A special routine is invoked to handle sums of products. *Distrib* is based on the fact that it is better to compute 'c1\*c2\*x + c1\*y' as 'c1\*(c2\*x + y)' and makes the divisibility tests required to assure the correctness of the transformation. This transformation is rarely possible with code directly written by the user, but it invariably occurs as a result of the implementation of multi-dimensional arrays.

Finally, *acommutate* reconstructs a tree from the list of expressions which result.

### Code Generation

The grand plan for code-generation is independent of any particular machine; it depends largely on a set of tables. But this fact does not necessarily make it very easy to modify the compiler to produce code for other machines, both because there is a good deal of machine-dependent structure in the tables, and because in any event such tables are non-trivial to prepare.

The arguments to the basic code generation routine *rcexpr* are a pointer to a tree representing an expression, the name of a code-generation table, and the number of a register in which the value of the expression should be placed. *Rcexpr* returns the number of the register in which the value actually ended up; its caller may need to produce a *mov* instruction if the value really needs to be in the given register. There are four code generation tables.

*Regtab* is the basic one, which actually does the job described above: namely, compile code which places the value represented by the expression tree in a register.

*Cctab* is used when the value of the expression is not actually needed, but instead the value of the condition codes resulting from evaluation of the expression. This table is used, for example, to evaluate the expression after *if*. It is clearly silly to calculate the value (0 or 1) of the expression 'a == b' in the context 'if (a == b) ...'

The *sprtab* table is used when the value of an expression is to be pushed on the stack, for example when it is an actual argument. For example in the function call 'f(a)' it is a bad idea to load *a* into a register which is then pushed on the stack, when there is a single instruction which does the job.

The *efftab* table is used when an expression is to be evaluated for its side effects, not its value. This occurs mostly for expressions which are statements, which have no value. Thus the code for the statement 'a = b' need produce only the appropriate *mov* instruction, and need not leave the value of *b* in a register, while in the expression 'a + (b = c)' the value of 'b = c' will appear in a register.

All of the tables besides *regtab* are rather small, and handle only a relatively few special cases. If one of these subsidiary tables does not contain an entry applicable to the given expression tree, *rcexpr* uses *regtab* to put the value of the expression into a register and then fixes things up; nothing need be done when the table was *efftab*, but a *lsl* instruction is produced when the table called for was *cctab*, and a *mov* instruction, pushing the register on the stack, when the table was *sprtab*.

The *rcexpr* routine itself picks off some special cases, then calls *cexpr* to do the real work. *Cexpr* tries to find an entry applicable to the given tree in the given table, and returns -1 if no such entry is found, letting *rcexpr* try again with a different table. A successful match yields a string containing both literal characters which are written out and pseudo-operations, or macros, which are expanded. Before studying the contents of these strings we will consider how table entries are matched against trees.

Recall that most non-leaf nodes in an expression tree contain the name of the operator, the type of the value represented, and pointers to the subtrees (operands). They also contain an estimate of the number of registers required to evaluate the expression, placed there by the expression-optimizer routines. The register counts are used to guide the code generation process, which is based on the Sethi-Ullman algorithm.

The main code generation tables consist of entries each containing an operator number and a pointer to a subtable for the corresponding operator. A subtable consists of a sequence of entries, each with a key describing certain properties of the operands of the operator involved; associated with the key is a code string. Once the subtable corresponding to the operator is found, the subtable is searched linearly until a key is found such that the properties demanded by the key are compatible with the operands of the tree node. A successful match returns the code string; an unsuccessful search, either for the operator in the main table or a compatible key in the subtable, returns a failure indication.

The tables are all contained in a file which must be processed to obtain an assembly language program. Thus they are written in a special-purpose language. To provide definiteness to the following discussion, here is an example of a subtable entry.

```
%n.aw
 F
 add A2,R
```

The '%' indicates the key; the information following (up to a blank line) specifies the code string. Very briefly, this entry is in the subtable for '+' of *regtab*; the key specifies that the left operand is any integer, character, or pointer expression, and the right operand is any word quantity which is directly addressible (e.g. a variable or constant). The code string calls for the generation of the code to compile the left (first) operand into the current register ('F') and then to produce an 'add' instruction which adds the second operand ('A2') to the register ('R'). All of the notation will be explained below.

Only three features of the operands are used in deciding whether a match has occurred. They are:

1. Is the type of the operand compatible with that demanded?
2. Is the 'degree of difficulty' (in a sense described below) compatible?
3. The table may demand that the operand have a '\*' (indirection operator) as its highest operator.

As suggested above, the key for a subtable entry is indicated by a '%,' and a comma-separated pair of specifications for the operands. (The second specification is ignored for unary operators). A specification indicates a type requirement by including one of the following letters. If no type letter is present, any integer, character, or pointer operand will satisfy the requirement (not float, double, or long).

- b A byte (character) operand is required.
- w A word (integer or pointer) operand is required.
- f A float or double operand is required.
- d A double operand is required.

l A long (32-bit integer) operand is required.

Before discussing the 'degree of difficulty' specification, the algorithm has to be explained more completely. *Rcexpr* (and *cexpr*) are called with a register number in which to place their result. Registers 0, 1, ... are used during evaluation of expressions; the maximum register which can be used in this way depends on the number of register variables, but in any event only registers 0 through 4 are available since r5 is used as a stack frame header and r6 (sp) and r7 (pc) have special hardware properties. The code generation routines assume that when called with register *n* as argument, they may use *n+1*, ... (up to the first register variable) as temporaries. Consider the expression 'X+Y', where both X and Y are expressions. As a first approximation, there are three ways of compiling code to put this expression in register *n*.

1. If Y is an addressible cell, (recursively) put X into register *n* and add Y to it.
2. If Y is an expression that can be calculated in *k* registers, where *k* smaller than the number of registers available, compile X into register *n*, Y into register *n+1*, and add register *n+1* to *n*.
3. Otherwise, compile Y into register *n*, save the result in a temporary (actually, on the stack) compile X into register *n*, then add in the temporary.

The distinction between cases 2 and 3 therefore depends on whether the right operand can be compiled in fewer than *k* registers, where *k* is the number of free registers left after registers 0 through *n* are taken: 0 through *n-1* are presumed to contain already computed temporary results; *n* will, in case 2, contain the value of the left operand while the right is being evaluated.

These considerations should make clear the specification codes for the degree of difficulty, bearing in mind that a number of special cases are also present:

- z is satisfied when the operand is zero, so that special code can be produced for expressions like 'x = 0'.
- l is satisfied when the operand is the constant 1, to optimize cases like left and right shift by 1, which can be done efficiently on the PDP-11.
- c is satisfied when the operand is a positive (16-bit) constant; this takes care of some special cases in long arithmetic.
- a is satisfied when the operand is addressible; this occurs not only for variables and constants, but also for some more complicated constructions, such as indirection through a simple variable, '\*p++' where *p* is a register variable (because of the PDP-11's auto-increment address mode), and \*(p+c) where *p* is a register and *c* is a constant. Precisely, the requirement is that the operand refers to a cell whose address can be written as a source or destination of a PDP-11 instruction.
- e is satisfied by an operand whose value can be generated in a register using no more than *k* registers, where *k* is the number of registers left (not counting the current register). The 'e' stands for 'easy.'
- n is satisfied by any operand. The 'n' stands for 'anything.'

These degrees of difficulty are considered to lie in a linear ordering and any operand which satisfies an earlier-mentioned requirement will satisfy a later one. Since the subtables are searched linearly, if a 'l' specification is included, almost certainly a 'z' must be written first to prevent expressions containing the constant 0 to be compiled as if the 0 were 1.

Finally, a key specification may contain a '\*' which requires the operand to have an indirection as its leading operator. Examples below should clarify the utility of this specification.

Now let us consider the contents of the code string associated with each subtable entry. Conventionally, lower-case letters in this string represent literal information which is copied directly to the output. Upper-case letters generally introduce specific macro-operations, some of which may be followed by modifying information. The code strings in the tables are written with tabs and new-lines used freely to suggest instructions which will be generated; the table-

compiling program compresses tabs (using the 0200 bit of the next character) and throws away some of the new-lines. For example the macro 'F' is ordinarily written on a line by itself; but since its expansion will end with a new-line, the new-line after 'F' itself is dispensable. This is all to reduce the size of the stored tables.

The first set of macro-operations is concerned with compiling subtrees. Recall that this is done by the *cexpr* routine. In the following discussion the 'current register' is generally the argument register to *cexpr*; that is, the place where the result is desired. The 'next register' is numbered one higher than the current register. (This explanation isn't fully true because of complications, described below, involving operations which require even-odd register pairs.)

- F causes a recursive call to the *rcexpr* routine to compile code which places the value of the first (left) operand of the operator in the current register.
- F1 generates code which places the value of the first operand in the next register. It is incorrectly used if there might be no next register; that is, if the degree of difficulty of the first operand is not 'easy,' if not, another register might not be available.
- FS generates code which pushes the value of the first operand on the stack, by calling *rcexpr* specifying *sprab* as the table.

Analogously,

S, S1, SS compile the second (right) operand into the current register, the next register, or onto the stack.

To deal with registers, there are

- R which expands into the name of the current register.
- R1 which expands into the name of the next register.
- R+ which expands into the the name of the current register plus 1. It was suggested above that this is the same as the next register, except for complications; here is one of them. Long integer variables have 32 bits and require 2 registers; in such cases the next register is the current register plus 2. The code would like to talk about both halves of the long quantity, so R refers to the register with the high-order part and R+ to the low-order part.
- R- This is another complication, involving division and mod. These operations involve a pair of registers of which the odd-numbered contains the left operand. *Cexpr* arranges that the current register is odd; the R- notation allows the code to refer to the next lower, even-numbered register.

To refer to addressible quantities, there are the notations:

- A1 causes generation of the address specified by the first operand. For this to be legal, the operand must be addressible; its key must contain an 'a' or a more restrictive specification.
- A2 correspondingly generates the address of the second operand providing it has one.

We now have enough mechanism to show a complete, if suboptimal, table for the + operator on word or byte operands.

```
%n,z
 F

%n,l
 F
 inc R

%n,aw
 F
 add A2,R

%n,e
 F
 S1
 add R1,R

%n,n
 SS
 F
 add (sp)+,R
```

The first two sequences handle some special cases. Actually it turns out that handling a right operand of 0 is unnecessary since the expression-optimizer throws out adds of 0. Adding 1 by using the 'increment' instruction is done next, and then the case where the right operand is addressible. It must be a word quantity, since the PDP-11 lacks an 'add byte' instruction. Finally the cases where the right operand either can, or cannot, be done in the available registers are treated.

The next macro-instructions are conveniently introduced by noticing that the above table is suitable for subtraction as well as addition, since no use is made of the commutativity of addition. All that is needed is substitution of 'sub' for 'add' and 'dec' for 'inc.' Considerable saving of space is achieved by factoring out several similar operations.

I is replaced by a string from another table indexed by the operator in the node being expanded. This secondary table actually contains two strings per operator.

I' is replaced by the second string in the side table entry for the current operator.

Thus, given that the entries for '+' and '-' in the side table (which is called *instab*) are 'add' and 'inc,' 'sub' and 'dec' respectively, the middle of of the above addition table can be written

```
%n,l
 F
 I' R

%n,aw
 F
 I A2,R
```

and it will be suitable for subtraction, and several other operators, as well.

Next, there is the question of character and floating-point operations.

B1 generates the letter 'b' if the first operand is a character, 'f' if it is float or double, and nothing otherwise. It is used in a context like 'movB1' which generates a 'mov', 'movb', or 'movf' instruction according to the type of the operand.



- B2 is just like B1 but applies to the second operand.
  - BE generates 'b' if either operand is a character and null otherwise.
  - BF generates 'f' if the type of the operator node itself is float or double, otherwise null.
- For example, there is an entry in *efftab* for the '=' operator

```
%a,aw
%ab,a
 IBE A2,A1
```

Note first that two key specifications can be applied to the same code string. Next, observe that when a word is assigned to a byte or to a word, or a word is assigned to a byte, a single instruction, a *mov* or *movb* as appropriate, does the job. However, when a byte is assigned to a word, it must pass through a register to implement the sign-extension rules:

```
%a,n
 S
 IB1 R,A1
```

Next, there is the question of handling indirection properly. Consider the expression 'X + \*Y', where X and Y are expressions, Assuming that Y is more complicated than just a variable, but on the other hand qualifies as 'easy' in the context, the expression would be compiled by placing the value of X in a register, that of \*Y in the next register, and adding the registers. It is easy to see that a better job can be done by compiling X, then Y (into the next register), and producing the instruction symbolized by 'add (R1),R'. This scheme avoids generating the instruction 'mov (R1),R1' required actually to place the value of \*Y in a register. A related situation occurs with the expression 'X + \*(p+6)', which exemplifies a construction frequent in structure and array references. The addition table shown above would produce

```
[put X in register R]
mov p,R1
add $6,R1
mov (R1),R1
add R1,R
```

when the best code is

```
[put X in R]
mov p,R1
add 6(R1),R
```

As we said above, a key specification for a code table entry may require an operand to have an indirection as its highest operator. To make use of the requirement, the following macros are provided.

- F\* the first operand must have the form \*X. If in particular it has the form \*(Y + c), for some constant c, then code is produced which places the value of Y in the current register. Otherwise, code is produced which loads X into the current register.
- F1\* resembles F\* except that the next register is loaded.
- S\* resembles F\* except that the second operand is loaded.
- S1\* resembles S\* except that the next register is loaded.
- FS\* The first operand must have the form \*\*X'. Push the value of X on the stack.
- SS\* resembles FS\* except that it applies to the second operand.

To capture the constant that may have been skipped over in the above macros, there are

#1 The first operand must have the form \*X; if in particular it has the form \*(Y + c) for c a constant, then the constant is written out, otherwise a null string.

#2 is the same as #1 except that the second operand is used.

Now we can improve the addition table above. Just before the '%n,e' entry, put

```
%n,ew*
 F
 S1*
 add #2(R1),R
```

and just before the '%n,n' put

```
%n,nw*
 SS*
 F
 add *(sp)+,R
```

When using the stacking macros there is no place to use the constant as an index word, so that particular special case doesn't occur.

The constant mentioned above can actually be more general than a number. Any quantity acceptable to the assembler as an expression will do, in particular the address of a static cell, perhaps with a numeric offset. If *x* is an external character array, the expression '*x*[*i*+5] = 0' will generate the code

```
mov i,r0
clrb x+5(r0)
```

via the table entry (in the '=' part of *efftab*)

```
%e*,z
 F
 l'B1 #1(R)
```

Some machine operations place restrictions on the registers used. The divide instruction, used to implement the divide and mod operations, requires the dividend to be placed in the odd member of an even-odd pair; other peculiarities of multiplication make it simplest to put the multiplicand in an odd-numbered register. There is no theory which optimally accounts for this kind of requirement. *Cexpr* handles it by checking for a multiply, divide, or mod operation; in these cases, its argument register number is incremented by one or two so that it is odd, and if the operation was divide or mod, so that it is a member of a free even-odd pair. The routine which determines the number of registers required estimates, conservatively, that at least two registers are required for a multiplication and three for the other peculiar operators. After the expression is compiled, the register where the result actually ended up is returned. (Divide and mod are actually the same operation except for the location of the result).

These operations are the ones which cause results to end up in unexpected places, and this possibility adds a further level of complexity. The simplest way of handling the problem is always to move the result to the place where the caller expected it, but this will produce unnecessary register moves in many simple cases; '*a* = *b*\**c*' would generate

```
mov b,r1
mul c,r1
mov r1,r0
mov r0,a
```

The next thought is used the passed-back information as to where the result landed to change the notion of the current register. While compiling the '=' operation above, which comes from a table entry like

```

%a,e
S
mov R,A1

```

it is sufficient to redefine the meaning of 'R' after processing the 'S' which does the multiply. This technique is in fact used; the tables are written in such a way that correct code is produced. The trouble is that the technique cannot be used in general, because it invalidates the count of the number of registers required for an expression. Consider just 'a\*b + X' where X is some expression. The algorithm assumes that the value of a\*b, once computed, requires just one register. If there are three registers available, and X requires two registers to compute, then this expression will match a key specifying '%n,e'. If a\*b is computed and left in register 1, then there are, contrary to expectations, no longer two registers available to compute X, but only one, and bad code will be produced. To guard against this possibility, *cexpr* checks the result returned by recursive calls which implement F, S and their relatives. If the result is not in the expected register, then the number of registers required by the other operand is checked; if it can be done using those registers which remain even after making unavailable the unexpectedly-occupied register, then the notions of the 'next register' and possibly the 'current register' are redefined. Otherwise a register-copy instruction is produced. A register-copy is also always produced when the current operator is one of those which have odd-even requirements.

Finally, there are a few loose-end macro operations and facts about the tables. The operators:

- V is used for long operations. It is written with an address like a machine instruction; it expands into 'adc' (add carry) if the operation is an additive operator, 'sbc' (subtract carry) if the operation is a subtractive operator, and disappears, along with the rest of the line, otherwise. Its purpose is to allow common treatment of logical operations, which have no carries, and additive and subtractive operations, which generate carries.
- T generates a 'tst' instruction if the first operand of the tree does not set the condition codes correctly. It is used with divide and mod operations, which require a sign-extended 32-bit operand. The code table for the operations contains an 'sxt' (sign-extend) instruction to generate the high-order part of the dividend.
- H is analogous to the 'F' and 'S' macros, except that it calls for the generation of code for the current tree (not one of its operands) using *regtab*. It is used in *cciab* for all the operators which, when executed normally, set the condition codes properly according to the result. It prevents a 'tst' instruction from being generated for constructions like 'if (a+b) ...' since after calculation of the value of 'a+b' a conditional branch can be written immediately.

All of the discussion above is in terms of operators with operands. Leaves of the expression tree (variables and constants), however, are peculiar in that they have no operands. In order to regularize the matching process, *cexpr* examines its operand to determine if it is a leaf; if so, it creates a special 'load' operator whose operand is the leaf, and substitutes it for the argument tree; this allows the table entry for the created operator to use the 'A1' notation to load the leaf into a register.

Purely to save space in the tables, pieces of subtables can be labelled and referred to later. It turns out, for example, that rather large portions of the the *effiab* table for the '=' and '=+' operators are identical. Thus '=' has an entry

```

%[move3:]
%a,aw
%ab,a
IBE A2,A1

```

while part of the '=+' table is

```
%aw,aw
% [move3]
```

Labels are written as '%[ ... : ]', before the key specifications; references are written with '% [ ... ]' after the key. Peculiarities in the implementation make it necessary that labels appear before references to them.

The example illustrates the utility of allowing separate keys to point to the same code string. The assignment code works properly if either the right operand is a word, or the left operand is a byte; but since there is no 'add byte' instruction the addition code has to be restricted to word operands.

### Delaying and reordering

Intertwined with the code generation routines are two other, interrelated processes. The first, implemented by a routine called *delay*, is based on the observation that naive code generation for the expression 'a = b++' would produce

```
mov b,r0
inc b
mov r0,a
```

The point is that the table for postfix ++ has to preserve the value of *b* before incrementing it; the general way to do this is to preserve its value in a register. A cleverer scheme would generate

```
mov b,a
inc b
```

*Delay* is called for each expression input to *rcexpr*, and it searches for postfix ++ and -- operators. If one is found applied to a variable, the tree is patched to bypass the operator and compiled as it stands; then the increment or decrement itself is done. The effect is as if 'a = b; b++' had been written. In this example, of course, the user himself could have done the same job, but more complicated examples are easily constructed, for example 'switch (x++)'. An essential restriction is that the condition codes not be required. It would be incorrect to compile 'if (a++) ...' as

```
tst a
inc a
beq ...
```

because the 'inc' destroys the required setting of the condition codes.

Reordering is a similar sort of optimization. Many cases which it detects are useful mainly with register variables. If *r* is a register variable, the expression 'r = x + y' is best compiled as

```
mov x,r
add y,r
```

but the codes tables would produce

```
mov x,r0
add y,r0
mov r0,r
```

which is in fact preferred if *r* is not a register. (If *r* is not a register, the two sequences are the same size, but the second is slightly faster.) The scheme is to compile the expression as if it had been written 'r = x; r = + y'. The *reorder* routine is called with a pointer to each tree that *rcexpr* is about to compile; if it has the right characteristics, the 'r = x' tree is constructed and passed recursively to *rcexpr*; then the original tree is modified to read 'r = + y' and the calling instance of *rcexpr* compiles that instead. Of course the whole business is itself recursive so that

more extended forms of the same phenomenon are handled, like 'r = x + y | z'.

Care does have to be taken to avoid 'optimizing' an expression like 'r = x + r' into 'r = x; r = + r'. It is required that the right operand of the expression on the right of the '=' be a ', distinct from the register variable.

The second case that *reorder* handles is expressions of the form 'r = X' used as a subexpression. Again, the code out of the tables for 'x = r = y' would be

```
mov y,r0
mov r0,r
mov r0,x
```

whereas if *r* were a register it would be better to produce

```
mov y,r
mov r,x
```

When *reorder* discovers that a register variable is being assigned to in a subexpression, it calls *rcexpr* recursively to compile the subexpression, then fiddles the tree passed to it so that the register variable itself appears as the operand instead of the whole subexpression. Here care has to be taken to avoid an infinite regress, with *rcexpr* and *reorder* calling each other forever to handle assignments to registers.

A third set of cases treated by *reorder* comes up when any name, not necessarily a register, occurs as a left operand of an assignment operator other than '=' or as an operand of prefix '++' or '--'. Unless condition-code tests are involved, when a subexpression like '(a = + b)' is seen, the assignment is performed and the argument tree modified so that *a* is its operand; effectively 'x + (y = + z)' is compiled as 'y = + z; x + y'. Similarly, prefix increment and decrement are pulled out and performed first, then the remainder of the expression.

Throughout code generation, the expression optimizer is called whenever *delay* or *reorder* change the expression tree. This allows some special cases to be found that otherwise would not be seen.









VOLUME II

Program Development Tools

Table of Contents

Part 1: System Programming Tools

1. Shell Introduction (Bourne shell)
2. C Shell
3. UNIX Programming
4. YACC
5. SED
6. AWK
7. LEX
8. DC - a desk calculator

Part 2: C Language and C Program Maintenance

1. C Reference Manual
2. ADB
3. LINT
4. MAKE

Part 3: UNIX Maintenance

1. UNIX Implementation
2. I/O
3. Regenerating System Software
4. UNIX C compiler



# A Tour Through the Portable C Compiler

S. C. Johnson

Bell Laboratories  
Murray Hill, New Jersey 07974

## Introduction

A C compiler has been implemented that has proved to be quite portable, serving as the basis for C compilers on roughly a dozen machines, including the Honeywell 6000, IBM 370, and Interdata 8/32. The compiler is highly compatible with the C language standard.<sup>1</sup>

Among the goals of this compiler are portability, high reliability, and the use of state-of-the-art techniques and tools wherever practical. Although the efficiency of the compiling process is not a primary goal, the compiler is efficient enough, and produces good enough code, to serve as a production compiler.

The language implemented is highly compatible with the current PDP-11 version of C. Moreover, roughly 75% of the compiler, including nearly all the syntactic and semantic routines, is machine independent. The compiler also serves as the major portion of the program *lint*, described elsewhere.<sup>2</sup>

A number of earlier attempts to make portable compilers are worth noting. While on CO-OP assignment to Bell Labs in 1973, Alan Snyder wrote a portable C compiler which was the basis of his Master's Thesis at M.I.T.<sup>3</sup> This compiler was very slow and complicated, and contained a number of rather serious implementation difficulties; nevertheless, a number of Snyder's ideas appear in this work.

Most earlier portable compilers, including Snyder's, have proceeded by defining an intermediate language, perhaps based on three-address code or code for a stack machine, and writing a machine independent program to translate from the source code to this intermediate code. The intermediate code is then read by a second pass, and interpreted or compiled. This approach is elegant, and has a number of advantages, especially if the target machine is far removed from the host. It suffers from some disadvantages as well. Some constructions, like initialization and subroutine prologs, are difficult or expensive to express in a machine independent way that still allows them to be easily adapted to the target assemblers. Most of these approaches require a symbol table to be constructed in the second (machine dependent) pass, and/or require powerful target assemblers. Also, many conversion operators may be generated that have no effect on a given machine, but may be needed on others (for example, pointer to pointer conversions usually do nothing in C, but must be generated because there are some machines where they are significant).

For these reasons, the first pass of the portable compiler is not entirely machine independent. It contains some machine dependent features, such as initialization, subroutine prolog and epilog, certain storage allocation functions, code for the *switch* statement, and code to throw out unneeded conversion operators.

As a crude measure of the degree of portability actually achieved, the Interdata 8/32 C compiler has roughly 600 machine dependent lines of source out of 4600 in Pass 1, and 1000 out of 3400 in Pass 2. In total, 1600 out of 8000, or 20%, of the total source is machine dependent (12% in Pass 1, 30% in Pass 2). These percentages can be expected to rise slightly as the compiler is tuned. The percentage of machine-dependent code for the IBM is 22%, for the Honeywell 25%. If the assembler format and structure were the same for all these machines,

perhaps another 5-10% of the code would become machine independent.

These figures are sufficiently misleading as to be almost meaningless. A large fraction of the machine dependent code can be converted in a straightforward, almost mechanical way. On the other hand, a certain amount of the code requires hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

## Overview

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.<sup>4</sup> One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.<sup>5</sup>

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the `#define` and `#include` statements, and related features (e.g., `#ifdef`, etc.). It is a nearly machine independent program, and will not be further discussed here.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program. This "one pass" operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined by the node

number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees that are built by the first pass have been declared to have room for the second pass information as well. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

### The Source Files

The compiler source consists of 22 source files. Two files, *manifest* and *macdefs*, are header files included with all other files. *Manifest* has declarations for the node numbers, types, storage classes, and other global data definitions. *Macdefs* has machine-dependent definitions, such as the size and alignment of the various data representations. Two machine independent header files, *mfile1* and *mfile2*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs*, contains declarations of register names, etc.

There is a file, *common*, containing (machine independent) routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. There are two dummy files, *comm1.c* and *comm2.c*, that simply include *common* within the scope of the appropriate pass1 or pass2 header files. When the compiler is loaded as a single pass, *common* only needs to be included once: *comm2.c* is not needed.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *scan.c*, *cgram.c*, *xdefs.c*, *pftn.c*, *trees.c*, *optim.c*, *local.c*, *code.c*, and *comm1.c*. *Scan.c* is the lexical analyzer, which is used by *cgram.c*, the result of applying *Yacc*<sup>6</sup> to the input grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Comm1.c* includes *common*, that contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like.

The second pass is produced by compiling and loading *reader.c*, *allo.c*, *match.c*, *comm1.c*, *order.c*, *local.c*, and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Comm2.c* includes the file *common*, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

### Data Structure Considerations.

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases; this requires some synchronization with the *Yacc* input file, *cgram.y*, as well.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, -=), and the operator ASG may be applied to such node names to generate new ones. e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *opype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number (=, +=, etc.), and *logop(o)* returns true if *o* is a relational or logical (&&, ||, or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN, and ARY represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus PTR+INT represents the type for an integer pointer, and

ARY + (PTR << 2) + (FTN << 4) + DOUBLE

represents the type of an array of pointers to functions returning doubles.

The type words are ordinarily manipulated by macros. If *t* is a type word, *BTYPE(t)* gives the basic type. *ISPTR(t)*, *ISARY(t)*, and *ISFTN(t)* ask if an object of this type is a pointer, array, or a function, respectively. *MODTYPE(t,b)* sets the basic type of *t* to *b*. *DECREF(t)* gives the type resulting from removing the first operator from *t*. Thus, if *t* is a pointer to *t'*, a function returning *t'*, or an array of *t'*, then *DECREF(t)* would equal *t'*. *INCREF(t)* gives the type representing a pointer to *t*. Finally, there are operators for dealing with the unsigned types. *ISUNSIGNED(t)* returns true if *t* is one of the four basic unsigned types; in this case, *DEUSIGN(t)* gives the associated 'signed' type. Similarly, *UNSIGNABLE(t)* returns true if *t* is one of the four basic types that could become unsigned, and *EVUNSIGN(t)* returns the unsigned analogue of *t* in this case.

The other important global data structure is that of expression trees. The actual shapes of the nodes are given in *mfile1* and *mfile2*. They are not the same in the two passes; the first pass nodes contain dimension and size information, while the second pass nodes contain register allocation information. Nevertheless, all nodes contain fields called *op*, containing the node number, and *type*, containing the type word. A function called *alloc()* returns a pointer to a new tree node. To free a node, its *op* field need merely be set to FREE. The other fields in the node will remain intact at least until the next allocation.

Nodes representing binary operators contain fields, *left* and *right*, that contain pointers to the left and right descendants. Unary operator nodes have the *left* field, and a value field called *rval*. Leaf nodes, with no descendants, have two value fields: *lval* and *rval*.

At appropriate times, the function *tcheck()* can be called, to check that there are no busy nodes remaining. This is used as a compiler consistency check. The function *icopy(p)* takes a pointer *p* that points to an expression tree, and returns a pointer to a disjoint copy of the tree. The function *walkf(p,f)* performs a postorder walk of the tree pointed to by *p*, and applies the function *f* to each node. The function *fwalk(p,f,d)* does a preorder walk of the tree pointed to by *p*. At each node, it calls a function *f*, passing to it the node pointer, a value passed down from its ancestor, and two pointers to values to be passed down to the left and right descendants (if any). The value *d* is the value passed down to the root. *Fwalk* is used for a number of tree labeling and debugging activities.

The other major data structure, the symbol table, exists only in pass one, and will be discussed later.

### Pass One

The first pass does lexical analysis, parsing, symbol table maintenance, tree building, optimization, and a number of machine dependent things. This pass is largely machine independent, and the machine independent sections can be pretty successfully ignored. Thus, they will be only sketched here.

### Lexical Analysis

The lexical analyzer is a conceptually simple routine that reads the input and returns the tokens of the C language as it encounters them: names, constants, operators, and keywords. The conceptual simplicity of this job is confounded a bit by several other simple jobs that unfortunately must go on simultaneously. These include

- Keeping track of the current filename and line number, and occasionally setting this information as the result of preprocessor control lines.
- Skipping comments.
- Properly dealing with octal, decimal, hex, floating point, and character constants, as well as character strings.

To achieve speed, the program maintains several tables that are indexed into by character value, to tell the lexical analyzer what to do next. To achieve portability, these tables must be initialized each time the compiler is run, in order that the table entries reflect the local character set values.

### Parsing

As mentioned above, the parser is generated by Yacc from the grammar on file *cgram.y*. The grammar is relatively readable, but contains some unusual features that are worth comment.

Perhaps the strangest feature of the grammar is the treatment of declarations. The problem is to keep track of the basic type and the storage class while interpreting the various stars, brackets, and parentheses that may surround a given name. The entire declaration mechanism must be recursive, since declarations may appear within declarations of structures and unions, or even within a *sizeof* construction inside a dimension in another declaration!

There are some difficulties in using a bottom-up parser, such as produced by Yacc, to handle constructions where a lot of left context information must be kept around. The problem is that the original PDP-11 compiler is top-down in implementation, and some of the semantics of C reflect this. In a top-down parser, the input rules are restricted somewhat, but one can naturally associate temporary storage with a rule at a very early stage in the recognition of that rule. In a bottom-up parser, there is more freedom in the specification of rules, but it is more

difficult to know what rule is being matched until the entire rule is seen. The parser described by *cgram.c* makes effective use of the bottom-up parsing mechanism in some places (notably the treatment of expressions), but struggles against the restrictions in others. The usual result is that it is necessary to run a stack of values "on the side", independent of the Yacc value stack, in order to be able to store and access information deep within inner constructions, where the relationship of the rules being recognized to the total picture is not yet clear.

In the case of declarations, the attribute information (type, etc.) for a declaration is carefully kept immediately to the left of the declarator (that part of the declaration involving the name). In this way, when it is time to declare the name, the name and the type information can be quickly brought together. The "\$0" mechanism of Yacc is used to accomplish this. The result is not pretty, but it works. The storage class information changes more slowly, so it is kept in an external variable, and stacked if necessary. Some of the grammar could be considerably cleaned up by using some more recent features of Yacc, notably actions within rules and the ability to return multiple values for actions.

A stack is also used to keep track of the current location to be branched to when a **break** or **continue** statement is processed.

This use of external stacks dates from the time when Yacc did not permit values to be structures. Some, or most, of this use of external stacks could be eliminated by redoing the grammar to use the mechanisms now provided. There are some areas, however, particularly the processing of structure, union, and enum declarations, function prologs, and switch statement processing, when having all the affected data together in an array speeds later processing; in this case, use of external storage seems essential.

The *cgram.y* file also contains some small functions used as utility functions in the parser. These include routines for saving case values and labels in processing switches, and stacking and popping values on the external stack described above.

## Storage Classes

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass; by the second pass, this information must have been totally dealt with. This means that all of the storage allocation must take place in the first pass, so that references to automatics and parameters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. Much of this transformation is machine dependent, and strongly depends on the storage class.

The classes include **EXTERN** (for externally declared, but not defined variables), **EXTDEF** (for external definitions), and similar distinctions for **USTATIC** and **STATIC**, **UFORTRAN** and **FORTRAN** (for fortran functions) and **ULABEL** and **LABEL**. The storage classes **REGISTER** and **AUTO** are obvious, as are **STNAME**, **UNAME**, and **ENAME** (for structure, union, and enumeration tags), and the associated **MOS**, **MOU**, and **MOE** (for the members). **TYPEDEF** is treated as a storage class as well. There are two special storage classes: **PARAM** and **NULL**. **NULL** is used to distinguish the case where no explicit storage class has been given; before an entry is made in the symbol table the true storage class is discovered. Similarly, **PARAM** is used for the temporary entry in the symbol table made before the declaration of function parameters is completed.

The most complexity in the storage class process comes from bit fields. A separate storage class is kept for each width bit field: a  $k$  bit bit field has storage class  $k$  plus **FIELD**. This enables the size to be quickly recovered from the storage class.



### Symbol Table Maintenance.

The symbol table routines do far more than simply enter names into the symbol table; considerable semantic processing and checking is done as well. For example, if a new declaration comes in, it must be checked to see if there is a previous declaration of the same symbol. If there is, there are many cases. The declarations may agree and be compatible (for example, an extern declaration can appear twice) in which case the new declaration is ignored. The new declaration may add information (such as an explicit array dimension) to an already present declaration. The new declaration may be different, but still correct (for example, an extern declaration of something may be entered, and then later the definition may be seen). The new declaration may be incompatible, but appear in an inner block: in this case, the old declaration is carefully hidden away, and the new one comes into force until the block is left. Finally, the declarations may be incompatible, and an error message must be produced.

A number of other factors make for additional complexity. The type declared by the user is not always the type entered into the symbol table (for example, if a formal parameter to a function is declared to be an array, C requires that this be changed into a pointer before entry in the symbol table). Moreover, there are various kinds of illegal types that may be declared which are difficult to check for syntactically (for example, a function returning an array). Finally, there is a strange feature in C that requires structure tag names and member names for structures and unions to be taken from a different logical symbol table than ordinary identifiers. Keeping track of which kind of name is involved is a bit of struggle (consider typedef names used within structure declarations, for example).

The symbol table handling routines have been rewritten a number of times to extend features, improve performance, and fix bugs. They address the above problems with reasonable effectiveness but a singular lack of grace.

When a name is read in the input, it is hashed, and the routine *lookup* is called, together with a flag which tells which symbol table should be searched (actually, both symbol tables are stored in one, and a flag is used to distinguish individual entries). If the name is found, *lookup* returns the index to the entry found; otherwise, it makes a new entry, marks it UNDEF (undefined), and returns the index of the new entry. This index is stored in the *rval* field of a NAME node.

When a declaration is being parsed, this NAME node is made part of a tree with UNARY MUL nodes for each \*, LB nodes for each array descriptor (the right descendant has the dimension), and UNARY CALL nodes for each function descriptor. This tree is passed to the routine *tymerge*, along with the attribute type of the whole declaration; this routine collapses the tree to a single node, by calling *tyreduce*, and then modifies the type to reflect the overall type of the declaration.

Dimension and size information is stored in a table called *dimtab*. To properly describe a type in C, one needs not just the type information but also size information (for structures and enums) and dimension information (for arrays). Sizes and offsets are dealt with in the compiler by giving the associated indices into *dimtab*. *Tymerge* and *tyreduce* call *dstash* to put the discovered dimensions away into the *dimtab* array. *Tymerge* returns a pointer to a single node that contains the symbol table index in its *rval* field, and the size and dimension indices in fields *csiz* and *cdim*, respectively. This information is properly considered part of the type in the first pass, and is carried around at all times.

To enter an element into the symbol table, the routine *defid* is called; it is handed a storage class, and a pointer to the node produced by *tymerge*. *Defid* calls *fixtype*, which adjusts and checks the given type depending on the storage class, and converts null types appropriately. It then calls *fixclass*, which does a similar job for the storage class; it is here, for example, that register declarations are either allowed or changed to auto.

The new declaration is now compared against an older one, if present, and several pages of validity checks performed. If the definitions are compatible, with possibly some added information, the processing is straightforward. If the definitions differ, the block levels of the

current and the old declaration are compared. The current block level is kept in *blevel*, an external variable; the old declaration level is kept in the symbol table. Block level 0 is for external declarations, 1 is for arguments to functions, and 2 and above are blocks within a function. If the current block level is the same as the old declaration, an error results. If the current block level is higher, the new declaration overrides the old. This is done by marking the old symbol table entry "hidden", and making a new entry, marked "hiding". *Lookup* will skip over hidden entries. When a block is left, the symbol table is searched, and any entries defined in that block are destroyed; if they hid other entries, the old entries are "unhidden".

This nice block structure is warped a bit because labels do not follow the block structure rules (one can do a *goto* into a block, for example); default definitions of functions in inner blocks also persist clear out to the outermost scope. This implies that cleaning up the symbol table after block exit is more subtle than it might first seem.

For successful new definitions, *defid* also initializes a "general purpose" field, *offset*, in the symbol table. It contains the stack offset for automatics and parameters, the register number for register variables, the bit offset into the structure for structure members, and the internal label number for static variables and labels. The offset field is set by *falloc* for bit fields, and *dclstruct* for structures and unions.

The symbol table entry itself thus contains the name, type word, size and dimension offsets, offset value, and declaration block level. It also has a field of flags, describing what symbol table the name is in, and whether the entry is hidden, or hides another. Finally, a field gives the line number of the last use, or of the definition, of the name. This is used mainly for diagnostics, but is useful to *lint* as well.

In some special cases, there is more than the above amount of information kept for the use of the compiler. This is especially true with structures; for use in initialization, structure declarations must have access to a list of the members of the structure. This list is also kept in *dimtab*. Because a structure can be mentioned long before the members are known, it is necessary to have another level of indirection in the table. The two words following the *csiz* entry in *dimtab* are used to hold the alignment of the structure, and the index in *dimtab* of the list of members. This list contains the symbol table indices for the structure members, terminated by a -1.

## Tree Building

The portable compiler transforms expressions into expression trees. As the parser recognizes each rule making up an expression, it calls *buildtree* which is given an operator number, and pointers to the left and right descendants. *Buildtree* first examines the left and right descendants, and, if they are both constants, and the operator is appropriate, simply does the constant computation at compile time, and returns the result as a constant. Otherwise, *buildtree* allocates a node for the head of the tree, attaches the descendants to it, and ensures that conversion operators are generated if needed, and that the type of the new node is consistent with the types of the operands. There is also a considerable amount of semantic complexity here: many combinations of types are illegal, and the portable compiler makes a strong effort to check the legality of expression types completely. This is done both for *lint* purposes, and to prevent such semantic errors from being passed through to the code generator.

The heart of *buildtree* is a large table, accessed by the routine *opact*. This routine maps the types of the left and right operands into a rather smaller set of descriptors, and then accesses a table (actually encoded in a switch statement) which for each operator and pair of types causes an action to be returned. The actions are logical or's of a number of separate actions, which may be carried out by *buildtree*. These component actions may include checking the left side to ensure that it is an lvalue (can be stored into), applying a type conversion to the left or right operand, setting the type of the new node to the type of the left or right operand, calling various routines to balance the types of the left and right operands, and suppressing the ordinary conversion of arrays and function operands to pointers. An important operation is OTHER, which causes some special code to be invoked in *buildtree*, to handle issues which are

unique to a particular operator. Examples of this are structure and union reference (actually handled by the routine *sref*), the building of NAME, ICON, STRING and FCON (floating point constant) nodes, unary \* and &, structure assignment, and calls. In the case of unary \* and &, *buildtree* will cancel a \* applied to a tree, the top node of which is &, and conversely.

Another special operation is PUN; this causes the compiler to check for type mismatches, such as intermixing pointers and integers.

The treatment of conversion operators is still a rather strange area of the compiler (and of C!). The recent introduction of type casts has only confounded this situation. Most of the conversion operators are generated by calls to *ymatch* and *pmatch*, both of which are given a tree, and asked to make the operands agree in type. *Pmatch* treats the case where one of the operands is a pointer; *ymatch* treats all other cases. Where these routines have decided on the proper type for an operand, they call *makeop*, which is handed a tree, and a type word, dimension offset, and size offset. If necessary, it inserts a conversion operation to make the types correct. Conversion operations are never inserted on the left side of assignment operators, however. There are two conversion operators used; PCONV, if the conversion is to a non-basic type (usually a pointer), and SCONV, if the conversion is to a basic type (scalar).

To allow for maximum flexibility, every node produced by *buildtree* is given to a machine dependent routine, *clocal*, immediately after it is produced. This is to allow more or less immediate rewriting of those nodes which must be adapted for the local machine. The conversion operations are given to *clocal* as well; on most machines, many of these conversions do nothing, and should be thrown away (being careful to retain the type). If this operation is done too early, however, later calls to *buildtree* may get confused about correct type of the subtrees; thus *clocal* is given the conversion ops only after the entire tree is built. This topic will be dealt with in more detail later.

## Initialization

Initialization is one of the messier areas in the portable compiler. The only consolation is that most of the mess takes place in the machine independent part, where it may be safely ignored by the implementor of the compiler for a particular machine.

The basic problem is that the semantics of initialization really calls for a co-routine structure: one collection of programs reading constants from the input stream, while another, independent set of programs places these constants into the appropriate spots in memory. The dramatic differences in the local assemblers also come to the fore here. The parsing problems are dealt with by keeping a rather extensive stack containing the current state of the initialization; the assembler problems are dealt with by having a fair number of machine dependent routines.

The stack contains the symbol table number, type, dimension index, and size index for the current identifier being initialized. Another entry has the offset, in bits, of the beginning of the current identifier. Another entry keeps track of how many elements have been seen, if the current identifier is an array. Still another entry keeps track of the current member of a structure being initialized. Finally, there is an entry containing flags which keep track of the current state of the initialization process (e.g., tell if a } has been seen for the current identifier.)

When an initialization begins, the routine *beginit* is called; it handles the alignment restrictions, if any, and calls *instk* to create the stack entry. This is done by first making an entry on the top of the stack for the item being initialized. If the top entry is an array, another entry is made on the stack for the first element. If the top entry is a structure, another entry is made on the stack for the first member of the structure. This continues until the top element of the stack is a scalar. *Instk* then returns, and the parser begins collecting initializers.

When a constant is obtained, the routine *doinit* is called; it examines the stack, and does whatever is necessary to assign the current constant to the scalar on the top of the stack. *gorscal* is then called, which rearranges the stack so that the next scalar to be initialized gets placed on top of the stack. This process continues until the end of the initializers; *endinit* cleans up. If

a { or } is encountered in the string of initializers, it is handled by calling *ibrace* or *irbrace*, respectively.

A central issue is the treatment of the "holes" that arise as a result of alignment restrictions or explicit requests for holes in bit fields. There is a global variable, *inoff*, which contains the current offset in the initialization (all offsets in the first pass of the compiler are in bits). *Doinit* figures out from the top entry on the stack the expected bit offset of the next identifier; it calls the machine dependent routine *inforce* which, in a machine dependent way, forces the assembler to set aside space if need be so that the next scalar seen will go into the appropriate bit offset position. The scalar itself is passed to one of the machine dependent routines *fincode* (for floating point initialization), *incode* (for fields, and other initializations less than an int in size), and *cinic* (for all other initializations). The size is passed to all these routines, and it is up to the machine dependent routines to ensure that the initializer occupies exactly the right size.

Character strings represent a bit of an exception. If a character string is seen as the initializer for a pointer, the characters making up the string must be put out under a different location counter. When the lexical analyzer sees the quote at the head of a character string, it returns the token *STRING*, but does not do anything with the contents. The parser calls *getstr*, which sets up the appropriate location counters and flags, and calls *lxstr* to read and process the contents of the string.

If the string is being used to initialize a character array, *lxstr* calls *putbyte*, which in effect simulates *doinit* for each character read. If the string is used to initialize a character pointer, *lxstr* calls a machine dependent routine, *bycode*, which stashes away each character. The pointer to this string is then returned, and processed normally by *doinit*.

The null at the end of the string is treated as if it were read explicitly by *lxstr*.

## Statements

The first pass addresses four main areas; declarations, expressions, initialization, and statements. The statement processing is relatively simple; most of it is carried out in the parser directly. Most of the logic is concerned with allocating label numbers, defining the labels, and branching appropriately. An external symbol, *reached*, is 1 if a statement can be reached, 0 otherwise; this is used to do a bit of simple flow analysis as the program is being parsed, and also to avoid generating the subroutine return sequence if the subroutine cannot "fall through" the last statement.

Conditional branches are handled by generating an expression node, *CBRANCH*, whose left descendant is the conditional expression and the right descendant is an *ICON* node containing the internal label number to be branched to. For efficiency, the semantics are that the label is gone to if the condition is *false*.

The switch statement is compiled by collecting the case entries, and an indication as to whether there is a default case; an internal label number is generated for each of these, and remembered in a big array. The expression comprising the value to be switched on is compiled when the switch keyword is encountered, but the expression tree is headed by a special node, *FORCE*, which tells the code generator to put the expression value into a special distinguished register (this same mechanism is used for processing the return statement). When the end of the switch block is reached, the array containing the case values is sorted, and checked for duplicate entries (an error); if all is correct, the machine dependent routine *genswitch* is called, with this array of labels and values in increasing order. *Genswitch* can assume that the value to be tested is already in the register which is the usual integer return value register.

## Optimization

There is a machine independent file, *optim.c*, which contains a relatively short optimization routine, *optim*. Actually the word optimization is something of a misnomer; the results are not optimum, only improved, and the routine is in fact not optional: it must be called for proper operation of the compiler.

*Optim* is called after an expression tree is built, but before the code generator is called. The essential part of its job is to call *clocal* on the conversion operators. On most machines, the treatment of & is also essential: by this time in the processing, the only node which is a legal descendant of & is NAME. (Possible descendants of \* have been eliminated by *buildtree*.) The address of a static name is, almost by definition, a constant, and can be represented by an ICON node on most machines (provided that the loader has enough power). Unfortunately, this is not universally true: on some machine, such as the IBM 370, the issue of addressability rears its ugly head; thus, before turning a NAME node into an ICON node, the machine dependent function *andable* is called.

The optimization attempts of *optim* are currently quite limited. It is primarily concerned with improving the behavior of the compiler with operations one of whose arguments is a constant. In the simplest case, the constant is placed on the right if the operation is commutative. The compiler also makes a limited search for expressions such as

$$(x + a) + b$$

where *a* and *b* are constants, and attempts to combine *a* and *b* at compile time. A number of special cases are also examined: additions of 0 and multiplications by 1 are removed, although the correct processing of these cases to get the type of the resulting tree correct is decidedly nontrivial. In some cases, the addition or multiplication must be replaced by a conversion op to keep the types from becoming fouled up. Finally, in cases where a relational operation is being done, and one operand is a constant, the operands are permuted, and the operator altered, if necessary, to put the constant on the right. Finally, multiplications by a power of 2 are changed to shifts.

There are dozens of similar optimizations that can be, and should be, done. It seems likely that this routine will be expanded in the relatively near future.

### Machine Dependent Stuff

A number of the first pass machine dependent routines have been discussed above. In general, the routines are short, and easy to adapt from machine to machine. The two exceptions to this general rule are *clocal* and the function prolog and epilog generation routines, *bfcodes* and *efcodes*.

*Clocal* has the job of rewriting, if appropriate and desirable, the nodes constructed by *buildtree*. There are two major areas where this is important: NAME nodes and conversion operations. In the case of NAME nodes, *clocal* must rewrite the NAME node to reflect the actual physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through the *rval* field of the node), and, based on the storage class of the node, the tree must be rewritten. Automatic variables and parameters are typically rewritten by treating the reference to the variable as a structure reference, off the register which holds the stack or argument pointer: the *stref* routine is set up to be called in this way, and to build the appropriate tree. In the most general case, the tree consists of a unary \* node, whose descendant is a + node, with the stack or argument register as left operand, and a constant offset as right operand. In the case of LABEL and internal static nodes, the *rval* field is rewritten to be the negative of the internal label number: a negative *rval* field is taken to be an internal label number. Finally, a name of class REGISTER must be converted into a REG node, and the *rval* field replaced by the register number. In fact, this part of the *clocal* routine is nearly machine independent; only for machines with addressability problems (IBM 370 again!) does it have to be noticeably different.

The conversion operator treatment is rather tricky. It is necessary to handle the application of conversion operators to constants in *clocal*, in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversion from byte pointer to short or long pointer might require a shift or rotate

operation, which would have to be generated here.

The extension of the portable compiler to machines where the size of a pointer depends on its type would be straightforward, but has not yet been done.

The other major machine dependent issue involves the subroutine prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence; this design issue is discussed elsewhere.<sup>5</sup> The routine *bfc* is called with the number of arguments the function is defined with, and an array containing the symbol table indices of the declared parameters. *Bfc* must generate the code to establish the new stack frame, save the return address and previous stack pointer value on the stack, and save whatever registers are to be used for register variables. The stack size and the number of register variables is not known when *bfc* is called, so these numbers must be referred to by assembler constants, which are defined when they are known (usually in the second pass, after all register variables, automatics, and temporaries have been seen). The final job is to find those parameters which may have been declared register, and generate the code to initialize the register with the value passed on the stack. Once again, for most machines, the general logic of *bfc* remains the same, but the contents of the *printf* calls in it will change from machine to machine. *efc* is rather simpler, having just to generate the default return at the end of a function. This may be nontrivial in the case of a function returning a structure or union, however.

There seems to be no really good place to discuss structures and unions, but this is as good a place as any. The C language now supports structure assignment, and the passing of structures as arguments to functions, and the receiving of structures back from functions. This was added rather late to C, and thus to the portable compiler. Consequently, it fits in less well than the older features. Moreover, most of the burden of making these features work is placed on the machine dependent code.

There are both conceptual and practical problems. Conceptually, the compiler is structured around the idea that to compute something, you put it into a register and work on it. This notion causes a bit of trouble on some machines (e.g., machines with 3-address opcodes), but matches many machines quite well. Unfortunately, this notion breaks down with structures. The closest that one can come is to keep the addresses of the structures in registers. The actual code sequences used to move structures vary from the trivial (a multiple byte move) to the horrible (a function call), and are very machine dependent.

The practical problem is more painful. When a function returning a structure is called, this function has to have some place to put the structure value. If it places it on the stack, it has difficulty popping its stack frame. If it places the value in a static temporary, the routine fails to be reentrant. The most logically consistent way of implementing this is for the caller to pass in a pointer to a spot where the called function should put the value before returning. This is relatively straightforward, although a bit tedious, to implement, but means that the caller must have properly declared the function type, even if the value is never used. On some machines, such as the Interdata 8/32, the return value simply overlays the argument region (which on the 8/32 is part of the caller's stack frame). The caller takes care of leaving enough room if the returned value is larger than the arguments. This also assumes that the caller know and declares the function properly.

The PDP-11 and the VAX have stack hardware which is used in function calls and returns; this makes it very inconvenient to use either of the above mechanisms. In these machines, a static area within the called function is allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is non-reentrant. However, the function can now be called as a subroutine without being properly declared, without the disaster which would otherwise ensue. No matter what choice is taken, the convention is that the function actually returns the address of the return structure value.

In building expression trees, the portable compiler takes a bit for granted about structures. It assumes that functions returning structures actually return a pointer to the structure, and it

assumes that a reference to a structure is actually a reference to its address. The structure assignment operator is rebuilt so that the left operand is the structure being assigned to, but the right operand is the address of the structure being assigned; this makes it easier to deal with

$$a = b = c$$

and similar constructions.

There are four special tree nodes associated with these operations: STASG (structure assignment), STARG (structure argument to a function call), and STCALL and UNARY STCALL (calls of a function with nonzero and zero arguments, respectively). These four nodes are unique in that the size and alignment information, which can be determined by the type for all other objects in C, must be known to carry out these operations; special fields are set aside in these nodes to contain this information, and special intermediate code is used to transmit this information.

### First Pass Summary

There are many other issues which have been ignored here, partly to justify the title "tour", and partially because they have seemed to cause little trouble. There are some debugging flags which may be turned on, by giving the compiler's first pass the argument

`-X[flags]`

Some of the more interesting flags are `-Xd` for the defining and freeing of symbols, `-Xi` for initialization comments, and `-Xb` for various comments about the building of trees. In many cases, repeating the flag more than once gives more information; thus, `-Xddd` gives more information than `-Xd`. In the two pass version of the compiler, the flags should not be set when the output is sent to the second pass, since the debugging output and the intermediate code both go onto the standard output.

We turn now to consideration of the second pass.

### Pass Two

Code generation is far less well understood than parsing or lexical analysis, and for this reason the second pass is far harder to discuss in a file by file manner. A great deal of the difficulty is in understanding the issues and the strategies employed to meet them. Any particular function is likely to be reasonably straightforward.

Thus, this part of the paper will concentrate a good deal on the broader aspects of strategy in the code generator, and will not get too intimate with the details.

### Overview.

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines, and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.

One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code, rather than to produce incorrect code. This goal is achieved by having a simple model of the job to be done (e.g., an expression tree) and a simple model of the machine state (e.g., which registers are free). The act of generating an instruction performs a transformation on the tree and the machine state; hopefully, the tree eventually gets reduced to a single node. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all the C operators. Thus the selection of which instruction/transformations to generate, and in what order, will have a heuristic flavor. If, for some expression tree, no transformation applies, or, more seriously, if the heuristics select a sequence of instruction/transformations that do not in fact reduce the tree, the compiler will report its inability to generate code, and abort.

A major part of the code generator is concerned with the model and the transformations, — most of this is machine independent, or depends only on simple tables. The flexibility comes from the heuristics that guide the transformations of the trees, the selection of subgoals, and the ordering of the computation.

### The Machine Model

The machine is assumed to have a number of registers, of at most two different types: *A* and *B*. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g., register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number in the *mac2defs* file; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table is the *rstatus* table on file *local2.c*. This table is indexed by register number, and contains expressions made up from manifest constants describing the register types: SAREG for dedicated AREG's, SAREGISTAREG for scratch AREGS's, and SBREG and SBREGISTBREG similarly for BREG's. There are macros that access this information: *isbreg(r)* returns true if register number *r* is a BREG, and *istreg(r)* returns true if register number *r* is a temporary AREG or BREG. Another table, *rnames*, contains the register names; this is used when putting out assembler code and diagnostics.

The usage of registers is kept track of by an array called *busy*. *Busy[r]* is the number of uses of register *r* in the current tree being processed. The allocation and freeing of registers will be discussed later as part of the code generation algorithm.

### General Organization

As mentioned above, the second pass reads lines from the intermediate file, copying through to the output unchanged any lines that begin with a ')', and making note of the information about stack usage and register allocation contained on lines beginning with '[' and '{'. The expression trees, whose beginning is indicated by a line beginning with '.', are read and rebuilt into trees. If the compiler is loaded as one pass, the expression trees are immediately available to the code generator.

The actual code generation is done by a hierarchy of routines. The routine *delay* is first given the tree: it attempts to delay some postfix ++ and -- computations that might reasonably be done after the smoke clears. It also attempts to handle comma (,) operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. *Delay* calls *codgen* to control the actual code generation process. *Codgen* takes as arguments a pointer to the expression tree, and a second argument that, for socio-historical reasons, is called a *cookie*. The cookie describes a set of goals that would be acceptable for the code generation: these are assigned to individual bits, so they may be logically or'ed together to form a large number of possible goals. Among the possible goals are FOREFF (compute for side effects only: don't worry about the value), INTEMP (compute and store value into a temporary location in memory), INAREG (compute into an A register), INTAREG (compute into a scratch A register), INBREG and INTBREG similarly, FORCC (compute for condition codes), and FORARG (compute it as a function argument; e.g., stack it if appropriate).

*Codgen* first canonicalizes the tree by calling *canon*. This routine looks for certain transformations that might now be applicable to the tree. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its



behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register. *Canon* also looks for such cases, calling the machine dependent routine *noroff* to decide if the offset is acceptable (for example, in the IBM 370 the offset must be between 0 and 4095 bytes). Another optimization is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine, *sucomp*, is called that computes the Sethi-Ullman numbers for the tree (see below).

After the tree is canonicalized, *codgen* calls the routine *store* whose job is to select a subtree of the tree to be computed and (usually) stored before beginning the computation of the full tree. *Store* must return a tree that can be computed without need for any temporary storage locations. In effect, the only store operations generated while processing the subtree must be as a response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator can operate under the assumption that there are enough registers to do its job, without worrying about temporary storage. If a store into a temporary appears in the output, it is always as a direct result of logic in the *store* routine; this makes debugging easier.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision.<sup>7</sup> It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

The *store* routine decides what subtrees, if any, should be stored by making use of numbers, called *Sethi-Ullman numbers*, that give, for each subtree of an expression tree, the minimum number of scratch registers required to compile the subtree, without any stores into temporaries.<sup>8</sup> These numbers are computed by the machine-dependent routine *sucomp*, called by *canon*. The basic notion is that, knowing the Sethi-Ullman numbers for the descendants of a node, and knowing the operator of the node and some information about the machine, the Sethi-Ullman number of the node itself can be computed. If the Sethi-Ullman number for a tree exceeds the number of scratch registers available, some subtree must be stored. Unfortunately, the theory behind the Sethi-Ullman numbers applies only to uselessly simple machines and operators. For the rich set of C operators, and for machines with asymmetric registers, register pairs, different kinds of registers, and exceptional forms of addressing, the theory cannot be applied directly. The basic idea of estimation is a good one, however, and well worth applying; the application, especially when the compiler comes to be tuned for high code quality, goes beyond the park of theory into the swamp of heuristics. This topic will be taken up again later, when more of the compiler structure has been described.

After examining the Sethi-Ullman numbers, *store* selects a subtree, if any, to be stored, and returns the subtree and the associated cookie in the external variables *stotree* and *stocook*. If a subtree has been selected, or if the whole tree is ready to be processed, the routine *order* is called, with a tree and cookie. *Order* generates code for trees that do not require temporary locations. *Order* may make recursive calls on itself, and, in some cases, on *codgen*, for example, when processing the operators *&&*, *||*, and comma (*','*), that have a left to right evaluation, it is incorrect for *store* examine the right operand for subtrees to be stored. In these cases, *order* will call *codgen* recursively when it is permissible to work on the right operand. A similar issue arises with the *? :* operator.

The *order* routine works by matching the current tree with a set of code templates. If a template is discovered that will match the current tree and cookie, the associated assembly language statement or statements are generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is made to find a match with a different cookie; for example, in order to compute an expression with cookie *INTMP* (store into a temporary storage location), it is usually necessary to compute the expression into a scratch register first. If all attempts to match the

tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call *order* to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate. For example, if the left operand of a  $+$  is a scratch register, the  $+$  can be replaced by a  $+=$  operator; the tree may then match a template.

To close this introduction, we will discuss the steps in compiling code for the expression

$$a + = b$$

where  $a$  and  $b$  are static variables.

To begin with, the whole expression tree is examined with cookie FOREFF, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the Interdata 8/32 for the moment. It is recognized that the left hand and right hand sides of the  $+=$  operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite this as

$$a = a + b$$

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it is recognized that the left hand side of the assignment is addressable, but the right hand side is not in a register, so *order* is called recursively, being asked to put the right hand side of the assignment into a register. This invocation of *order* searches the tree for a match, and fails. The machine dependent rule for  $+$  notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call to *order* is made, with the tree consisting solely of the leaf  $a$ , and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted. The node consisting of  $a$  is rewritten in place to represent the register into which  $a$  is loaded, and this third call to *order* returns. The second call to *order* now finds that it has the tree

$$\text{reg} + b$$

to consider. Once again, there is no match, but the default rewriting rule rewrites the  $+$  as a  $+=$  operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

$$\text{reg} += b$$

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

$$a = \text{reg}$$

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed: we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different;  $b$  is loaded in to a register, and then an add to storage instruction generated to add this register in to  $a$ . The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the

heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

### The Templates

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

If we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), then we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```

ASG PLUS, INAREG,
 SAREG, TINT,
 SNAME, TINT,
 0, RLEFT,
 " add AL,AR\n",

```

The top line specifies the operator (+ =) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule (replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim*. to the output; upper case letters trigger various macro-like expansions. Thus, AL would expand into the Address form of the Left operand — presumably the register number. Similarly, AR would expand into the name of the right operand. The *add* instruction of the last section might well be emitted by this template.

In principle, it would be possible to make separate templates for all legal combinations of operators, cookies, types, and shapes. In practice, the number of combinations is very large. Thus, a considerable amount of mechanism is present to permit a large number of subtrees to be matched by a single template. Most of the shape and type specifiers are individual bits, and can be logically or'ed together. There are a number of special descriptors for matching classes of operators. The cookies can also be combined. As an example of the kind of template that really arises in practice, the actual template for the Interdata 8/32 that subsumes the above example is:

```

ASG OPSIMP, INAREG|FORCC,
 SAREG, TINT|TUNSIGNED|TPOINT,
 SAREG|SNAME|SOREG|SCON, TINT|TUNSIGNED|TPOINT,
 0, RLEFT|RESCC,
 " OI AL,AR\n",

```

Here, OPSIMP represents the operators +, -, !, &, and ^ . The OI macro in the output string expands into the appropriate Integer Opcode for the operator. The left and right sides can be integers, unsigned, or pointer types. The right side can be, in addition to a name, a register, a memory location whose address is given by a register and displacement (OREG), or a constant. Finally, these instructions set the condition codes, and so can be used in condition contexts: the cookie and rewriting rules reflect this.

### The Template Matching Algorithm.

The heart of the second pass is the template matching algorithm, in the routine *match*. *Match* is called with a tree and a cookie; it attempts to match the given tree against some template that will transform it according to one of the goals given in the cookie. If a match is successful, the transformation is applied; *expand* is called to generate the assembly code, and then *reclaim* rewrites the tree, and reclaims the resources, such as registers, that might have become free as a result of the generated code.

This part of the compiler is among the most time critical. There is a spectrum of implementation techniques available for doing this matching. The most naive algorithm simply looks at the templates one by one. This can be considerably improved upon by restricting the search for an acceptable template. It would be possible to do better than this if the templates were given to a separate program that ate them and generated a template matching subroutine. This would make maintenance of the compiler much more complicated, however, so this has not been done.

The matching algorithm is actually carried out by restricting the range in the table that must be searched for each opcode. This introduces a number of complications, however, and needs a bit of sympathetic help by the person constructing the compiler in order to obtain best results. The exact tuning of this algorithm continues; it is best to consult the code and comments in *match* for the latest version.

In order to match a template to a tree, it is necessary to match not only the cookie and the op of the root, but also the types and shapes of the left and right descendants (if any) of the tree. A convention is established here that is carried out throughout the second pass of the compiler. If a node represents a unary operator, the single descendant is always the "left" descendant. If a node represents a unary operator or a leaf node (no descendants) the "right" descendant is taken by convention to be the node itself. This enables templates to easily match leaves and conversion operators, for example, without any additional mechanism in the matching program.

The type matching is straightforward; it is possible to specify any combination of basic types, general pointers, and pointers to one or more of the basic types. The shape matching is somewhat more complicated, but still pretty simple. Templates have a collection of possible operand shapes on which the opcode might match. In the simplest case, an *add* operation might be able to add to either a register variable or a scratch register, and might be able (with appropriate help from the assembler) to add an integer constant (ICON), a static memory cell (NAME), or a stack location (OREG).

It is usually attractive to specify a number of such shapes, and distinguish between them when the assembler output is produced. It is possible to describe the union of many elementary shapes such as ICON, NAME, OREG, AREG or BREG (both scratch and register forms), etc. To handle at least the simple forms of indirection, one can also match some more complicated forms of trees: STARNM and STARREG can match more complicated trees headed by an indirection operator, and SFLD can match certain trees headed by a FLD operator: these patterns call machine dependent routines that match the patterns of interest on a given machine. The shape SWADD may be used to recognize NAME or OREG nodes that lie on word boundaries: this may be of some importance on word-addressed machines. Finally, there are some special shapes: these may not be used in conjunction with the other shapes, but may be defined and extended in machine dependent ways. The special shapes SZERO, SONE, and SMONE are predefined and match constants 0, 1, and -1, respectively; others are easy to add and match by using the machine dependent routine *special*.

When a template has been found that matches the root of the tree, the cookie, and the shapes and types of the descendants, there is still one bar to a total match: the template may call for some resources (for example, a scratch register). The routine *allo* is called, and it attempts to allocate the resources. If it cannot, the match fails; no resources are allocated. If successful, the allocated resources are given numbers 1, 2, etc. for later reference when the

assembly code is generated. The routines *expand* and *reclaim* are then called. The *match* routine then returns a special value, MDONE. If no match was found, the value MNOPE is returned; this is a signal to the caller to try more cookie values, or attempt a rewriting rule. *Match* is also used to select rewriting rules, although the way of doing this is pretty straightforward. A special cookie, FORREW, is used to ask *match* to search for a rewriting rule. The rewriting rules are keyed to various opcodes; most are carried out in *order*. Since the question of when to rewrite is one of the key issues in code generation, it will be taken up again later.

### Register Allocation.

The register allocation routines, and the allocation strategy, play a central role in the correctness of the code generation algorithm. If there are bugs in the Sethi-Ullman computation that cause the number of needed registers to be underestimated, the compiler may run out of scratch registers; it is essential that the allocator keep track of those registers that are free and busy, in order to detect such conditions.

Allocation of registers takes place as the result of a template match; the routine *allo* is called with a word describing the number of A registers, B registers, and temporary locations needed. The allocation of temporary locations on the stack is relatively straightforward, and will not be further covered: the bookkeeping is a bit tricky, but conceptually trivial, and requests for temporary space on the stack will never fail.

Register allocation is less straightforward. The two major complications are *pairing* and *sharing*. In many machines, some operations (such as multiplication and division), and/or some types (such as longs or double precision) require even/odd pairs of registers. Operations of the first type are exceptionally difficult to deal with in the compiler; in fact, their theoretical properties are rather bad as well.<sup>9</sup> The second issue is dealt with rather more successfully; a machine dependent function called *szn(*t*)* is called that returns 1 or 2, depending on the number of A registers required to hold an object of type *t*. If *szn* returns 2, an even/odd pair of A registers is allocated for each request.

The other issue, sharing, is more subtle, but important for good code quality. When registers are allocated, it is possible to reuse registers that hold address information, and use them to contain the values computed or accessed. For example, on the IBM 360, if register 2 has a pointer to an integer in it, we may load the integer into register 2 itself by saying:

```
L 2,0(2)
```

If register 2 had a byte pointer, however, the sequence for loading a character involves clearing the target register first, and then inserting the desired character:

```
SR 3,3
IC 3,0(2)
```

In the first case, if register 3 were used as the target, it would lead to a larger number of registers used for the expression than were required; the compiler would generate inefficient code. On the other hand, if register 2 were used as the target in the second case, the code would simply be wrong. In the first case, register 2 can be *shared* while in the second, it cannot.

In the specification of the register needs in the templates, it is possible to indicate whether required scratch registers may be shared with possible registers on the left or the right of the input tree. In order that a register be shared, it must be scratch, and it must be used only once, on the appropriate side of the tree being compiled.

The *allo* routine thus has a bit more to do than meets the eye: it calls *freereg* to obtain a free register for each A and B register request. *Freereg* makes multiple calls on the routine *usable* to decide if a given register can be used to satisfy a given need. *Usable* calls *shareit* if the register is busy, but might be shared. Finally, *shareit* calls *ushare* to decide if the desired register is actually in the appropriate subtree, and can be shared.

Just to add additional complexity, on some machines (such as the IBM 370) it is possible

to have "double indexing" forms of addressing; these are represented by OREGS's with the base and index registers encoded into the register field. While the register allocation and deallocation *per se* is not made more difficult by this phenomenon, the code itself is somewhat more complex.

Having allocated the registers and expanded the assembly language, it is time to reclaim the resources; the routine *reclaim* does this. Many operations produce more than one result. For example, many arithmetic operations may produce a value in a register, and also set the condition codes. Assignment operations may leave results both in a register and in memory. *Reclaim* is passed three parameters; the tree and cookie that were matched, and the rewriting field of the template. The rewriting field allows the specification of possible results; the tree is rewritten to reflect the results of the operation. If the tree was computed for side effects only (FOREFF), the tree is freed, and all resources in it reclaimed. If the tree was computed for condition codes, the resources are also freed, and the tree replaced by a special node type, FORCC. Otherwise, the value may be found in the left argument of the root, the right argument of the root, or one of the temporary resources allocated. In these cases, first the resources of the tree, and the newly allocated resources, are freed; then the resources needed by the result are made busy again. The final result must always match the shape of the input cookie; otherwise, the compiler error "cannot reclaim" is generated. There are some machine dependent ways of preferring results in registers or memory when there are multiple results matching multiple goals in the cookie.

### The Machine Dependent Interface

The files *order.c*, *local2.c*, and *table.c*, as well as the header file *mac2defs*, represent the machine dependent portion of the second pass. The machine dependent portion can be roughly divided into two: the easy portion and the hard portion. The easy portion tells the compiler the names of the registers, and arranges that the compiler generate the proper assembler formats, opcode names, location counters, etc. The hard portion involves the Sethi-Ullman computation, the rewriting rules, and, to some extent, the templates. It is hard because there are no real algorithms that apply; most of this portion is based on heuristics. This section discusses the easy portion; the next several sections will discuss the hard portion.

If the compiler is adapted from a compiler for a machine of similar architecture, the easy part is indeed easy. In *mac2defs*, the register numbers are defined, as well as various parameters for the stack frame, and various macros that describe the machine architecture. If double indexing is to be permitted, for example, the symbol R2REGS is defined. Also, a number of macros that are involved in function call processing, especially for unusual function call mechanisms, are defined here.

In *local2.c*, a large number of simple functions are defined. These do things such as write out opcodes, register names, and address forms for the assembler. Part of the function call code is defined here: that is nontrivial to design, but typically rather straightforward to implement. Among the easy routines in *order.c* are routines for generating a created label, defining a label, and generating the arguments of a function call.

These routines tend to have a local effect, and depend on a fairly straightforward way on the target assembler and the design decisions already made about the compiler. Thus they will not be further treated here.

### The Rewriting Rules

When a tree fails to match any template, it becomes a candidate for rewriting. Before the tree is rewritten, the machine dependent routine *nexcook* is called with the tree and the cookie; it suggests another cookie that might be a better candidate for the matching of the tree. If all else fails, the templates are searched with the cookie FORREW, to look for a rewriting rule. The rewriting rules are of two kinds: for most of the common operators, there are machine dependent rewriting rules that may be applied; these are handled by machine dependent functions that are called and given the tree to be computed. These routines may recursively call

*order* or *codgen* to cause certain subgoals to be achieved; if they actually call for some alteration of the tree, they return 1, and the code generation algorithm recanonicalizes and tries again. If these routines choose not to deal with the tree, the default rewriting rules are applied.

The assignment ops, when rewritten, call the routine *serasg*. This is assumed to rewrite the tree at least to the point where there are no side effects in the left hand side. If there is still no template match, a default rewriting is done that causes an expression such as

$$a += b$$

to be rewritten as

$$a = a + b$$

This is a useful default for certain mixtures of strange types (for example, when *a* is a bit field and *b* an character) that otherwise might need separate table entries.

Simple assignment, structure assignment, and all forms of calls are handled completely by the machine dependent routines. For historical reasons, the routines generating the calls return 1 on failure, 0 on success, unlike the other routines.

The machine dependent routine *setbin* handles binary operators; it too must do most of the job. In particular, when it returns 0, it must do so with the left hand side in a temporary register. The default rewriting rule in this case is to convert the binary operator into the associated assignment operator; since the left hand side is assumed to be a temporary register, this preserves the semantics and often allows a considerable saving in the template table.

The increment and decrement operators may be dealt with with the machine dependent routine *setincr*. If this routine chooses not to deal with the tree, the rewriting rule replaces

$$x ++$$

by

$$( (x += 1) - 1 )$$

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of *x* is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code. *Offstar* is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREGS with a constant offset from a register, and *offstar* is called with a tree of the form

$$expr + const$$

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

### The Sethi-Ullman Computation

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The

theory also does not account for side effects: this rules out various forms of pathology that arise from assignment and assignment ops. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a "useless" theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries "on the fly") is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no "right" way to write them: it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problem have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue: operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized: it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

$$A - B$$

where  $A$  and  $B$  are expressions: suppose further that  $B$  takes two registers, and  $A$  one. It is possible to compute the full expression in two registers by first computing  $B$ , and then, using the scratch register used by  $B$ , but not containing the answer, compute  $A$ . The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine *setbin*, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

$$*(p + i)$$

where  $p$  is a pointer and  $i$  an integer, this can probably be done in one register on most machines. Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both  $p$  and  $i$  into



registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause *offstar* to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

### Register Allocation

After the Sethi-Ullman numbers are computed, *order* calls a routine, *rallo*, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; switch and function return are the two major places. The expression tree has a field, *rall*, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in *rall*, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine *rmove*. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

### Compiler Bugs

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form "no match for op ...", or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in

the code generator, and follow the tree through the process of being matched and rewritten. Some flags of interest are `-e`, which prints the expression tree, `-r`, which gives information about the allocation of registers, `-a`, which gives information about the performance of *rallo*, and `-o`, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

### Summary and Conclusion

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable "peephole" optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

Finally, I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed "into the pits" with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun.

## References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (1978).
3. A. Snyder, *A Portable Compiler for the Language C*, Master's Thesis, M.I.T., Cambridge, Mass. (1974).
4. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp. 97-104 (January 1978).
5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence*, Bell Laboratories internal memorandum (1977).
6. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
7. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.* 23(3) pp. 488-501 (1975). Also in *Proc. ACM Symp. on Theory of Computing*, pp. 207-217, 1975.
8. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. Assoc. Comp. Mach.* 17(4) pp. 715-728 (October 1970). Reprinted as pp. 229-247 in *Compiler Techniques*, ed. B. W. Pollack, Auerbach, Princeton NJ (1972).
9. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Proc. 4th ACM Symp. on Principles of Programming Languages*, pp. 21-28 (January 1977).







## A Dial-Up Network of UNIX™ Systems

*D. A. Nowitz*

*M. E. Lesk*

Bell Laboratories  
Murray Hill, New Jersey 07974

### *ABSTRACT*

A network of over eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

August 18, 1978

---

†UNIX is a Trademark of Bell Laboratories.

# A Dial-Up Network of UNIX™ Systems

*D. A. Nowitz*

*M. E. Lesk*

Bell Laboratories  
Murray Hill, New Jersey 07974

## 1. Purpose

The widespread use of the UNIX† system<sup>1</sup> within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back and forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.<sup>2,3</sup> Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

## 2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes: in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there: this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster

---

†UNIX is a Trademark of Bell Laboratories.



than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.<sup>4,5</sup> Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustrates the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

### 3. Processing

The user has two commands which set up communications. *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

## File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

*Uucico* may be started in several ways;

- a) by a system daemon,
- b) by one of the *uucp* or *uux* programs,
- c) by a remote system.

## Scan For Work

The file names in the spool directory are constructed to allow the daemon programs (*uucico*, *uuxqv*) to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

## Call Remote System

The call is made using information from several files which reside in the *uucp* program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

- [1] system name,
- [2] times to call the system (days-of-week and times-of-day),
- [3] device or device type to be used for call,
- [4] line speed,
- [5] phone number,
- [6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

### Line Protocol Selection

The remote system sends a message

*P proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the *proto-list* for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

*U code*

where *code* is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the *uucp* transmission program. Other protocols may be added by individual installations.

### Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

- S send a file,
- R receive a file,
- C copy complete,
- H hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE*'s spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

### Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

### 4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in

and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.<sup>6</sup> Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

### 5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

## 6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.<sup>7</sup> There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

## 7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

## Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

### References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).
2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.* 57(6) pp. 2177-2200 (1978).
3. G. L. Chesson, "The Network UNIX System," *Operating Systems Review* 9(5) pp. 60-66 (1975). Also in *Proc. 5th Symp. on Operating Systems Principles*.
4. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications*, p. 21F (June 1974). IEEE Cat. No. 74CH0859-9-CSCB.
5. A. G. Fraser, "A Virtual Channel Network," *Datamation*, pp. 51-56 (February 1975).
6. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," *Comp. Sci. Tech. Rep. No. 41*, Bell Laboratories, Murray Hill, New Jersey (June 1976).
7. F. P. Brooks, Jr., *The Mythical Man-Month*. Addison-Wesley, Reading, Mass. (1975).

# **Uucp Implementation Description**

*D. A. Nowitz*

## *ABSTRACT*

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hardwired communication lines. This document gives a detailed implementation description of the current (second) implementation of uucp.

This document is for use by an administrator/installer of the system. It is not meant as a user's guide.

October 31, 1978

# Uucp Implementation Description

D. A. Nowitz

## Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.<sup>1</sup> This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execution files* are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

- uucp      This program creates work and gathers data files in the spool directory for the transmission of files.
- uux        This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.
- uucico    This program executes the work files for data transmission.
- uuxqt     This program executes the execution files for UNIX command execution.

The secondary programs are:

- uulog     This program updates the log file with new entries and reports on the status of uucp requests.
- uuclean   This program removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

## 1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

```
uucp [option] ... source ... destination
```

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

- c        Make directories when necessary for copying the file.

---

†UNIX is a Trademark of Bell Laboratories.

<sup>1</sup>M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.



- c Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.
  - g*letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)
  - m Send mail on completion of the work.
- The following options are used primarily for debugging:
- r Queue the job but do not start *uucico* program.
  - s*dir* Use directory *dir* for the spool directory.
  - x*num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as "*\*[/]*". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

```
uucp *.c usg!/usr/dan
```

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the "usg" machine.

The source and/or destination names may also contain a *~user* prefix. This translates to the login directory on the specified system. For names with partial path-names, the current directory is prepended to the file name. File names with *../* are not permitted.

The command

```
uucp usg!~dan/*.h ~dan
```

will set up the transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

- [1] Copy source to destination on local system.
- [2] Receive files from other systems.
- [3] Send files to a remote systems.
- [4] Send files from remote systems to another remote system.
- [5] Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the *-r* option was specified).

#### Type 1

A *cp* command is used to do the work. The *-d* and the *-m* options are not honored in this case.

#### Type 2

A one line *work file* is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

- [1] R

- [2] The full path-name of the source or a `~user/path-name`. The `~user` part will be expanded on the remote system.
- [3] The full path-name of the destination file. If the `~user` notation is used, it will be immediately expanded to be the login directory for the user.
- [4] The user's login name.
- [5] A `"-"` followed by an option list. (Only the `-m` and `-d` options will appear in this list.)

### Type 3

For each source file, a *work file* is created and the source file is copied into a *data file* in the spool directory. (A `"-c"` option on the *uucp* command will prevent the *data file* from being made.) In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

- [1] S
- [2] The full-path name of the source file.
- [3] The full-path name of the destination or `~user/file-name`.
- [4] The user's login name.
- [5] A `"-"` followed by an option list.
- [6] The name of the *data file* in the spool directory.
- [7] The file mode bits of the source file in octal print format (e.g. 0666).

### Type 4 and Type 5

*Uucp* generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

## 2. Uux - UNIX To UNIX Execution

The *uux* command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the *uux* command is

```
uux [-] [option] ... command-string
```

where the command-string is made up of one or more arguments. All special shell characters such as `"<>|"` must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a `"!"` will not be treated as files. (They will not be copied to the execution machine.) The `"-"` is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The options, essentially for debugging, are:

- `-r` Don't start *uucico* or *uuxqt* after queuing the job;
- `-xnum` Num is the level of debugging output desired.

The command

```
pr abc | uux - usg!lpr
```

will set up the output of `"pr abc"` as standard input to an `lpr` command to be executed on system `"usg"`.

*Uux* generates an *execute file* which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed

by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE*.)

The *execute file* will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

#### User Line

U user system

where the *user* and *system* are the requester's login name and system.

#### Required File Line

F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

#### Standard Input Line

I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "-" option is used. If a standard input is not specified, "/dev/null" is used.

#### Standard Output Line

O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note - the use of ">>" is not implemented.)

#### Command Line

C command | arguments | ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

### 3. Uucico - Copy In, Copy Out

The *uucico* program will perform the following major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

*Uucico* may be started in several ways:

- a) by a system daemon,
- b) by one of the *uucp*, *uux*, *uuxqt* or *uucico* programs,
- c) directly by the user (this is usually for testing),
- d) by a remote system. (The *uucico* program should be specified as the "shell" field in the "/etc/passwd" file for the "uucp" logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (*-s* option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

- rl* Start the program in *MASTER* mode. This is used when *uucico* is started by a program or "cron" shell.
- ssys* Do work only for system *sys*. If *-s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

- ddir* Use directory *dir* for the spool directory.
- xnum* *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

### Scan For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where:

- Type* is an upper case letter, ( *C* - copy command file, *D* - data file, *X* - execute file);
- System-name* is the remote system;
- Grade* is a character;
- Number* is a four digit, padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix "C."). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

### Call Remote System

The call is made using information from several files which reside in the *uucp* program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is:

- [1] system name.
- [2] times to call the system (days-of-week and times-of-day),
- [3] device or device type to be used for call,
- [4] line speed.
- [5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU*,
- [6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

### Line Protocol Selection

The remote system sends a message

*P proto-list*

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks the *proto-list* for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

*U code*

where *code* is either a one character protocol letter or *N* which means there is no common protocol.

### Work Processing

The initial roles (*MASTER* or *SLAVE*) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the "-r1" *uucico* option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

- S* send a file.
- R* receive a file.
- C* copy complete.
- X* execute a *uucp* command.
- H* hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, *XY*, *XN*, corresponding to yes or no for each request.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the case of *CN*, the transferred file will be in the spool directory with a name beginning with "TM".) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE*'s spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

### Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a *-s* option was specified.

### 4. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the "Uux" section above.

### Command Execution

The execution is accomplished by executing a *sh -c* of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

### 5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

- ssys* Print entries where *sys* is the remote system name;
- user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

### 6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

- ddir* The directory to be scanned is *dir*.
- m* Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the *uucp* programs since the *setuid* bit will be set on these programs. The mail will therefore most often go to the owner of the *uucp* programs.)

- *n hours* Change the aging time from 72 hours to *hours* hours.
- *ppre* Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
- *x num* This is the level of debugging output desired.

## 7. Security

**The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.**

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. (See the "Files required for execution" section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specified in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs *uucp*, *uucico*, *uux*, *uuxqt* should be also owned by uucp and have the setuid bit set.)

## 8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name*.

The four directories are:

- lib* (/usr/src/cmd/uucp) This directory contains the source files for generating the *uucp* system.
- program* (/usr/lib/uucp) This is the directory used for the executable system programs and the system files.
- spool* (/usr/spool/uucp) This is the spool directory used during *uucp* execution.
- xqtdir* (/usr/spool/uucp/.XQTDIR) This directory is used during execution of *execute files*.

The names given in parentheses above are the default values for the directories. The italicized named *lib*, *program*, *xqtdir*, and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

### Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

### makefile modification

There are several *make* variable definitions which may need modification.

- INSDIR This is the *program* directory (e.g. INSDIR = /usr/lib/uucp). This parameter is used if "make cp" is used after the programs are compiled.
- IOCTL This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard "C" library; the statement "IOCTL = ioctl.o" is required in this case.
- PKON The statement "PKON = pkon.o" is required if the packet driver is not in the kernel.

### Compile the system The command

make

will compile the entire system. The command

make cp

will copy the commands to the to the appropriate directories.

The programs *uucp*, *uux*, and *uulog* should be put in "/usr/bin". The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

### Files required for execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

### L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

line call-unit speed

where:

- line is the device for the line (e.g. cu10),
- call-unit is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number "0" in this field.),
- speed is the line speed.

The line

cu10 cua0 300

would be set up for a system which had device cu10 wired to a call-unit cua0 for use at 300 baud.

### L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is



abb dial-seq

where;

abb is the abbreviation,

dial-seq is the dial sequence to call that location.

The line

py 165-

would be set up so that entry py7777 would send 165-7777 to the dial-unit.

## LOGIN/SYSTEM NAMES

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique *system name* which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

## USERFILE

This file contains user accessibility information. It specifies four types of constraint;

- [1] which files can be accessed by a normal user of the local machine,
- [2] which files can be accessed from a remote computer,
- [3] which login name is used by a particular remote computer,
- [4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

login.sys [ c ] path-name [ path-name ] ...

where:

login is the login name for a user or the remote computer,

sys is the system name for a remote computer,

c is the optional *call-back required* flag,

path-name is a path-name prefix that is acceptable for *user*.

The constraints are implemented as follows.

- [1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.
- [2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a *null* system name is used.
- [3] When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.
- [4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

```
u,m /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with `"/usr/xyz"`.

The line

```
dan, /usr/dan
```

allows the ordinary user *dan* to issue commands for files whose name starts with `"/usr/dan"`.

The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with `"/usr/spool"`.

The lines

```
root, /
, /usr
```

allows any user to transfer files beginning with `"/usr"` but the user with login *root* can transfer any file.

## L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

### system name

The name of the remote system.

### time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800-1730).

The day portion may be a list containing some of

*Su Mo Tu We Th Fr Sa*

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800-1230). If no time portion is specified, any time of day is assumed to be ok for the call.

### device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

### speed

This is the line speed for the call (e.g. 300).

### phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995-9980).

For the hardwired devices, this field contains the same string as used for the *device* field.

### login

The login information is given as a series of fields and subfields in the format

expect send [ expect send ] ...

where: *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect1-send-expect1...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file would be

```
sys Any ACU 300 mh7654 login uucp ssword: word
```

The expect algorithm looks at the last part of the string as illustrated in the password field.

## 9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

### SQFILE — sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

### TM — temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the *spool* directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

```
uuclean -pTM
```

will remove all *TM* files older than three days.

### LOG — log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

```
uulog
```

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new *LOG* entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

### STST — system status files

These files are created in the *spool* directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when machines are conversing. The form of the file name is

```
STST.sys
```

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

### LCK — lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

```
LCK..str
```

where *str* is either a device or system name. The files may be left in the *spool* directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

### Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a "shell" file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

```
program/uulog
program/uucico -rl
```

Note that the "-rl" option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C*, or *D*, files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
program/uuclean -pTM -pC. -pD.
program/uuclean -pST -pLCK -n12
```

can be used. Note the "-n12" option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the "-n" option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILES*. A shell like

```
cp spool/LOGFILE spool/o.LOGFILE
rm spool/LOGFILE
```

can be used.

### Login Entry

One or more logins should be set up for *uucp*. Each of the "/etc/passwd" entries should have the "*program/uucico*" as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should be the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of *uucp* (*uucico*) only.

### File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp*, *uux*, *uucico* and *uuxqt* should be owned by the *uucp* login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys*, *SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

