THE JOVIAL CHECKER
AN AUTOMATIC CHECKOUT SYSTEM
FOR HIGHER LEVEL LANGUAGE PROGRAMS

Mildred Wilkerson
System Development Corporation
Paramus, N. J.

Knowledge of specific machine language is still required for checkout of higher level programs. Consequently, several potential advantages of higher level programming languages have not materialized. These include shorter and less technical programming training, and faster program checkout.

JOVIAL is a higher level programming language. The "item" is its basic unit of data. The Checker is a program which executes translated JOVIAL programs and selectively records test results in JOVIAL language. Most non-essential information is eliminated from printouts. Check of actual results against expected results may be done automatically.

The Checker is part of a utility system which includes a Compiler and the Checker. The Compiler checks legality of JOVIAL statements, translates to binary code for a specific machine, and produces a JOVIAL program listing.

The Checker operates the object program and "snaps" every modified value of selected items. Basis of selection may be items of interest to the programmer, items whose final values deviate from programmer-supplied, expected values, or both. Each modified item value is printed with the statement which modified its value at that point and an associated label. Final values only of any or all tables may also be recorded.

Introduction

Although higher level languages have been in use scarcely two years, the contributions of FORTRAN, COBOL, JOVIAL, ALTAC and other such languages are now beginning to be realized by their users. Despite continuing machine obsolescence, the problem of program obsolescence due to the differing languages of various computers is now soluble with higher level language programs and compilers.

Most programmers observe that, while overall output may not yet reflect the speed-up, coding with a higher level language is considerably faster than coding in machine language. More time is now available for such problematic areas as problem analysis, program design, and modifications resulting from system design changes.

The pitfalls leading to trivial, clerical type coding errors also have been radically reduced by higher level language. This is generally attributed to the sheer reduction in the number of higher level language instructions required to program a given problem, to the more flexible format of instructions, and to the greater readibility of the language.

On the other hand, one major, potential advantage of higher level languages has not been realized. This is the elimination of machine language as a requisite.

The Problem and Its Consequences

Higher level languages have not yet replaced machine languages for a single programmer. The expectation in this respect was that machine languages could be omitted from the training of all programmers except those involved in programming and maintaining compilers and special utility programs. The existing situation, however, is that programmers must still be taught the symbolic language of at least one specific machine, as well as the new higher level language. Consequently programming training today is more time-consuming and more complicated than ever before.

Closely associated was the hope that higher level languages would further progress toward a common communication link between man and machines, between non-specialists and computer specialists, between management and programmers. One of the deterrents toward this goal is that while such a language may exist, conventional training is still dominated by machine languages. This training is neither appealing nor expedient for management and non-specialists. It is time-consuming and tedious. It requires a capacity for rote memorization and a fastidiousness for clerical detail. The very technical nature of machine language furthermore restricts the type of people who may be selected for programmer training, and may, in fact, discourage many creative people from entering this profession.

It is not resistance to higher level languages which has prevented relinquishment of machine languages from programmer-training. It is the fact that no method had been devised to produce test results from higher level language programs that did not depend upon a thorough knowledge of machine language by individual programmers.

The JOVIAL Checker is designed to solve this technical problem. Its consequences, or any

similar solution, cannot be displayed as a cure for the problems just mentioned. It can, however, be regarded as one stepping stone toward full realization of the advantages of higher level languages. To our knowledge, it is the first utility program designed to eliminate the last remaining traces of machine language from higher level language programming.

The Problems at SDC

The problems that gave rise to the Checker were much more modest. Program checkout has long been one of the most time-consuming headaches of the programming field. While higher level languages have significantly reduced the number of errors made in the original program, virtually no time has been saved in debugging and checking out programs.

Professor Wrubel[1] cited the problem in apt words when discussing one of the better known higher languages. "It is a frustrating thing," he writes, 'to have computed the answers correctly only to find them printed on the output page in an all but incomprehensible jumble."

A survey of the practices within the field yielded no satisfactory solution. Generally, test results are printed in the machine language format. Instructions are sometimes traced, but this produces stacks of printout paper with no clue to the origins of errors. An individual programmer with a great deal of foresight may leave holes in his program for insertion of instructions that could produce test results in any format, provided these were prepared by the programmers and later deleted from his program.

Due to the efforts of Jules Schwartz and others at System Development Corporation, we have been programming in a higher level language called JOVIAL for well over a year. Various utility programs for program checkout have been developed, but none embodied all of the needed capabilities. Martin Blauer, therefore, initiated the efforts that led to the design of the Checker to meet the following requirements:

1. All test results appearing on the printout, including instructions, data or other information, should appear in the JOVIAL format.
2. Adequate information should be provided to locate the origin of errors, but otherwise unneeded or unwanted results should be omitted from the printed test results.

The JOVIAL Checker

The JOVIAL Checker was designed and developed by members of System Development Corporation for use in checking out programs written in the JOVIAL language. It will be available for use in March 1961, and will operate upon programs translated by the JOVIAL-to-IBM 7090 Compiler. The Checker is also suitable for use with the AN/FSQ-31V military computer and is readily adaptable for use on any other computer for which a JOVIAL compiler is available.

The combined Compiler and Checker system is designed to translate programs written in JOVIAL language, execute the translated instructions, and produce test results in JOVIAL format. A brief examination of the organization of JOVIAL variables and one type of JOVIAL instruction, as well as an outline of the Compiler's functions, will be helpful in understanding the Checker's operations and output.

Organization of Variables

All input and output data, as well as variables manipulated internally by a JOVIAL program are organized into items, entries and tables. The item is the basic unit of data. Its size may range from one bit to the total number of bits in the machine word, depending upon the size of the data it will contain.

An entry is comprised of one or more items. Two or more items collected in the entry are usually related, i.e., a payroll code, an employee number, a tax deduction rate, etc. relate to one employee.

A table is comprised of one or more entries, usually repeating the same group of items contained in the first entry. Each entry, however, contains a different set of variables. There is no practical limit to the size of either an entry or a table.

All items and tables used by a JOVIAL program are defined according to the basic characteristics of the data they will contain and are assigned symbolic names. Thereafter they are conveniently called upon by name. Entries are referenced by integer values in the form of constants, subscripts or other variables.

Assignment Statement

Dynamic JOVIAL statements may be classified according to two basic types--those which control sequence of operations and those which modify the language may be used to modify the value of any variable--the assignment statement and the exchange statement. The latter is a type of two-way, restricted assignment statement.

The assignment statement places the value named by the right term into the location of the variable named by the left term, altering the format of the right term to fit, if necessary.

```
example:  (Left Term)   (Right Term)
            NUMBER   =      1 $
             ABLE    =   ABLE + BAKER$
```

The item named NUMBER in the first example is assigned the decimal value of 1. In example two, item ABLE is set to its own value plus the value of item BAKER.

Since modification of the value of any JOVIAL variable must be performed with this type of statement, the assignment statement, and parti-

cularly its left term, is vital to the operation of the Checker.

The Compiler

The JOVIAL Compiler accepts a program written in JOVIAL, analyzes its statements for illegalities, generates an intermediate language version of this program and then translates from this language to the binary code of the specific machine. Four of the Compiler's working tables are saved for subsequent use by the Checker. One relates to statement labels; one gives references to items and tables; another, to so-called status items; and the last refers to intermediate language statements.

The output from the Compiler is the object program and test data, the above tables and a printer destined tape, used also by the Checker, listing each JOVIAL statement with its equivalent symbolic instructions and octal machine code. The listing also provides a record of input test data in JOVIAL format and, if any illegalities were detected, error messages in context. When the JOVIAL program is corrected of all errors, it is ready to be run with the Checker.

The Checker Options

The general functions of the Checker are to operate the object program with the supplied test data and to record selective test results in JOVIAL format on a printer-destined tape. Results may also be printed on-line, if desired.

To use the Checker, the programmer creates two or three control cards to specify the method of selecting test results for recording. From three basic options of selecting test results, the programmer may choose any, all, or none. By answering 'yes' or "no" to each of the following questions, the programmer has eight combinations of recorded results from which to choose:

Unconditional Trace: Are dynamic snaps of selected items for which the programmer has not supplied expected final values wanted? (Let us call this an "unconditional trace." Dynamic snap, as opposed to final or static snap, is used here to mean that every value of a selected item is recorded each time it is modified throughout the entire operation of the program.

This option also applies to items in selected entries. If, for example, the program operates upon every other entry containing the selected item, instead of every entry, the programmer has no need for any dynamic snaps of item values located in half of the entries. The programmer then specifies the item name followed by the entry number. This selectiveness is important because the total number of items selected for an unconditional trace is limited to one-hundred items, and each iteration of the same item in different entries is counted as one item.

Strings and items located in tables whose entry lengths or entry structures vary from entry to entry may also be traced.

To initiate this type of trace, the programmer creates an unconditional trace control card, followed by a sufficient number of cards to list every item he wants traced in this manner. (See figures 1 and 2).

As a result of the unconditional trace, the values of all modifications of selected items are recorded for printout. Each value is accompanied by the item name and entry number, the assignment statement which modified the item at that point, and the closest preceding JOVIAL statement label.

Discrepancy Trace: Are dynamic snaps wanted only in the event that final values of items within selected tables deviated from expected values? (Let us call this a "discrepancy trace.")

For the discrepancy trace, the programmer supplies a control card with the words, "Discrepancy trace," and defines two sets of tables as part of his organization of variables with the original JOVIAL program. One set of tables defines and names all items selected for dynamic snaps in the event their final values are in error. These tables are named "ACTØ, ATCl," etc. After the object program has been operated, the actual final values of the items defined within these tables automatically will be placed in the item's assigned location.

The second set of tables are given the names, "EXPØ, EXPl," etc., and contains the programmer-supplied expected final values for all items named in the ACT tables. The values of items within the ACT tables must correspond exactly with the positioning of expected values in the EXP tables, and all recurrences of the selected items in every entry of the tables must be provided for.

After operation of the object program, actual final values of all items in the ACT tables are compared with the expected values in the EXP tables. Only in the event that a discrepancy occurs between any of the correspondingly positioned values, is a trace initiated.

Except that only those items in error are traced, this trace is performed in the same way as an unconditional trace, and recordings will also be accompanied by the modifying assignment statement and closest preceding statement label. Although any number of items may be placed in the ACT-EXP tables, only the first one-hundred discrepant final values will be traced.

When an item to be checked is already organized within an entry containing different items which need not be checked, the programmer may

remove the selected item from its original table and define it within an ACT table. This reorganization in no way alters the operation of the program or the results obtained.

Final Snaps: Are final values only of selected tables wanted? This option will usually be employed in conjunction with one or both of the options already discussed. In effect, it is a 'static snap" of the values in preselected tables at the end of the program's operation, therefore no assignment statements or labels accompany these recordings. Table names, entry numbers and item names are provided. On the control card the programmer may specify that final snaps be made of all tables defined with his program, no tables, only the tables named, or all tables excluding those named. (Figure 3)

With any of the three options named, special information must be supplied to the Checker if recordings are requested from tables whose entry lengths or entry structures vary from entry to entry. One control card is needed, one card for each table name, and one or more cards per item. Control items--or those items containing information about the length or structure of each entry --are designated by their absence or presence in floating fields. Control information on strings is also specified on these cards.

Highlights of Checker Operation

Three major routines called Control, Tracer, and Record constitute the Checker program. These routines, in turn, are modularly constructed of multiple subroutines for both flexibility of operation and ease of modification. The broad flow of operations is illustrated in Figure 4 to depict the sequence of the operational highlights only.

One pass is required through the Checker program unless actual values deviate from expected values, in which case two passes are made. The Checker may be operated in a strictly unconditional mode, strictly discrepant mode, or a combined mode. If both modes are desired, an unconditional trace control card is used, but the first pass succeeds in performing all the operations required of the first pass of both modes.

For an unconditional trace, selected item names are read in from card reader or from script tape. These item names and entry numbers, if entry references are furnished, are entered into a table called "Trace."

Statement References: With the aid of tables furnished by the Compiler, the items named in table Trace are then used to locate all JOVIAL assignment statements which contain these items as their left terms. As these assignment statements are located, they are placed in a table called "Refer." In addition to all JOVIAL statements which modify the items selected for trace, the Refer table contains the relative location in the binary program of the machine instruction which modifies the value of the item. This instruction is usually a "store" class instruction. The Refer table is subsequently used to insert traps in the object program, create a table consisting of displaced "store" instructions, and finally, its assignment statements are recorded for printout with corresponding item values.

Statement Labels: Two compiler tables are used to associate the closest preceding JOVIAL statement label with each assignment statement in the Refer Table. A search of the intermediate language table yields only the operator "label" and a reference to another table containing all statement labels. JOVIAL labels are easily recognized, however, and these are saved until associated with an assignment statement to be traced or until another JOVIAL statement label is encountered. The last label saved is thus automatically associated with the next assignment statement under trace.

Imbedding Traps: So that all modifications of an item under trace may be saved before the item is subjected to further modification, the object program is imbedded with "traps." Traps may be defined as instructions which effect an unconditional transfer of control to the snap recording routine.

Traps are imbedded in the object program to replace each "store" class instruction whose relative location is furnished by the Refer table. The store instructions, in turn, are relocated in another table and are operated upon from within this table prior to operation of the Snap routine.

Snap Tables: Recordings of the modifications of all items under trace are saved in a snap table which has a capacity of 400 snaps per Checker pass. If snaps exceed this capacity, the contents of the filled snap table are repeatedly buffered onto a scratch tape and brought back into memory just before the final recordings for printout are made.

Operation and Recording: The Checker then operates the object program with imbedded traps. If control information is present regarding variable length or variable structure entry tables, this is tabulated. Final snaps of selected tables are processed as requested and recorded on the printout tape. If no ACT tables are present, snaps resulting from the unconditional trace are grouped with appropriate statement labels and assignment statements. These are recorded on the output tape and the job is logged complete.

Discrepancies: If ACT tables are present and one or more values deviate from the EXP values, the Trace table is recreated. This time, however, instead of containing items requested for an unconditional trace, the Trace table contains only the names of items which revealed discrepancies. Again, traps are imbedded, and again, the program is operated. For the second pass, all parts of the Record routine are omitted except the final

recording of snaps and associated information destined for printout.

Checker Printout

The printout of the Checker provides the programmer with the following information in the JOVIAL format: (Figure 5)

1. Final values of any tables specified for final snaps. The word "Table" precedes the table name, the word "Item," its name, 'String," its name, etc. Fixed entry length tables are printed first, listing all values for each item sequentially. Variable structure or variable entry length tables follow, however, these values are listed entry by entry because the same items may not be present in all entries.
2. Unconditional traces are so labeled and are followed by all such traces in the format previously described.
3. Discrepancy traces are listed last in the same format.

Checker Restrictions

Restrictions imposed upon JOVIAL programs by the Checker at this time are approximate since the Checker, which is itself a JOVIAL program, has not been compiled at the time of this writing. Restrictions estimated for the Checker and the JOVIAL program when compiled on one computer will not be the same for another computer with greater capabilities.

When compiled on the IBM 7090, the Checker is expected to occupy about 16,000 registers, reserving 10,000 for the object program and test data. Most of the remaining core space will probably be occupied by the Compiler tables and the tables created by the Checker. Actually no core space is left unused, thanks to one whimsical programmer who filled in remaining space with transfer to a routine which prints the message, "Dear Programmer. You transferred out of your program. How about that?"

Practical limits to the lengths of tables created internally by the Checker impose two rather generous limits on the JOVIAL program. It is assumed that the number of assignment statements which refer to any item under trace will not exceed an average of four. JOVIAL assignment statements are also limited in length to an average of six registers each.

Evaluation

In light of the stated objectives of the Checker, comments must be withheld until programmer use of the utility program is observed and indications of time-savings are available. Similarly, the overall efficiency of the Checker is more convincingly reported after compilation figures and timing results are tabulated.

If, however, a bad plan admits of no modification, with a slight twist of logic we can believe the Checker is a good plan in at least one respect. Two proposals are now under investigation for possible modification of this or future Checkers.

One proposal considers that the programmer may wish to check out his program several times with different sets of test values, all of which may not necessarily be included with the original program. Once the program has been compiled and corrected, the present system necessitates recompiling to include such changes. The modification under consideration would enable the programmer to insert, delete, or change values of items by means of an additional Checker sub-routine which would operate at the beginning of the Control routine.

The second proposal permits the programmer to designate certain items as 'conditioning items." Conditioning items are those items not selected for a trace, but whose use in the program affects the values of items selected for a discrepancy trace. Conditioning items would be traced unconditionally in the event that the item with which they were associated were found to be in error.

The present system accommodates such items only as items selected for an unconditional trace. Should the main item selected for a discrepancy trace not be in error, unneeded results would be produced.

Reference

1. Wrubel, Marshall H., A Primer of Programming for Digital Computers, McGraw-Hill Book Company, Inc., N.Y., N.Y., 1959, p. 122.

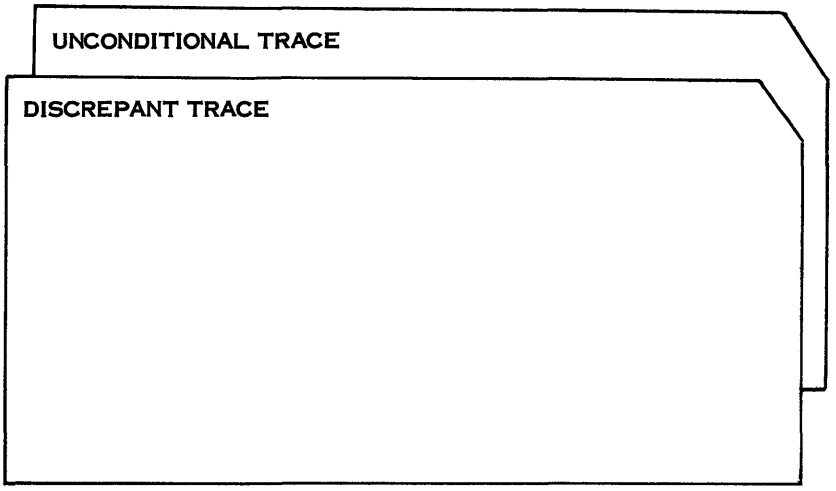

FIGURE I TRACE CONTROL CARDS

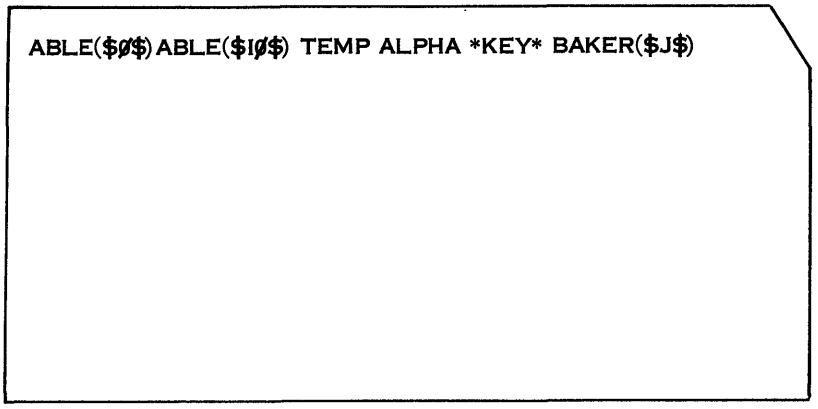

FIGURE 2 AN ITEM CARD FOR UNCONDITIONAL TRACE

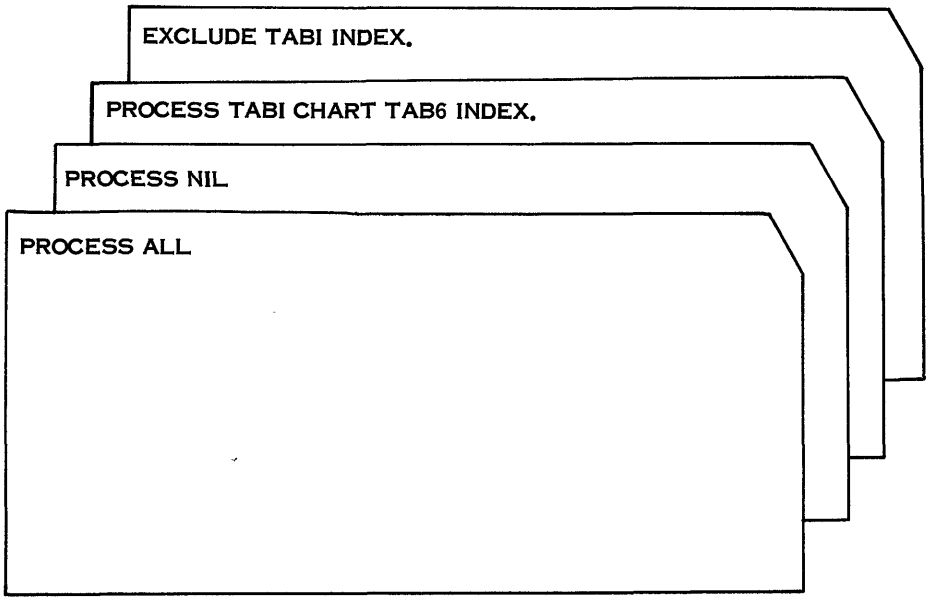

FIGURE 3 FINAL SNAP CONTROL CARDS

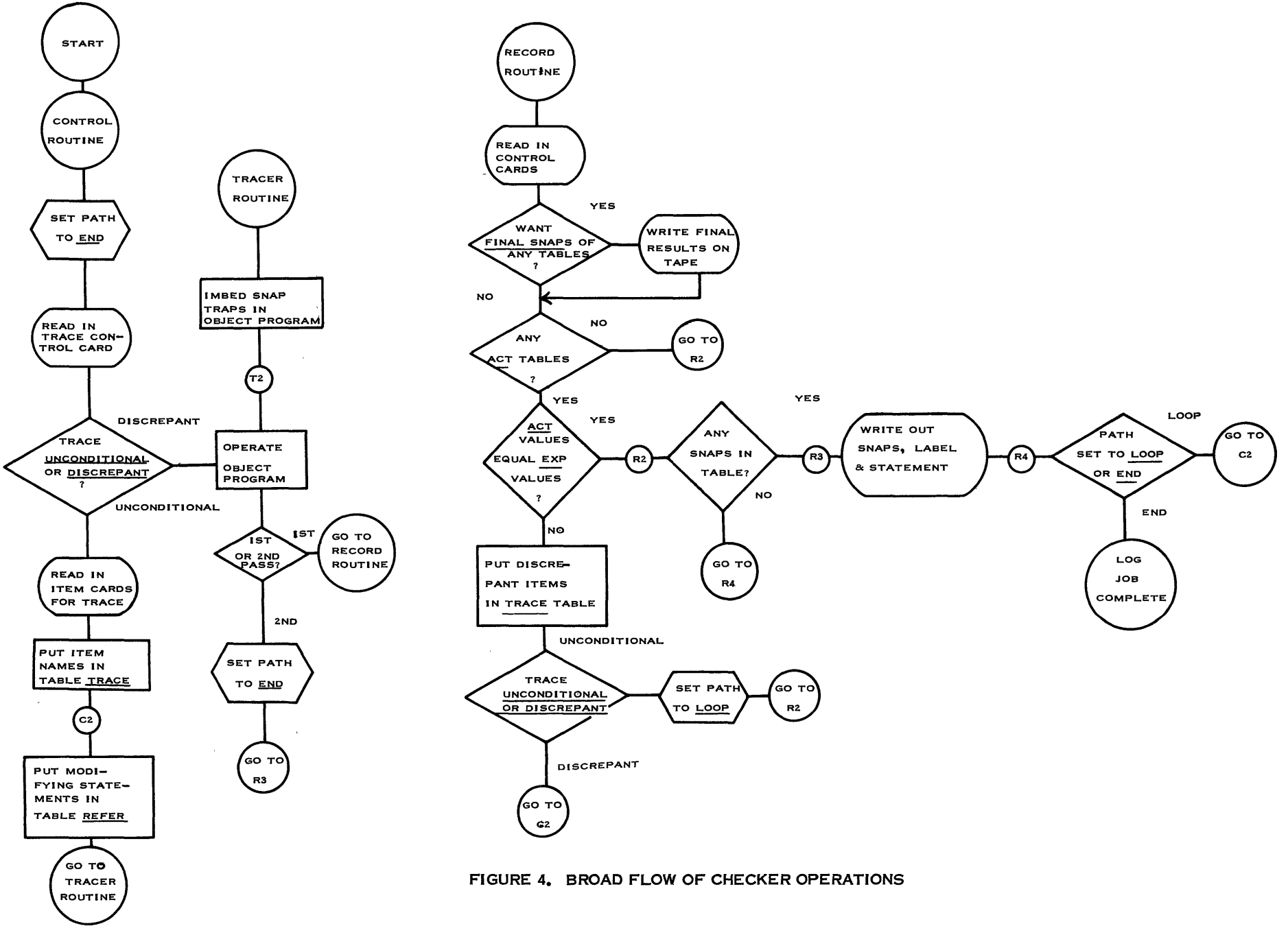

FIGURE 4. BROAD FLOW OF CHECKER OPERATIONS

FIGURE 5. SAMPLE CHECKER PRINTOUT

```
TABLE DECOR
   ITEM COLORS
   ENTRY   (0) BLUE      (1) GREEN      (2) ORANGE
           (3) BLACK     (4) PURPLE     (5) PINK
           (6) RED       (7) YELLOW
   ITEM PATTERN
   ENTRY   (0) .135663E3     (1) .06025E2     (2) .2EI
           (3) .55E3         (4) .200E3       (5) .5E-2
           (6) .60IE3        (7) .325E0
   ITEM NAMEP
   ENTRY   (0) BLUBEL        (1) SPRING       (2) SUNSET
           (3) EBONY         (4) DUSK         (5) SUNRIS
           (6) BLAZE         (7) JONQUL
```

FINAL VALUES IN TABLE WITH FIXED LENGTH ENTRIES.

```
TABLE MILLS
   ENTRY   (0)
      ITEM PATTNO       .135E3
      ITEM TLMILS       ONE
      STRING LOCATN     SAVANA
   ENTRY   (1)
      ITEM PATTNO       .2EI
      ITEM TLMILS       NONE
   ENTRY   (2)
      ITEM PATTNO       .6025E2
      ITEM TLMILS       THREE
      STRING LOCATN         LOUSVL
                            SAVANA
                            BIRMHM
```

FINAL VALUES IN TABLE WITH VARIABLE LENGTH AND VARIABLE STRUCTURE ENTRIES.

```
UNCONDITIONAL TRACES FOLLOW

   10A. BYTE ($B, E$) (PRICE ($C$))     TEMP ($A$)
      (2) (3) (PRICE (12))       .99
   10A. BYTE ($B, E$) (PRICE ($C$))     TEMP ($A$)
      (2) (3) (PRICE (13))       .87
   10A. BYTE ($E, E$) (PRICE ($C$))     TEMP ($A$)
      (2) (3) (PRICE (14))       .54
```

DYNAMIC SNAPS OF VALUES OF PARTS OF ITEM PRICE.

```
DISCREPANCY TRACES FOLLOW
   CF8.  ABLE ($L$)  COUNT $
         ABLE (0)    2
   CF8.  ABLE ($L$)  COUNT $
         ABLE (1)    3
   CF8.  ABLE ($L$)  COUNT $
         ABLE (2)    4
```

DYNAMIC SNAPS OF VALUES OF ITEM ABLE