PLEXUS

Sys5 UNIX Programmer's Guide

98-05080.1 Ver. B May, 1986

Sys5 UNIX Programmer's Guide

98-05080.1 Ver. B May, 1986

PLEXUS COMPUTERS, INC.
3833 North First Street
San Jose, CA 95134
408/943-9433

Printed in the United States of America

CONTENTS

10. RATFOR

11. PROGRAMMING LANGUAGE EFL

12. CURSES & TERMINFO PACKAGE

13. CURSES EXAMPLES

1. INTRODUCTION

This volume describes two main programming languages supported on the UNIX system. The languages include:

- C Language — A medium-level programming language which was used to write most of the UNIX operating system. Chapter 2 describes the C language. Chapters 3 through 7 describe the libraries and support tools available with the UNIX system for the benefit of the C language programmer. These chapters contain the following:

  C LANGUAGE— Chapter 2 provides a summary of the grammar and rules of the C programming language. Chapter 2 describes the C language.

  LIBRARIES— Chapters 3 and 4 describe functions and declarations that support the C Language and how to use these functions. Chapter 3 describes the C Library and Chapter 4 describes the Object File and Math Libraries.

  THE "cc" COMMAND— Chapter 5 describes the command used to compile C language programs, produce assembly language programs, and produce executable programs.

  A C PROGRAM CHECKER - "lint"— Chapter 6 describes a program that attempts to detect compile-time bugs and non-portable features in C programs.

  A SYMBOLIC DEBUGGER - "sdb" — Chapter 7 describes a symbolic debugging program that is used to debug compiled C language programs.

- Fortran — Fortran 77, a rational Fortran preprocessor (Ratfor), and EFL are described as follows:

  UNIX SYSTEM COMMANDS FOR FORTRAN— Chapter 8 describes the various commands that may be used with Fortran on a UNIX system.

  FORTRAN 77 — Chapter 9 describes the implementation of Fortran 77 on the UNIX system in terms of the variations from the American National Standard.

---

* Trademark of "AT&T".

RATFOR— Chapter 10 describes the Ratfor preprocessor. This preprocessor provides a means for writing Fortran in a fashion similar to the C language. This preprocessor provides (among other things) simplified control-flow statements.

EFL— Chapter 11 describes the programming language EFL.

Chapter 12 describes the **curses** and **terminfo** package that provides the programmer with screen-oriented programming capabilities. Chapter 13 provides examples of **curses** programs.

It is assumed that the user of this document has at least two years of specialized training in computer-related fields. The user is also expected to use the UNIX system for software development. Chapters 8, 9, 10 and 11 assume that the user is already familiar with Fortran 77. If not familiar, review one of the many texts that describes Fortran 77. The following texts are suggested:

FORTRAN 77
Harry Katzan, Jr.
Van Nostrand Reinhold

FORTRAN 77 - FEATURING STRUCTURED PROGRAMMING
Loren P. Meissner and Elliot I. Organick
Addison-Wesly

AMERICAN NATIONAL STANDARD PROGRAMMING
LANGUAGE FORTRAN
ANSI x3.9 - 1978
American National Standards Institute

Throughout this document, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *UNIX System Administrator Reference Manual*. Each reference of the form **name**(1) and **name**(6) refers to entries in the *UNIX System Reference Manual*. All other references to entries of the form **name**(N), where possibly followed by a letter, refer to entry **name** in section **N** of the *UNIX System Programmer Reference Manual.*

2. C LANGUAGE

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, new-lines, 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 Tokens

2.1.1 Comments

The characters /* introduce a comment which terminates with the characters /fB*/. Comments do not nest.

2.1.2 Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names.

2.1.3 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

auto	**do**	**for**	**return**	**typedef**
break	**double**	**goto**	**short**	**union**
case	**else**	**if**	**sizeof**	**unsigned**
char	**enum**	**int**	**static**	**void**
continue	**external**	**long**	**struct**	**while**
default	**float**	**register**	**switch**	

Some implementations also reserve the words **fortran** and **asm**.

2.1.4 Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "NAMES."

2.1.4.1 Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with **0** (digit zero). An octal constant consists of the digits **0**

through **7** only. 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. Otherwise, the integer constant is taken to be decimal. 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**. Otherwise, integer constants are **int**.

2.1.4.2 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.

2.1.4.3 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 nongraphic characters, the single quote (') and the backslash (\), may be represented according to the following table of escape sequences:

new-line	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

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 behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **int**.

2.1.4.4 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 has type **double**.

2.1.4.5 Enumeration Constants

Names declared as enumerators (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") have type **int**.

2.1.5 Strings

A string is a sequence of characters surrounded by double quotes, as in "...". A string has type "array of **char**" and storage class **static** (see "NAMES") and is initialized with the given characters. 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.

A \ and the immediately following new-line are ignored. All strings, even when written identically, are distinct.

2.2 Syntax Notation

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 nonterminal symbol is indicated by the subscript "opt," so that

$$\{\ expression_{opt}\ \}$$

indicates an optional expression enclosed in braces. The syntax is summarized in "SYNTAX SUMMARY".

2.3 Names

The C language 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.

2.3.1 Storage Class

There are four declarable storage classes:

- Automatic
- Static
- External
- Register.

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "STATEMENTS") 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.

2.3.2 Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** 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. In particular, **char** may be signed or unsigned by default.

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.

The properties of **enum** types (see "Structure, Union, and Enumeration Declarations" under "DECLARATIONS") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation.

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. **Char**, **int** of all sizes whether **unsigned** or not, and **enum** will collectively be called *integral* types. The **float** and **double** types will collectively be called *floating* types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

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.

2.4 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.

2.5 Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

2.5.1 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 preserves sign. 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.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, 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 integer or to a **char**, it is truncated on the left. Excess bits are simply discarded.

2.5.2 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. This result is undefined if it cannot be represented as a float.

2.5.3 Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

2.5.4 Pointers and Integers

An expression of integral type 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.

2.5.5 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 $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** 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.

2.5.6 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."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.
2. Then, if either operand is **double,** the other is converted to **double** and that is the type of the result.
3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.
4. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.
5. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.

6. Otherwise, if either operand is **unsigned,** the other is converted to **unsigned** and that is the type of the result.
7. Otherwise, both operands must be **int**, and that is the type of the result.

2.5.7 Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "STATEMENTS") or as the left operand of a comma expression (see "Comma Operator" under "EXPRESSIONS").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

2.6 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 + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, 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 are summarized in the grammar of "SYNTAX SUMMARY".

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 subexpression evaluation takes 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 undefined. Most 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.

2.6.1 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$_{opt}$ *)*
primary-expression . 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 ...", 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** and floating constants have type **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 "Initialization" under "DECLARATIONS.")

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 subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, * and +, respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "TYPES REVISITED."

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**. 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 "Unary Operators" and "Type Names" under "DECLARATIONS."

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. 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 a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from - and >) 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 "Structure, Union, and Enumeration Declarations" under "DECLARATIONS."

2.6.2 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 the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator **!** is one if the value of its operand is zero, zero if the value of its operand is nonzero. 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**=**x**+**1**. See the discussions "Additive Operators" and "Assignment Operators" 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 "Type Names" under "Declarations."

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 **unsigned** 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**.

2.6.3 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.

The binary **%** operator yields the remainder from the division of the first expression by the second. The operands must be integral.

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).

2.6.4 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 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 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.

2.6.5 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 arithmetic.

2.6.6 Relational Operators

The relational operators group left to right.

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.

2.6.7 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 only if 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.

2.6.8 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.

2.6.9 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.

2.6.10 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.

2.6.11 Logical AND Operator

logical-and-expression:
expression && expression

The **&&** operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 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**.

2.6.12 Logical OR Operator

logical-or-expression:
expression | *expression*

The | operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, | guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

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**.

2.6.13 Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, 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. If both are structures or unions of the same type, the result

has the type of the structure or union. If both pointers are 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.

2.6.14 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. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* = **E2** may be inferred by taking it as equivalent to **E1** = **E1** *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 "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

2.6.15 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, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "DECLARATIONS"), the comma operator as described in this subpart 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.

2.7 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$_{opt}$;

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$_{opt}$
sc-specifier decl-specifiers$_{opt}$

The list must be self-consistent in a way described below.

2.7.1 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. See "Typedef" for more information. The meanings of the various storage classes were discussed in "Names."

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 (see "External Definitions") 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 in each function are effective. Moreover, only variables of certain types will be stored in registers. 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.

2.7.2 Type Specifiers

The type-specifiers are

type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier
basic-type-specifier:
    basic-type
    basic-type basic-type-specifiers
basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void

At most one of the words **long** or **short** may be specified in conjunction with **int**; the meaning is the same as if **int** were not mentioned. The word **long** may be specified in conjunction with **float**; the meaning is the same as **double**. The word **unsigned** may be specified alone, or in conjunction with **int** or any of its short or long varieties, or with **char**.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of **long**, **short**, or **unsigned** is not permitted with **typedef** names. If the type-specifier is missing from a declaration, it is taken to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure, Union, and Enumeration Declarations." Declarations with **typedef** names are discussed in "Typedef."

2.7.3 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 initializer$_{opt}$

Initializers are discussed in "Initialization". 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-expression$_{opt}$ *]*

The grouping is the same as in expressions.

2.7.4 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 , whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multidimensional 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 multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; 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())**. The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**. 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 3-dimensional array of integers, with rank 3×5×7. 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" and the last has type **int**.

2.7.5 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, a non-negative constant expression, 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 the declarations are read left to right. Each nonfield 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.

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, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

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. For these reasons, it is strongly recommended that fields be declared as **unsigned**. 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. Structure tags 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 third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
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**.

2.7.6 Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding 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.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

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=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

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 possible values are drawn from the set {0,1,20,21}.

2.7.7 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* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in "CONSTANT EXPRESSIONS", 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 that are not initialized are guaranteed to start off as zero. Automatic and register variables that 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), 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 zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. 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 one-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, three 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.

2.7.8 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*$_{opt}$]

To avoid ambiguity, in the construction

(abstract-declarator)

the abstract-declarator is required to be nonempty. 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 (*)()
int (*[3])()

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

2.7.9 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 becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators." For example, after

typedef int MILES, *KLICKSP;
typedef struct { **double re, im;** } **complex;**

the constructions

MILES distance;
extern KLICKSP 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. The **zp** is a pointer to such a structure.

The **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.

2.8 Statements

Except as indicated, statements are executed in sequence.

2.8.1 Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

2.8.2 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*$_{opt}$ *statement-list*$_{opt}$ }

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.

2.8.3 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 nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The "else" ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

2.8.4 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 nonzero. The test takes place before each execution of the statement.

2.8.5 Do Statement

The **do** statement has the form

do *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

2.8.6 For Statement

The **for** statement has the form:

for ($exp\text{-}1_{opt}$; $exp\text{-}2_{opt}$; $exp\text{-}3_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-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 incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied **while** clause equivalent to **while(1)**; other missing expressions are

simply dropped from the expansion above.

2.8.7 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 "CONSTANT EXPRESSIONS."

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.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see "Break Statement."

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.

2.8.8 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.

2.8.9 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, see "Null Statement".)

2.8.10 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 function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

2.8.11 Goto Statement

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

2.8.12 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 subblocks in which the same identifier has been redeclared. See "SCOPE RULES."

2.8.13 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**.

2.9 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 (see "Type Specifiers" in "DECLARATIONS") 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.

2.9.1 External Function Definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "Scope of Externals" in "SCOPE RULES" 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$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

The function-body has the form

function-body:
declaration-list$_{opt}$ *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.

The C program converts all **float** actual parameters to **double**, so formal parameters declared **float** have their declaration adjusted to read **double**. All **char** and **short** formal parameter declarations are similarly adjusted to read **int**. 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 ...."

2.9.2 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**.

2.10 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 scopes 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.

2.10.1 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 a block persists until the end of the block. The lexical scope of labels is the whole of the function 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 (see "Structure, Union, and Enumeration Declarations" in "DECLARATIONS") that tags, identifiers associated with ordinary variables, and identities associated with structure and union members form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **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**.

2.10.2 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 at least one 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.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated

elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

2.11 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. There may be any number of blanks and horizontal tabs between the # and the directive. 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.

2.11.1 Token Replacement

A compiler-control line of the form

#define *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

#define *identifier(identifier, ...)token-string*$_{opt}$

where there is no space between the first identifier and the **(**, is a macro definition with arguments. There may be zero or more formal parameters. 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. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers 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 (if any) to be forgotten.

If a **#define**d identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

2.11.2 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 file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include *<filename>*

searches only the specified or standard places and not the directory of the **#include**. (How the places are specified is not part of the language.)

#includes may be nested.

2.11.3 Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "CONSTANT EXPRESSIONS"; the following additional restrictions apply here: the constant expression may not contain **sizeof** casts, or an enumeration constant.)

A restricted constant expression may also contain the additional unary expression

defined *identifier*
or
defined(*identifier*

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant-expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#ifdef(***identifier***)**. A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if!defined(***identifier***)**.

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 a **#else** or, lacking a **#else**, the **#endif** are ignored.

These constructions may be nested.

2.11.4 Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

#line *constant "filename"*

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 *"filename"*. If *"filename"* is absent, the remembered file name does not change.

2.12 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 because **auto** functions do not exist. 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**.´´

2.13 TYPES REVISITED

This part summarizes the operations which can be performed on objects of certain types.

2.13.1 Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the -> or the . must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int     type;
        } n;
        struct
        {
                int     type;
                int     intnode;
        } ni;
        struct
        {
                int     type;
                float   floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

2.13.2 Functions

There are only two things that can be done with a function m 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 **(.**

2.13.3 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 multidimensional arrays. If **E** is an *n*-dimensional array of rank i×j×...×k, then **E** appearing in an expression is converted to a pointer to an (n-1)-dimensional array with rank j×...×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 3×5 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 five 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.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

2.13.4 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, see "Unary Operators" under"EXPRESSIONS" and "Type Names"under "DECLARATIONS."

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;
```

The **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 measures bytes. The **char**'s have no alignment requirements; everything else must have an even address.

2.14 CONSTANT EXPRESSIONS

In several places C requires expressions that 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, casts to integral types, enumeration 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 use floating constants and arbitrary casts and 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.

2.15 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 nuisances 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. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine dependent 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 some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., 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.

2.16 SYNTAX SUMMARY

This summary of C syntax is intended more for aiding comprehension than

as an exact statement of the language.

2.16.1 Expressions

The basic expressions are:

expression:
primary
** expression*
&lvalue
- expression
! expression
~ expression
++ lvalue
--lvalue
lvalue ++
lvalue --
sizeof *expression*
sizeof (*type-name***)**
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$ *)*
primary [expression]
primary . 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.

binop:

```
* / %
+ -
>> <<
< > <= >=
== !=
&
^
|
&&
|
```

The conditional operator groups right to left.

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.

2.16.2 Declarations

declaration:

decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:

type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:

auto
static
extern
register
typedef

type-specifier:
    struct-or-union-specifier
    typedef-name
    enum-specifier
basic-type-specifier:
    basic-type
    basic-type basic-type-specifiers
basic-type:
    char
    short
    int
    long
    unsigned
    float
    double
    void

enum-specifier:
    enum { *enum-list* }
    enum *identifier* { *enum-list* }
    enum *identifier*

enum-list:
    enumerator
    enum-list , enumerator

enumerator:
    identifier
    identifier = constant-expression

init-declarator-list:
    init-declarator
    init-declarator , init-declarator-list

init-declarator:
    declarator initializer$_{opt}$

declarator:
    identifier
    (declarator)
    ** declarator*
    declarator ()
    declarator [constant-expression$_{opt}$ *]*

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* }
{ *initializer-list* , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
* *abstract-declarator*
abstract-declarator ()
abstract-declarator [constant-expression$_{opt}$ *]*

typedef-name:
identifier

2.16.3 Statements

compound-statement:
{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

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 *(exp*$_{opt}$*;exp*$_{opt}$*;exp*$_{opt}$*) statement*
switch *(expression) statement*
case *constant-expression : statement*
default *: statement*
break ;
continue ;
return ;
return *expression ;*
goto *identifier ;*
identifier : statement
;

2.16.4 External definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
extern *declaration* ;
static *declaration* ;

2.16.5 Preprocessor

#define *identifier token-string*$_{opt}$
#define *identifier*(*identifier*,...)*token-string*$_{opt}$
#undef *identifier*
#include *"filename"*
#include <*filename*>
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant "filename"*

3. C LIBRARIES

This chapter and Chapter 4 describe the libraries that are supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort by linking only what is needed, allowing use of locally produced functions, etc. All of the functions described are also described in Part 3 of the *Sys5 UNIX Programmer Reference Manual*. Most of the declarations described are in Part 5 of the *Sys5 UNIX Programmer Reference Manual*. The three main libraries on the UNIX system are:

C library — This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described later in this chapter.

Object file — This library provides functions for the access and manipulation of object files. This library is described in Chapter 4.

Math library — This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is described in Chapter 4.

Some libraries consist of two portions - functions and declarations. In some cases, the user must request that the functions (and/or declarations) of a specific library be included in a program being compiled. In other cases, the functions (and/or declarations) are included automatically.

3.0.1 Including Functions

When a program is being compiled, the compiler will automatically search the C language library to locate and include functions that are used in the program. This is the case only for the C library and no other library. In order for the compiler to locate and include functions from other libraries, the user must specify these libraries on the command line for the compiler. For example, when using functions of the math library, the user must request that the math library be searched by including the argument **-lm** on the command line, such as:

```
cc file.c -lm
```

The argument **-lm** must come after all files that reference functions in the math library in order for the link editor to know which functions to include in the a.out file.

This method should be used for all functions that are not part of the C language library.

3.0.2 Including Declarations

Some functions require a set of declarations in order to operate properly. A set of declarations is stored in a file under the */usr/include* directory. These files are referred to as *header files*. In order to include a certain header file, the user must specify this request within the C language program. The request is in the form:

#include <file.h>

where *file.h* is the name of the file. Since the header files define the type of the functions and various preprocessor constants, they must be included before invoking the functions they declare.

The remainder of this chapter describes the functions and header files of the C Library. The description of the library begins with the actions required by the user to include the functions and/or header files in a program being compiled (if any). Following the description of the actions required is information in three-column format of the form:

function **reference**(N) Brief description.

The functions are grouped by type while the reference refers to section 'N' in the *Sys5 UNIX Programmer Reference Manual*. Following this, are descriptions of the header files associated with these functions (if any).

3.1 THE C LIBRARY

The C library consists of several types of functions. All the functions of the C library are loaded automatically by the compiler. Various declarations must sometimes be included by the user as required. The functions of the C library are divided into the following types:

- Input/output control
- String manipulation
- character manipulation
- Time functions
- Miscellaneous functions.

3.1.1 Input/Output Control

These functions of the C library are automatically included as needed during the compiling of a C language program. No command line request is needed.

The header file required by the input/output functions should be included in the program being compiled. This is accomplished by including the line:

#include <stdio.h>

near the beginning of each file that references an input or output function.

The input/output functions are grouped into the following categories:

- File access
- File status
- Input
- Output
- Miscellaneous.

3.1.2 File Access Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
fclose	**fclose**(3S)	Close an open stream.
fdopen	**fopen**(3S)	Associate stream with an **open**(2) ed file.
fileno	**ferror**(3S)	File descriptor associated with an open stream.
fopen	**fopen**(3S)	Open a file with specified permissions. **Fopen** returns a pointer to a stream which is used in subsequent references to the file.
freopen	**fopen**(3S)	Substitute named file in place of open stream.
fseek	**fseek**(3S)	Reposition the file pointer.
pclose	**popen**(3S)	Close a stream opened by **popen**.
popen	**popen**(3S)	Create pipe as a stream between calling process and command.
rewind	**fseek**(3S)	Reposition file pointer at beginning of file.
setbuf	**setbuf**(3S)	Assign buffering to stream.
vsetbuf	**setbuf**(3S)	Similar to **setbuf**, but allowing finer control.

3.1.3 File Status Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
clearerr	**ferror**(3S)	Reset error condition on stream.
feof	**ferror**(3S)	Test for "end of file" on stream.
ferror	**ferror**(3S)	Test for error condition on stream.
ftell	**fseek**(3S)	Return current position in the file.

3.1.4 Input Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
fgetc	**getc**(3S)	True function for **getc** (3S).
fgets	**gets**(3S)	Read string from stream.
fread	**fread**(3S)	General buffered read from stream.
fscanf	**scanf**(3S)	Formatted read from stream.
getc	**getc**(3S)	Read character from stream.
getchar	**getc**(3S)	Read character from standard input.
gets	**gets**(3S)	Read string from standard input.
getw	**getc**(3S)	Read word from stream.
scanf	**scanf**(3S)	Read using format from standard input.
sscanf	**scanf**(3S)	Formatted from string.
ungetc	**ungetc**(3S)	Put back one character on stream.

3.1.5 Output Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
fflush	**fclose**(3S)	Write all currently buffered characters from stream.
fprintf	**printf**(3S)	Formatted write to stream.
fputc	**putc**(3S)	True function for **putc** (3S).
fputs	**puts**(3S)	Write string to stream.
fwrite	**fread**(3S)	General buffered write to stream.
printf	**printf**(3S)	Print using format to standard output.
putc	**putc**(3S)	Write character to standard output.
putchar	**putc**(3S)	Write character to standard output.
puts	**puts**(3S)	Write string to standard output.
putw	**putc**(3S)	Write word to stream.
sprintf	**printf**(3S)	Formatted write to string.
vfprintf	**vprint**(3C)	Print using format to stream by **varargs**(5) argument list.
vprintf	**vprint**(3C)	Print using format to standard output by **varargs**(5) argument list.
vsprintf	**vprintf**(3C)	Print using format to stream string by **varargs**(5) argument list.

3.1.6 Miscellaneous Functions

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ctermid	**ctermid**(3S)	Return file name for controlling terminal.
cuserid	**cuserid**(3S)	Return login name for owner of current process.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
system	**system**(3S)	Execute shell command.
tempnam	**tempnam**(3S)	Create temporary file name using directory and prefix.
tmpnam	**tmpnam**(3S)	Create temporary file name.
tmpfile	**tmpfile**(3S)	Create temporary file.

3.1.7 String Manipulation Functions

These functions are used to locate characters within a string, copy, concatenate, and compare strings. These functions are automatically located and loaded during the compiling of a C language program. No command line request is needed since these functions are part of the C library. The string manipulation functions are declared in a header file that may be included in the program being compiled. This is accomplished by including the line:

#include <string.h>

near the beginning of each file that uses one of these functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
strcat	**string**(3C)	Concatenate two strings.
strchr	**string**(3C)	Search string for character.
strcmp	**string**(3C)	Compares two strings.
strcpy	**string**(3C)	Copy string.
strcspn	**string**(3C)	Length of initial string not containing set of characters.
strlen	**string**(3C)	Length of string.
strncat	**string**(3C)	Concatenate two strings with a maximum length.
strncmp	**string**(3C)	Compares two strings with a maximum length.
strncpy	**string**(3C)	Copy string over string with a maximum length.
strpbrk	**string**(3C)	Search string for any set of characters.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
strrchr	**string**(3C)	Search string backwards for character.
strspn	**string**(3C)	Length of initial string containing set of characters.
strtok	**string**(3C)	Search string for token separated by any of a set of characters.

3.1.8 Character Manipulation

The following functions and declarations are used for testing and translating ASCII characters. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The declarations associated with these functions should be included in the program being compiled. This is accomplished by including the line:

#include <ctype.h>

near the beginning of the file being compiled.

3.1.9 Character Testing Functions

These functions can be used to identify characters as uppercase or lowercase letters, digits, punctuation, etc.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
isalnum	**ctype**(3C)	Is character alphanumeric?
isalpha	**ctype**(3C)	Is character alphabetic?
isascii	**ctype**(3C)	Is integer ASCII character?
iscntrl	**ctype**(3C)	Is character a control character?
isdigit	**ctype**(3C)	Is character a digit?
isgraph	**ctype**(3C)	Is character a printable character?
islower	**ctype**(3C)	Is character a lowercase letter?

FUNCTION	REFERENCE	BRIEF DESCRIPTION
isprint	**ctype**(3C)	Is character a printing character including space?
ispunct	**ctype**(3C)	Is character a punctuation character?
isspace	**ctype**(3C)	Is character a white space character?
isupper	**ctype**(3C)	Is character an uppercase letter?
isxdigit	**ctype**(3C)	Is character a hex digit?

3.1.10 Character Translation Functions

These functions provide translation of uppercase to lowercase, lowercase to uppercase, and integer to ASCII.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
toascii	**conv**(3C)	Convert integer to ASCII character.
tolower	**conv**(3C)	Convert character to lowercase.
toupper	**conv**(3C)	Convert character to uppercase.

3.1.11 Time Functions

These functions are used for accessing and reformatting the systems idea of the current date and time. These functions are located and loaded automatically during the compiling of a C language program. No command line request is needed since these functions are part of the C library.

The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

#include <time.h>

near the beginning of any file using the time functions. These functions (except **tzset**) convert a time such as returned by **time**(2).

FUNCTION	REFERENCE	BRIEF DESCRIPTION
asctime	**ctime**(3C)	Return string representation of date and time.
ctime	**ctime**(3C)	Return string representation of date and time, given integer form.
gmtime	**ctime**(3C)	Return Greenwich Mean Time.
localtime	**ctime**(3C)	Return local time.
tzset	**ctime**(3C)	Set time zone field from environment variable.

3.1.12 Miscellaneous Functions

These functions support a wide variety of operations. Some of these are numerical conversion, password file and group file access, memory allocation, random number generation, and table management. These functions are automatically located and included in a program being compiled. No command line request is needed since these functions are part of the C library.

Some of these functions require declarations to be included. These are described following the descriptions of the functions.

3.1.13 Numerical Conversion

The following functions perform numerical conversion.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
a64l	**a64l**(3C)	Convert string to base 64 ASCII.
atof	**atof**(3C)	Convert string to floating.
atoi	**atof**(3C)	Convert string to integer.
atol	**atof**(3C)	Convert string to long.
frexp	**frexp**(3C)	Split floating into mantissa and exponent.
l3tol	**l3tol**(3C)	Convert 3-byte integer to long.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ltol3	**l3tol**(3C)	Convert long to 3-byte integer.
ldexp	**frexp**(3C)	Combine mantissa and exponent.
l64a	**a64l**(3C)	Convert base 64 ASCII to string.
modf	**frexp**(3C)	Split mantissa into integer and fraction.

3.1.14 DES Algorithm Access

The following functions allow access to the Data Encryption Standard (DES) algorithm used on the UNIX operating system. The DES algorithm is implemented with variations to frustrate use of hardware implementations of the DES for key search.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
crypt	**crypt**(3C)	Encode string.
encrypt	**crypt**(3C)	Encode/decode string of 0s and 1s.
setkey	**crypt**(3C)	Initialize for subsequent use of **encrypt**.

3.1.15 Group File Access

The following functions are used to obtain entries from the group file. Declarations for these functions must be included in the program being compiled with the line:

#include <grp.h>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
endgrent	**getgrent**(3C)	Close group file being processed.
getgrent	**getgrent**(3C)	Get next group file entry.
getgrgid	**getgrent**(3C)	Return next group with matching gid.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
getgrnam	**getgrent**(3C)	Return next group with matching name.
setgrent	**getgrent**(3C)	Rewind group file being processed.
fgetgrent	**getgrent**(3C)	Get next group file entry from a specified file.

3.1.16 Password File Access

These functions are used to search and access information stored in the password file (/etc/passwd). Some functions require declarations that can be included in the program being compiled by adding the line:

#include <pwd.h>

FUNCTION	REFERENCE	BRIEF DESCRIPTION
endpwent	**getpwent**(3C)	Close password file being processed.
getpw	**getpw**(3C)	Search password file for uid.
getpwent	**getpwent**(3C)	Get next password file entry.
getpwnam	**getpwent**(3C)	Return next entry with matching name.
getpwuid	**getpwent**(3C)	Return next entry with matching uid.
putpwent	**putpwent**(3C)	Write entry on stream.
setpwent	**getpwent**(3C)	Rewind password file being accessed.
fgetpwent	**getpwent**(3C)	Get next password file entry from a specified file.

3.1.17 Parameter Access

The following functions provide access to several different types of paramenters. None require any declarations.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
getopt	**getopt**(3C)	Get next option from option list.
getcwd	**getcwd**(3C)	Return string representation of current working directory.
getenv	**getenv**(3C)	Return string value associated with environment variable.
getpass	**getpass**(3C)	Read string from terminal without echoing.
putenv	**putenv**(3C)	Change or add value of an environment variable.

3.1.18 Hash Table Management

The following functions are used to manage hash search tables. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

#include <search.h>

near the beginning of any file using the search functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
hcreate	**hsearch**(3C)	Create hash table.
hdestroy	**hsearch**(3C)	Destroy hash table.
hsearch	**hsearch**(3C)	Search hash table for entry.

3.1.19 Binary Tree Management

The following functions are used to manage a binary tree. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

#include <search.h>

near the beginning of any file using the search functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
tdelete	**tsearch**(3C)	Deletes nodes from binary tree.
tfind	**tsearch**(3C)	Find element in binary tree.
tsearch	**tsearch**(3C)	Look for and add element to binary tree.
twalk	**tsearch**(3C)	Walk binary tree.

3.1.20 Table Management

The following functions are used to manage a table. Since none of these functions allocate storage, sufficient memory must be allocated before using these functions. The header file associated with these functions should be included in the program being compiled. This is accomplished by including the line:

#include <search.h>

near the beginning of any file using the search functions.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
bsearch	**bsearch**(3C)	Search table using binary search.
lfind	**lsearch**(3C)	Find element in library tree.
lsearch	**lsearch**(3C)	Look for and add element in binary tree.
qsort	**qsort**(3C)	Sort table using quick-sort algorithm.

3.1.21 Memory Allocation

The following functions provide a means by which memory can be dynamically allocated or freed.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
calloc	**malloc**(3C)	Allocate zeroed storage.
free	**malloc**(3C)	Free previously allocated storage.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
malloc	**malloc**(3C)	Allocate storage.
realloc	**malloc**(3C)	Change size of allocated storage.

The following is another set of memory allocation functions available.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
calloc	**malloc**(3X)	Allocate zeroed storage.
free	**malloc**(3X)	Free previously allocated storage.
malloc	**malloc**(3X)	Allocate storage.
mallopt	**malloc**(3X)	Control allocation algorithm.
mallinfo	**malloc**(3X)	Space usage.
realoc	**malloc**(3X)	Change size of allocated storage.

3.1.22 Pseudorandom Number Generation

The following functions are used to generate pseudorandom numbers. The functions that end with **48** are a family of interfaces to a pseudorandom number generator based upon the linear congruent algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions provide an interface to a multiplicative congruential random number generator with period of 232.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
drand48	**drand48**(3C)	Random double over the interval [0 to 1).
lcong48	**drand48**(3C)	Set parameters for **drand48**, **lrand48**, and **mrand48**.
lrand48	**drand48**(3C)	Random long over the interval [0 to 2^{31}).
mrand48	**drand48**(3C)	Random long over the interval [-2^{31} to 2^{31}).
rand	**rand**(3C)	Random integer over the interval [0 to 32767).

FUNCTION	REFERENCE	BRIEF DESCRIPTION
seed48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48**.
srand	**rand**(3C)	Seed the generator for **rand**.
srand48	**drand48**(3C)	Seed the generator for **drand48**, **lrand48**, and **mrand48** using a long.

3.1.23 Signal Handling Functions

The functions **gsignal** and **ssignal** implement a software facility similar to **signal**(2) in the *Sys5 UNIX Programmer Reference Manual*. This facility enables users to indicate the disposition of error conditions and allows users to handle signals for their own purposes. The declarations associated with these functions can be included in the program being complied by the line

#include <signal.h>

These declarations define ASCII names for the 15 software signals.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
gsignal	**ssignal**(3C)	Send a software signal.
ssignal	**ssignal**(3C)	Arrange for handling of software signals.

3.1.24 Miscellaneous

The following functions do not fall into any previously described category.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
abort	**abort**(3C)	Cause an IOT signal to be sent to the process.
abs	**abs**(3C)	Return the absolute integer value.
ecvt	**ecvt**(3C)	Convert double to string.
fcvt	**ecvt**(3C)	Convert double to string using Fortran Format.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
gcvt	**ecvt**(3C)	Convert double to string using Fortran F or E format.
isatty	**ttyname**(3C)	Test whether integer file descriptor is associated with a terminal.
mktemp	**mktemp**(3C)	Create file name using template.
monitor	**monitor**(3C)	Cause process to record a histogram of program counter location.
swab	**swab**(3C)	Swap and copy bytes.
ttyname	**ttyname**(3C)	Return pathname of terminal associated with integer file descriptor.

4. OBJECT AND MATH LIBRARIES

This chapter describes the Object and Math Libraries that are supported on the UNIX operating system. A library is a collection of related functions and/or declarations that simplify programming effort. All of the functions described are also described in Part 3 of the *UNIX System Programmer Reference Manual*. Most of the declarations described are in Part 5 of the *UNIX System Programmer Reference Manual*. The three main libraries on the UNIX system are:

C library This is the basic library for C language programs. The C library is composed of functions and declarations used for file access, string testing and manipulation, character testing and manipulation, memory allocation, and other functions. This library is described in Chapter 3.

Object file This library provides functions for the access and manipulation of object files. This library is described later in this chapter.

Math library This library provides exponential, bessel functions, logarithmic, hyperbolic, and trigonometric functions. This library is also described later in this chapter.

4.1 THE OBJECT FILE LIBRARY

The object file library provides functions for the access and manipulation of object files. Some functions locate portions of an object file such as the symbol table, the file header, sections, and line number entries associated with a function. Other functions read these types of entries into memory. For a description of the format of an object file, see "The Common Object File Format" in the *UNIX System Support Tools Guide*.

This library consists of several portions. The functions reside in */usr/lib/libld.a* and are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

cc file **-lld**

which causes the link editor to search the object file library. The argument **-lld** must appear after all files that reference functions in *libld.a*.

In addition, various header files must be included. This is accomplished by including the line:

```
#include <stdio.h>
#include <a.out.h>
#include <ldfcn.h>
```

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ldaclose	**ldclose**(3X)	Close object file being processed.
ldahread	**ldahread**(3X)	Read archive header.
ldaopen	**ldopen**(3X)	Open object file for reading.
ldclose	**ldclose**(3X)	Close object file being processed.
ldfhread	**ldfhread**(3X)	Read file header of object file being processed.
ldgetname	**ldgetname**(3X)	Retrieve the name of an object file symbol table entry.
ldlinit	**ldlread**(3X)	Prepare object file for reading line number entries via **ldlitem**.
ldlitem	**ldlread**(3X)	Read line number entry from object file after **ldlinit**.
ldlread	**ldlread**(3X)	Read line number entry from object file.
ldlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed.
ldnlseek	**ldlseek**(3X)	Seeks to the line number entries of the object file being processed given the name of a section.
ldnrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed given the name of a section.
ldnshread	**ldshread**(3X)	Read section header of the named section of the object file being processed.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ldnsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed given the name of a section.
ldohseek	**ldohseek**(3X)	Seeks to the optional file header of the object file being processed.
ldopen	**ldopen**(3X)	Open object file for reading.
ldrseek	**ldrseek**(3X)	Seeks to the relocation entries of the object file being processed.
ldshread	**ldshread**(3X)	Read section header of an object file being processed.
ldsseek	**ldsseek**(3X)	Seeks to the section of the object file being processed.
ldtbindex	**ldtbindex**(3X)	Returns the long index of the symbol table entry at the current position of the object file being processed.
ldtbread	**ldtbread**(3X)	Reads a specific symbol table entry of the object file being processed.
ldtbseek	**ldtbseek**(3X)	Seeks to the symbol table of the object file being processed.
sgetl	**sputl**(3X)	Access long integer data in a machine independant format.
sputl	**sputl**(3X)	Translate a long integer into a machine independant format.

4.1.1 Common Object File Interface Macros (ldfcn.h)

The interface between the calling program and the object file access routines is based on the defined type **LDFILE** which is defined in the header file **ldfcn.h** (see **ldfcn**(4)). The primary purpose of this structure is to

provide uniform access to both simple object files and to object files that are members of an archive file.

The function **ldopen**(3X) allocates and initializes the **LDFILE** structure and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure may be accessed individually through the following macros: the **type** macro returns the magic number of the file, which is used to distinguish between archive files and simple object files. The *OPTR* macro returns the file pointer which was opened by **ldopen**(3X) and is used by the input/output functions of the C library. The **OFFSET** macro returns the file address of the beginning of the object file. This value is non-zero only if the object file is a member of the archive file. The **HEADER** macro accesses the file header structure of the object file.

Additional macros are provided to access an object file. These macros parallel the input/output functions in the C library; each macro translates a reference to an **LDFILE** structure into a reference to its file descriptor field. The available macros are described in **ldfcn**(4) in the *UNIX System Reference Manual.*

4.2 THE MATH LIBRARY

The math library consists of functions and a header file. The functions are located and loaded during the compiling of a C language program by a command line request. The form of this request is:

cc file **-lm**

which causes the link editor to search the math library. In addition to the request to load the functions, the header file of the math library should be included in the program being compiled. This is accomplished by including the line:

#include <math.h>

near the beginning of the (first) file being compiled.

The functions are grouped into the following categories:

- Trigonometric functions
- Bessel functions
- Hyperbolic functions
- Miscellaneous functions.

4.2.1 Trigonometric Functions

These functions are used to compute angles (in radian measure), sines, cosines, and tangents. All of these values are expressed in double precision,

FUNCTION	REFERENCE	BRIEF DESCRIPTION
acos	**trig**(3M)	Return arc cosine.
asin	**trig**(3M)	Return arc sine.
atan	**trig**(3M)	Return arc tangent.
atan2	**trig**(3M)	Return arc tangent of a ratio.
cos	**trig**(3M)	Return cosine.
sin	**trig**(3M)	Return sine.
tan	**trig**(3M)	Return tangent.

4.2.2 Bessel Functions

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are **j0**, **j1**, **jn**, **y0**, **y1**, and **yn**. The functions are located in section **bessel**(3M).

4.2.3 Hyperbolic Functions

These functions are used to compute the hyperbolic sine, cosine, and tangent for real values.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
cosh	**sinh**(3M)	Return hyperbolic cosine.
sinh	**sinh**(3M)	Return hyperbolic sine.
tanh	**sinh**(3M)	Return hyperbolic tangent.

4.2.4 Miscellaneous Functions

These functions cover a wide variety of operations, such as natural logarithm, exponential, and absolute value. In addition, several are provided to truncate the integer portion of double precision numbers.

FUNCTION	REFERENCE	BRIEF DESCRIPTION
ceil	**floor**(3M)	Returns the smallest integer not less than a given value.
exp	**exp**(3M)	Returns the exponential function of a given value.
fabs	**floor**(3M)	Returns the absolute value of a given value.
floor	**floor**(3M)	Returns the largest integer not greater than a given value.
fmod	**floor**(3M)	Returns the remainder produced by the division of two given values.
gamma	**gamma**(3M)	Returns the natural log of the absolute value of the result of applying the gamma function to a given value.
hypot	**hypot**(3M)	Return the square root of the sum of the squares of two numbers.
log	**exp**(3M)	Returns the natural logarithm of a given value.
log10	**exp**(3M)	Returns the lorarithm base ten of a given value.
matherr	**matherr**(3M)	Error-handling function.
pow	**exp**(3M)	Returns the result of a given value raised to another given value.
sqrt	**exp**(3M)	Returns the square root of a given value.

5. COMPILER AND C LANGUAGE

This chapter describe the UNIX System's C compiler, **cc**, and the C programming language that the compiler translates. The compiler is part of the UNIX System Software Generation System (SGS).

The SGS is a package of tools used to create and test programs for UNIX Systems. These tools allow high-level program coding and source-level testing of code. The C language is implemented for high-level programming; it contains many control and structuring facilities that greatly simplify the task of algorithm construction. Within the SGS, a C compiler converts C programs into assembly language programs that are ultimately translated into object files by the assembler, **as**. The link editor, **ld**, collects and merges object files into executable load modules. Each of these tools preserves all symbolic information necessary for meaningful symbolic testing at C-language source level. In addition, a utility package aids in testing and debugging.

The current manual page for the C compiler can be obtained with the SGS command:

man cc

5.1 USE OF THE COMPILER

The main command of the SGS is **cc**; it operates much like the UNIX system **cc** command. To use the compiler, first creat a file (typically by using the UNIX system text editor) containing C source code. The name of the file created must have a special format; the last two characters of the file name must be **.c** as in *file1.c*.

Next, enter the SGS command

cc *options file.c*

to invoke the compiler on the C source file *file.c* with the appropriate *options* selected. The compilation process creates an absolute binary file named **a.out** that reflects the contents of *file.c* and any referenced library routines. The resulting binary file, **a.out**, can then be executed on the target system.

Options can control the steps in the compilation process. When none of the controlling options are used, and only one file is named, **cc** automatically calls the assembler, **as**, and the link editor, **ld**, thus resulting in an executable file, named **a.out**. If more than one file is named in a command,

cc *file1.c file2.c file3.c*

then the output will be placed on files *file1.o*, *file2.o*, and *file3.o*. These files can then be linked and executed through the **ld** command.

The **cc** compiler also accepts input file names with the last two characters **.s**. The **.s** signifies a source file in assembly language. The **cc** compiler passes this type of file directly to **as**, which assembles the file and places the output on a file of the same name with **.o** substituted for **.s**.

Cc is based on a portable C compiler and translates C source files into assembly code. Whenever the command **cc** is used, the standard C preprocessor (which resides on the file **/lib/cpp**) is called. The preprocessor performs file inclusion and macro substitution. The preprocessor is always invoked by **cc** and need not be called directly by the programmer. Then, unless the appropriate flags are set, **cc** calls the assembler and the link editor to produce an executable file.

5.2 COMPILER OPTIONS

All options recognized by the **cc** command are listed below:

Option	Argument	Description
-c	none	Suppress the link-editing phase of compilation and force an object file to be produced even if only one file is compiled.
-g	none	Produce symbolic debugging information.
-p	none	Reserved for invoking a profiler.
-D	*identifier[=constant]*	Define the external symbol *identifier* to the preprocessor, and give it the value *constant* (if specified).
-E	none	Same as the **-P** option except output is directed to the standard output.

Option	Argument	Description
-I	*directory*	Change the algorithm that searches for **#include** files whose names do not begin with / to look in the named *directory* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" are searched for first in the directory of the file being compiled, then in directories named by the **-I** options, and last in directories on the standard list. For **#include** files whose names are enclosed in <>, the directory of the *file* argument is not searched.
-O	none	Invoke an object code optimizer.
-P	none	Suppress compilation and loading; i.e., invoke only the preprocessor and leave out the output on corresponding files suffixed .i.
-U	*identifier*	Undefine the named *identifier* to the preprocessor.
-V	none	Print the version of the assembler that is invoked.
-W	*c,arg1[,arg2...]*	Pass along the argument(s) *argi* to pass *c*, where *c* is one of **[p012al]**, indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively.

This part provides additional information for those options not completely described above.

By using appropriate options, compilation can be terminated early to produce one of several intermediate translations such as relocatable object files (**-c** option), assembly source expansions for C code (**-S** option), or the output of the preprocessor (**-P** option). In general, the intermediate files may be saved and later resubmitted to the **cc** command, with other files or libraries included as necessary.

When compiling C source files, the most common practice is to use the **-c** option to save relocatable files. Subsequent changes to one file do not then require that the others be recompiled. A separate call to **cc** without the **-c** option then creates the linked executable **a.out** file. A relocatable object file created under the **-c** option is named by adding a **.o** suffix to the source file name.

The **-W** option provides the mechanism to specify options for each step that is normally invoked from the **cc** command line. These steps are preprocessing, the first pass of the compiler, the second pass of the compiler, optimization, assembly, and link editing. At this time, only assembler and link editor options can be used with the **-W** option. The most common example of use of the **-W** option is "**-W***a,-m*", which passes the **-m** option to the assembler. Specifying "**-wl,-m**" passes the **-m** option to the link editor.

When the **-P** option is used, the compilation process stops after only preprocessing, with output left on *file.i*. This file will be unsuitable for subsequent processing by **cc**.

The **-O** option decreases the size and increases the execution speed of programs by moving, merging, and deleting code. However, line numbers used for symbolic debugging may be transposed when the optimizer is used.

The **-g** option produces information for a symbolic debugger. The SGS currently supports the SDB symbolic debugger.

6. A C PROGRAM CHECKER (lint)

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. 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 legal. The **lint** program accepts multiple input files and library specifications and checks them for consistency.

6.0.1 Usage

The **lint** command has the form:

lint [options] files ... library-descriptors ...

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are:

-a Suppress messages about assignments of long values to variables that are not long.

-b Suppress messages about break statements that cannot be reached.

-c Only check for intra-file bugs; leave external information in files suffixed with **.ln**.

-h Do not apply heuristics (which attempt to detect bugs, improve style, and reduce waste).

-n Do not check for compatibility with either the standard or the portable **lint** library.

-0 *name* Create a lint library from input files named **llib-l***name***.ln**.

-p Attempt to check portability to other dialects of C language.

-u Suppress messages about function and external variables used and not defined or defined and not used.

-v Suppress messages about unused arguments in functions.

-x Do not report variables referred to by external declarations but never used.

When more than one option is used, they should be combined into a single argument, such as, **-ab** or **-xha**.

The names of files that contain C language programs should end with the suffix **.c** which is mandatory or **lint** and the C compiler.

The **lint** program accepts certain arguments, such as:

–ly

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment:

/* 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 comments can be used to specify features of the library functions.

The **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 do not result in messages. The **lint** program does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

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 language program is run. When the **–p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **–n** option can be used to suppress all library checking.

6.1 TYPES OF MESSAGES

The following paragraphs describe the major categories of messages printed by **lint**.

6.1.1 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 types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

The **lint** program prints messages 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 double 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 and can be discovered by using the **–x** option with the **lint** command.

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 messages 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.

Messages about unused arguments can be suppressed for one function by adding the comment:

/* ARGSUSED */

to the program before the function. This has the effect of the **-v** option for only one function. Also, the comment:

/* VARARGS */

can be used to suppress messages about variable number of arguments in calls to a function. The comment should be added before 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 with a digit giving the number of arguments which should be checked. For example:

/* VARARGS2 */

will cause only the first two arguments to be checked.

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. Conversely, many functions and variables defined elsewhere may be used. The **–u** option may be used to suppress the spurious messages which might otherwise appear.

6.1.2 Set/Used Information

The **lint** program attempts to detect cases where a variable is used before it is set. The **lint** program 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 print messages about some programs which are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

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.

6.1.3 Flow of Control

The **lint** program attempts to detect unreachable portions of the programs which it processes. It will print messages 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 and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. The **lint** program also prints messages about loops which cannot be entered at the top. Some valid programs may have such loops which are considered to be bad style at best and bugs at worst.

The **lint** program has no way of detecting functions which are called and never returned. Thus, a call to **exit** may cause an unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program cannot be reached but it is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached.

The **lint** program will not print a message about unreachable **break** statements. Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements. The **–O** option in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached 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.

6.1.4 Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function "values" that have never been returned. The **lint** program 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; the **lint** program 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 message 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.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, 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.

6.1.5 Type Checking

The **lint** program enforces the type checking rules of C language 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

- 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 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.

If it is desired to turn off strict type checking for an expression, the comment

/* NOSTRICT */

should be added to the program immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

6.1.6 Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. The **lint** program will print a message as a result of detecting this. 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 print messages 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 messages; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

6.1.7 Nonportable Character Use

On some systems, characters are signed quantities with a range from –128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar()) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message "nonportable character comparison".

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**

6.1.8 Assignments of "longs" to "ints"

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. 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**, which are truncated. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the **–a** option.

6.1.9 Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. The messages hopefully encourage better code quality, clearer style, and may even point out bugs. The **–h** option is used to supress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message "null effect" from **lint**. The following program fragment:

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. The **lint** program will print the message "degenerate unsigned comparison" in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message "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 statement

```
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** option has not been used, **lint** prints messages 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 to be bad style, usually unnecessary, and frequently a bug.

6.1.10 Old Syntax

Several forms of older syntax are now illegal. 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 (e.g., +=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages 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, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

and the compiler must read past *x* in order to determine the correct meaning. 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.

6.1.11 Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. The **lint** program 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.

6.1.12 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 language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. 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.

The **lint** program checks for the important special case where a simple scalar variable is affected. For example, the statement

```
a[i] = b[i++];
```

will cause **lint** to print the message

warning: i evaluation order undefined

in order to call attention to this condition.

7. sdb

The symbolic debugger **sdb**(1) is not implemented by Plexus.

If you have purchased a version of CDB separately you may want to include that information in this section.

8. FORTRAN UNIX SYSTEM COMMANDS

A UNIX system Fortran 77 user should be familiar with the following commands:

- **f77** [options] files - This command invokes the UNIX system Fortran 77 compiler
- **ratfor** [options] [files] - This command invokes the Ratfor preprocessor
- **efl** [options] [files] - This command compiles a program written in Extended Fortran Language (EFL) into clean Fortran
- **asa** [files] - This command interprets the output of Fortran programs that utilize ASA carriage control characters
- **fsplit** options files - This command splits the named file(s) into separate files, with one procedure per file.

For more information about the above commands, see the *UNIX System User Reference Manual.*

9. FORTRAN 77

This chapter describes the compiler and run-time system for Fortran 77 as implemented on the UNIX system. This chapter also describes the interfaces between procedures and the file formats assumed by the I/O system.

9.1 USAGE

The command to run the compiler is

f77 options file

The **f77**(1) command is a general purpose command for compiling and loading Fortran and Fortran-related files into an executable module. EFL (compiler) and Ratfor (preprocessor) source files will be translated into Fortran before being presented to the Fortran compiler. The **f77** command invokes the C compiler to translate C source files and invokes the assembler to translate assembler source files. Object files will be link edited. [The **f77**(1) and **cc**(1) commands have slightly different link editing sequences. Fortran programs need two extra libraries (libI77.a, libF77.a) and an additional startup routine.] The following file name suffixes are understood:

.f	Fortran source file
.e	EFL source file
.r	Ratfor source file
.c	C language source file
.s	Assembler source file
.o	Object file.

9.2 LANGUAGE EXTENSIONS

Fortran 77 includes almost all of Fortran 66 as a subset. The most important additions are a character string data type, file-oriented input/output statements, and random access I/O. Also, the language has been cleaned up considerably.

In addition to implementing the language specified in the Fortran 77 American National Standard, this compiler implements a few extensions. Most are useful additions to the language. The remainder are extensions to make it easier to communicate with C language procedures or to permit compilation of old (1966 Standard Fortran) programs.

9.2.1 Double Complex Data Type

The data type **double complex** is added. Each datum is represented by a pair of double-precision real variables. A double complex version of every **complex** built-in function is provided.

9.2.2 Internal Files

The Fortran 77 American National Standard introduces *internal files* (memory arrays) but restricts their use to formatted sequential I/O statements. This I/O system also permits internal files to be used in direct and unformatted reads and writes.

9.2.3 Implicit Undefined Statement

Fortran has a rule that the type of a variable that does not appear in a type statement is **integer** if its first letter is *i*, *j*, *k*, *l*, *m* or *n*. Otherwise, it is **real**. Fortran 77 has an **implicit** statement for overriding this rule. An additional type statement, *undefined*, is permitted. The statement

```
implicit undefined(a-z)
```

turns off the automatic data typing mechanism, and the compiler will issue a diagnostic for each variable that is used but does not appear in a type statement. Specifying the **-u** compiler option is equivalent to beginning each procedure with this statement.

9.2.4 Recursion

Procedures may call themselves directly or through a chain of other procedures.

9.2.5 Automatic Storage

Two new keywords recognized are **static** and **automatic.** These keywords may appear as "types" in type statements and in **implicit** statements. Local variables are static by default; there is exactly one copy of the datum, and its value is retained between calls. There is one copy of each variable declared **automatic** for each invocation of the procedure. Automatic variables may not appear in **equivalence, data,** or **save** statements.

9.2.6 Variable Length Input Lines

The Fortran 77 American National Standard expects input to the compiler to be in a 72-column format: except in comment lines, the first five characters are the statement number, the next is the continuation character, and the next 66 are the body of the line. (If there are fewer than 72 characters on a line, the compiler pads it with blanks; characters after the first 72 are ignored.) In order to make it easier to type Fortran programs, this compiler also accepts input in variable length lines. An ampersand (**&**) in the first position of a line indicates a continuation line; the remaining characters form

the body of the line. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line; the remaining characters form the body of the line. A tab elsewhere on the line is treated as another kind of blank by the compiler.

In the Fortran 77 Standard, there are only 26 letters

— Fortran is a one-case language. Consistent with ordinary system usage, the new compiler expects lowercase input. By default, the compiler converts all uppercase characters to lowercase except those inside character constants. However, if the **-U** compiler option is specified, uppercase letters are not transformed. In this mode, it is possible to specify external names with uppercase letters in them and to have distinct variables differing only in case. Regardless of the setting of the option, keywords will only be recognized in lowercase.

9.2.7 Include Statement

The statement

include "stuff"

is replaced by the contents of the file *stuff*. Includes may be nested to a reasonable depth, currently ten.

9.2.8 Binary Initialization Constants

A **logical, real,** or **integer** variable may be initialized in a **data** statement by a binary constant, denoted by a letter followed by a quoted string. If the letter is *b*, the string is binary, and only zeroes and ones are permitted. If the letter is *o*, the string is octal with digits *zero* through *seven*. If the letter is **z** or **x**, the string is hexadecimal with digits *zero* through *nine*, *a* through *f*. Thus, the statements

```
integer a(3)
data a/b'1010',o'12',z'a'/
```

initialize all three elements of **a** to ten.

9.2.9 Character Strings

For compatibility with C language usage, the following backslash escapes are recognized:

\n	New-line
\t	Tab
\b	Backspace
\f	Form feed

\0	Null
\'	Apostrophe (does not terminate a string)
\"	Quotation mark (does not terminate a string)
\\	\
\x	Where **x** is any other character.

Fortran 77 only has one quoting character — the apostrophe ('). This compiler and I/O system recognize both the apostrophe and the double quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escapes.

Every unequivalenced scalar local character variable and every character string constant is aligned on an **integer** word boundary. Each character string constant appearing outside a **data** statement is followed by a null character to ease communication with C language routines.

9.2.10 Hollerith

Fortran 77 does not have the old Hollerith **(nh)** notation though the new Standard recommends implementing the old Hollerith feature in order to improve compatibility with old programs. In this compiler, Hollerith data may be used in place of character string constants and may also be used to initialize non character variables in data statements.

9.2.11 Equivalence Statements

This compiler permits single subscripts in **equivalence** statements under the interpretation that all missing subscripts are equal to 1. A warning message is printed for each such incomplete subscript.

9.2.12 One-Trip DO Loops

The Fortran 77 American National Standard requires that the range of a **do** loop not be performed if the initial value is already past the limit value, as in

```
do 10 i = 2, 1
```

The 1966 Standard stated that the effect of such a statement was undefined, but it was common practice that the range of a **do** loop would be performed at least once. In order to accommodate old programs though they were in violation of the 1966 Standard, the **-onetrip** compiler option causes nonstandard loops to be generated.

9.2.13 Commas in Formatted Input

The I/O system attempts to be more lenient than the Fortran 77 American National Standard when it seems worthwhile. When doing a formatted read of non-character variables, commas may be used as value separators in the input record overriding the field lengths given in the format statement. Thus, the format

(i10, f20.10, i4)

will read the record

-345,.05e-3,12

correctly.

9.2.14 Short Integers

On machines that support half word integers, the compiler accepts declarations of type **integer*2.** (Ordinary integers follow the Fortran rules about occupying the same space as a REAL variable; they are assumed to be of C language type **long int;** half word integers are of C language type **short int.**) An expression involving only objects of type **integer*2** is of that type. Generic functions return short or long integers depending on the actual types of their arguments. If a procedure is compiled using the **-I2** flag, all small integer constants will be of type **integer*2.** If the precision of an integer-valued intrinsic function is not determined by the generic function rules, one will be chosen that returns the prevailing length (**integer*2** when the **-I2** command flag is in effect). When the **-I2** option is in effect, all quantities of type **logical** will be short. Note that these short integer and logical quantities do not obey the standard rules for storage association.

9.2.15 Additional Intrinsic Functions

This compiler supports all of the intrinsic functions specified in the Fortran 77 Standard. In addition, there are functions for performing bitwise Boolean operations (**or, and, xor,** and **not**) and for accessing the command arguments (**getarg** and **iargc**).

The following lists the Fortran intrinsic function library plus some additional functions. These functions are automatically available to the Fortran programmer and require no special invocation of the compiler. The asterisk (*) beside some of the commands indicate they are not part of standard F77. In parenthesis beside each function description listed below is the location for the command in the *Sys5 UNIX Programmer Reference Manual*. These functions are as follows:

abort*	Terminate program (ABORT(3F))
abs	Absolute value (MAX(3F)

acos	Arccosine (ACOS(3F))
aimag	Imaginary part of complex argument (AIMAG(3F))
aint	Integer part (AINT(3F))
alog	Natural logarithm (LOG(3F))
alog10	Common logarithm (ALOG10(3F))
amax0	Maximum value (MAX(3F))
amax1	Maximum value (MAX(3F))
amin0	Minimum value (MIN(3F))
amin1	Minimum value (MIN(3F))
amod	Remaindering (MOD(3F))
and*	Bitwise boolean (BOOL(3F))
anint	Nearest integer (ROUND(3F))
asin	Arcsine (ASIN(3F))
atan	Arctangent (ATAN(3F))
atan2	Arctangent (ATAN2(3F))
cabs	Complex absolute value (ABS(3F))
ccos	Complex cosine (COS(3F))
cexp	Complex exponential (EXP(3F))
char	Explicit type conversion (FTYPE(3F))
clog	Complex natural logarithm (LOG(3F))
cmplx	Explicit type conversion (FTYPE(3F))
conjg	Complex conjugate (CONJG(3F))
cos	Cosine (COS(3F))
cosh	Hyperbolic cosine (COSH(3F))
csin	Complex sine (SIN(3F))
csqrt	Complex square root (SQRT(3F))
dabs	Absolute value (ABS(3F))
dacos	Arccosine (ACOS(3F))
dasin	Arcsine (ASIN(3F))
datan	Arctangent (ATAN(3F))
datan2	Double precision arctangent (ATAN2(3F))
dble	Explicit type conversion (FTYPE(3F))
dcmplx*	Explicit type conversion (FTYPE(3F))
dconjg*	Complex conjugate (CONJG(3F))
dcos	Cosine (DCOS(3F))
dcosh	Hyperbolic cosine (COSH(3F))
ddim	Positive difference (DIM(3F))
dexp	Exponential (EXP(3F))
dim	Positive difference (DIM(3F))
dimag*	Imaginary part of complex argument ((AIMAG(3F))
dint	Integer part (AINT(3F))
dlog	Natural logarithm (LOG(3F))
dlog10	Common logarithm (LOG10(3F))
dmax1	Maximum value (MAX(3F))

dmin1	Minimum value (MIN(3F))
dmod	Remaindering (DMOD(3F))
dnint	Nearest integer (ROUND(3F))
dprod	Double precision product (DPROD(3F))
dsign	Transfer of sign (SIGN(3F))
dsin	Sine (SIN(3F))
dsinh	Hyperbolic sine (SINH(3F))
dsqrt	Square root (SQRT(3F))
dtan	Tangent (TAN(3F))
dtanh	Hyperbolic tangent (TANH(3F))
exp	Exponential (EXP(3F))
float	Explicit type conversion (FTYPE(3F))
getarg*	Return command-line argument (GETARG(3F))
getenv*	Return environment variable (GETENV(3F))
iabs	Absolute value (ABS(3F))
iargc	Return number of arguments (IARGC(3F))
ichar	Explicit type conversion (FTYPE(3F))
idim	Positive difference (DIM(3F))
idint	Explicit type conversion (FTYPE(3F))
idnint	Nearest integer (ROUND(3F))
ifix	Explicit type conversion (FTYPE(3F))
index	Return location of substring (INDEX(3F))
int	Explicit type conversion (FTYPE(3F))
irand*	Random number generator
isign	Transfer of sign (SIGN(3F))
len	Return location of string (LEN(3F))
lge	String comparison (STRCMP(3F))
lgt	String comparison (STRCMP(3F))
lle	String comparison (STRCMP(3F))
llt	String comparison (STRCMP(3F))
log	Natural logarithm (LOG(3F))
log10	Common logarithm (LOG10(3F))
lshift*	Bitwise boolean (BOOL(3F))
max	Maximum value (MAX(3F))
max0	Maximum value (MAX(3F))
max1	Maximum value (MAX(3F))
mclock*	Return Fortran time accounting (MCLOCK(3F))
min	Minimum value (MIN(3F))
min0	Minimum value (MIN(3F))
min1	Minimum value (MIN(3F))
mod	Remaindering (MOD(3F))
nint	Nearest integer (BOOL(3F))
not*	Bitwise boolean (BOOL(3F))
or*	Bitwise boolean (BOOL(3F))

rand*	Random number generator (RAND(3F))
real	Explicit type conversion (FTYPE(3F))
rshift*	Bitwise boolean (BOOL(3F))
sign	Transfer of sign (SIGN(3F))
signal*	Specify action on receipt of system signal (SIGNAL(3F))
sin	Sine (SINE(3F))
sinh	Hyperbolic sine (SINH(3F))
sngl	Explicit type conversion (FTYPE(3F))
sqrt	Square root (SQRT(3F))
srand*	Random number generator (RAND(3F))
system*	Issue a shell command (SYSTEM(3F))
tan	Tangent (TAN(3F))
tanh	Hyperbolic tangent (TANH(3F))
xor*	Bitwise boolean (BOOL(3F))
zabs*	Complex absolute value (ABS(3F)).

For more information on the Fortran intrinsic function commands, see the *Sys5 UNIX Programmer Reference Manual*.

9.3 VIOLATIONS OF THE STANDARD

The following paragraphs describe only three known ways in which the UNIX system implementation of Fortran 77 violates the new American National Standard.

9.3.1 Double Precision Alignment

The Fortran 77 American National Standard permits **common** or **equivalence** statements to force a double precision quantity onto an odd word boundary, as in the following example:

```
real a(4)
double precision b,c
equivalence (a(1),b), (a(4),c)
```

Some machines require that double precision quantities be on double word boundaries; other machines run inefficiently if this alignment rule is not observed. It is possible to tell which equivalenced and common variables suffer from a forced odd alignment, but every double-precision argument would have to be assumed on a bad boundary. To load such a quantity on some machines, it would be necessary to use two separate operations. The first operation would be to move the upper and lower halves into the halves of an aligned temporary. The second would be to load that double-precision temporary. The reverse would be needed to store a result. All double-precision real and complex quantities are required to fall on even word boundaries on machines with corresponding hardware requirements and to

issue a diagnostic if the source code demands a violation of the rule.

9.3.2 Dummy Procedure Arguments

If any argument of a procedure is of type character, all dummy procedure arguments of that procedure must be declared in an **external** statement. This requirement arises as a subtle corollary of the way we represent character string arguments. A warning is printed if a dummy procedure is not declared **external.** Code is correct if there are no **character** arguments.

9.3.3 T and TL Formats

The implementation of the **t** (absolute tab) and **tl** (leftward tab) format codes is defective. These codes allow rereading or rewriting part of the record which has already been processed. The implementation uses "seeks"; so if the unit is not one which allows seeks (such as a terminal) the program is in error. A benefit of the implementation chosen is that there is no upper limit on the length of a record nor is it necessary to predeclare any record lengths except where specifically required by Fortran or the operating system.

9.4 INTERPROCEDURE INTERFACE

To be able to write C language procedures that call or are called by Fortran procedures, it is necessary to know the conventions for procedure names, data representation, return values, and argument lists that the compiled code obeys.

9.4.1 Procedure Names

On UNIX systems, the name of a common block or a Fortran procedure has an underscore appended to it by the compiler to distinguish it from a C language procedure or external variable with the same user-assigned name. Fortran library procedure names have embedded underscores to avoid clashes with user-assigned subroutine names.

9.4.2 Data Representations

The following is a table of corresponding Fortran and C language declarations:

Fortran	C Language
integer*2 x	**short int x;**
integer x	**long int x;**
logical x	**long int x;**
real x	**float x;**
double precision x	**double x;**
complex x	**struct { float r, i; } x;**

double complex x	**struct { double dr, di; } x;**
character*6 x	**char x[6];**

By the rules of Fortran, **integer, logical,** and **real** data occupy the same amount of memory.

9.4.3 Return Values

A function of type **integer, logical, real,** or **double precision** declared as a C language function returns the corresponding type. A **complex** or **double complex** function is equivalent to a C language routine with an additional initial argument that points to the place where the return value is to be stored. Thus, the following:

```
complex function f( . . . )
```

is equivalent to

```
struct { float r, i; } temp;
f_(&temp, . . .)
 . . .
```

A character-valued function is equivalent to a C language routine with two extra initial arguments - a data address and a length. Thus,

```
character*15 function g( . . . )
```

is equivalent to

```
char result[ ];
long int length;
g_(result, length, . . .)
 . . .
```

and could be invoked in C language by

```
char chars[15];
 . . .
g_(chars, 15L, . . . );
```

Subroutines are invoked as if they were **integer**-valued functions whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the function but are used to do an indexed branch in the calling procedure. (If the subroutine has no entry points with alternate return arguments, the returned value is undefined.) The statement

```
call nret(*1, *2, *3)
```

is treated exactly as if it were the computed **goto**

```
goto (1, 2, 3), nret( )
```

9.4.4 Argument Lists

All Fortran arguments are passed by address. In addition, for every argument that is of type character or that is a dummy procedure, an argument giving the length of the value is passed. (The string lengths are **long int** quantities passed by value.) The order of arguments is then:

Extra arguments for complex and character functions
Address for each datum or function
A **long int** for each character or procedure argument

Thus, the call in

```
external f
character*7 s
integer b(3)
. . .
call sam(f, b(2), s)
```

is equivalent to that in

```
int f();
char s[7];
long int b[3];
. . .
sam_(f, &b[1], s, 0L, 7L);
```

Note that the first element of a C language array always has subscript 0, but Fortran arrays begin at 1 by default. Fortran arrays are stored in column-major order; C language arrays are stored in row-major order.

9.5 FILE FORMATS

9.5.1 Structure of Fortran Files

Fortran requires four kinds of external files: *sequential formatted* and *unformatted*, and *direct formatted* and *unformatted*. On UNIX systems, these are all implemented as ordinary files which are assumed to have the proper internal structure.

Fortran I/O is based on "records." When a direct file is opened in a Fortran program, the record length of the records must be given; and this is used by the Fortran I/O system to make the file look as if it is made up of records of the given length. In the special case that the record length is given as 1, the files are not considered to be divided into records but are treated as byte-addressable byte strings; i.e., as ordinary files on the UNIX system. (A read or write request on such a file keeps consuming bytes until satisfied rather than being restricted to a single record.)

The peculiar requirements on sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

The Fortran I/O system breaks sequential formatted files into records while reading by using each new-line as a record separator. The result of reading off the end of a record is undefined according to the Fortran 77 American National Standard. The I/O system is permissive and treats the record as being extended by blanks. On output, the I/O system will write a new-line at the end of each record. It is also possible for programs to write new-lines for themselves. This is an error, but the only effect will be that the single record the user thought was written will be treated as more than one record when being read or backspaced over.

9.5.2 Preconnected Files and File Positions

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input, unit 6 is connected to the standard output, and unit 0 is connected to the standard error unit. All are connected for sequential formatted I/O.

All the other units are also preconnected when execution begins. Unit *n* is connected to a file named **fort.***n*. These files need not exist nor will they be created unless their units are used without first executing an **open.** The default connection is for sequential formatted I/O.

The Fortran 77 Standard does not specify where a file which has been explicitly **open**ed for sequential I/O is initially positioned. In fact, the I/O system attempts to position the file at the end. A **write** will append to the file and a **read** will result in an "end of file" indication. To position a file to its beginning, use a **rewind** statement. The preconnected units 0, 5, and 6 are positioned as they come from the parent process.

10. RATFOR

This chapter describes the Ratfor(1) preprocessor. It is assumed that the user is familiar with the current implementation of Fortran 77 on the UNIX system.

The Ratfor language allows users to write Fortran programs in a fashion similar to C language. The Ratfor program is implemented as a preprocessor that translates this "simplified" language into Fortran. The facilities provided by Ratfor are:

- Statement grouping
- **if–else** and **switch** for decision making
- **while**, **for**, **do**, and **repeat–until** for looping
- **break** and **next** for controlling loop exits
- Free form input such as multiple statements/lines and automatic continuation
- Simple comment convention
- Translation of >, >=, etc., into .gt., .ge., etc.
- **return** statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files.

10.1 USAGE

The Ratfor program takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **-6x**, which uses **x** as a continuation character in column 6 (the UNIX system uses & in column 1), **-h**, which causes quoted strings to be turned into *n*H constructs and **-C**, which causes Ratfor comments to be copied into the generated Fortran.

10.2 STATEMENT GROUPING

The Ratfor language provides a statement grouping facility. A group of statements can be treated as a unit by enclosing them in the braces { and }. For example, the Ratfor code

```
if (x > 100)
   { call error("x>100"); err = 1; return }
```

will be translated by the Ratfor preprocessor into **Fortran** equivalent to

```
      if (x .le. 100) goto 10
         call error(5hx>100)
         err = 1
         return
10    ...
```

which should simplify programming effort. By using { and }, a group of statements can be used instead of a single statement.

Also note in the previous Ratfor example that the character > was used instead of **.GT.** in the **if** statement. The Ratfor preprocessor translates this C language type operator to the appropriate Fortran operator. More on relationship operators later.

In addition, many Fortran compilers permit character strings in quotes (like "*x>100*"). But others, like ANSI Fortran 66, do not. Ratfor converts it into the right number of *Hs.*

The Ratfor language is free form. Statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The previous example could also be written as

```
if (x > 100) {
    call error("x>100")
    err = 1
    return
}
```

which shows grouped statements spread over several lines. In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement, no braces are needed.

10.3 THE "if-else" CONSTRUCTION

The Ratfor language provides an **else** statement. The syntax of the **if-else** construction is:

if (*legal Fortran condition*)
    ratfor statement
else
    ratfor statement

where the **else** part is optional. The **legal Fortran condition** is anything that can legally go into a Fortran Logical *F* statement. The Ratfor preprocessor does not check this clause since it does not know enough Fortran to know what is permitted. The "ratfor" **statement** is any Ratfor or Fortran statement or any collection of them in braces. For example:

```
if (a <= b)
    { sw = 0; write(6, 1) a, b }
else
    { sw = 1; write(6, 1) b, a }
```

is a valid Ratfor **if-else** construction. This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed.

10.3.1 Nested "if" Statements

The statement that follows an **if** or an **else** can be any Ratfor statement including another **if** or **else** statement. In general, the structure

```
if (condition) action
else if (condition) action
else action
```

provides a way to write a multibranch in Ratfor. (The Ratfor language also provides a **switch** statement which could be used instead, under certain conditions.) The last **else** handles the "default" condition. If there is no default action, this final **else** can be omitted. Thus, only the actions associated with the valid condition are performed. For example:

```
if (x < 0)
    x = 0
else if (x > 100)
    x = 100
```

will ensure that x is not less than 0 and not greater than 100.

Nested **if** and **else** statements could result in ambiguous code. In Ratfor when there are more **if** statements than **else** statements, **else** statements are associated with the closest previous **if** statement that currently does not have an associated **else** statement. For example:

```
if (x > 0)
if (y > 0)
write(6,1) x, y
else
write(6,2) y
```

is interpreted by the Ratfor preprocessor as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
    else
        write(6, 2) y
}
```

in which the braces are assumed. If the other association is desired it **must** be written as

```
if (x > 0) {
    if (y > 0)
        write(6, 1) x, y
}
else
    write(6, 2) y
```

with the braces specified.

10.4 THE "switch" STATEMENT

The **switch** statement provides a way to express multiway branches which branch on the value of some *integer*-valued expression. The syntax is

switch (*expression***)** {
    case *expr1* :
    statements
    case *expr2, expr3* :
    statements
    ...
    default:
    statements
}

where each **case** is followed by an integer expression (or several integer expressions separated by commas). The **switch** *expression* is compared to each **case** *expr* until a match is found. Then the *statements* following that **case** are executed. If no **cases** match *expression*, then the *statements* following **default** are executed. The **default** section of a **switch** is optional.

When the *statements* associated with a **case** are executed, the entire **switch** is exited immediately. This is different from C language.

10.5 THE "do" STATEMENT

The **do** statement in Ratfor is quite similar to the **DO** statement in Fortran except that it uses no statement number (braces are used to mark the end of the **do** instead of a statement number). The syntax of the **ratfor do** statement is

do *legal-Fortran-DO-text* {
  ratfor statements
}

The *legal-Fortran-DO-text* must be something that can legally be used in a Fortran **DO** statement. Thus if a local version of Fortran allows **DO** limits to be expressions (which is not currently permitted in ANSI Fortran 66), they can be used in a **ratfor do** statement. The *ratfor statements* are enclosed in braces; but as with the **if**, a single statement need not have braces

around it. For example, the following code sets an array to zero:

```
do i = 1, n
   x(i) = 0.0
```

and the code

```
do i = 1, n
  do j = 1, n
    m(i, j) = 0
```

sets the entire array *m* to zero.

10.6 THE "break" AND "next" STATEMENTS

The Ratfor **break** and **next** statements provide a means for leaving a loop early and one for beginning the next iteration. The **break** causes an immediate exit from the **do**; in effect, it is a branch to the statement *after* the **do**. The **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array

```
do i = 1, n {
  if (x(i) < 0.0)
    next
  process positive element
}
```

The **break** and **next** statements will also work in the other Ratfor looping constructions and will be discussed with each looping construction.

The **break** and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop. For example:

```
break 2
```

exits from two levels of enclosing loops, and

```
break 1
```

is equivalent to **break**. The

```
next 2
```

iterates the second enclosing loop.

10.7 THE "while" STATEMENT

The Ratfor language provides a **while** statement. The syntax of the **while** statement is

while (*legal-Fortran-condition*)
  ratfor statement

As with the **if**, **legal–Fortran–condition** is something that can go into a Fortran Logical *F*, and **ratfor statement** is a single statement which may be multiple statements enclosed in braces.

For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a **while** loop to find the first nonblank character could be

```
while (nextch(ich) == iblank)
    ;
```

where a semicolon by itself is a null statement (which is necessary here to mark the end of the **while**). If the semicolon were not present, the **while** would control the next statement. When the loop is exited, **ich** contains the first nonblank.

10.8 THE "for" STATEMENT

The **for** statement is another Ratfor loop. A **for** statement allows explicit initialization and increment steps as part of the statement.

The syntax of the **for** statement is

for (*init* **;** *condition* **;** *increment* **)**
  ratfor statement

where *init* is any single Fortran statement which is executed once before the loop begins. The **increment** is any single Fortran statement that is executed at the end of each pass through the loop before the test. The *condition* is again anything that is legal in a Fortran Logical *F*. Any of **init**, **condition**, and **increment** may be omitted although the semicolons **must** always be present. A nonexistent **condition** is treated as always true, so

```
for (;;)
```

is an infinite loop.

For example, a Fortran **DO** loop could be written as

```
for (i = 1; i <= n; i = i + 1) ...
```

which is equivalent to

```
i = 1
while (i <= n) {
    ...
    i = i + 1
}
```

The initialization and increment of *i* have been moved into the **for** statement.

The **for**, **do**, and **while** versions have the advantage that they will be done zero times if *n* is less than 1. In addition, the **break** and **next** statements work in a **for** loop.

The *increment* in a **for** need not be an arithmetic progression. The program

```
sum = 0.0
for (i = first; i > 0; i = ptr(i))
    sum = sum + value(i)
```

steps through a list (stored in an integer array *ptr*) until a zero pointer is found while adding up elements from a parallel array of values. Notice that the code also works correctly if the list is empty.

10.9 THE "repeat-until" STATEMENT

There are times when a test needs to be performed at the bottom of a loop after one pass through. This facility is provided by the **repeat-until** statement. The syntax for the **repeat-until** statement is

repeat
ratfor statement
until (*legal-Fortran-condition* **)**

where **ratfor–statement** is done once, then the *condition* is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a **repeat** by itself is an infinite loop. A **repeat-until** loop can be exited by the use of a **stop**, **return**, or **break** statement or an implicit stop such as running out of input with a **READ** statement.

As stated before, a **break** statement causes an immediate exit from the enclosing **repeat-until** loop. A **next** statement will cause a skip to the bottom of a **repeat-until** loop (i.e., to the **until** part).

10.10 THE "return" STATEMENT

The standard Fortran mechanism for returning a value from a routine uses the name of the routine as a variable. This variable can be assigned a value. The last value stored in it is the value returned by the function. For example, in a Fortran routine named *equal*, the statements

```
equal = 0
return
```

cause *equal* to return zero.

The Ratfor language provides a **return** statement similar to the C language **return** statement. In order to return a value from any routine, the **return** statement has the syntax

return (*expression*)

where *expression* is the value to be returned.

If there is no parenthesized expression after **return**, no value is returned.

10.11 THE "define" STATEMENT

The Ratfor language provides a **define** statement similar to the C language version. Any string of alphanumeric characters can be defined as a name. Whenever that name occurs in the input (delimited by nonalphanumerics), it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off.) A defined name can be arbitrarily long and must begin with a letter.

Usually the **define** statement is used for symbolic parameters. The syntax of the **define** statement is

define name value

where *name* is a symbolic name that represents the quantity of *value*. For example:

```
define ROWS 100
define CLOS 50
dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

causes the preprocessor to replace the name *ROWS* with the value *100* and the name *COLS* with the value *50*. Alternately, definitions may be written as

```
define(ROWS, 100)
```

in which case the defining text is everything after the comma up to the right parenthesis. This allows multiple-line definitions.

10.12 THE "include" STATEMENT

The Ratfor language provides an **include** statement similar to the **#include** <...> statement in C language. The syntax for this statement is

include *file*

which inserts the contents of the named file into the Ratfor input file in place of the **include** statement. The standard usage is to place **COMMON** blocks on a file and use the **include** statement to include the common code whenever needed.

10.13 FREE-FORM INPUT

In Ratfor, statements can be placed anywhere on a line. Long statements are continued automatically as are long conditions in **if**, **for**, and **until**

statements. Blank lines are ignored. Multiple statements may appear on one line if they are separated by semicolons. No semicolon is needed at the end of a line if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

= + - * , | & (_

are assumed to be continued on the next line. Underscores are discarded wherever they occur. All other characters remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and placed in columns 1 through 5 upon output. Thus:

write(6, 100); 100 format("hello")

is converted into

```
      write(6, 100)
100   format(5hhello)
```

10.14 TRANSLATIONS

When the **-h** option is chosen, text enclosed in matching single or double quotes is converted to *nH...* but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash (\) serves as an escape character; i.e., the next character is taken literally. This provides a way to get quotes and the backslash itself into quoted strings. For example:

"\'"

is a string containing a backslash and an apostrophe. (This is **not** the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character **%** is left absolutely unaltered except for stripping off the **%** and moving the line one position to the left. This is useful for inserting control cards and other things that should not be preprocessed (like an existing Fortran program). Use **%** only for ordinary statements not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made (except within single or double quotes or on a line beginning with a **%**):

== .eq.

!= .ne.

> .gt.

>= .ge.

< .lt.

<= .le.

& .and.

| .or.

! .not.

In addition, the following translations are provided for input devices with restricted character sets:

[{

] }

$({

$) }

10.15 WARNINGS

The Ratfor preprocessor catches certain syntax errors (such as missing braces), **else** statements without **if** statements, and most errors involving missing parentheses in statements.

All other errors are reported by the Fortran compiler. Unfortunately, the Fortran compiler prints messages in terms of generated Fortran code and not in terms of the Ratfor code. This makes it difficult to locate Ratfor statements that contain errors.

The keywords are deserved. Using **if**, **else**, **while**, etc., as variable names will cause considerable problems. Likewise, spaces within keywords and use of the Arithmetic *F* will cause problems.

The Fortran *n*H convention is not recognized by Ratfor. Use quotes instead.

10.16 EXAMPLE OF RATFOR CONVERSION

As an example of how to use the Ratfor program, the following program **prog.r** (where the **.r** indicates a Ratfor source program), is written in the Ratfor language:

```
    ICNT=0
  10 WRITE(6,31)
  31 FORMAT("INPUT FIRST NUMBER")
    READ(5,32) A
  32 FORMAT(F10.2)
    WRITE(6,33)
  33 FORMAT("INPUT SECOND NUMBER")
    READ(5,34) B
  34 FORMAT(F10.2)
    IF(A<B)
      WRITE(6,36) A,B
    ELSE WRITE(6,37)A,B
  36 FORMAT(F10.2," < ",F10.2)
  37 FORMAT(F10.2," >= ",F10.2)
    ICNT=ICNT+1
    IF(ICNT.EQ.5)
      GOTO 100
    GOTO 10
 100 END
```

The command

ratfor prog.r > prog.f

causes the Fortran translation program **prog.f** to be produced. (The Ratfor program **prog.r** remains intact.) The Fortran program **prog.f** follows:

```
    icnt=0
10   write(6,31)
31   format("INPUT FIRST NUMBER")
    read(5,32) a
32   format(f10.2)
    write(6,33)
33   format("INPUT SECOND NUMBER")
    read(5,34) b
34   format(f10.2)
    if(.not.(a.lt.b))goto 23000
    write(6,36) a,b
    goto 23001
23000 continue
    write(6,37)a,b
23001 continue
36   format(f10.2," < ",f10.2)
37   format(f10.2," >= ",f10.2)
    icnt=icnt+1
    if(.not.(icnt.eq.5))goto 23002
    goto 100
23002 continue
    goto 10
100  end
```

The Fortran program **prog.f** is compiled using the command

f77 prog.f

An object program file **prog.o** and a final output file **a.out** are produced. Since the output file **a.out** is an executable file, the command

a.out

causes the program to run.

The Ratfor program **prog.r** can also be translated and compiled with the single command

f77 prog.r

where the **.r** indicates a Ratfor source program. An object file **prog.o** and a final output file **a.out** are produced.

11. EFL

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

The name EFL originally stood for "Extended Fortran Language." The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions.

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression.* A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation

[*item* **]**

could refer to any of the following:

item
item, *item*
item, *item*, *item*

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

11.1 LEXICAL FORM

11.1.1 Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	**' "**
sharp	**#**
continuation	**_**

braces	{ }
parentheses	( )
other	, ; : . + - * /
	= < > & | $

Letter case (upper or lower) is ignored except within strings, so "a" and "A" are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ("!") may be used in place of a tilde (" "). Square brackets ("[" and "]") may be used in place of braces ("{" and "}").

11.1.2 Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

11.1.2.1 White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

11.1.2.2 Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

11.1.2.3 Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

include joe

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

11.1.2.4 Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of a line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
  000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

11.1.2.5 Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

11.1.3 Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

11.1.3.1 Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

11.1.3.2 Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks ('), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

'hello there'
"ain't misbehavin'"

11.1.3.3 Integer Constants

An integer constant is a sequence of one or more digits.

0
57
123456

11.1.3.4 Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

.I
I.
I.J
IE
I.E
.IE
I.JE

11.1.3.5 Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

11.1.3.6 Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  -  *  /  **
<  <=  >  >=  ==  =
&&  ||  &  |
+=  -=  /=  **=
&&=  ||=  &=  |=
->  .  $
```

A dot (".") is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see "ATAVISMS") in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.,* **.lt.**).

11.1.4 Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the

statement

n += 1

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

11.2 PROGRAM FORM

11.2.1 Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

11.2.2 Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in "PROCEDURES."

11.2.3 Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See "Blocks" under "EXECUTABLE STATEMENTS.") An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *K* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
...
if(x > 2)
        {           # new block
        integer x   # a different variable
        do x = 1,7
                    write(,x)
        ...
        }           # end of block
end     # end of procedure, return to block 0
```

11.2.4 Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in "COMPILER OPTIONS". The **include, define,** and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statement and finishes with an **end** statement; these are discussed in "PROCEDURES". Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

11.2.5 Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
        read(, x)
        if(x < 3) goto error
        ...
error:  fatal("bad input")
```

11.3 DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

11.3.1 Basic Types

The basic types are

logical
integer
field(*m* :*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values *true* and *false*. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m* :*n*]). A "real" quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

11.3.2 Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
–94
+6
0

A long real ("double precision") constant is a floating point constant containing an exponent field that begins with the letter **d.** A real ("single precision") constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid **real** constants:

17.3
–.4
7.9e–6 (= 7.9×10^{6})
14e9 (= 1.4×10^{10})

The following are valid **long real** constants

7.9d–6 (= 7.9×10^{6})
5d3

A character constant is a quoted string.

11.3.3 Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

11.3.3.1 Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

11.3.3.2 Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

11.3.3.3 Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

11.3.4 Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

11.3.5 Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are

called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

11.4 EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression* **)**
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in "Unary Operators" and "Binary Operators" under "EXPRESSIONS".

```
-> .
**
* /    unary + - ++ --
+ -
< <= > >= == =
& &&
| ||
$
= += -= *= /= **=  &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

11.4.1 Primaries

Primaries are the basic elements of expressions. They include constants, variables, array elements, structure members, procedure invocations, input/output expressions, coercions, and sizes.

11.4.1.1 Constants

Constants are described in "Constants" under "DATA TYPES AND VARIABLES".

11.4.1.2 Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may appear only as procedure arguments and in input/output lists.

11.4.1.3 Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

a(5)
b(6, –3, 4)

11.4.1.4 Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

a.b
x(3).y(4).z(5)

11.4.1.5 Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename **( )**
procedurename **(** *expression* **)**
procedurename **(** *expression-1*, ..., *expression-n* **)**

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see "Known Functions" under "PROCEDURES"), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

f(x)
work()
g(x, y+3, 'xx')

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See "PROCEDURES".

11.4.1.6 Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See "Input/Output Statements" under "EXECUTABLE STATEMENTS".

11.4.1.7 Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes **(** *expression* **)**

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = **5**
long real(5) = **5.0d0**
complex(5,3) = **5+3i**

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

11.4.1.8 Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

11.4.2 Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

11.4.3 Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

11.4.3.1 Arithmetic

Unary + has no effect. A unary – yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator — subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

11.4.3.2 Logical

The only logical unary operator is complement ( ). This operator is defined by the equations

true = **false**
false = **true**

11.4.4 Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

11.4.4.1 Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	i	r	l r	c	l c
i	i	r	l r	c	l c
r	r	r	l r	c	l c
l r	l r	l r	l r	l c	l c
c	c	c	l c	c	l c
l c	l c	l c	l c	l c	l c

i = integer
r = real
l r = long real
c = complex
l c = long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

11.4.4.2 Logical

The two binary logical operations in EFL, **and** and **or,** are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise, the expression has the value of **b.** The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise, the expression has the value of **b.** The other forms of the operators (**&** for **and** and **|** for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

11.4.4.3 Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator		Meaning
<	<	less than
<=	≤	less than or equal to
==	=	equal to
=		not equal to
>	>	greater than
>=	≥	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and = . The character collating sequence is not defined.

11.4.4.4 Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = expression

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op = b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

11.4.5 Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside –> structurename

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside.* In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) –> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place.**

11.4.6 Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

11.4.7 Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

11.5 DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

11.5.1 Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two forms:

attributes variable-list
attributes { *declarations* }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

11.5.2 Attributes

11.5.2.1 Basic Types

The following are basic types in declarations

logical
integer
field(*m* **:***n***)**
character(*k***)**
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

11.5.2.2 Arrays

The dimensionality may be declared by an **array** attribute

array(b_1,...,b_n)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper* − *lower* + 1 is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as **(0:n−1)**. The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

array(5)
array(5, 1:5, −3:0)
array(5, *)
array(0:m−1, m)

11.5.2.3 Structures

A structure declaration is of the form

struct *structname* **{** *declaration statements* **}**

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
      {
      integer a, b
      real x(5)
      }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

11.5.2.4 Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

11.5.2.5 Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname* **)**

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

11.5.2.6 External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external [[*name*]]

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding **procedure** statement.

11.5.3 Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an **array** attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

11.5.4 The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial [[*var* = *val*]]

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

11.6 EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements, otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

11.6.1 Expression Statements

11.6.1.1 Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see "Input/Output Statements" under "EXECUTABLE STATEMENTS") resemble procedure invocations but do not yield a value. If an error occurs the program stops.

11.6.1.2 Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

11.6.2 Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i  # this variable is unknown
        # outside the braces
big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

11.6.3 Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

11.6.3.1 If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression* **)** □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

11.6.3.2 If-Else

A more general statement is of the form

if (*logical-expression* **)** □ *statement-1* □
else □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise, *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-**else**d **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

11.6.3.3 Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression* **)** □ *block*

Inside the block two special types of labels are recognized. A prefix of the

form

case [[*constant*]] **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The **else-if** example above is better written as

```
select(x)
  {
  case 1:
    k = 1
  case 3,5:
    k = 2
  default:
    k = 3
  }
```

Note that control does not "fall through" to the next case.

11.6.4 Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

11.6.4.1 While Statement

This construct has the form

while (*logical-expression* **)** □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

11.6.5 For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* **,** □ *logical-expression* **,**
□ *iteration-statement* **)** □ *body-statement*

Except for the behavior of the **next** statement (see "Branch Statement" under "EXECUTABLE STATEMENTS"), this construct is equivalent to

initial-statement
while (*logical-expression* **)**
{
body-statement
iteration-statement
}

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
        n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
        ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

11.6.5.1 Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

11.6.5.2 Repeat ... Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until (** *logical-expression* **)**

executes the *statement,* then evaluates the logical; if the logical is true the loop is complete; otherwise, control returns to the *statement.* Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until.** In practice, this appears to be the least frequently used looping construct.

11.6.5.3 Do Loop

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable* = *expression-1, expression-2, expression-3*
statement

The variable is first given the value *expression-1.* The statement is

executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2.* If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable=expression-1, variable<=t2, variable+=t3)
    statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2.* The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
        n += i
```

11.6.6 Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

11.6.6.1 Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a **select** the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4
```

```
        case 4:
                fixup(k)
                goto default

        default:
                prmsg("ouch")
        }
```

(If two **select** statements are nested, the case labels of the outer **select** are not accessible from the inner one.)

11.6.6.2 Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
  {
  do a computation
  if ( finished )
      break
  }
```

More general forms permit controlling a branch out of more than one construct.

break 3

transfers control to the statement following the third loop and/or **select** surrounding the statement. It is possible to specify which type of construct (**for**, **while**, **repeat**, **do**, or **select**) is to be counted. The statement

break while

breaks out of the first surrounding **while** statement. Either of the statements

break 3 for
break for 3

will transfer to the statement after the third enclosing **for** loop.

11.6.6.3 Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while,** the *iteration-statement* of a **for,** the body of a **repeat,** the test of a **repeat...until,** or the increment of a **do.** Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

11.6.6.4 Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression* **)**

11.6.7 Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

11.6.7.1 Input/Output Units

Each I/O statement refers to a "unit," identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit.* These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records.* These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file.*

11.6.7.2 Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* **,** *binary-output-list* **)**
readbin(*unit* **,** *binary-input-list* **)**

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist*

(see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

11.6.7.3 Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

11.6.7.4 Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

11.6.7.5 Formats

The following are permissible *format-specifiers.* The quantities *w* , *d* , and *k* must be integer constant expressions.

i(*w***)**	integer with *w* digits
f(*w***,***d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w***,***d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter **e**
l(*w***)**	logical field of width *w* characters, the first of which is **t** or **f**

	(the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(*w*)	character string of width *w*
s(*k*)	skip *k* lines
x(*k*)	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

11.6.7.6 Manipulation Statements

The three input/output statements

backspace(*unit***)**
rewind(*unit***)**
endfile(*unit***)**

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

11.7 PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

11.7.1 Procedures Statement

Each procedure begins with a statement of one of the forms

procedure
attributes **procedure** *procedurename*
attributes **procedure** *procedurename* **( )**
attributes **procedure** *procedurename* **(** [*name*] **)**

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may

be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

11.7.2 End Statement

Each procedure terminates with a statement

end

11.7.3 Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

11.7.4 Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

11.7.5 Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

11.7.5.1 Minimum and Maximum Functions

The generic functions are **min** and **max.** The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real.** Otherwise, if any of the arguments are **real** then the result is **real;** otherwise all the arguments and the result must be **integer.**

Examples are

min(5, x, –3.20)
max(i, z)

11.7.5.2 Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

11.7.5.3 Elementary Functions

The following generic functions take arguments of **real**, **long real**, or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only **real** or **long real** arguments:

atan	$atan(x) = \tan^{-1} x$
atan2	$atan2(x,y) = \tan^{-1} \frac{x}{y}$

11.7.5.4 Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

11.8 ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

11.8.1 Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ("%") is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

11.8.2 Call Statement

A subroutine call may be preceded by the keyword **call.**

call joe
call work(17)

11.8.3 Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

11.8.4 Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

11.8.5 Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list* **)** *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

implicit (a–h, o–z) real
implicit (i–n) integer

11.8.6 Computed Goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto ([[*label*]] **),** *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

11.8.7 Goto Statement

In unconditional and computed **goto** statements, it is permissible to separate the **go** and **to** words, as in

go to xyz

11.8.8 Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see "COMPILER OPTIONS") which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
=	**.ne.**
&	**.and.**
|	**.or.**
&&	**.andand.**
||	**.oror.**
	.not.
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt, le,** etc. The readable forms in the left column are always recognized.

11.8.9 Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

11.8.10 Function Values

The preferred way to return a value from a function in EFL is the **return**(*value*) construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

11.8.11 Equivalence

A statement of the form

equivalence $v_1, v_2, \ldots, v_n$

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

11.8.12 Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	*Argument Type*	*Result Type*
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

11.9 COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option [*opt* **]**

where each *opt* is of one of the forms

optionname
optionname = *optionvalue*

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names **yes** and **no** apply to a number of options.

11.9.1 Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

11.9.2 Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

11.9.3 Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

11.9.4 Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (**&**) in the first column of the continued lines instead.

11.9.5 Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	*Type*
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

11.9.6 Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	*Size Option*	*Alignment Option*
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

11.9.7 Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

11.9.8 Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11.10 EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.10.1 File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure  # main program
character(100) line

while( read( , line) == 0 )
                    write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.10.2 Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
        {
        c(i,j) = 0
        do k = 1,n
                c(i,j) += a(i,k) * b(k,j)
        }
end
```

11.10.3 Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST         0
define NOTFOUND     -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value,
#  an x, and a y value.
struct
        {
        integer nextindex
        integer x, y
        } list(*)
integer first, p, arg

for(p = first , p =LAST && list(p).x<=x ,
   p = list(p).nextindex)
        if(list(p).x == x)
                return( list(p).y )

return(NOTFOUND)
end
```

The search is a single **for** loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using

an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

11.10.4 Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)   # print an expression tree

integer first           # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
        {
        character(1) op
        integer leftp, rightp
        real val
        } tree(100)     # array of structures

struct
        {
        integer nextstate
        integer nodep
        } stackframe(100)

define NODE   tree(currentnode)
define STACK  stackframe(stackdepth)

#  nextstate values
define DOWN  1
define LEFT  2
define RIGHT  3
```

```
# initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first

while( stackdepth > 0 )
        {
        currentnode = STACK.nodep
        select(STACK.nextstate)
                {
                case DOWN:
                        if(NODE.op == " ")  # a leaf
                                {
                                outval( NODE.val )
                                stackdepth -= 1
                                }
                        else    { # a binary operator node
                                outch( "(" )
                                STACK.nextstate = LEFT
                                stackdepth += 1
                                STACK.nextstate = DOWN
                                STACK.nodep = NODE.leftp
                                }

                case LEFT:
                        outch( NODE.op )
                        STACK.nextstate = RIGHT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.rightp

                case RIGHT:
                        outch( ")" )
                        stackdepth -= 1
                }
        }
end
```

11.11 PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

11.11.1 Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

11.11.1.1 Character String Copying

The subroutine **ef1asc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine ef1asc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a.**

11.11.1.2 Character String Comparisons

The function **ef1cmc** is invoked to determine the order of two character strings. The declaration is

```
integer function ef1cmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb.** It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

11.12 DIFFERENCES BETWEEN RATFOR AND EFL

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the "ATAVISMS" are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry

about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

11.13 COMPILER

11.13.1 Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers.

11.13.2 Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

11.13.3 Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (See "EXAMPLES".)

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do  3 i = 1, m
         do  2 j = 1, p
            c(i, j) = 0
            do  1 k = 1, n
               c(i, j) = c(i, j)+a(i, k)*b(k, j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk:

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c    nextstate values
c    initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
1     if (stapth .le. 0) goto  9
         curode = staame(2, stapth)
         goto  7
2           if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto  4
3              call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
```

```
                staame(2, stapth) = tree(2, curode)
4           goto  8
5           call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto  8
6           call outch(1h))
            stapth = stapth-1
            goto  8
7           if (staame(1, stapth) .eq. 3) goto  6
            if (staame(1, stapth) .eq. 2) goto  5
            if (staame(1, stapth) .eq. 1) goto  2
8       continue
        goto  1
9   continue
    end
```

11.14 CONSTRAINTS ON EFL

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

11.14.1 External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

11.14.2 Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

11.14.3 Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

11.14.4 Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

11.14.5 Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

12. CURSES AND TERMINFO PACKAGE

This chapter is an introduction to **curses**(3X) and **terminfo**(4). It is intended for the programmer who must write a screen-oriented program using the **curses** package. Several example programs are discussed. The example programs can be found in Chapter 13. This chapter also documents each **curses** function. It is intended as a reference.

For **curses** to be able to produce terminal dependent output, it has to know what kind of terminal you have. The UNIX system convention for this is to put the name of the terminal in the variable TERM in the environment. Thus, a user on a DEC VT100 would set **TERM=vt100** when logging in. Curses uses this convention.

12.0.1 Output

A program using **curses** always starts by calling **initscr()**. (See Figure 12-1.) Other modes can then be set as needed by the program. Possible modes include **cbreak()**, and **idlok(stdscr, TRUE)**. These modes will be explained later. During the execution of the program, output to the screen is done with routines such as **addch(ch)** and **printw(fmt,args)**. (These routines behave just like **putchar** and **printf** except that they go through **curses**.) The cursor can be moved with the call **move(row,col)**. These routines only output to a data structure called a *window*, not to the actual screen. A window is a representation of a CRT screen, containing such things as an array of characters to be displayed on the screen, a cursor, a current set of video attributes, and various modes and options. You don't need to worry about windows unless you use more than one of them, except to realize that a window is buffering your requests to output to the screen.

To send all accumulated output, it is necessary to call **refresh()**. (This can be thought of as a **flush**.) Finally, before the program exits, it should call **endwin()**, which restores all terminal settings and positions the cursor at the bottom of the screen.

```
#include <curses.h>
...
    initscr(); /* Initialization */

    cbreak();/* Various optional mode settings */
    nonl();
    noecho();
...
    while (!done) { /* Main body of program */
        ...
        /* Sample calls to draw on screen */
        move(row, col);
        addch(ch);
        printw("Formatted print with value %d\n", value);
        ...
        /* Flush output */
        refresh();
        ...
    }

    endwin();    /* Clean up */
    exit(0);
```

Figure 12-1. Framework of a Curses Program

See the program **scatter** in Chapter 13 for an example program. This program reads a file, and displays the file in a random order on the screen. Some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not. The variables **LINES** and **COLS** are defined by **initscr** with the current screen size. Programs should use them instead of assuming a 24x80 screen.

No output to the terminal actually happens until **refresh** is called. Instead, routines such as **move** and **addch** draw on a window data structure called **stdscr** (standard screen). **Curses** always keeps track of what is on the physical screen, as well as what is in **stdscr**.

When **refresh** is called, **curses** compares the two screen images and sends a stream of characters to the terminal that will turn the current screen into what is desired. Curses considers many different ways to do this, taking into account the various capabilities of the terminal, and similarities between what is on the screen and what is desired. It usually outputs as few

characters as is possible. This function is called *cursor optimization* and is the source of the name of the **curses** package.

NOTE: Due to the hardware scrolling of terminals, writing to the lower righthand character position is impossible.

12.0.2 Input

Curses can do more than just draw on the screen. Functions are also provided for input from the keyboard. The primary function is **getch()** which waits for the user to type a character on the keyboard, and then returns that character. This function is like **getchar** except that it goes through **curses**. Its use is recommended for programs using the **cbreak()** or **noecho()** options, since several terminal or system dependent options become available that are not possible with **getchar**.

Options available with **getch** include **keypad** which allows extra keys such as arrow keys, function keys, and other special keys that transmit escape sequences, to be treated as just another key. (The values returned for these keys are listed below.) **KEY_LEFT** in **curses.h**. The values for these keys are over octal 400, so they should be stored in a variable larger than a **char**.) *nodelay* mode causes the value –1 to be returned if there is no input waiting. Normally, **getch** will wait until a character is typed. Finally, the routine **getstr(str)** can be called, allowing input of an entire line, up to a newline. This routine handles echoing and the erase and kill characters of the user. Examples of the use of these options are in later example programs.

The following function keys might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported, due to lack of definitions in **terminfo** or the terminal not transmitting a unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character

KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	Soft (partial) reset (unreliable)
KEY_RESET	0531	Reset or hard reset (unreliable)
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower left)

See the program **show** in Chapter 13 for an example use of **getch**. **Show** pages through a file, showing one screen full each time the user presses the space bar. By creating an input file for **show** made up of 24 line pages, each segment varying slightly from the previous page, nearly any exercise for **curses** can be created. Such input files are called *show scripts*.

In the **show** program, **cbreak** is called so that the user can press the space bar without having to hit return. The **noecho** function is called to prevent the space from echoing in the middle of a **refresh**, messing up the screen. The **nonl** function is called to enable more screen optimization. The **idlok** function is called to allow insert and delete line, since many show scripts are constructed to duplicate bugs caused by that feature. The **clrtoeol** and **clrtobot** functions clear from the cursor to the end of the line and screen, respectively.

12.0.3 Highlighting

The function **addch** always draws two things on a window. In addition to the character itself, a set of *attributes* is associated with the character. These attributes cover various forms of highlighting of the character. For example, the character can be put in reverse video, bold, or be underlined. You can think of the attributes as the color of the ink used to draw the character.

A window always has a set of *current attributes* associated with it. The current attributes are associated with each character as it is written to the window. The current attributes can be changed with a call to **attrset(attrs)**. (Think of this as dipping the window's pen in a particular color ink.) The

names of the attributes are **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_INVIS**, and **A_UNDERLINE**. For example, to put a word in bold, the code in Figure 12-2 might be used. The word "boldface" will be shown in bold.

```
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
...
refresh();
```

Figure 12-2. Use of attributes

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, **curses** will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

One particular attribute is called *standout*. This attribute is used to make text attract the attention of the user. The particular hardware attribute used for standout varies from terminal to terminal, and is chosen to be the most visually pleasing attribute the terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or inverse video, but instead just need to highlight some text. For such applications, the **A_STANDOUT** attribute is recommended. Two convenient functions, **standout()** and **standend()** turn on and off this attribute.

Attributes can be turned on in combination. Thus, to turn on blinking bold text, use **attrset(A_BLINK|A_BOLD)**. Individual attributes can be turned on and off with **attron** and **attroff** without affecting other attributes.

For an example program using attributes, see **highlight**. The program takes a text file as input and allows embedded escape sequences to control attributes. In this example program, **\U** turns on underlining, **\B** turns on bold, and **\N** restores normal text. Note the initial call to **scrollok**. This allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **curses** will automatically scroll the terminal up a line and call **refresh**.

Highlight comes about as close to being a filter as is possible with **curses**. It is not a true filter, because **curses** must "take over" the CRT screen. In

order to determine how to update the screen, it must know what is on the screen at all times. This requires **curses** to clear the screen in the first call to **refresh**, and to know the cursor position and screen contents at all times.

12.0.4 Multiple Windows

A window is a data structure representing all or part of the CRT screen. It has room for a two dimensional array of characters, attributes for each character (a total of 16 bits per character: 7 for text and 9 for attributes) a cursor, a set of current attributes, and a number of flags. Curses provides a full screen window, called **stdscr**, and a set of functions that use **stdscr**. Another window is provided called **curscr**, representing the physical screen.

It is important to understand that a window is only a data structure. Use of more than one window does not imply use of more than one terminal, nor does it involve more than one process. A window is merely an object which can be copied to all or part of the terminal screen. The current implementation of **curses** does not allow windows which are bigger than the screen.

The programmer can create additional windows with the function **newwin(lines, cols, begin_row, begin_col)** will return a pointer to a newly created window. The window will be **lines** by **cols**, and the upper left corner of the window will be at screen position **(begin_row, begin_col)**. All operations that affect **stdscr** have corresponding functions that affect an arbitrary named window. Generally, these functions have names formed by putting a "w" on the front of the **stdscr** function, and the window name is added as the first parameter. Thus, **waddch(mywin, c)** would write the character **c** to window **mywin**. The **wrefresh(win)** function is used to flush the contents of a window to the screen.

Windows are useful for maintaining several different screen images, and alternating the user among them. Also, it is possible to subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be the more recently refreshed window.

In all cases, the non-w version of the function calls the w version of the function, using **stdscr** as the additional argument. Thus, a call to **addch(c)** results in a call to **waddch(stdscr, c)**.

The program **window** is an example of the use of multiple windows. The main display is kept in **stdscr**. When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen. A call to **wrefresh** on that window causes the window to be written over **stdscr** on the screen. Calling **refresh** on **stdscr** results in the original window being redrawn on the screen. Note the calls to **touchwin** before writing out an overlapping window. These are necessary to defeat an

optimization in **curses**. If you have trouble refreshing a new window which overlaps an old window, it may be necessary to call **touchwin** on the new window to get it completely written out.

For convenience, a set of "move" functions are also provided for most of the common functions. These result in a call to **move** before the other function. For example, **mvaddch(row, col, c)** is the same as **move(row, col); addch(c)**. Combinations, e.g. **mvwaddch(row, col, win, c)** also exist.

12.0.5 Multiple Terminals

Curses can produce output on more than one terminal at once. This is useful for single process programs that access a common database, such as multi-player games. Output to multiple terminals is a difficult business, and **curses** does not solve all the problems for the programmer. It is the responsibility of the program to determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking **$TERM** in the environment, does not work, since each process can only examine its own environment. Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. Nonetheless, a program wishing to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. However, for some applications, such as an inter-terminal communication program, or a program that takes over unused tty lines, it would be appropriate.) A typical solution requires the user logged in on each line to run a program that notifies the master program that the user is interested in joining the master program, telling it the notification program's process id, the name of the tty line and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program, and all programs exit.

Curses handles multiple terminals by always having a *current terminal*. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals have type **struct screen ***. A new terminal is initialized by calling **newterm(type, fd)**. **newterm** returns a screen reference to the terminal being set up. **type** is a character string, naming the kind of terminal being used. **fd** is a stdio file descriptor to be used for input and output to the terminal. (If only output is needed, the file can be open for output only.) This call replaces the normal call to **initscr**, which calls **newterm(getenv("TERM"), stdout)**.

To change the current terminal, call "**set_term(*sp*)**" where **sp** is the screen reference to be made current. **set_term** returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**. Options such as **cbreak** and **noecho** must be set separately for each terminal. The functions **endwin** and **refresh** must be called separately for each terminal. See Figure 12-3 for a typical scenario to output a message to each terminal.

```
for (i=0; i<nterm; i++) {
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 12-3. Sending a message to several terminals

See the sample program **two** for a full example. This program pages through a file, showing one page to the first terminal and the next page to the second terminal. It then waits for a space to be typed on either terminal, and shows the next page to the terminal typing the space. Each terminal has to be separately put into nodelay mode. Since no standard multiplexor is available in current versions of the UNIX system, it is necessary to either busy wait, or call **sleep(1);**, between each check for keyboard input. This program sleeps for a second between checks.

The **two** program is just a simple example of two terminal **curses**. It does not handle notification, as described above, instead it requires the name and type of the second terminal on the command line. As written, the command **sleep 100000** must be typed on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the second terminal.

12.0.6 Low Level Terminfo Usage

Some programs need to use lower level primitives than those offered by **curses**. For such programs, the *terminfo level* interface is offered. This interface does not manage your CRT screen, but rather gives you access to strings and capabilities which you can use yourself to manipulate the terminal.

Programmers are discouraged from using this level. Whenever possible, the higher level **curses** routines should be used. This will make your program more portable to other UNIX systems and to a wider class of terminals. Curses takes care of all the glitches and misfeatures present in physical terminals, but at the terminfo level you must deal with them yourself. Also, it cannot be guaranteed that this part of the interface will not change or be upward compatible with previous releases.

There are two circumstances when it is proper to use terminfo. The first is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second situation is when writing a filter. A typical filter does one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of terminfo is indicated.

A program writing at the terminfo level uses the framework shown in Figure 12-4.

```
•
#include <curses.h>
#include <term.h>
...
    setupterm(0, 1, 0);
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);
```

Figure 12-4. Terminfo level framework

Initialization is done by calling **setupterm**. Passing the values 0, 1, and 0 invoke reasonable defaults. If **setupterm** can't figure out what kind of terminal you are on, it will print an error message and exit. The program should call **reset_shell_mode** before it exits.

Global variables with names like **clear_screen** and **cursor_address** are defined by the call to **setupterm**. They can be output using **putp**, or also using **tputs**, which allows the programmer more control. These strings *should not* be directly output to the terminal using **printf** since they contain padding information. A program that directly outputs strings will fail on terminals that require padding, or that use the xon/xoff flow control protocol.

In the terminfo level, the higher level routines described previously are not available. It is up to the programmer to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4).

The example program **termhl** shows simple use of terminfo. It is a version of **highlight** that uses terminfo instead of **curses**. This version can be used as a filter. The strings to enter bold and underline mode, and to turn off all attributes, are used.

This program is more complex than it need be in order to illustrate some properties of terminfo. The routine **vidattr** could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did since there are several ways to change video attribute modes. This program was written to illustrate typical use of terminfo.

The function **tputs(cap, affcnt, outc)** applies padding information. Some capabilities contain strings like $<20>, which means to pad for 20 milliseconds. **tputs** generates enough pad characters to delay for the appropriate time. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention **affcnt** is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since **affcnt** is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, **affcnt** is always 1 and **outc** always just calls **putchar**. For these programs, the routine **putp(cap)** is a convenient abbreviation. **termhl** could be simplified by using **putp**.

Note also the special check for the **underline_char** capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, will output **underline_char** if necessary. Low level details such as this are precisely why the **curses** level is recommended over the terminfo level. Curses takes care of terminals with different methods of underlining and other CRT functions. Programs at the terminfo level must handle such details themselves.

12.0.7 A Larger Example

For a final example, see the program **editor**. This program is a very simple screen editor, patterned after the **vi** editor. The program illustrates how to use **curses** to write a screen editor. This editor keeps the buffer in **stdscr**

to keep the program simple – obviously a real screen editor would keep a separate data structure. Many simplifications have been made here – no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. The routine to write out the file illustrates the use of the **mvinch** function, which returns the character in a window at a given position. The data structure used here does not have a provision for keeping track of the number of characters in a line, or the number of lines in the file, so trailing blanks are eliminated when the file is written out.

The program uses built-in **curses** functions **insch**, **delch**, **insertln**, and **deleteln**. These functions behave much as the similar functions on intelligent terminals behave, inserting and deleting a character or line.

The command interpreter accepts not only ASCII characters, but also special keys. This is important – a good program will accept both. (Some editors are *modeless*, using nonprinting characters for commands. This is largely a matter of taste – the point being made here is that both arrow keys and ordinary ASCII characters should be handled.) It is important to handle special keys because this makes it easier for a new user to learn to use your program if he can use the arrow keys, instead of having to memorize that "h" means left, "j" means down, "k" means up, and "l" means right. On the other hand, not all terminals have arrow keys, so your program will be usable on a larger class of terminals if there is an ASCII character which is a synonym for each special key. Also, experienced users dislike having to move their hands from the "home row" position to use special keys, since they can work faster with alphabetic keys.

Note the call to **mvaddstr** in the input routine. **addstr** is roughly like the C **fputs** function, which writes out a string of characters. Like **fputs**, **addstr** does not add a trailing newline. It is the same as a series of calls to **addch** using the characters in the string. **mvaddstr** is the mv version of **addstr**, which moves to the given location in the window before writing.

The control-L command illustrates a feature most programs using **curses** should add. Often some program beyond the control of **curses** has written something to the screen, or some line noise has messed up the screen beyond what **curses** can keep track of. In this case, the user usually types control-L, causing the screen to be cleared and redrawn. This is done with the call to **clearok(curscr)**, which sets a flag causing the next **refresh** to first clear the screen. Then **refresh** is called to force the redraw.

Note also the call to **flash()**, which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within

earshot of the user. The routine **beep()** can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep** will flash the screen.)

Another important point is that the input command is terminated by control-D, not escape. It is very tempting to use escape as a command, since escape is one of the few special keys which is available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape ("escape sequences") to control the terminal, and have special keys that send escape sequences to the computer. If the computer sees an escape coming from the terminal, it cannot tell for sure whether the user pushed the escape key, or whether a special key was pressed. Curses handles the ambiguity by waiting for up to one second. If another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to one second for each character) until either a full special key is read, one second passes, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes **curses** to think a special key has been pressed. Also, there is a one second pause until the escape can be passed to the user program, resulting in slower response to the escape key. Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. Such programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a timeout solution. The moral is clear: when designing your program, avoid the escape key.

12.1 LIST OF ROUTINES

This section describes all the routines available to the programmer in the **curses** package. The routines are organized by function. For an alphabetical list, see **curses**(3X).

12.1.1 Structure

All programs using **curses** should include the file <**curses.h**>. This file defines several **curses** functions as macros, and defines several global variables and the datatype **WINDOW**. References to windows are always of type **WINDOW** *. Curses also defines **WINDOW** * constants **stdscr** (the standard screen, used as a default to routines expecting a window), and **curscr** (the current screen, used only for certain low level operations like clearing and redrawing a garbaged screen). Integer constants **LINES** and **COLS** are defined, containing the size of the screen. Constants **TRUE** and **FALSE** are defined, with values 1 and 0, respectively. Additional constants which are values returned from most **curses** functions are **ERR** and **OK**.

OK is returned if the function could be properly completed, and **ERR** is returned if there was some error, such as moving the cursor outside of a window.

The include file <**curses.h**> automatically includes <**stdio.h**> and an appropriate tty driver interface file, currently either <**sgtty.h***> or <**termio.h**>. Including <**stdio.h**> again is harmless but wasteful, including <**sgtty.h**> again will usually result in a fatal error.

A program using **curses** should include the loader option **-lcurses** in the makefile. This is true for both the **terminfo** level and the **curses** level. The compilation flag **-DMINICURSES** can be included if you restrict your program to a small subset of **curses** concerned primarily with screen output and optimization. The routines possible with mini-curses are listed in "Mini-Curses" under "OPERATION DETAILS."

12.1.2 Initialization

These functions are called when initializing a program.

initscr()

The first function called should always be **initscr**. This will determine the terminal type and initialize **curses** data structures. **initscr** also arranges that the first call to **refresh** will clear the screen.

endwin()

A program should always call **endwin** before exiting. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper non-visual mode, and tear down all appropriate data structures.

newterm(type, fd)

A program which outputs to more than one terminal should use **newterm** instead of **initscr**. **newterm** should be called once for each terminal. It returns a variable of type **SCREEN** * which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and a stdio file descriptor (**FILE***) for output to the terminal. The file descriptor should be open for both reading and writing if input from the terminal is desired. The program should also call **endwin** for each terminal being used (see **set_term** below). If an error occurs, the value **NULL** is returned.

---

* The driver interface <**sgtty.h**> is a tty driver interface used in other versions of the UNIX system.

set_term(new)
This function is used to switch to a different terminal. The screen reference **new** becomes the new current terminal. The previous terminal is returned by the function. All other calls affect only the current terminal.

longname()
This function returns a pointer to a static area containing a verbose description of the current terminal. It is defined only after a call to **initscr**, **newterm**, or **setupterm**.

12.1.3 Option Setting

These functions set options within **curses**. In each case, **win** is the window affected, and **bf** is a boolean flag with value **TRUE** or **FALSE** indicating whether to enable or disable the option. All options are initially **FALSE**. It is not necessary to turn these options off before calling **endwin**.

clearok(win,bf)
If set, the next call to **wrefresh** with this window will clear the screen and redraw the entire screen. If **win** is **curscr**, the next call to **wrefresh** with any window will cause the screen to be cleared. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win,bf)
If enabled, **curses** will consider using the hardware insert/delete line feature of terminals so equipped. If disabled, **curses** will never use this feature. The insert/delete character feature is always considered. Enable this option only if your application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, **curses** will redraw the changed portions of all lines that do not match the desired line.

keypad(win,bf)
This option enables the keypad of the users terminal. If enabled, the user can press a function key (such as an arrow key) and **getch** will return a single value representing the function key. If disabled, **curses** will not treat function keys specially. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will turn on the terminal keypad.

leaveok(win,bf)
Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the

update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

meta(win,bf)
If enabled, characters returned by **getch** are transmitted with all 8 bits, instead of stripping the highest bit. The value **OK** is returned if the request succeeded, the value **ERR** is returned if the terminal or system is not capable of 8-bit input.

Meta mode is useful for extending the non-text command set in applications where the terminal has a meta shift key. Curses takes whatever measures are necessary to arrange for 8-bit input. On other versions of UNIX systems, raw mode will be used. On our systems, the character size will be set to 8, parity checking disabled, and stripping of the 8th bit turned off.

Note that 8-bit input is a fragile mode. Many programs and networks only pass 7 bits. If any link in the chain from the terminal to the application program strips the 8th bit, 8-bit input is impossible.

nodelay(win,bf)
This option causes **getch** to be a non-blocking call. If no input is ready, **getch** will return –1. If disabled, **getch** will hang until a key is pressed.

intrflush(win,bf)
If this option is enabled when an interrupt key is pressed on the keyboard (interrupt, quit, suspend), all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default is for the option to be enabled. This option depends on support in the underlying teletype driver.

typeahead(fd)
Sets the file descriptor for typeahead check. **fd** should be an integer returned from **open** or **fileno**. Setting typeahead to –1 will disable typeahead check. By default, file descriptor 0 (stdin) is used. Typeahead is checked independently for each screen, and for multiple interactive terminals it should probably be set to the appropriate input for each screen. A call to **typeahead** always affects only the current screen.

scrollok(win,bf)
This option controls what happens when the cursor of a window is moved off the edge of the window, either from a newline on the bottom line, or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, **wrefresh** is called on the window, and then the physical terminal and window are scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**.

setscrreg(t,b)
wsetscrreg(win,t,b)
These functions allow the user to set a software scrolling region in a window **win** or **stdscr**. **t** and **b** are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok** are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the VT100. Only the text of the window is scrolled. If **idlok** is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.

12.1.4 Terminal Mode Setting

These functions are used to set modes in the tty driver. The initial mode usually depends on the setting when the program was called: the initial modes documented here represent the normal situation.

cbreak()
nocbreak()
These two functions put the terminal into and out of **CBREAK** mode. In this mode, characters typed by the user are immediately available to the program. When out of this mode, the teletype driver will buffer characters typed until newline is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal is not in CBREAK mode. Most interactive programs using **curses** will set this mode.

echo()
noecho()
These functions control whether characters typed by the user are echoed as typed. Initially, characters typed are echoed by the teletype driver. Authors of many interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing.

nl()
nonl()
These functions control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion.

raw()
noraw()
The terminal is placed into or out of raw mode. Raw mode is similar to cbreak mode in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, and suspend characters are passed through uninterpreted instead of

generating a signal. RAW mode also causes 8 bit input and output. The behavior of the BREAK key may be different on different systems.

resetty()
savetty()
These functions save and restore the state of the tty modes. **savetty** saves the current state in a buffer, **resetty** restores the state to what it was at the last call to **savetty**.

12.1.5 Window Manipulation

newwin(num_lines, num_cols, beg_row, beg_col)
Create a new window with the given number of lines and columns. The upper left corner of the window is at line **beg_row** column **beg_col**. If either **num_lines** or **num_cols** is zero, they will be defaulted to **LINES-beg_row** and **COLS-beg_col**. A new full-screen window is created by calling **newwin(0,0,0,0).**

newpad(num_lines, num_cols)
Creates a new *pad* data structure. A pad is like a window, except that it is not restricted by the screen size, and is not associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call **refresh** with a pad as an argument, the routines **prefresh** or **pnoutrefresh** should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

subwin(orig, num_lines, num_cols, begy, begx)
Create a new window with the given number of lines and columns. The window is at position (*begy, begx*) on the screen. (It is relative to the screen, not **orig**.) The window is made in the middle of the window **orig**, so that changes made to one window will affect both windows. When using this function, often it will be necessary to call **touchwin** before calling **wrefresh**.

delwin(win)
Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

mvwin(win, br, bc)
Move the window so that the upper left corner will be at position **(br, bc)**. If the move would cause the window to be off the screen, it is an error and the window is not moved.

touchwin(win)
Throw away all optimization information about which parts of the window have been touched, by pretending the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

overlay(win1, win2)
overwrite(win1, win2)
These functions overlay **win1** on top of **win2**; that is, all text in **win1** is copied into **win2**. The difference is that **overlay** is nondestructive (blanks are not copied) while **overwrite** is destructive.

12.1.6 Causing Output to the Terminal

refresh()
wrefresh(win)
These functions must be called to get any output on the terminal, as other routines merely manipulate data structures. **wrefresh** copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. **refresh** is the same, using **stdscr** as a default screen. Unless **leaveok** has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

doupdate()
wnoutrefresh(win)
These two functions allow multiple updates with more efficiency than **wrefresh**. To use them, it is important to understand how **curses** works. In addition to all the window structures, **curses** keeps two data structures representing the terminal screen: a *physical* screen, describing what is actually on the screen, and a *virtual* screen, describing what the programmer *wants* to have on the screen. **wrefresh** works by first copying the named window to the virtual screen **(wnoutrefresh)**, and then calling the routine to update the screen **(doupdate)**. If the programmer wishes to output several windows at once, a series of calls to **wrefresh** will result in alternating calls to **wnoutrefresh** and **doupdate**, causing several bursts of output to the screen. By calling **wnoutrefresh** for each window, it is then possible to call **doupdate** once, resulting in only one burst of output, with probably fewer total characters transmitted.

prefresh(pad,pminrow,pmincol,sminrow,smincol,smaxrow,smaxcol)
pnoutrefresh(pad,pminrow,pmincol,sminrow,smincol,smaxrow,smaxcol)
These routines are analogous to **wrefresh** and **wnoutrefresh** except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. **pminrow** and **pmincol** specify the upper left corner, in the pad, of the rectangle to be displayed. **sminrow**, **smincol**, **smaxrow**, and **smaxcol** specify the edges,

on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

12.1.7 Writing on Window Structures

These routines are used to "draw" text on windows. In all cases, a missing **win** is taken to be **stdscr**. **y** and **x** are the row and column, respectively. The upper left corner is always (0,0), not (1,1). The **mv** functions imply a call to **move** before the call to the other function.

12.1.7.1 Moving the Cursor

move(y, x)
wmove(win, y, x)
The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until **refresh** is called. The position specified is relative to the upper left corner of the window.

12.1.7.2 Writing One Character

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)
The character **ch** is put in the window at the current cursor position of the window. If **ch** is a tab, newline, or backspace, the cursor will be moved appropriately in the window. If **ch** is a different control character, it will be drawn in the ^X notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok** is enabled, the scrolling region will be scrolled up one line.

The **ch** parameter is actually an integer, not a character. Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another with **inch** and **addch**.)

12.1.7.3 Writing a String

addstr(str)
waddstr(win,str)
mvaddstr(y,x,str)
mvwaddstr(win,y,x,str)
These functions write all the characters of the null terminated character string **str** on the given window. They are identical to a series of calls to **addch**.

12.1.7.4 Clearing Areas of the Screen

erase()
werase(win)
These functions copy blanks to every position in the window.

clear()
wclear(win)
These functions are like **erase** and **werase** but they also call **clearok**, arranging that the screen will be cleared on the next call to **refresh** for that window.

clrtobot()
wclrtobot(win)
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

clrtoeol()
wclrtoeol(win)
The current line to the right of the cursor is erased.

12.1.7.5 Inserting and Deleting Text

delch()
wdelch(win)
mvdelch(y,x)
mvwdelch(win,y,x)
The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. This does not imply use of the hardware delete character feature.

deleteln()
wdeleteln(win)
The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not imply use of the hardware delete line feature.

insch(c)
winsch(win, c)
mvinsch(y,x,c)
mvwinsch(win,y,x,c)
The character **c** is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not imply use of the hardware insert character feature.

insertln()
winsertln(win)
A blank line is inserted above the current line. The bottom line is lost. This does not imply use of the hardware insert line feature.

12.1.7.6 Formatted Output

printw(fmt, args)
wprintw(win, fmt, args)
mvprintw(y, x, fmt, args)
mvwprintw(win, y, x, fmt, args)
These functions correspond to **printf**. The characters which would be output by **printf** are instead output using **waddch** on the given window.

12.1.7.7 Miscellaneous

box(win, vert, hor)
A box is drawn around the edge of the window. **vert** and **hor** are the characters the box is to be drawn with.

scroll(win)
The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

12.1.8 Input from a Window

getyx(win,y,x)
The cursor position of the window is placed in the two integer variables **y** and **x**. Since this is a macro, no & is necessary.

inch()
winch(win)
mvinch(y,x)
mvwinch(win,y,x)
The character at the current position in the named window is returned. If any attributes are set for that position, their values will be or-ed into the value returned. The predefined constants **A_ATTRIBUTES** and **A_CHARTEXT** can be used with the & operator to extract the character or attributes alone.

12.1.9 Input from the Terminal

getch()
wgetch(win)
mvgetch(y,x)
mvwgetch(win,y,x)
A character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value –1 is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**, this will be after one character, or after the first newline.

If **keypad** mode is enabled, and a function key is pressed, the code for that function key will be returned instead of the raw characters. Possible function keys are defined with integers beginning with 0401, whose names begin with **KEY_**. These are listed in "Input" under "INTRODUCTION." If a character is received that could be the beginning of a function key (such as escape), **curses** will set a 1-second timer. If the remainder of the sequence does not come in within 1 second, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a one second delay after a user presses the escape key. (Use by a programmer of the escape key for a single character function is discouraged.)

getstr(str)
wgetstr(win,str)
mvgetstr(y,x,str)
mvwgetstr(win,y,x,str)
A series of calls to **getch** is made, until a newline is received. The resulting value is placed in the area pointed at by the character pointer **str**. The users' erase and kill characters are interpreted.

scanw(fmt, args)
wscanw(win, fmt, args)
mvscanw(y, x, fmt, args)
mvwscanw(win, y, x, fmt, args)
This function corresponds to **scanf**. **wgetstr** is called on the window, and the resulting line is used as input for the scan.

12.1.10 Video Attributes

attroff(at)
wattroff(win, attrs)
attron(at)
wattron(win, attrs)
attrset(at)
wattrset(win, attrs)
standout()
standend()
wstandout(win)
wstandend(win)
These functions set the *current attributes* of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, and **A_UNDERLINE**. These constants are defined in <**curses.h**> and can be combined with the C | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

attrset(at) sets the current attributes of the given window to **at**. **attroff**(at) turns off the named attributes without affecting any other attributes. **attron**(at) turns on the named attributes without affecting any others. **standout** is the same as **attron**(A_STANDOUT) **standend** is the same as **attrset**(0), that is, it turns off all attributes.

12.1.11 Bells and Flashing Lights

beep()
flash()
These functions are used to signal the programmer. **beep** will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash** will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

12.1.12 Portability Functions

These functions do not directly involve terminal dependent character output but tend to be needed by programs that use **curses**. Unfortunately, their

† UNIX is a Trademark of Bell Laboratories.

implementation varies from one version of UNIX† to another. They have been included here to enhance the portability of programs using **curses**.

baudrate()
baudrate returns the output speed of the terminal. The number returned is the integer baud rate, for example, 9600, rather than a table index such as **B9600**.

erasechar()
The erase character chosen by the user is returned. This is the character typed by the user to erase the character just typed.

killchar()
The line kill character chosen by the user is returned. This is the character typed by the user to forget the entire line being typed.

flushinp()
flushinp throws away any typeahead that has been typed by the user and has not yet been read by the program.

12.1.13 Delays

These functions are highly unportable, but are often needed by programs that use **curses**, especially real time response programs. Some of these functions require a particular operating system or a modification to the operating system to work. In all cases, the routine will compile and return an error status if the requested action is not possible. It is recommended that programmers avoid use of these functions if possible.

draino(ms) The program is suspended until the output queue has drained enough to complete in **ms** additional milliseconds. Thus, **draino**(50) at 1200 baud would pause until there are no more than 6 characters in the output queue, because it would take 50 milliseconds to output the additional 6 characters. The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the ioctls needed to implement **draino**, the value ERR is returned; otherwise, OK is returned.

napms(ms) This function suspends the program for **ms** milliseconds. It is similar to **sleep** except with higher resolution. The resolution actually provided will vary with the facilities available in the operating system, and often a change to the operating system will be necessary to produce good results. If resolution of at least .1 second is not possible, the routine will round to the next higher second, call sleep, and return ERR. Otherwise, the value OK is returned. Often the resolution provided is 1/60th second.

12.1.14 Lower Level Functions

These functions are provided for programs not needing the screen optimization capabilities of **curses**. Programs are discouraged from working

at this level, since they must handle various glitches in certain terminals. However, a program can be smaller if it only brings in the low level routines.

12.1.14.1 Cursor Motion

mvcur(oldrow, oldcol, newrow, newcol)

This routine optimally moves the cursor from (oldrow, oldcol) to (newrow, newcol). The user program is expected to keep track of the current cursor position. Note that unless a full screen image is kept, **curses** will have to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion. For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over, but if **curses** does not have access to the screen image, it doesn't know what these characters are.

12.1.14.2 Terminfo Level

These routines are called by low level programs that need access to specific capabilities of **terminfo**. A program working at this level should include both <**curses.h**> and <**term.h**> in that order. After a call to **setupterm**, the capabilities will be available with macro names defined in <**term.h**>. See **terminfo**(4) for a detailed description of the capabilities.

Boolean valued capabilities will have the value 1 if the capability is present, 0 if it is not. Numeric capabilities have the value –1 if the capability is missing, and have a value at least 0 if it is present. String capabilities (both those with and without parameters) have the value **NULL** if the capability is missing, and otherwise have type **char** * and point to a character string containing the capability. The special character codes involving the \ and ^ characters (such as \r for return, or ^A for control A) are translated into the appropriate ASCII characters. Padding information (of the form $<time>) and parameter information (beginning with %) are left uninterpreted at this stage. The routine **tputs** interprets padding information, and **tparm** interprets parameter information.

If the program only needs to handle one terminal, the definition **–DSINGLE** can be passed to the C compiler, resulting in static references to capabilities instead of dynamic references. This can result in smaller code, but prevents use of more than one terminal at a time. Very few programs use more than one terminal, so almost all programs can use this flag.

setupterm(term, filenum, errret)

This routine is called to initialize a terminal. **term** is the character string representing the name of the terminal being used. **filenum** is the UNIX file descriptor of the terminal being used for output. **errret** is a pointer to an integer, in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or –1 (some problem locating the **terminfo** database).

The value of **term** can be given as 0, which will cause the value of TERM in the environment to be used. The **errret** pointer can also be given as 0, meaning no error code is wanted. If **errret** is defaulted, and something goes wrong, **setupterm** will print an appropriate error message and exit, rather than returning. Thus, a simple program can call **setupterm(0, 1, 0)** and not worry about initialization errors.

If the variable TERMINFO is set in the environment to a path name, **setupterm** will check for a compiled **terminfo** description of the terminal under that path, before checking /etc/term. Otherwise, only /etc/term is checked.

setupterm will check the tty driver mode bits, using **filenum**, and change any that might prevent the correct operation of other low level routines. Currently, the mode that expands tabs into spaces is disabled, because the tab character is sometimes used for different functions by different terminals. (Some terminals use it to move right one space. Others use it to address the cursor to row or column 9.) If the system is expanding tabs, **setupterm** will remove the definition of the **tab** and **backtab** functions, making the assumption that since the user is not using hardware tabs, they may not be properly set in the terminal. Other system dependent changes, such as disabling a virtual terminal driver, may be made here.

As a side effect, **setupterm** initializes the global variable **ttytype**, which is an array of characters, to the value of the list of names for the terminal. This list comes from the beginning of the **terminfo** description.

After the call to **setupterm**, the global variable **cur_term** is set to point to the current structure of terminal capabilities. By calling **setupterm** for each terminal, and saving and restoring **cur_term**, it is possible for a program to use two or more terminals at once.

The mode that turns newlines into CRLF on output is not disabled. Programs that use **cursor_down** or **scroll_forward** should avoid these capabilities if their value is linefeed unless they disable this mode. **setupterm** calls **reset_prog_mode** after any changes it makes.

reset_prog_mode()
reset_shell_mode()
def_prog_mode()
def_shell_mode()
These routines can be used to change the tty modes between the two states: *shell* (the mode they were in before the program was started) and *program* (the mode needed by the program). **def_prog_mode** saves the current terminal mode as program mode. **setupterm** and **initscr** call def_shell_mode automatically. **reset_prog_mode** puts the terminal into program mode, and **reset_shell_mode** puts the terminal into normal mode.

A typical calling sequence is for a program to call **initscr** (or **setupterm** if a **terminfo** level program), then to set the desired program mode by calling routines such as **cbreak** and **noecho**, then to call **def_prog_mode** to save the current state. Before a shell escape or control-Z suspension, the program should call **reset_shell_mode**, to restore normal mode for the shell. Then, when the program resumes, it should call **reset_prog_mode**. Also, all programs must call **reset_shell_mode** before they exit. (The higher level routine **endwin** automatically calls **reset_shell_mode**.)

Normal mode is stored in **cur_term->Ottyb**, and program mode is in **cur_term->Nttyb**. These structures are both of type **SGTTYB** (which varies depending on the system). Currently the possible types are **struct sgttyb** (on some other systems) and **struct termio** (on this version of the UNIX system). **def_prog_mode** should be called to save the current state in **Nttyb**.

vidputs(newmode, putc)
newmode is any combination of attributes, defined in <**curses.h**>. **putc** is a putchar-like function. The proper string to put the terminal in the given video mode is output. The previous mode is remembered by this routine. The result characters are passed through **putc**.

vidattr(newmode)
The proper string to put the terminal in the given video mode is output to **stdout**.

tparm(instring, p1, p2, p3, p4, p5, p6, p7, p8, p9)
tparm is used to instantiate a parameterized string. The character string returned has the given parameters applied, and is suitable for **tputs**. Up to 9 parameters can be passed, in addition to the parameterized string.

tputs(cp, affcnt, outc)
A string capability, possibly containing padding information, is processed. Enough padding characters to delay for the specified time replace the padding specification, and the resulting string is passed, one character at a time, to the routine **outc**, which should expect one character parameter. (This routine often just calls **putchar.**) **cp** is the capability string. **affcnt** is the number of units affected by the capability, which varies with the particular capability. (For example, the **affcnt** for **insert_line** is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal.) **affcnt** is used by the padding information of some terminals as a multiplication factor. If the capability does not have a factor, the value 1 should be passed.

putp(str)
This is a convenient function to output a capability with no **affcnt**. The string is output to **putchar** with an **affcnt** of 1. It can be used in simple applications that do not need to process the output of **tputs**.

delay_output(ms)
A delay is inserted into the output stream for the given number of milliseconds. The current implementation inserts sufficient pad characters for the delay. This should not be used in place of a high resolution sleep, but rather for delay effects in the output. Due to buffering in the system, it is unlikely that this call will result in the process actually sleeping. Since large numbers of pad characters can be output, it is recommended that **ms** not exceed 500.

12.2 OPERATION DETAILS

These paragraphs describe many of the details of how the **curses** and terminfo package operates.

12.2.1 Insert and Delete Line and Character
The algorithm used by **curses** takes into account insert and delete line and character functions, if available, in the terminal. Calling the routine

idlok(stdscr, TRUE);

will enable insert/delete line. By default, **curses** will not use insert/delete line. This was not done for performance reasons, since there is no speed penalty involved. Rather, experience has shown that some programs do not need this facility, and that if **curses** uses insert/delete line, the result on the screen can be visually annoying. Since many simple programs using **curses** do not need this, the default is to avoid insert/delete line. Insert/delete character is always considered.

12.2.2 Additional Terminals
Curses will work even if absolute cursor addressing is not possible, as long as the cursor can be moved from any location to any other location. It considers local motions, parameterized motions, home, and carriage return.

Curses is aimed at full duplex, alphanumeric, video terminals. No attempt is made to handle half-duplex, synchronous, hard copy, or bitmapped terminals. Bitmapped terminals can be handled by programming the bitmapped terminal to emulate an ordinary alphanumeric terminal. This does not take advantage of the bitmap capabilities, but it is the fundamental nature of **curses** to deal with alphanumeric terminals.

The **curses** handles terminals with the "magic cookie glitch" in their video attributes. The term "magic cookie" means that a change in video attributes is implemented by storing a "magic cookie" in a location on the screen. This "cookie" takes up a space, preventing an exact implementation of what

the programmer wanted. Curses takes the extra space into account, and moves part of the line to the right, as necessary. In some cases, this will unavoidably result in losing text from the right hand edge of the screen. Advantage is taken of existing spaces.

12.2.3 Multiple Terminals

Some applications need to display text on more than one terminal, controlled by the same process. Even if the terminals are of different types, **curses** can handle this.

All information about the current terminal is kept in a global variable

struct screen *SP;

Although the screen structure is hidden from the user, the C compiler will accept declarations of variables which are pointers. The user program should declare one screen pointer variable for each terminal it wishes to handle. The routine

struct screen *
newterm(type, fd)

will set up a new terminal of the given terminal type which does output on file descriptor fd. A call to **initscr** is essentially **newterm(getenv("TERM"),stdout)**. A program wishing to use more than one terminal should use **newterm** for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call

set_term(term)

The old value of SP will be returned. The programmer should not assign directly to SP because certain other global variables must also be changed.

All **curses** routines always affect the current terminal. To handle several terminals, switch to each one in turn with **set_term**, and then access it. Each terminal must be set up with **newterm**, and closed down with **endwin**.

12.2.4 Video Attributes

Video attributes can be displayed in any combination on terminals with this capability. They are treated as an extension of the standout capability, which is still present.

Each character position on the screen has 16 bits of information associated with it. Seven of these bits are the character to be displayed, leaving separate bits for nine video attributes. These bits are used for standout, underline, reverse video, blink, dim, bold, blank, protect, and alternate character set. Standout is taken to be whatever highlighting works best on the terminal, and should be used by any program that does not need

specific or combined attributes. Underlining, reverse video, blink, dim, and bold are the usual video attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the particular terminal. The use of these last three bits is subject to change and not recommended. Note also that not all terminals implement all attributes – in particular, no current terminal implements both dim and bold.

The routines to use these attributes include

attrset(attrs)
attron(attrs)
attroff(attrs)
standout()
standend()
wattrset(win, attrs)
wattron(win, attrs)
wattroff(win, attrs)
wstandout(win)
wstandend(win)

Attributes, if given, can be any combination of **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**. These constants, defined in **curses**.h, can be combined with the C | (or) operator to get multiple attributes. **attrset** sets the current attributes to the given **attrs**; **attron** turns on the given **attrs** in addition to any attributes that are already on; **attroff** turns off the given attributes, without affecting any others. **standout** and **standend** are equivalent to **attron(A_STANDOUT)** and **attrset(A_NORMAL)**.

If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place. If the terminal has no highlighting at all, all attributes will be ignored.

12.2.5 Special Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differs from terminal to terminal. **Curses** allows the programmer to handle these keys.

A program using special keys should turn on the keypad by calling

keypad(stdscr, TRUE)

at initialization. This will cause special characters to be passed through to the program by the function **getch**. These keys have constants which are listed in "Input" under "INTRODUCTION." They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program using special keys should avoid using the **escape** key, since most sequences start with escape, creating an ambiguity. **Curses** will set a one second alarm to deal with this ambiguity, which will cause delayed

response to the escape key. It is a good idea to avoid escape in any case, since there is eventually pressure for nearly *any* screen oriented program to accept arrow key input.

12.2.6 Scrolling Region

There is a programmer accessible scrolling region. Normally, the scrolling region is set to the entire window, but the calls

setscrreg(top, bot)
wsetscrreg(win, top, bot)

set the scrolling region for **stdscr** or the given window to any combination of top and bottom margins. When scrolling past the bottom margin of the scrolling region, the lines in the region will move up one line, destroying the top line of the region. If scrolling has been enabled with **scrollok**, scrolling will take place only within that window. Note that the scrolling region is a software feature, and only causes a window data structure to scroll. This may or may not translate to use of the hardware scrolling region feature of a terminal, or insert/delete line.

12.2.7 Mini-Curses

Curses copies from the current window to an internal screen image for every call to **refresh**. If the programmer is only interested in screen output optimization, and does not want the windowing or input functions, an interface to the lower level routines is available. This will make the program somewhat smaller and faster. The interface is a subset of full **curses**, so that conversion between the levels is not necessary to switch from mini-curses to full **curses**.

The following functions of **curses** and terminfo are available to the user of minicurses:

addch(ch)	addstr(str)	attroff(at)	attron(at)
attrset(at)	clear()	erase()	initscr
move(y, x)	mvaddch(y,x,ch)	mvaddstr(y,x,str)	newterm
refresh()	standend()	standout()	

The following functions of **curses** and terminfo are *not* available to the user of minicurses:

box	clrtobot	clrtoeol	delch
deleteln	delwin	getch	getstr
inch	insch	insertln	longname
makenew	mvdelch	mvgetch	mvgetstr
mvinch	mvinsch	mvprintw	mvscanw
mvwaddch	mvwaddstr	mvwdelch	mvwgetch
mvwgetstr	mvwin	mvwinch	mvwinsch
mvwprintw	mvwscanw	newwin	overlay
overwrite	printw	putp	scanw
scroll	setscrreg	subwin	touchwin
vidattr	waddch	waddstr	wclear
wclrtobot	wclrtoeol	wdelch	wdeleteln
werase	wgetch	wgetstr	winsch
winsertln	wmove	wprintw	wrefresh
wscanw	wsetscrreg		

The subset mainly requires the programmer to avoid use of more than the one window **stdscr**. Thus, all functions beginning with "w" are generally undefined. Certain high level functions that are convenient but not essential are also not available, including **printw** and **scanw**. Also, the input routine **getch** cannot be used with mini-curses. Features implemented at a low level, such as use of hardware insert/delete line and video attributes, are available in both versions. Also, mode setting routines such as **crmode** and **noecho** are allowed.

To access mini-curses, add **-DMINICURSES** to the CFLAGS in the makefile. If routines are requested that are not in the subset, the loader will print error messages such as

Undefined:
m_getch
m_waddch

to tell you that the routines **getch** and **waddch** were used but are not available in the subset. Since the preprocessor is involved in the implementation of mini-curses, the entire program must be recompiled when changing from one version to the other.

12.2.8 TTY Mode Functions

In addition to the save/restore routines **savetty()** and **resetty()**, standard routines are available for going into and out of normal tty mode. These routines are **resetterm()**, which puts the terminal back in the mode it was in when **curses** was started; **fixterm()**, which undoes the effects of **resetterm**,

that is, restores the "current **curses** mode"; and **saveterm()**, which saves the current state to be used by **fixterm()**. **endwin** automatically calls **resetterm**, and the routine to handle control-Z (on other systems that have process control) also uses **resetterm** and **fixterm**. Programmers should use these routines before and after shell escapes, and also if they write their own routine to handle control-Z. These routines are also available at the *terminfo* level.

12.2.9 Typeahead Check

If the user types something during an update, the update will stop, pending a future update. This is useful when the user hits several keys, each of which causes a good deal of output. For example, in a screen editor, if the user presses the "forward screen" key, which draws the next screen full of text, several times rapidly, rather than drawing several screens of text, the updates will be cut short, and only the last screen full will actually be displayed. This feature is automatic and cannot be disabled. The feature only works on versions of the UNIX system with the necessary support in the operating system.

12.2.10 getstr

No matter what the setting of *echo* is, strings typed in here are echoed at the current cursor location. The users erase and kill characters are understood and handled. This makes it unnecessary for an interactive program to deal with erase, kill, and echoing when the user is typing a line of text.

12.2.11 longname

The **longname** function does not need any arguments. It returns a pointer to a static area containing the actual long name of the terminal.

12.2.12 Nodelay Mode

The call

nodelay(stdscr, TRUE)

will put the terminal in "nodelay mode". While in this mode, any call to **getch** will return –1 if there is nothing waiting to be read immediately. This is useful for writing programs requiring "real time" behavior where the users watch action on the screen and press a key when they want something to happen. For example, the cursor can be moving across the screen, in real time. When it reaches a certain point, the user can press an arrow key to change direction at that point.

12.2.13 Portability

Several useful routines are provided to improve portability. The implementation of these routines is different from system to system, and the differences can be isolated from the user program by including them in **curses.**

Functions **erasechar()** and **killchar()** return the characters which erase one character, and kill the entire input line, respectively. The function **baudrate()** will return the current baud rate, as an integer. (For example, at 9600 baud, the integer 9600 will be returned, not the value **B9600** from <**sgtty.h**>.) The routine **flushinp()** will cause all typeahead to be thrown away.

13. CURSES EXAMPLE

The following examples are provided to demonstrate uses of **curses**. They are for illustration purposes only. A good programmer would expand the programs presented here before using them.

13.1 EXAMPLE PROGRAM 'editor'

```
/*
 * editor: A screen-oriented editor.  The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr itself to simplify
 * the program.
 */

#include <curses.h>

#define CTRL(c) ('c' & 037)

main(argc, argv)
char **argv;
{
        int i, n, l;
        int c;
        FILE *fd;

        if (argc != 2) {
                fprintf(stderr, "Usage: edit file0);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL) {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
```

```
        while ((c = getc(fd)) != EOF)
                addch(c);
        fclose(fd);

        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l=0; l<23; l++) {
                n = len(l);
                for (i=0; i<n; i++)
                        putc(mvinch(l, i), fd);
                putc('0, fd);
        }
        fclose(fd);

        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS-1;

        while (linelen >=0 && mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;

        for (;;) {
                move(row, col);
                refresh();
                c = getch();
                switch (c) {    /* Editor commands */
```

```
/* hjkl and arrow keys: move cursor */
/* in direction indiated */
case 'h':
case KEY_LEFT:
     if (col > 0)
          col--;
     break;

case 'j':
case KEY_DOWN:
     if (row < LINES-1)
          row++;
     break;

case 'k':
case KEY_UP:
     if (row > 0)
          row--;
     break;

case 'l':
case KEY_RIGHT:
     if (col < COLS-1)
          col++;
     break;

/* i: enter input mode */
case KEY_IC:
case 'i':
     input();
     break;

/* x: delete current character */
case KEY_DC:
case 'x':
     delch();
     break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
     move(++row, col=0);
     insertln();
     input();
```

```
                break;

        /* d: delete current line */
        case KEY_DL:
        case 'd':
                deleteln();
                break;

        /* ^L: redraw screen */
        case KEY_CLEAR:
        case CTRL(L):
                clearok(curscr);
                refresh();
                break;

        /* w: write and quit */
        case 'w':
                return;

        /* q: quit without writing */
        case 'q':
                endwin();
                exit(1);
        default:
                flash();
                break;
        }
    }
}

/*
 * Insert mode: accept characters and insert them.
 *  End with ^D or EIC
 */
input()
{
    int c;

    standout();
    mvaddstr(LINES-1, COLS-20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;) {
```

```
            c = getch();
            if (c == CTRL(D) | c == KEY_EIC)
                break;
            insch(c);
            move(row, ++col);
            refresh();
        }
        move(LINES-1, COLS-20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

13.2 EXAMPLE PROGRAM 'highlight'

```
/*
 * highlight: a program to turn U, B, and
 * N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */
#include <curses.h>

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;

    if (argc != 2) {
        fprintf(stderr, "Usage: highlight file0);
        exit(1);
    }

    fd = fopen(argv[1], "r");
    if (fd == NULL) {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
        if (c == '\') {
            c2 = getc(fd);
            switch (c2) {
            case 'B':
                attrset(A_BOLD);
                continue;
            case 'U':
                attrset(A_UNDERLINE);
                continue;
            case 'N':
                attrset(0);
```

```
                    continue;
                }
                addch(c);
                addch(c2);
            }
            else
                addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

13.3 EXAMPLE PROGRAM 'scatter'

```
/*
 *    SCATTER. This program takes the first
 *    23 lines from the standard
 *    input and displays them on the
 *    VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */

main()
{
      register int row=0,col=0;
      register char c;
      int char_count=0;
      long t;
      char buf[BUFSIZ];

      initscr();
      for(row=0;row<MAXLINES;row++)
            for(col=0;col<MAXCOLS;col++)
                  s[row][col]=' ';

      row = 0;
      /* Read screen in */
      while( (c=getchar()) != EOF && row < LINES ) {
            if(c != '0) {
                  /* Place char in screen array */
                  s[row][col++] = c;
                  if(c != ' ')
                        char_count++;
            } else {
                  col=0;
                  row++;
            }
      }

      time(&t); /* Seed the random number generator */
      srand((int)(t&0177777L));
```

```
                    continue;
                }
                addch(c);
                addch(c2);
            }
            else
                addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

13.3 EXAMPLE PROGRAM 'scatter'

```
/*
 *    SCATTER. This program takes the first
 *    23 lines from the standard
 *    input and displays them on the
 *    VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */

main()
{
      register int row=0,col=0;
      register char c;
      int char_count=0;
      long t;
      char buf[BUFSIZ];

      initscr();
      for(row=0;row<MAXLINES;row++)
            for(col=0;col<MAXCOLS;col++)
                  s[row][col]=' ';

      row = 0;
      /* Read screen in */
      while( (c=getchar()) != EOF && row < LINES ) {
            if(c != '0) {
                  /* Place char in screen array */
                  s[row][col++] = c;
                  if(c != ' ')
                        char_count++;
            } else {
                  col=0;
                  row++;
            }
      }

      time(&t); /* Seed the random number generator */
      srand((int)(t&0177777L));
```

```
                    continue;
                }
                addch(c);
                addch(c2);
            }
            else
                addch(c);
        }
        fclose(fd);
        refresh();
        endwin();
        exit(0);
}
```

13.3 EXAMPLE PROGRAM 'scatter'

```
/*
 *    SCATTER. This program takes the first
 *    23 lines from the standard
 *    input and displays them on the
 *    VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */

main()
{
      register int row=0,col=0;
      register char c;
      int char_count=0;
      long t;
      char buf[BUFSIZ];

      initscr();
      for(row=0;row<MAXLINES;row++)
            for(col=0;col<MAXCOLS;col++)
                  s[row][col]=' ';

      row = 0;
      /* Read screen in */
      while( (c=getchar()) != EOF && row < LINES ) {
            if(c != '0) {
                  /* Place char in screen array */
                  s[row][col++] = c;
                  if(c != ' ')
                        char_count++;
            } else {
                  col=0;
                  row++;
            }
      }

      time(&t); /* Seed the random number generator */
      srand((int)(t&0177777L));
```

```
        while(char_count) {
                row=rand() % LINES;
                col=(rand()>>2) % COLS;
                if(s[row][col] != ' ')
                {
                        move(row, col);
                        addch(s[row][col]);
                        s[row][col]=EOF;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

13.4 EXAMPLE PROGRAM 'show'

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if(argc != 2)
    {
        fprintf(stderr,"usage: %s file0, argv[0]);
        exit(1);
    }
    if((fd=fopen(argv[1],"r")) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for(line=0; line<LINES; line++)
        {
            if(fgets(linebuf, sizeof linebuf, fd) == NULL)
            {
                clrtobot();
                done();
            }
            move(line, 0);
            printw("%s", linebuf);
        }
```

```
            refresh();
            if(getch() == 'q')
                done();
        }
}


void
done()
{
        move(LINES-1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

13.5 EXAMPLE PROGRAM 'termhl'

```
/*
 * A terminfo level version of highlight.
 */
#include <curses.h>
#include <term.h>

int ulmode = 0;          /* Currently underlining */

main(argc, argv)
char **argv;
{
     FILE *fd;
     int c, c2;
     int outch();

     if (argc > 2) {
          fprintf(stderr, "Usage: termhl [file]0);
          exit(1);
     }

     if (argc == 2) {
          fd = fopen(argv[1], "r");
          if (fd == NULL) {
               perror(argv[1]);
               exit(2);
          }
     } else {
          fd = stdin;
     }

     setupterm(0, 1, 0);

     for (;;) {
          c = getc(fd);
          if (c == EOF)
               break;
          if (c == '\') {
               c2 = getc(fd);
               switch (c2) {
               case 'B':
                    tputs(enter_bold_mode, 1, outch);
                    continue;
               case 'U':
```

```
                    tputs(enter_underline_mode, 1, outch);
                    ulmode = 1;
                    continue;
               case 'N':
                    tputs(exit_attribute_mode, 1, outch);
                    ulmode = 0;
                    continue;
               }
               putch(c);
               putch(c2);
          }
          else
               putch(c);
     }
     fclose(fd);
     fflush(stdout);
     resetterm();
     exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
int c;
{
     outch(c);
     if (ulmode && underline_char) {
          outch('
)
;
          tputs(underline_char, 1, outch);
     }
}

/*
 * Outchar is a function version of putchar that can be passed to
 * tputs as a routine to call.
 */
outch(c)
int c;
{
     putchar(c);
}
```

13.6 EXAMPLE PROGRAM 'two'

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
    int done();
    int c;

    if (argc != 4) {
        fprintf(stderr, "Usage: two othertty otherttytype inputfile0);
        exit(1);
    }

    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "w+");
    signal(SIGINT, done);   /* die gracefully */

    me = newterm(getenv("TERM"), stdout);/* initialize my tty */
    you = newterm(argv[2], fdyou);/* Initialize his terminal */

    set_term(me);           /* Set modes for my terminal */
    noecho();           /* turn off tty echo */
    cbreak();           /* enter cbreak mode */
    nonl();             /* Allow linefeed */
    nodelay(stdscr,TRUE); /* No hang on input */

    set_term(you);          /* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);

    /* Dump second screen full on his terminal */
```

```
        dump_page(you);

        for (;;) {              /* for each screen full */
                set_term(me);
                c = getch();
                if (c == 'q')           /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(me);

                set_term(you);
                c = getch();
                if (c == 'q')           /* wait for user to read it */
                        done();
                if (c == ' ')
                        dump_page(you);
                sleep(1);
        }
}

dump_page(term)
struct screen *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line=0; line<LINES-1; line++) {
                if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
                        clrtobot();
                        done();
                }
                mvprintw(line, 0, "%s", linebuf);
        }
        standout();
        mvprintw(LINES-1, 0, "--More--");
        standend();
        refresh();      /* sync screen */
}

/*
 * Clean up and exit.
 */
done()
```

```
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES-1,0);  /* to lower left corner */
    clrtoeol();       /* clear bottom line */
    refresh();        /* flush out everything */
    endwin();         /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES-1,0);  /* to lower left corner */
    clrtoeol();       /* clear bottom line */
    refresh();        /* flush out everything */
    endwin();         /* curses cleanup */

    exit(0);
}
```

13.7 EXAMPLE PROGRAM 'window'

```
#include <curses.h>

WINDOW *cmdwin;

main()
{
    int i, c;
    char buf[120];

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
    for (i=0; i<LINES; i++)
        mvprintw(i, 0, "This is line %d of stdscr", i);

    for (;;) {
        refresh();
        c = getch();
        switch (c) {
        case 'c':  /* Enter command from keyboard */
            werase(cmdwin);
            wprintw(cmdwin, "Enter command:");
            wmove(cmdwin, 2, 0);
            for (i=0; i<COLS; i++)
                waddch(cmdwin, '-');
            wmove(cmdwin, 1, 0);
            touchwin(cmdwin);
            wrefresh(cmdwin);
            wgetstr(cmdwin, buf);
            touchwin(stdscr);
            /*
             * The command is now in buf.
             * It should be processed here.
```

```
                */
                break;
            case 'q':
                endwin();
                exit(0);
            }
        }
}
```