GPMX—A portable general purpose macro processor adapted for preprocessing FORTRAN

by ROBERT C. GAMMILL
The Rand Corporation
Santa Monica, California

ABSTRACT

GPMX is an extension of GPM, a simple, elegant yet powerful language independent macro processor described by Strachey. Unextended, GPM is not suited for preprocessing languages which use column position and end of record to delimit statements. Examples are FORTRAN and many assembly languages. Many programmers are constrained to work in such limited languages and GPMX is a simple yet powerful tool for extending and modifying these languages. Others have developed preprocessors dedicated to a particular language. This has advantages for the implementor, but requires the user to learn a different preprocessor for each language he uses. GPMX is designed to work on any language so that the (non-trivial) effort of learning to use it need not be repeated later. Extensions in GPMX include macro control over: files, record input and output, spacing, conditional macro processing and compilation, access to input and output buffers, and dynamic changing of the macro flag characters. Most of the extensions are accomplished simply by putting the control information on the macro stack where the processor has access to it (ala von Neumann). GPMX has been implemented in ANS FORTRAN for portability. Several applications are shown, including GO-TO free control structures for FORTRAN. Source is available.

GOALS AND PHILOSOPHY

An important goal of this work was the development of a flexible software tool. To be called a tool (using a restrictive definition) software must be suitable for a variety of tasks, on the scale of a single human being (the user) and not dependent upon any unusual aspect of the environment (i.e., adaptable). To make an analogy, a Boeing 747 would not qualify as a tool, since it is suitable to many tasks, at a human scale (e.g., purchasable, maintainable and portable) and can be used in almost any environment (e.g., under water). Many language processors do not qualify as tools. They have been designed for a specific task and environment, so when the situation changes they prove to be clumsy or useless. Then, the large size of most language processors makes them difficult to modify or adapt to the new situation. The lack of adaptable tools has made software development needlessly complex.

GPMX qualifies as a tool. It is suitable to a number of tasks and on a human scale. The original implementation of GPM in CPL was extremely compact, and was listed in the last three pages of Reference 1. Gries suggests its implementation as a student exercise (page 433). GPMX appears, to the untrained eye, to be unchanged from GPM. This is because the extensions have been carefully designed to minimize the structural change in GPM. Most of the changes have been accomplished by adding a few primitive macros and by putting the processor input-output and control information on the stack as the values of defined macros, available for use and modification. This is a simple application of the von Neumann stored program concept, and it provides the same startling increase of power to GPM that it provided to the early computers. Finally GPMX has been implemented in ANS FORTRAN for portability.

Although GPMX is a tool, we do not want to imply that it is simple for an inexperienced programmer to learn to use all its power. Simple applications of GPMX can be easy to understand, but an application that uses its full power can be extremely difficult to follow. Perhaps it is best to characterize GPMX as a tool for the advanced scientific or systems programmer, who finds it necessary to write programs in limited languages (such as FORTRAN and assembly) because of special properties they possess, yet desires to have a flexible and powerful macro processor for manipulating his source text.

GPMX provides the following capabilities:

(a) Conditional compilation—so alternative versions of a subroutine need not require multiple source files.

(b) Macro processing—allowing insertion of the same declarations in many subroutines or selective use of open or closed code.
(c) Text compacting—allowing reduction of blanks and other unused redundancy.
(d) Syntax extension—such as adding GO-TO free control constructs to FORTRAN.
(e) Language independence—since information about the language processed is carried in macros not in the processor.
(f) Power and flexibility—the elegance and simplicity of GPM has been kept while its considerable power has been increased.

DESCRIPTION OF GPM

In recent years a number of language independent macro processors have been introduced. GPM is one of these. In this section we will briefly survey the characteristics of GPM. GPM uses marker characters to indicate where a macro call begins "[", ends "]" and the separations between arguments ",". GPM has only six primitive (built-in) macros. An example showing the use of three of these is:

\[
\text{[DEC: \text{BAR:} \text{+:} \text{[BIN:3]} \text{:[BIN:2]]]} \quad (1)
\]

GPM has two data types, "binary" integer and character string. BIN converts from character string to integer and DEC does the reverse. BAR allows integer arithmetic (+, -). Example (1) produces the character string "5" as its result. The most important primitive macro DEF allows definition of new macros.

\[
\text{[DEF: +: < \text{[DEC: \text{BAR:} \text{+:} \text{[BIN:1]} : \text{[BIN:2]]]} > ]} \quad (2)
\]

A call on the defined name "+" causes evaluation of its body (second argument of DEF).

\[
[ + : 3 : 2 ] \quad (3)
\]

Example (3) will also produce "5" as its result. Example (2) introduced string quotes (< and >) and formal parameters (i.e., %1). Evaluation of nested quotes removes the outermost pair. Macro bodies are normally protected from evaluation at definition time by quoting them. Access to the body of a macro without evaluation is provided by VAL.

\[
\text{[VAL: +]} \quad (4)
\]

Example (4) will return the body of the + macro, the quoted string from example (2). UPDATE stores new values in the bodies of macros.

\[
\text{UPDATE: +: <<This is a new body for +.>>} \quad (5)
\]

The marker character set being used here is different from that used in GPM. GPMX allows these to be changed at will, so their selection is dictated only by esthetics and local constraints.

An important feature of GPM is that macro calls are allowed anywhere. For example, \[[A]:[B]\] is a call on a macro whose name is the result of \([A]\) and whose argument is the result of \([B]\). Also, since macro definition is done by a macro call, macro definitions may be created in unusual ways. One use of this in GPM is the temporary macro definition. If a macro definition is created in the argument sequence of a macro call, that definition will exist only so long as the called macro is being evaluated.

\[
\text{[A: [DEF: N: [BIN: 3]]]} \quad (6)
\]

In example (6) the macro N will be defined (with value 3) only so long as macro A is being evaluated. If there was a previous definition for N, it will be temporarily superseded by this one. One of the important uses of temporary definitions is for self-defining macros, where the definition of the macro occurs in its own argument sequence.

\[
\text{[DEF: EVAL: < [X: [DEF: X: %1]] > ]} \quad (7)
\]

Every time the macro EVAL is called, the macro X is temporarily defined and then called. Since a macro body is evaluated when its name is called, this means that the result of a call on EVAL will be the result of evaluation of its first argument. Using EVAL to cause evaluation and string quotes to defer it gives the ability to specify when evaluation is to be carried out. Thus, although Brown characterizes GPM as passing parameters "by value", as shown in (10), it is possible to pass them "by name", as shown in example (9).

\[
\text{[A: [DEF: N: 3]} \quad (8)
\]

Another important result from the ability to create temporary definitions, is that a conditional macro can be defined.

\[
\text{[DEF: COND: < [\%1: [DEF: \%1: \%4] >] \quad < [DEF: \%2: \%3]] > ]} \quad (11)
\]

The COND macro depends upon the fact that the most recent definition is used if two macros have the same name. COND executes its third argument if its first and second arguments match, and its fourth argument otherwise. Thus, \text{[COND: A: B: C: D]} will produce result D, while \text{[COND: A: C: D]} will produce result C. Conditional execution gives access to the recursive calling capability of GPM by allowing termination of recursive loops. The standard example is the recursive computation of the factorial function (fact(i) := if \(i = 0\) then 1 else \(i \times \text{fact}(i - 1)\)).

\[
\]

A more practical example would show how data tables may be generated using recursion, but such examples become extremely involved.

From the collection of the Computer History Museum (www.computerhistory.org)
EXTENSIONS IN GPMX

In order to make GPM usable as a FORTRAN preprocessor, new capabilities have been added. The elegance and simplicity of GPM makes this easy. In adding new capabilities to GPM care was taken to minimize the set of additional primitive macros, as the elegance of the original set is one of GPM's best features.

The most obvious area of need when using GPM as a FORTRAN preprocessor results from the fact that FORTRAN is column and line (record) oriented, whereas GPM deals with a character string. The solution implemented in GPMX is the introduction of eleven new primitive macros which cope with the problems of record input, output and blank fill. Table I gives the macros and the output or side effect yielded by execution of each.

These new primitive macros use a special marker character for brevity and so as not to conflict with user defined macros which have one character names. In other words, [5] will not give the same result as $5, but will seek a user defined macro named 5. An early version of GPMX used [R] for $1 and [W] for $2, but access to these was often destroyed by subsequent user macro definitions. Thus, the $0 through $9 are primitive macros whose meanings cannot be superseded. The user of GPMX who does not need line and column control and fears that a $ macro may occur in the input text, can disable this feature by changing the definition of the $ character to an illegal value. The utility of very short macros in simple text substitution is apparent, so any GPMX macro with a single character name (not a digit) and no arguments can be called in this manner. For example, the macro [DEF:S:<$2$6>], which can be called either as [S] or $S, is useful for moving to column 7 of the next line.

Another mechanism which has been provided in GPMX is the line distributor, which controls the distribution of GPMX input records. We are going to want to make occasional use of GPMX macros in large existing FORTRAN programs. However, scanning every character of a large program which contains only sparse occurrences of macros will be very expensive.

<table>
<thead>
<tr>
<th>TABLE I—Semantics of Special $ Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MACRO</strong></td>
</tr>
<tr>
<td>$0</td>
</tr>
<tr>
<td>$1</td>
</tr>
<tr>
<td>$2</td>
</tr>
<tr>
<td>$3</td>
</tr>
<tr>
<td>$4</td>
</tr>
<tr>
<td>$5</td>
</tr>
<tr>
<td>$6</td>
</tr>
<tr>
<td>$7</td>
</tr>
<tr>
<td>$8</td>
</tr>
<tr>
<td>$9</td>
</tr>
<tr>
<td>$$</td>
</tr>
</tbody>
</table>

The line distributor allows specially marked GPMX lines to be inserted in ordinary FORTRAN, so that only the marked lines need be scanned. Column 1 of each line is reserved for this purpose. The character punched in column 1 controls the handling of the line, much the way a C in column 1 of a FORTRAN line specifies it to be a comment. However, in this extension to GPM, the mode of a character may be set to any desired value (from 1 to 6), allowing succeeding lines starting with that character to be treated in a specified manner. Table II gives the mode values and actions.

<table>
<thead>
<tr>
<th>TABLE II—Line Distributor Actions Specified by Character Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

The line distributor mechanism has several uses. One of these is to provide GPMX comment lines (mode 2). Another is conditional compilation. Putting a character in mode 4 will cause all lines with that character in column 1 to be compiled. If that character is set to mode 3 on a subsequent pass, all those lines will become FORTRAN comments. GPMX lines can produce widely varying numbers of output characters, and if blanks are taken as significant, much of the output becomes unwanted blanks. Thus, it was decided in GPMX to ignore unquoted blanks in the input. This is only done on mode 6 lines, which are passed to the GPMX scanner. For those who use GPMX with another language, the column position of the line distributor control character can be changed to any desired value, through use of the UPDATE macro. Also, should the user desire to disable the line distributor mechanism, and operate in GPM mode (scanning every line), that too can be accomplished through macro calls.

Besides the primitive line format control macros, several other primitive macros have been added. These are listed below. Each has been chosen to be of general use. The second through fourth macros are the character string primitives of FL/1.

(a) [SETMODE:[BIN:5]:<0123456789>] sets mode values of line distributor control characters. In this case the characters normally found in column 1 of a FORTRAN line are being set to mode 5.

(b) [LENGTH:[VAL:8]] returns a value (binary integer) which is the length of its argument. In this case the result will be 4 (see preceding DEF of $).

(c) [SUBSTR:<ABCDEFG>:[BIN:3]:[BIN:2]]
returns a substring of its first argument. The second argument tells where the substring begins and the third tells how long it is. The result is CD.

(d) [INDEX:<ABCDEFG>:<CDE>] returns a binary integer which gives the position where its second argument first occurs as a substring of its first. In this case the result will be a binary 3. If there are no occurrences the result is zero.

(e) [WEF:[BIN:7]] writes an end-of-file on logical unit 7.

(f) [REW:[BIN:8]] rewinds logical unit 8.

Besides the addition of new primitive macros and the line distributor mechanism, a very important change has been made in GPMX. This involved moving most of the important control variables, switches and buffers to the stack, as ordinary GPMX macros whose values may be changed by UPDATE. This is a very important improvement. For example, it allows the macro flag characters to be changed during macro processing, and with no support mechanism required.

* CHANGE CONTROL CHARACTERS
  - [UPDATE:CONCHR:@,?()&>&1]
  - * SET THEM BACK
  - @UPDATE,CONCHR,([:%<><])$:1

Another important use of this facility is changing logical unit numbers for the various input and output streams by means of UPDATE. Logical unit number zero has been taken to mean that no I/O should occur on the particular stream. For example, the following statements will turn on the output listing and turn off the output, for testing without compilation of the output.

  - [UPDATE:OUTPUT:[BIN:0]]
  - [UPDATE:OUTLST:[BIN:6]]

The ability to change the logical unit numbers for the input and output, in conjunction with the primitive macros WEF and REW, allows the GPMX programmer complete control over input and output files. Partitioning of output into two files, one for declarations and another for executable code (a frequently needed capability in a preprocessor) is easy to program at the macro level. Also, multiple passes through the input text can be carried out using these facilities.

Other important GPMX control elements can also be changed using UPDATE or accessed using VAL. One particularly important macro, COLUMN, contains the integer which tells in what column the next output character will go. This is especially useful when writing macros to format the output correctly (e.g., tabbing). The contents of the input buffer, INBUF, can be manipulated as a character string (using INDEX, SUBSTR, VAL and UPDATE) before any macro scanning takes place, so GPMX can be used as a general text processor with controlled input. This is a capability that Brown\[sup 31\] specifically notes as missing from GPM, when comparing it with the TRAC language.

**GPMX APPLICATIONS**

In this section we will examine some GPMX applications. In all cases the examples deal with FORTRAN. This was done because FORTRAN is widely known and because it involves most of the unpleasant problems that will be encountered by a language independent preprocessor. The first example shows the generation of local variables and statement numbers as they can be applied in a DO macro. This example uses the fact that a macro definition (DEF) which is created as part of the argument sequence of another macro will exist only as long as that macro is being evaluated. This allows temporary variables to be created. Here K is used to store the local variable name and M is used to store the local statement number. (Note: % %1 means the first argument of the caller’s caller).

* GENERATE STATEMENT NUMBER
  * OR INTEGER VARIABLE
  - [DEF:GENSN:<[DEF:ro1:<77»]
  - [DEF:GENIV:<[DEF:ro1:<IV>]
  - [DEF:B:[BIN:1]] [DEF:N:[BIN:10]]
  * GEN RETURNS THE VALUE
  * OF N AND INCREMENTS N.
  - [DEF:GEN:<[VAL:N] [UPDATE:N:]
  <[BAR:+:$N:$B]>]
  * EVAL MACRO EVALUATES ITS
  * ARGUMENT.
  - [DEF:EVAL:<[8:[DEF:8:%1]>]
  * CONT MACRO PRODUCES A CONTINUE
  * CARD
  - [DEF:CONT:<[$2[ %1]< CONTINUE>>>]
  * DO MACRO
  - [DEF:DO:<[DOBODY:[GENIV:K]>]
  - [DEF:DOBODY:<DO $M $K=1,% %1>>
  - <[EVAL:% %2][CONT:M]>]
  * CALL THE DO MACRO
  - $S [DO:<32>:<[$SB($K)=A($K)>]

The preceding code yields:

```
DO 7711 IV10=1,32
B(IV10) =A(IV10)
7711 CONTINUE
```

In this example the body of the DO is not evaluated until it is passed into the interior of the DO macro. This is an example of a useful application of passing
arguments “by name.” Here it allows the temporary macro K to carry the local name for the integer DO variable. A disadvantage of this example is that the complete body of the DO must be passed as the second argument of the DO macro, enclosed in string quotes. This would be extremely awkward for long DO loops.

One of the goals set for GPMX was syntax extension of FORTRAN, to allow GO-TO-free constructs such as if-then-else and while-do to be introduced into the language. Initially we tried to do this in a manner similar to that shown in the preceding example for DO loops. This proved awkward, since the body of code in each part of an if-then-else had to be passed as an argument to a macro, tending to overload the stack. Furthermore, it is very desirable to be able to insert these control statements within ordinary FORTRAN code in the following manner:

- [IF] I. NE. 3 .AND. F(I,J).GT. 3. 4 [THEN]
  a sequence of FORTRAN statements
- [ELSE]
  another sequence of FORTRAN statements
- [FI]

To implement the preceding idea, we create a special macro whose body will serve as a short push-down stack for statement labels. Then we define macros to allow us to PUSH and POP this special stack, as well as one to allow us to pick up a copy of the TOP element (without changing the stack).

- [DEF :PUSH: <$S<IF (.NOT. (> >
- [DEF :POP: <$S$G<77> 
- [DEF :TOP: <$B<77>

Using the preceding macros, we implement the if-then-else-fi construct as four separate macros, in the following manner. The definition for GEN was given earlier in this section.

- [DEF :IF: <$B<77>[DEC:>
- [DEF :STACK:$9]
- [DEF :S:[COND:[BIN:7]:>
- [DEF :B:[COND:[BIN:7]>[VAL:COLUMN]
- [DEF :TOP:[GEN]:[BIN:1]>

In this example we have introduced new macros S and B for controlling our position in the output card image. $S moves us to column 7 of the next line, if we are not already there. $B moves us to column 1 of the next line, adding a CONTINUE statement if we are presently in column 7 of a line. These two macros demonstrate how we can use COLUMN (the position in OUTBUF where the next output character goes) to control positioning of output information.

As we used GPMX on actual programs, written in structured FORTRAN, it became clear that our macro definitions for GO-TO-free FORTRAN had still not adequately solved all the problems. As a result, we changed from a single stack to double stacks, one containing the label desired for exit from a control structure and the other containing the label desired when the next cycle (pass) through a control structure is to be started. The EXIT and NEXT macros, which generate FORTRAN GO-TO’s, require a first argument which specifies how far out in a nested control structure the jump is to go. We also implemented automatic indentation of the output text (as dictated by the nested control structure) and added most of the commonly mentioned control structures (e.g. CASE, REPEAT, WHILE, FOR). All of this was carried out completely at the macro level. Figure 1 shows the use of some of these capabilities in a simple program. The library of macros is given in the Appendix.

We now feel that we have achieved a useable set of macros for GO-TO-free FORTRAN. However, many further improvements can still be introduced. We note with satisfaction, however, that although we write and rewrite macros quite often, changes in the processor have become an infrequent event. An example of such an event occurred when moving GPMX to PDP-11 UNIX. It became desirable to allow the GPMX programmer to associate a filename with a logical unit number, providing complete macro time control over input and output files. This required the addition of a new primitive macro which is system dependent (not definable inside FORTRAN). The macro is [SET-FILE:[BIN:<digit>:]<filename>]. Code written using this macro acts much like a job control language. It seems likely that with the addition of yet another system dependent primitive macro (EXEC perhaps) allowing execution of assemblers, compilers, loaders and other system utilities, GPMX could be used as an extensible job control monitor for minicomputers.

Recalling our earlier description of a tool, we feel...
that GPM and its offspring GPMX are definitely tools. As we have indicated, with minor modifications these macro processors seem capable of use as general text processors, as language translators or as job control monitors. Of course, the limitations must be recognized (as when racing bicycles against cars) but the flexibility and simplicity seems astonishing.

ASSESSMENT

GPMX is an adaptable tool, easily modified or controlled by a sophisticated user. It has been moved to a wide variety of computers (including minis) in a few man-hours. A disadvantage of implementation in ANS FORTRAN is that handling card images via formatted I/O is inefficient, although essential for portability. However, the I/O in GPMX is localized in a few routines, so substitution of non-portable but efficient I/O is easy.

GPMX is dramatically different from other FORTRAN preprocessors, since it is a general purpose macro processor extended for that use. GPM was not designed for ease of use by unsophisticated FORTRAN...
applications programmers. Thus, preprocessors (such as MORTRAN2) which were designed for that goal, tend to be more suitable for use by FORTRAN programmers unfamiliar with general purpose macro preprocessors. However, GPMX has substantially more computational power than most other FORTRAN preprocessors, and it is language independent besides. Although MORTRAN2 and some other FORTRAN preprocessors also have macro capabilities, in most cases this is of the pattern match and simple text substitution variety. None of these have sufficient computational power to implement stacks or source symbol generation at the macro level, as demonstrated here. These advantages make GPMX especially suitable for use by advanced applications and systems programmers, for whom FORTRAN preprocessing is only one of the desired applications. A special advantage enjoyed by GPMX is that most extensions can be produced by defining new macros, while extensions in other preprocessors often involve rewriting the processor. The most obvious disadvantage of GPMX (like GPM) is its ugly syntax. The many good features should overcome this irritation in the appropriate applications.

REFERENCES
APPENDIX

* GENERATE A LOCAL STATEMENT NUMBER.
- [DEF:NXTNAM: [BIN: 10]] s1 INITIALIZE INTEGER FOR LOCAL NAMES
- [DEF:INCR: [UPDATE: z1: [BAR: ++: [VAL: z1: > [BIN: 1]]]]] s1
- [DEF:GEN: [VAL: NXTNAM: [INCR: NXTNAM]]] s1 GENERATE A NUMBER

* DEFINE LINE AND SPACING CONTROL MACROS S, B AND Z.
- [DEF:S: [COND: ] [BIN: 7] [VAL: COLUMN: > [SZ: > [S$6$sz]]]] s1
- [DEF:B: [COND: ] [BIN: 7] [VAL: COLUMN: > [SZ: > [SZ$cs$2]]]] s1
- [DEF:Z: $9$9$9] [UPDATE: z:] s1 Z INITIALLY EMPTY STRING
- [DEF:DNZ: [UPDATE: z: [SUBSTR: sz: >>]]] s1
  [BIN: 1] [BAR: --] [LENGTH: sz: > [BIN: 3]]] s1
- [DEF:UPZ: [UPDATE: z: 2$3]] s1

* DEFINE MACROS FOR FORTRAN TO HANDLE KEY-WORDS.
- [DEF:G: [GO TO >>]] [DEF: C: [CONTINUE]] [DEF: M: ] [S$COMMON]] s1
- [DEF:Q: ] [S$EQUIVALENCE >>]] [DEF: I: ] [S$INTEGER]] s1
- [DEF:D: ] [S$DATA >>]] [DEF: E: ] [S$END$2] [DEF: ] [S$5$2]] s1
- [DEF: #: ] [IF (NOT (. )]] [DEF: ] [IF ( ] [DEF: ] [S$LOGICAL >>]] s1

* DEFINE STACK HANDLING MACROS.
- [DEF:PUSH: [SUBSTR: [VAL: 1]] [BAR: ] [BIN: 1]] [BAR: ] s1 PUT ITEM ON TOP
- [DEF:POP: [SUBSTR: [VAL: 1]] [BAR: ] [BIN: 1]]] s1 DISCARD TOP
- [DEF:GET: [SUBSTR: [VAL: 1]] [BIN: 2] [BAR: ] > s1 GET ITEM
- [DEF:PUT: [SUBSTR: [VAL: 1]] [BIN: 3]]] s1 MAKE STATEMENT NUMBER STACKS.

* DEFINE CONTROL STRUCTURE MACROS FOR GO-TO FREE FORTRAN.
- [DEF:IF: ] [S$#] [PUSH: O: STCKX] [UPZ: ] [TOP: STCK] [STACK: ] [UPZ: ] s1
- [DEF:THEN: ] [TOP: STCK: 1]]> s1 NEXT: 1]] s1
- [DEF:ELSE: ] [S$EXIT: 1] [NLBL] [PUT: [TOP: STCKX] [STACK: 1]]] s1
- [DEF:FI: ] [COND: ] [TOP: STACK: ] [TOP: STCKX: ] [NLBL] ]> s1
  [XLBL: ] [POP: STACK: ] [DNZ: ] s1 END OF IF
- [DEF:REPEAT: ] [S$77] [DEC: ] [TOP: STCKX: ] [STACK: ] [UPZ: ] [PUSH: ] [GEN: ] [STACK: ] s1
- [DEF:UNTIL: ] [DNZ: ] [NLBL] [S$2$#] ] s1
- [DEF:TILNOT: ] [DNZ: ] [NLBL] [S$2$2] ] s1
- [DEF:ENDREP: ] [SUBSTR: ] [SUBSTR: ] [TOP: STACK: ] [STACK: ] [UPZ: ] s1
- [DEF:LPBACK: ] [S$77] [DEC: ] [BAR: ] [TOP: STACK: ] [BIN: 1]]] s1
  [POP: STACK: ] [XLBL: ] s1

* DEFINE THE EXIT AND NEXT FUNCTIONS AND LABEL MACROS.
- [DEF:EXIT: ] [COND: ] [GET: STCKX: ] [TOP: STACK: ] [TOP: STCKX: ] [BAR: ]> ]> s1
- [DEF:EXIT: ] [COND: ] [GET: STCKX: ] [BAR: ] > s1 GENERATE EXIT LABEL IF NOT 0.
- [DEF:NEXT: ] [S$77] [DEC: ] [GET: STCKX: ] ] ] ] ] s1 GO TO NEXT CASE