THE BIGMAC USER'S MANUAL

by

Eugene Myers
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-145-78 November, 1978

This work was supported by NSF grant MCS77-02194 and
Army Grant DAAG29-78-G-0046.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 1978		**00-11-1978 to 00-11-1978**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The BIGMAC user's Manual	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Colorado at Boulder,Department of Computer Science,Boulder,CO,80309-0430**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**95**	

TABLE OF CONTENTS

I. SYSTEM OUTLINE

0. OVERVIEW 1
1. SCANNER: RECOGNITION COMPONENT 4
2. SCANNER: TRANSFORMATION COMPONENT 6
3. MACRO ACTION SET STRUCTURE 9
4. MACRO TRANSFORMATION PRIMITIVES 13
5. NAME GENERATION AND EXPRESSION ANALYSIS . . . 16

II. THE STREX MACRO LANGUAGE

0. PRELIMINARIES 17
1. FORTRAN BASE 19
2. STRING FEATURES 20
   1. String Declarations 21
   2. String Assignments 22
   3. String Expressions 24
   4. External Functions 27
3. MACRO FEATURES 28
   1. Transformation Primitives 29
   2. Unique Name Generator 31
   3. Expression Analysis Primitives 32
   4. Environment Primitives 36
4. MACRO FILES 39

III. APPENDICES

A. SCANNER TRANSFORMATIONS 40
B. ERROR MESSAGES 42
C. BIGMAC ON THE C.U. CDC-7600 48
D. SAMPLE APPLICATIONS 50

I. SYSTEM OUTLINE

0// OVERVIEW

The BIGMAC system is a programmable utility for performing textual transformations on ANSI FORTRAN code. BIGMAC was developed for the specific purpose of replacing procedure calls with in-line code. For the purposes of modularity and hierarchical development it is frequently desirable to code simple routines for low level data abstractions such as stacks and lists. However, it is undesirable to pay the runtime costs of parameter passing and routine linkage for such frequently invoked routines. BIGMAC remedies the situation by allowing one to program and develop a prototype of the modular variety and then produce the efficient production code by transforming the prototype with BIGMAC. The degree of speed-up will depend on the machine and compiler in question. In a recent large scale application, BIGMAC speeded up the DAVE system by 47%. BIGMAC can, of course, be used for conventional macro applications. Of greater interest, is that BIGMAC is general enough to enable the programming of limited language extensions.

The design of BIGMAC incorporates many of the principles found in typical macro facilities. It is rather unusual in that macros are not templates for textual substitution but are executable routines. This very dynamic approach affords a great deal of flexibility with low developmemt overhead, as an existing language can be used as the basis for the macro language. The base language provides conditional execution and local and global data management. The developer need only concern himself with the macro system interface. However, in the case of a language with weak string capabilities (e.g., FORTRAN), the designer must also bolster the base language's capabilities in this regard. BIGMAC macros are routines written in such an extension of FORTRAN.

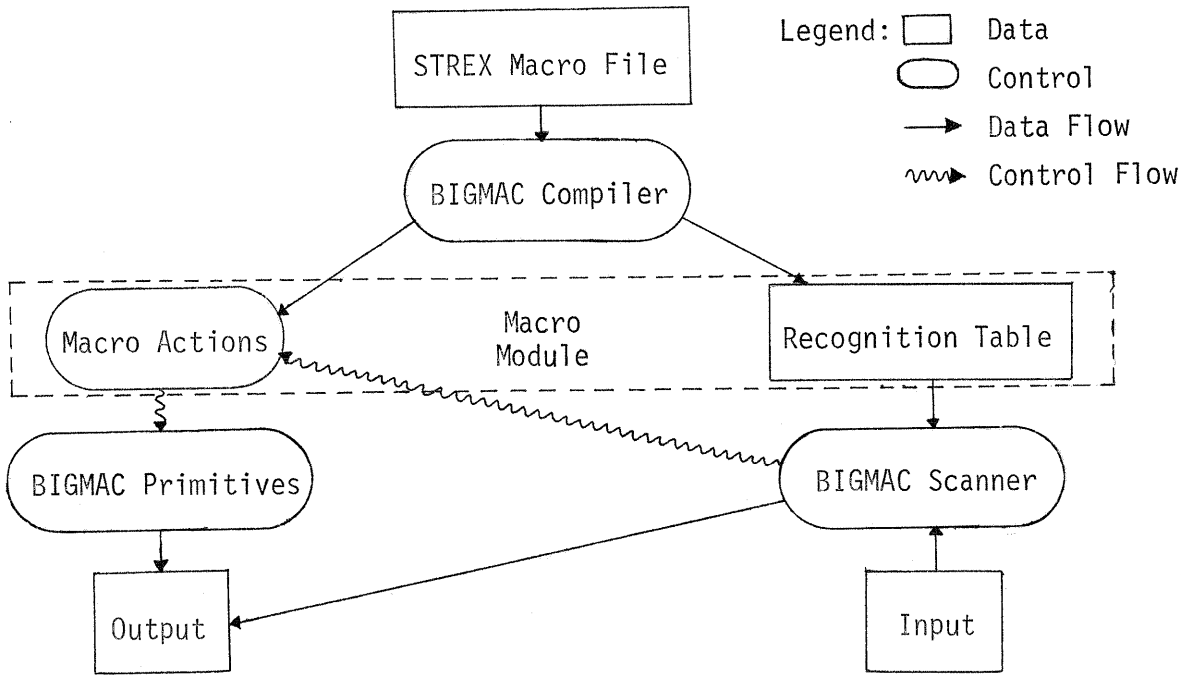


Figure 1: BIGMAC Outline

The structure of the BIGMAC system is depicted in Figure 1. At the head of BIGMAC is a table-driven scanner. This scanner in conjunction with a user-programmed macro module forms a text transforming machine. A macro module consists of a recognition table and a set of macro actions. A recognition table is an array of phrases to be recognized, called the recognition domain, and a function mapping each phrase into the name of a macro action in the macro module. Each macro action is a program for performing a transformation on the text stream to which the scanner is currently being applied. These transformations are carried out with the aid of a number of system primitives.

The operation of a BIGMAC text transforming machine can now be described as follows. The scanner will search a submitted input in a forward scan until either --

1: The end of input is reached --
The machine stops.

2: A suffix of the text thus far scanned matches a phrase in the recognition domain --

The associated macro action is invoked. Upon return the scanner resumes where it left off. The BIGMAC scanner does *not* rescan the recognized phrase.

A BIGMAC user writes macro modules in a language called STREX (STRing EXtended) FORTRAN. This linguistic framework is the subject of the second part of this manual. For the present it suffices to know that there is a system compiler for transforming STREX specifications into macro modules.

1// SCANNER: RECOGNITION COMPONENT

The BIGMAC scanner can recognize any subroutine call or function reference in an input of ANSI FORTRAN code. To specify a particular routine invocation for recognition a user just gives the name of the routine. Furthermore, a user restricts the admissible syntax of recognized invocations by indicating --

1: whether the routine is a subroutine or function.

2: the number of arguments in the invocation.

If the scanner recognizes an invocation which is not admissible (i.e., violates either 1 or 2) then a message is issued and the associated macro is not invoked.

An activation phrase is the text of a recognized and admissible invocation. More explicitly, in the case of a function reference, it is the text from the leftmost letter of the routine name to the right parenthesis ending the list of actual arguments. For a subroutine call it is the text from the 'C' in 'CALL' to the end of the line, i.e., it is the entire CALL statement.

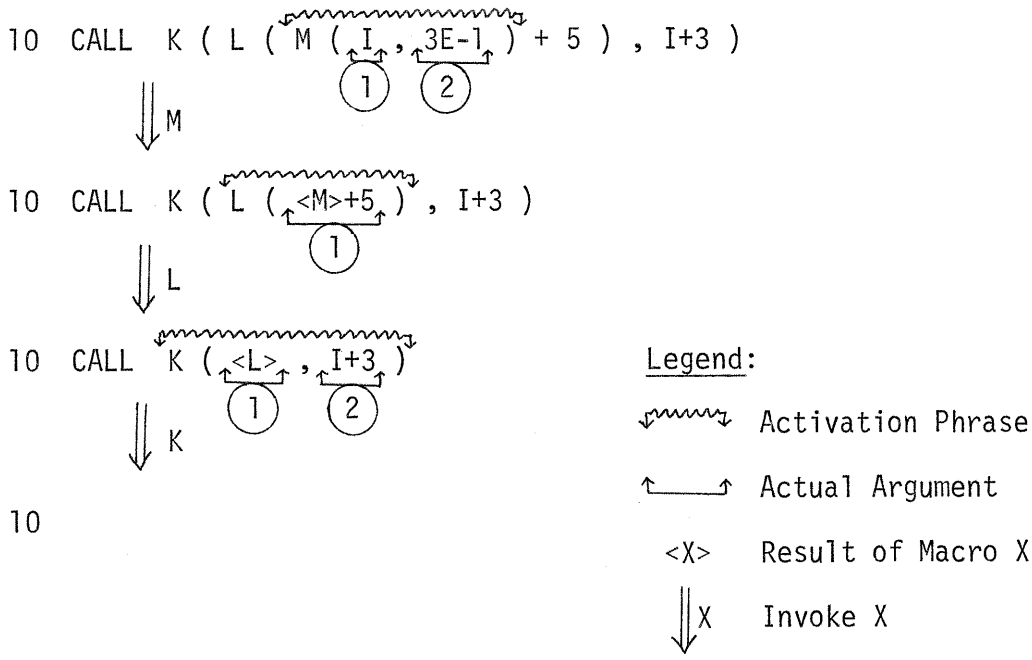
The rule for associating an activation phrase with a macro action is particularly simple. The name of the routine in the activating phrase is taken to be the name of the macro action to be invoked. The actual arguments of a particular invocation are passed to the macro action as text strings.

For function references, the associated macro returns a string which is substituted for the activation phrase. For subroutine calls the activation phrase is deleted.

Example 1: Suppose a user has specified the recognition table --

Macro Name	S/F	# of Args.
K	S	2
L	F	1
M	F	2

The action of the scanner with this recognition table is illustrated below --



2// SCANNER: TRANSFORMATION COMPONENT

The BIGMAC scanner views an input text stream as a sequence of text fragments. There are four types of fragments --

1: Comment - An ANSI Standard comment line.

2: Label - An ANSI statement label.

3: N-Statement - An ANSI statement containing no activation phrases.

4: A-Statement - An ANSI statement containing at least one activation phrase.

Note that these fragment definitions will unambigously partition any input stream of ANSI Standard code.

The BIGMAC scanner is unusual in that before invoking a macro action it transforms the input fragments so that the following condition is satisfied --

A-Statement Condition: The placement of a text fragment just before an A-statement fragment in the transformed input sequence, must guarantee that the fragment is executed just before the execution of the A-statement fragment.

This preconditioning of the input considerably simplifies the required text transforming primitives (see section 4). Henceforth, the discussion of the input sequence will always refer to the preconditioned input sequence. Those interested in the exact transforms employed are referred to Appendix A.

Example 2: Suppose 'SUB' is in the recognition domain. The original text on the left is transformed by the scanner into the text on the right.

Original

```
    DO  10 I = 1,3
    DO 20 J = 1,4
    IF (I.EQ.J) CALL SUB(I)
    IF (I.EQ.3) GO TO 10
20  IF (J.EQ.4) CALL SUB(4)
10  CONTINUE
```

Conditioned

```
    DO <&1> I = 1,3
    DO <&2> J = 1,4
    IF (.NOT.(I.EQ.J.)) GO TO <&3>
    CALL SUB(I)
<&3> CONTINUE
    IF (I.EQ.3) GO TO 10
    IF (.NOT.(J.EQ.4)) GO TO <&2>
    CALL SUB(4)
<&2> CONTINUE
 10  CONTINUE
<&1> CONTINUE
```

Legend:

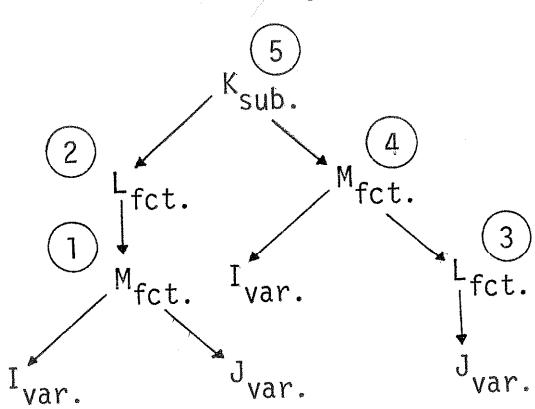
<&i> A label guaranteed to be unique with respect to those already in the program unit containing it.

In section 0 it was stated that the scanner activates a macro as soon as an activation phrase is reached. This implies that if a number of function references are imbedded in an expression, then they will be recognized and activated in the order that they would be reached by a left to right post-order traversal of the expression's parse diagram. Example 3 illustrates this.

Example 3: Assume the recognition table of Example 1. Then in the statement --

```
CALL K(L(M(I,J)), M(I,L(J)))
```

--the scanner will activate the macros in the order indicated on the parse tree below --



Thus if a number of different macro actions place text fragments just prior to the same A-statement fragment, then those fragments will occur in the order of activation of the corresponding routine references. But by the observation above this order is a reflection of the order of semantic evaluation of the A-statement's parse tree. Hence the requirement of the A-Statement Condition is in essence extended to the expression level.

3// MACRO ACTION SET STRUCTURE

Not every macro name in a macro action set need be in the recognition table. Those names that are in the recognition table are termed primary. All other macros in the action set are auxillary and are presumably used to modularize a complicated macro action. A frequent user can build libraries of auxillary macro actions and use them to augment the action set of a particular macro module. Of course, a macro module will have a set of action names which is a superset of the names in the recognition table.

The compiled macro actions utilize a number of system routines which implement --

1: A string data type sub-language.

2: A text transformation package.

3: A variety of primitives for analyzing the structure of the input stream as a FORTRAN program.

The semantics of BIGMAC's string extension are quite conventional and are discussed at length in the second part of this manual. The other features are the subject of the following sections.

The BIGMAC scanner recognizes a number of special activation phrases. These phrases are provided to coordinate macro actions with respect to the program structure of the input text. The macro actions which correspond to these phrases have been given special reserved names. They must be subroutines with no arguments. We have in tabular form --

Macro Name	Activation Phrase (AP)	Substitution for AP
SYSBEG	Empty string just left of the first character in input file.	As for a subroutine.
SYSEND	Empty string just right of the last character in input file.	As for a subroutine.
BLKMAC	Empty string just left of the first character in the END statement of a BLOCK DATA program unit.	Empty string.

(contd)

Macro Name	Activation Phrase (AP)	Substitution for AP
MANBEG	Empty string just left of the first character in the first executable statement in a main program.	As for a subroutine.
MANEND	A STOP statement in a main program.	'STOP'
ROUBEG	Empty string just left of the first character in the first executable statement in a subprogram.	As for a subroutine.
ROUEND	A RETURN statement in a subprogram.	'RETURN'
PRGEND	An END statement in an executable program.	'END'

Example 4: The contrived program below has been annotated to show the points where the special macros would be invoked if present in the action set --

```
SYSBEG ■
       COMMON /IT/ I
       INTEGER I,J,K
MANBEG ■
       READ (5,100) K
  100    FORMAT (I2)
       DO  10  J = 1,K
   10     J = I + SUM(J-1)
       WRITE (6,200) K,I
  200    FORMAT (1H1,2HF(,I2,2H)=,I4)
MANEND [STOP]
PRGEND [END]
       INTEGER FUNCTION SUM(J)
       INTEGER I,J
ROUBEG ■
       SUM = 0
       IF (J.EQ.0) [RETURN] ROUEND
       DO 10 I = 1,J
   10    SUM = SUM + J
ROUEND [RETURN]
PRGEND [END]
       BLOCK DATA
       COMMON /IT/I
       DATA I/0/
BLKMAC ■
PRGEND [END]
SYSEND ■
```

Legend:

xxxx Special Macro

▭ AP

■ Null string AP

The placement of commons in the SYSBEG macro makes those commons (and only those commons) global to the macro process. That is, the extent of these commons will be that of BIGMAC's entire execution. These common storage cells can be used for global communications between macro actions.

For example, one may wish to have a flag, INMAIN, which is true while the scanner is in the main routine of the input text and false at all other times. Putting INMAIN in a common block in the SYSBEG macro will guarantee that INMAIN will be available to all other macros containing the common block and further that the value of INMAIN will be preserved from invocation to invocation. INMAIN is initially set to false by the SYSBEG macro. The MANBEG macro should turn it on and the PRGEND macro will turn it off. It should be noted that common variables which are strings may not be equivalenced either implicitly or explicitly.

4// MACRO TRANSFORMATION PRIMITIVES

Just as the input stream is viewed as a sequence of text fragments, so is the output of the system. These fragments are also of four kinds --

1: Comment - An ANSI Standard comment line.

2: Label - An ANSI statement label.

3: Statement - An ANSI statement.

4: Error - An error message. These messages are printed out but do not occur in the final BIGMAC output.

The output sequence is built by appending fragments to the right end of the current sequence. These fragments are appended by the scanner and also the user via the text transformation primitives.

The scanner traverses the input in a forward scan. Every input fragment is appended to the output sequence *after* the scanner has traversed it. Thus if the scanner detects any errors while traversing a fragment then the scanner's error message fragments will preceed the fragment as the scanner appends the error fragments first. Similarly, for A-statement fragments the scanner will first activate all the associated macro actions before the resulting neutralized A-statement is appended as a statement fragment. Hence, if the macro actions append some fragments then these fragments will immediately preceed their respective neutralized A-statement fragment. Thus the necessity and utility of the A-Statement Condition.

The text transformation primitives allow the user's macro actions to append any of the fragment types onto the output sequence. There is a primitive for each type, i.e., the COMMENT, LABEL, EXECUTE, and ERROR statements of STREX described in the second part of this manual. Each primitive takes a user-constructed string and regardless of its syntax appends it to output as a fragment of the type specified by the primitive.

Any sequence of these primitives can be executed. A critical feature of BIGMAC is that it places a CONTINUE statement fragment between any two consecutive label fragments. This is semantically natural if one considers a FORTRAN in which statements can have any number of labels. Along the same lines, if the right most fragment

of the current output sequence is a label fragment and one is about to append either a comment or error fragment then this fragment is *not appended*, but placed to the immediate left of the label fragment. This guarantees that the label will refer to the next statement fragment appended and again is semantically natural if one considers a FORTRAN in which comments can occur arbitrarily within the code. These two precautions insure that if a user generates syntactically correct fragments, then the result of appending an arbitrary sequence of such fragments is also syntactically correct.

The above mechanism only allows the insertion of text into program parts, as activation phrases only occur in program parts. This implies that a user has no way of inserting specification statements into a program unit. This situation is alleviated by a special primitive which allows a macro action to insert statement fragments just before the first specification statement (an executable statement if there are no specification statements) in the program unit containing the associated activation phrase. The location for insertion is referred to as the declaration insertion point (DIP) of the program unit. See the DECLARE statement of STREX in the second part of this manual.

Example 5: Assume the recognition table of Example 1. Moreover assume that each macro action executes the sequence of primitives --

Macro Name	Action Sequence
M	<Spec.>,<Comm.>
L	<Stmt.>,<Label>
K	<Label>,<Stmt.>

The preprocessor will transform code as illustrated below --

```
   SUBROUTINE SIMPLE(I,J)                                SUBROUTINE SIMPLE(I,J)
   {DIP}                                                 <Spec(M1)>
10 CALL K(L(M(I,J)),M(I,L(J)))                           <Spec(M2)>
   RETURN                                                <Comm(M1)>
   END                                               10  <Stmt(L1)>
                                  <Label(L1)>            <Stmt(L2)>
                                                         <Comm(M2)>
                                  <Label(L2)>            CONTINUE
                                  <Label(K1)>            <Stmt(K1)>
                                                         RETURN
                                                         END
```

Legend:

(X_j) denotes the j^{th} activation of X.

5// NAME GENERATION AND EXPRESSION ANALYSIS

A generator for symbolic names and labels is available to the BIGMAC macro writer. The generator distinguishes four classes of identifiers as follows --

1:	Label	- An ANSI Standard statement label.
2:	Integer Name	- An ANSI symbolic name whose first character is one of the letters I, J, K, L, M, or N.
3:	Real Name	- An ANSI symbolic name which is not an integer name.
4:	Block Name	- An ANSI symbolic name.

For classes 1 through 3, each invocation of the generator returns an identifier of the appropriate type which is unique with respect to those of the program unit currently being scanned and those generated since the program unit's first character was traversed. For the class of block names, an invocation produces an identifier which is unique with respect to all the common block names in the input file and those generated thus far. See the GENER function in the second part of this manual.

A parser for FORTRAN expressions is provided primarily so that a user may analyze the arguments of a macro action. For example, one may wish to determine if an argument is a simple variable or a valid subscript expression. The parser will produce post-fix polish parse strings which may be examined and modified by the sophisticated macro writer via the string manipulation components of the STREX language. Any well-formed parse string may then be requested to yield the string it derives, i.e., the reverse of parsing. See the functions PARSE, GETTOK, ADDTOK, and RPARSE in the second part of this manual.

The BIGMAC user may also examine the environment surrounding a macro activation phrase. One may inquire as to the existence and usage of common block and variable names in the (input-text) procedure containing the activation phrase of the current macro invocation. See the functions VTHERE, CTHERE, and VTYPE in the latter part of this manual.

II. THE STREX MACRO LANGUAGE

0// PRELIMINARIES

STREX (STRing EXtended) FORTRAN provides the linguistic framework for programming BIGMAC macro-modules. Formally, STREX is a superset of a subset of ANSI FORTRAN. Section 1 outlines the relevant subset of ANSI FORTRAN. The other sections describe the superset in three parts --

1: String data type concepts.

2: Macro concepts.

3: Macro module concepts.

Throughout this part of the manual, there arise many occasions to reference the ANSI FORTRAN Standard (ANSI X3.9-1966). Such references will be abbreviated to lists of section numbers in square brackets.

An extended version of Backus Naur Form grammar rules will be used to describe the syntactic form of STREX features. Nonterminals will consist of a description of the token surrounded by angle brackets, e.g.,

<variable-reference>

Terminal strings will be enclosed in single quotes, e.g.,

'DO'

A plus sign will be used to compress several rules having the same left-hand sides, e.g.,

<A> ← <B> + <C> ≡ <A> ← <B>
<A> ← <C>

A superscript star will denote zero or more repititions, e.g.,

<A> ← <B>* ≡ <A> ← '' + <A> <B>

Curly braces will be used for bracketing purposes, e.g.,

<A> ← {<B> <C>}* <D> ≡ <A> ← <D>
<A> ← <B> <C> <A>

In STREX FORTRAN there are six different kinds of data types or modes --

INTEGER [4.2.1]	COMPLEX [4.2.4]
REAL [4.2.2]	LOGICAL [4.2.5]
DOUBLE PRECISION [4.2.3]	STRING

The term STANDARD will denote all the modes except STRING. In many cases the mode of the datum that a nonterminal will yield upon semantic evaluation must be specified. This will be done by writing -- <A ε TYPE> -- which specified that the nonterminal <A> when evaluated will yield a datum of mode TYPE.

1// FORTRAN BASE

The subset of ANSI FORTRAN that constitutes the base of STREX FORTRAN consists of all FORTRAN constructions which do not involve any of the forms listed below --

1: READ, WRITE, REWIND, BACKSPACE, & ENDFILE statements [7.1.3]

2: FORMAT statements [7.2.3]

3: Hollerith constants [4.2.6,5.1.1.6,7.2.2.,8.4.2]

The primary purpose of a macro action is to transform the input stream on the basis of the parameter strings passed to the macro by the BIGMAC scanner. These side effects are handled entirely by the text transformation primitives of STREX (see section 3.1). Hence I/O statements are not necessary. STREX's string capabilities supersede the need for Hollerith constants.

2// STRING FEATURES

Due to the text manipulative nature of macro processing, the notion of string data is fundamental. ANSI FORTRAN is very weak in this regard. STREX must and does provide a more powerful linguistic medium for expressing string manipulations.

A string datum is defined as an ordered sequence of FORTRAN characters [3.1]. In what follows a string value will be written as its sequence of characters surrounded by vertical bars. Blank characters *are* significant. The convention is adopted that the i^{th} character of a string value is the i^{th} counted from the left. The length of a string value is the total number of characters in its sequence.

Example 1: | A STRING | denotes a string value whose 3rd character is 'S' and whose length is 8.

The string data type is supported in STREX by the addition of constructions for declaring, assigning, and manipulating string data. These constructions are described in the following subsections.

STREX's string component is implemented in an interpretive fashion. This allows the semantic routines to trap run-time errors such as an attempt to use an undefined variable in an expression or an out-of-bounds subscript. However, a semantic error of this sort does not halt execution. An error fragment is issued, some conservative recovery action is taken, and execution continues. In the semantic descriptions that follow, error conditions are only noted. For a detailed description of the errors and their recovery actions see Appendix B.

2.1 // String Declarations:

Type statements are defined in section 7.2.1.6 of the ANSI Standard. In STREX type statements can also declare simple variables and arrays to be of string type according to the syntax --

1a. <type_statement> ← 'STRING' {<declaration> ','}* <declaration>

String mode functions referenced in a program unit may be declared in that program unit with the syntax --

1b. <type_statement> ← 'FSTRING' {<name> ','}* <name>

String statement functions are not allowed.

Function statements are introduced in section 8.3.1 of the ANSI Standard. STREX allows functions to return string values via the syntax --

2. <fct_statement> ← 'STRING FUNCTION' <fct_name>
'('{<formal_param>','}*<formal_param>')'

The semantic implications of these forms should be obvious from analogy with their Standard couterparts cited above.

Example 2:

```
STRING FUNCTION SFUN(AVAR,ALEN,SVAR)
STRING AVAR(10),SVAR
FSTRING SFUN2
INTEGER ALEN
DO 10 I=1,ALEN
10    AVAR(I) = SFUN2(I)
SFUN = SVAR
RETURN
END
```

2.2// String Assignments:

In STREX there are two types of string assignment statements. The syntax and semantics of the first form is a direct extension of the assignment statements defined in the ANSI Standard [7.1.1.1, 7.1.1.2] --

1. <assignment_statement> ← <variable_reference ε STRING>
'='<expression ε STRING>

If the variable reference is also a parameter of the program unit containing the assignment and further if this parameter's value was obtained by the pass-by-value mechanism then an error has occurred and is flagged. Otherwise, execution of this construction causes the referenced variable to bind to the string value resulting from the evaluation of the expression.

The second assignment form stems from the requirement in general string manipulators of being able to replace substrings of a string. STREX uses the syntax --

2. <assignment_statement> ←<variable reference ε STRING>
'('<expression$_1$εINTEGER>'..'<expression$_2$εINTEGER>')'
'='<expression$_3$εSTRING>

The semantics of this statement are somewhat involved. If the referenced variable is undefined or is a parameter satisfying the condition given in the paragraph above then an error is issued. Now suppose the variable is bound to a string datum S, that expression$_1$ evaluated to L, and expression$_2$ evaluates to R. If L<0 or R>length (S)+1 or L≥R then an error occurs. Otherwise the substring lying exclusively between the Lth and Rth characters of S is replaced by the value of expression$_3$.

Example 3:

① SVAR = AVAR

② SVAR (0..1) = AVAR

③ SVAR (1..3) = AVAR

④ AVAR (0..3) = SVAR

Value of SVAR	Value of AVAR	
undefined	|ABC|	initially
|ABC|	"	after ①
|ABCABC|	"	after ②
|AABCCABC|	"	after ③
"	|AABCCABCC|	after ④

2.3// String Expressions:

The syntax of STREX string expressions forms a simple 3-level precedence grammar --

Level 1:

<expression ε STRING> ← <term ε STRING> {'.'<term ε STRING>}*

(Concatenation)

Level 2:

<term ε STRING> ← <primary ε STRING>

{'('<expression$_1$ ε INTEGER>'..'<expression$_2$ ε INTEGER>')'}*

(Substring Selection)

Level 3:

<primary ε STRING> ← <reference ε STRING> (Reference)

+ '(' <expression ε STRING>')' (Parenthesis)

+ <constant ε STRING> (Constant)

+ '='<expression ε STANDARD>'=' (Conversion)

We discuss each of the constructions in the paragraphs below.

1. Reference:

As for arithmetic primaries [6.1], string primaries can be a variable, an array element, or a function reference whose mode is STRING. Exectution of a reference yields the value to which the reference is currently bound. If the reference is undefined then the recovery action is dependent on the semantic action requiring its value (see appendix B).

2. Parenthesis:

As for all ANSI expressions, parentheses can be used in string expressions to enforce an arbitrary order of evaluation.

3. Constant:

A string constant denotation has the following syntax --

<constant ε STRING> ← '$'<character_less_dollar>*'$'

<character_less_dollar> ← <{Fortran Characters}-{$}> + '$$'

Since dollar-sign is the delimiting character, one has to use two dollar-signs to denote one within the constant string.

Example 4: $HELLO$ has the value |HELLO|

$3$$$ has the value |3$|

A string constant may not be used in a data initialization statement. This restriction is somewhat cumbersome, but the effect is easily simulated by explicitly initializing the string variables in the SYSBEG macro (see part 1, section3).

4. Conversion:

An arithmetic or logical expression between equal-signs yields a string which is a constant denotation for the value of that expression. In the case of COMPLEX, DOUBLE PRECISION, and REAL expressions the denotation is only a close approximation and is machine dependent.*

Example 5:

=4/3= evaluates to |.133333333333333E+1| on the CDC 6600, but on IBM 370 the result is |.1333333E+1|

=1.GT.0= evaluates to |.TRUE.| on any machine.

5. Substring Selection:

Suppose the primary evaluates to the string S and further that $expression_1$ evaluates to L and $expression_2$ evaluates to R. If S is undefined or L < 0 or R > length (S) + 1 or L ≤ R then an error is issued. Otherwise the resulting term is the substring of S lying exclusively between the Lth and Rth characters. If more than one pair of selection indices is involved then they group from left to

* Currently, only INTEGER expressions can be converted.

right. For example, if V has the value |ABC| then V (1..4) (1..3) is equivalent to (V(1..4)) (1..3) and has the value |C|.

6. Concatenation:

The concatenation of a sequence of strings is expressed by separating each term by periods. If any term is undefined an error is noted. If subexpressions involving concatenation occur at more than fifty different and nested levels, then a stack error will occur. This should never happen in practice, e.g.

$$.($$.($$.$$)) has just 3 levels.

$$.$$.$$.$$ has just 1 level.

2.4// External Functions:

STREX provides three simple external functions. These are outlined in the chart below.

	Name	No. of arguments	Type of arguments	Type of result	Definition
1.	LENSTR	1	STRING	INTEGER	The length of the argument.
2.	EQSTR	2	STRING	LOGICAL	True only if the two string arguments are identical.
3.	SEARCH	2	STRING	INTEGER	0 if the first argument is not a substring of the second. Otherwise, SEARCH gives the index of the leftmost character in the leftmost occurrence of the first argument in the second.

3// MACRO FEATURES

BIGMAC's macro capabilities are discussed at some length in the last three sections of the first part of this manual. In particular, the semantics of text transformations and name generation are specified to the extent that this section only presents the STREX syntax for these primitives. The expression analysis package will be presented more thoroughly in subsection 3.3 below.

3.1// Transformation Primitives:

The text transformation primitives of BIGMAC are realized syntactically as STREX statements. The syntax of each statement is listed below along with a brief semantic description.

1. 'COMMENT' <expression ε STRING>

   - If the length of the expression is over 77 characters, an error message is issued and the string is truncated on the right. A fragment of the comment type is appended to output. The 'C' in column 1 and blanks in columns 2 and 3 are automatically generated before the fragment is printed by an output formatter. These fragments are not a part of the final output.

2. 'LABEL' <expression ε STRING>

   - If the expression is over 5 characters long, an error is issued and the string truncated on the right. If the string is the empty string then no action is taken. Otherwise, a fragment of the label type is appended to output. Labels are right justified by the BIGMAC formatter.

3. 'ERROR' <expression ε STRING>

   - The string may be of arbitrary length. The BIGMAC formatter breaks the string into a number of lines if necessary and tacks a header in front of it. These fragments are not a part of the final output.

4. 'EXECUTE' <expression ε STRING>

   - If the expression is over 1320 characters long an error is issued and the expression is truncated on the right. A fragment of the statement type is append to output unless the string is the empty string in which case no action is taken. The BIGMAC formatter automatically breaks the string into continuation lines if necessary.

5. 'DECLARE' <expression ε STRING>

   - The effect is the same as EXECUTE above except that the fragment is inserted at the DIP of the program unit currently being scanned.

6. 'EXECUTE' <expression$_1$ ε STRING>','<expression$_2$ ε STRING>

   - This statement is an abbreviation for the two statement sequence --

```
LABEL <expression1>
EXECUTE <expression2>
```

3.2// Unique Name Generator:

The BIGMAC unique name generator is a STREX-supported explicit function whose result is a unique name. Formally --

<expression ε STRING> ← 'GENER('<type>')'.

<type> ← 'L' + 'I' + 'R' + 'C'.

The type argument specifies which class of name is being requested according to the scheme --

L - Label

I - Integer Name

R - Real Name

C - (Common) Block Name

If it happens that the generator has exhausted all the possible names within a given program unit (it probably never will) then an error message is issued and the generator starts to recycle through the names for the overflowing class.

3.3// Expression Analysis Primitives:

In order to syntactically analyze an expression one must have the information provided by a parsing structure. A parse tree is too complex a structure to support. For an expression, its Polish prefix form conveniently conveys its structure in a one-dimensional form. Additionally, the Polish representation can be encoded as a *string* of tokens, thus allowing one to examine and modify it with the string manipulation features of STREX.

The components of a FORTRAN expression are grouped into a number of tokens. Each token has a unique integer code. The tokens and their codes are --

Operands		Operators	
Code	Token	Code	Token
0	<function or array reference argument>	10	'**'
1	<simple variable>	11	'*'
2	<array reference>	12	'/'
3	<function reference>	13	'+' (binary)
4	<unsigned constant ε INTEGER>	14	'-' (binary)
5	<unsigned constant ε REAL>	15	<relational-operator>
6	<constant ε LOGICAL>	16	'.AND.'
7	<unsigned constant ε DOUBLE PRECISION>	17	'.OR.'
8	<constant ε COMPLEX>	18	'.NOT.'
9	<constant ε HOLLERITH>	19	'-' (unary)

Note that function or array arguments are not further decomposed but left intact. The other pieces of information contained in a token are a pair of indices which indicate the positions within the original expression between which lies the actual string corresponding to the token. In summary a token is encoded as a triple of integers, the first element of which is the token code, the second is the left index, and the third the right index. For example, the token for the array reference in the string |3+AB(4)| is <2,2,8>.

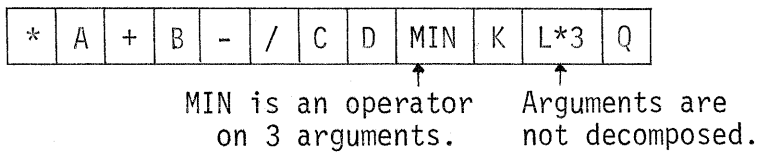
An expression is parsed as a string of token triples in Polish prefix form hereafter referred to as a token-string. The boundaries of the triples are not distinguished. That is, a string of n tokens is physically a string of 3n integers; the user must break the string into triples with his code. Further, the arguments of a function or array reference immediately follow the reference token. This is consistent with prefix notation if one considers an array a function reference as an *operator* on the arguments.

Example 6:

The expression --

```
| A * ( B + ( ( C / D - M I N ( K , L * 3 , Q ) ) ) |
  - - - - 5 - - - - 0 - - - - 5 - - - - 0 - - - - 5
```

is in Polish prefix form --



where each block is a token.

The expression's final token-string encoding is --

```
| 11, 1, 3 | 1, 0, 2 | 13, 4, 6 | 1, 3, 5 | 14,11,13
| 12, 8,10 | 1, 7, 9 |  1, 9,11 | 3,12,25 |  0,16,18
|  0,18,22 | 0,22,24 |
```

The parsing package consists of four routines. The header section of each routine is given below along with a discussion of its effect. Note the close association required between a token-string and the original string from which it was created by the PARSE routine.

1. STRING FUNCTION PARSE (EXPSTR)

STRING EXPSTR

EXPSTR is an input parameter whose value is a string denoting a valid ANSI FORTRAN expression. The result of PARSE is the token-string for the expression in EXPSTR. The index pairs of every token refer to locations in EXPSTR.

2. INTEGER FUNCTION GETTOK(TOKSTR,I,LINDEX,RINDEX)

INTEGER I, LINDEX, RINDEX

STRING TOKSTR

TOKSTR is an input parameter which is a token-string of say N tokens. I is an input parameter and LINDEX and RINDEX are output parameters. If I ≤ 0 or I > N then GETTOK returns -1. Otherwise, GETTOK returns the token code of the Ith token (from the left) and also has the side-effect of setting LINDEX and RINDEX to the left and right indices of the Ith token.

3. STRING FUNCTION ADDTOK(TOKCOD,ACTSTR,EXPSTR)

STRING ACTSTR, EXPSTR

INTEGER TOKCOD

TOKCOD is an input parameter which should be a valid token code. ACTSTR is an input string which should represent an instance of the token specified by TOKCOD. EXPSTR is used for both input and output. ADDTOK returns a token triple whose code is TOKCOD and whose index pair delimit the string ACTSTR. ADDTOK has the side-effect of appending ACTSTR to the end of EXPSTR, i.e., 'EXPSTR = EXPSTR.ACTSTR'.

4. STRING FUNCTION RPARSE(TOKSTR,EXPSTR)

STRING TOKSTR, EXPSTR

TOKSTR and EXPSTR are both input parameters. TOKSTR must be a token-string for which all index pairs refer to locations in EXPSTR (hence the necessity for the EXPSTR argument in ADDTOK). The result of RPARSE is the string derived by the token-string TOKSTR, i.e., RPARSE performs the reverse of the action of PARSE.

Token-strings are strings of integers *not* characters. Thus although one can manipulate token-strings with STREX string features, the user must be careful not to apply any features which actually examines the value of a cell in the string, e.g., =<expression>= or EXECUTE <expression>. Of course one shouldn't mix strings and token-strings.

Example 7:

The function REPL replaces all occurrences of the variable LEN with the constant 9 --

```
      STRING FUNCTION REPL(EXPR)
      STRING EXPR, TOKSTR
      TOKSTR = PARSE(EXPR)
      K = LENSTR(TOKSTR)/3
      IF (K.EQ.0) GO TO 20
      DO 10 I = 1,K
         IT = GETTOK(TOKSTR,I,M,N)
         IF (ITOK.NE.1) GO TO 10
         IF (.NOT.EQSTR(EXPR(M..N),$LEN$)) GO TO 10
         TOKSTR(3*I-3 .. 3*I+1) = ADDTOK(4,$9$,EXPR)
   10 CONTINUE
   20 REPL = RPARSE(TOKSTR,EXPR)
      RETURN
      END
```

A call of the form --

```
S = REPL($3*LEN+A(LEN)$)
```

results in S having the value --

```
|3*9+A(9)|
```

3.4// Environment Primitives

A number of system functions are provided for analyzing the local and global variables of the procedure currently being scanned when a macro expansion takes place. There are functions for determining whether or not a given common block or variable name is present in the procedure. One can also determine the type of a variable which is local to the scanned program unit. The functions are --

1. LOGICAL FUNCTION CTHERE(NAME)

   STRING NAME

   CTHERE is true if and only if a common block named NAME is in the currently scanned procedure. If name is undefined CTHERE is false.

2. LOGICAL FUNCTION VTHERE(NAME)

   STRING NAME

   VTHERE is true if and only if a variable named NAME appears in the scanned procedure. If name is undefined VTHERE is false.

3. INTEGER FUNCTION VTYPE(NAME,ARRAY)

   STRING NAME

   LOGICAL ARRAY

   VTYPE is an integer code for the type of the variable named NAME --

VTYPE	
0	NAME is undefined.
1	NAME does not appear as a variable name in the procedure
2	NAME appears as an INTEGER variable.
3	NAME appears as a REAL variable.
4	NAME appears as a DOUBLE PRECISION variable.
5	NAME appears as a COMPLEX variable.
6	NAME appears as a LOGICAL variable

The flag ARRAY will be true if and only if NAME is used as an array name within the program unit.

Example 8:

Text:
```
SUBROUTINE EXAMP
COMMON /A/ X,Y,Z(3)
 .
 .
 .
I=FUNMAC(J)
 .
 .
 .
END
```

Macro:
```
STRING FUNCTION FUNMAC(S)
STRING S
LOGICAL CTHERE,VTHERE,L(7)
INTEGER VTYPE
 .
 .
 .
DECLARE $COMMON /B/ R,S,T(3)$
L(1) = CTHERE ($A$)
L(2) = CTHERE ($B$)
L(3) = CTHERE ($/A/$)
L(4) = VTHERE ($T$)
L(5) = VTHERE ($A$)
I1   = VTYPE ($T$, L(6))
I2   = VTYPE ($Z$, L(7))
 .
 .
 .
END
```

When the macro FUNMAC is invoked the values of L, I1, and I2 will be --

L(1) - .TRUE.

L(2-6) - .FALSE.

L(7) - .TRUE.

I1 - 1

I2 - 3

4// MACRO FILES

A STREX macro file is a file or other line-oriented text stream consisting of a number of MACRO lines and STREX program units terminated by a line containing a dollar-sign in column 7. A STREX macro file is a complete specification of a macro module.

A MACRO line is a line containing the phrase 'MACRO' in columns 7 through 11. These lines must occur just before a program unit. They indicate that the following STREX routine is a primary macro action. The name of the routine is put in the recognition table along with its type (subroutine or function) and the number of arguments it requires. The formal parameters of such a routine must all be simple string variables and if the unit is a function it must return a string value.

Any routine in a macro file with the name --

SYSBEG,	PRGEND,	ROUBEG,
SYSEND,	MANBEG,	or ROUEND
BLKMAK,	MANEND,	

must be a primary action which is a subroutine with no arguments. Recall that these are the names of the special actions discussed in section 3 of the first part of this manual.

III. APPENDICES

A// SCANNER TRANSFORMATIONS:

In section 2 of part 1 the A-statement condition is presented. In order to meet this condition it suffices to insure that

1: Each A-statement is a basic block.*

2: Any label preceeding an A-statement, is not referenced by a DO-statement.

The transforms employed by the scanner are given below. The following legend is used

|| - text fragment boundaries.

<> - non-terminal syntactic tokens.

&lab - a label guaranteed to be unique with respect to those already in the containing program unit.

A logical IF statement is the only statement which is compound in terms of basic blocks. Hence to achieve 1 above it suffices to 'split' every logical IF statement in which the statement part is an A-statement. The diagram below indicates the transform used --

```
||IF (<expr>) <A-statement> ||
          ≋
          ▼
||IF (.NOT.(<expr>)) GO TO &lab
||<A-statement>
|| &lab || CONTINUE ||
```

The solution for 2 is difficult to motivate it as it was primarily the result of implementation issues. The set of transforms employed by the BIGMAC scanner are just listed below.

---

* L. D. Fosdick, 'BRNANL, A Fortran Program to Identify Basic Blocks in Fortran Programs', Univ. of Colo., Tech. Rpt. #CU-CS-040-74(1974)

Case 1: DO label is referenced only by DO statement

Case 1.1: DO terminal statement is an N-statement

```
|| DO <DO-label> <loop> ||                    || DO &lab <loop> ||
        :                    ~~~~>                  :
|| <DO-label> || < N-statement> ||            || &lab || < N-Statement> ||
```

Case 1.2: DO terminal statement is an A-statement

```
|| DO <DO-label> <loop> ||                    || DO &lab <loop> ||
        :                    ~~~~>                  :
|| <DO-label> || <A-Statement> ||             || <A-Statement> ||
                                                || &lab ||CONTINUE ||
```

Case 2: DO label is referenced by a statement other than a DO-statement.

```
|| DO <DO-label> <loop> ||                    || DO &lab <loop> ||
        :                    ~~~~>                  :
|| <DO-label> || < Statement> ||              || <DO-label>|| <Statement>
                                                || &lab      || CONTINUE ||
```

Conceptually the solutions for 1 and 2 can be applied in parallel. However, some inefficiencies result. Consider then that case where a DO terminal statement is also a logical IF requiring splitting. The appropriate DO transform is applied (either case 1.2 or 2 above) and then the following abbreviated transform is used for the logical IF statement --

```
|| IF (<expr>) <A-statement> || ~~~~> || IF (.NOT.(<expr>)) TO TO &lab
                                      || <A-statement> ||
```

Where &lab is the *same* as that used in the DO transform. The reader is now referred to example 2 in the first part of this manual.

B// ERROR MESSAGES

There are two phases to using BIGMAC. One first compiles a macro module using the STREX compiler and then one uses the compiled macro module to expand some input text. Errors are detected at both stages. The STREX compiler's error checking is not very extensive. It basically only tries to insure that the string and macro components of STREX programs are properly translated. The compiler error set is listed below in the section entitled "Compiler Errors."

The errors that occur at the time a macro module is applied to a stream of text are somewhat more extensive. There are two classes of such errors. The first set of errors are those that arise when the input text is found not to be ANSI FORTRAN code (Text Errors). The second type of errors are macro routine errors trapped by the BIGMAC interpretive primitives. They involve such items as undefined variables and subscript range errors. The semantic action in progress when an error is detected is given in parentheses along with the error message. A corrective action is taken by the primitive and processing continues. The specific recovery actions are given in the list below.

Compiler Errors

Message	Meaning
NAME TOO LONG - <name>	The variable <name> is more than 6 characters long.
LABEL TOO LONG - <label>	The label <label> is more than 5 characters long.
END OF FILE	An end of file mark was reached while still in the middle of a macro.
NO EXECUTABLE STATEMENT	A program unit with no executable statements (not even a RETURN) was encountered.
NO SYSBEG MACRO	At least one common was used in the macro file, but none were declared in a SYSBEG macro.
TOO MANY MACROS	The STREX compiler's internal structures have overflowed (not likely to happen).

DUPLICATE MACRO - <macro name>

A name with the name <macro name> is declared more than once.

MACRO ARG NOT STRING

All macro arguments and macro functions must be declared to be STRING.

ILLEGAL SPECIFICATION - <name>

The variable <name> has been given conflicting type attributes in different type statements.

SYSTEM COMMON NAME - <common-block name>

Certain common block names are used by the system and must not be used by the user. They are -- BUFP, CONCAT, GLOP, ICFILE, IOFILE, LEXIC, LIF, LISTC, MAC, MACSAR, MFLAGS, PTR, PTRS, SYST, TABLE, WRKKEY.

IMPROPER STATEMENT

The compiler cannot make sense of the statement.

STMT ILLEGAL HERE

A statement is out of place, e.g. a declarative statement following an executable one.

ILLEGAL EXPRESSION AT <number> <type code>

The expression is ill-formed. The error was first detected at the <number>th character of the expression and at that point the expression was of type <type code> where the code is --

N - no type

S - STRING

I - INTEGER

R - REAL

C - COMPLEX

D - DOUBLE PRECISION

L - LOGICAL

Text Errors

Message	Meaning
NAME TOO LONG TRUNCATED TO 8 CHARACTERS - <name>	Obvious
LABEL TOO LONG TRUNCATED TO 5 CHARACTERS - <label>	Obvious
END OF FILE	End of file mark does not directly follow an END statement.
NO EXECUTABLE STATEMENT	A program unit with no executable statements was encountered.
IMPROPER MACRO REFERENCE - <macro name> TYPE DOES NOT MATCH	Macro is a subroutine while reference is a function or vice versa.
IMPROPER MACRO REFERENCE - <macro name> WRONG NUMBER OF ARGUMENTS	Reference has more or less arguments than the macro invoked.
NAME ALREADY TYPED - <name>	The variable name has been typed twice.
BLOCK NAME TOO LONG TRUNCATED TO 6 CHARACTERS	Obvious

Run Time Errors

Message (Sources)	Meaning (Recovery Action)
PASS 2 - UNDEFINED VARIABLE	A string variable is undefined -
(ADDTOK)	Returns a token which has a negative token code.
(ASSIGN)	No assignment takes place.
(ASSIGNI)	No assignment takes place.
(COMMENT)	No comment is placed on output.
(CONCATE)	Null string is substituted.
(CTHERE)	.FALSE. is returned.
(DECLARE)	No declaration is placed on output.
(EQUALS)	.FALSE. is returned.
(ERROR)	No error is placed on output.
(EXECUTE)	No statement is placed on output.
(FUNCTION)	Null string is the result of the function.
(GETTOK)	Returns a negative token code.
(LABEL)	No label is placed on output.
(LENGTH)	Ø is returned.
(PARSE)	Null string is the result.
(RPARSE)	Null string is returned.
(SELECTION)	Null string is the result.
(VTHERE)	.FALSE. is returned.
(VTYPE)	Ø is returned.
PASS 2 - SUBSCRIPT OUT OF BOUNDS	Obvious -
(ASSIGNI)	No assignment takes place.
(GETTOK)	Negative token code is returned.
(SELECTION)	Null string is the result.

PASS2 - LABEL TOO LONG	Obvious -
(EXECUTE)	Label is truncated to 5 characters.
(LABEL)	Label is truncated to 5 characters.
PASS2 - STATEMENT TOO LONG	Obvious -
(DECLARE)	Declaration is truncated to 1320 characters.
(EXPANSION)	Statement is truncated to 1320 characters.
(EXECUTE)	Statement is truncated to 1320 characters.
PASS2 - COMMENT TOO LONG	Obvious -
(COMMENT)	Comment is truncated to 77 charters.
PASS2 - EXPRESSION TOO DEEP	Nesting depth of concatenation expression is too deep -
(CONCATE)	Null string is the result.
PASS2 - NAME GENERATOR OVERFLOW	Name generator has exhausted all possible names of the indicated type -
(INTEGER) (GENERATE)	
(REAL) (GENERATE)	Generator recycles, i.e., starts over again.
(LABEL) (GENERATE)	
(COMMON) (GENERATE)	
PASS2 - CANNOT ASSIGN AN EXPRESSION	A parameter passed by value is the left-handside of an assignment -
(ADDTOK)	Returns a negative token code.
(ASSIGN)	No assignment takes place.
(ASSIGNI)	No assignment takes place.
PASS2 - BAD TOKEN	An invalid token occurs in a token list --
PASS2 - BAD TOKEN LIST	The syntactic structure of a token list is invalid --
(RPARSE)	Returns the null string.

The pneumonics for the source of the errors are indicated below by giving section and item numbers --

ADDTOK	3.3.3	FUNCTION	2.1.2
ASSIGN	2.2.1	GENERATE	3.2
ASSIGNI	2.2.2	GETTOK	3.3.2
COMMENT	3.1.1	LABEL	3.1.2 and 6
CONCATE	2.3.6	LENGTH	2.4.1
CTHERE	3.4.1	PARSE	3.3.1
DECLARE	3.1.5	RPARSE	3.3.4
EQUALS	2.4.2	SELECTION	2.3.5
ERROR	3.1.3	VTHERE	3.4.2
EXECUTE	3.2.4 and 6	VTYPE	3.4.3

The one pneumonic which cannot be described this way is EXPANSION which is the action of substituting an A-statement fragment by its neutralized counterpart.

C// BIGMAC ON THE C.U. CDC-7600

Using the BIGMAC system takes two steps. The first is to try to compile a STREX macro file into a macro module. A macro module is a multi-record file consisting of three object modules and one data record. These modules and data when combined with the rest of the BIGMAC expansion component, implement the macro process specified by the STREX file. Applying this processor to some AINSI FORTRAN text file constitutes the second step.

In terms of KRONOS control language the first step takes the form -

```
GET (BIGCMP/PJ = PAIX,ID = A246)
CALL (BIGCMP, RENAME/S = ?, M = ?)
```

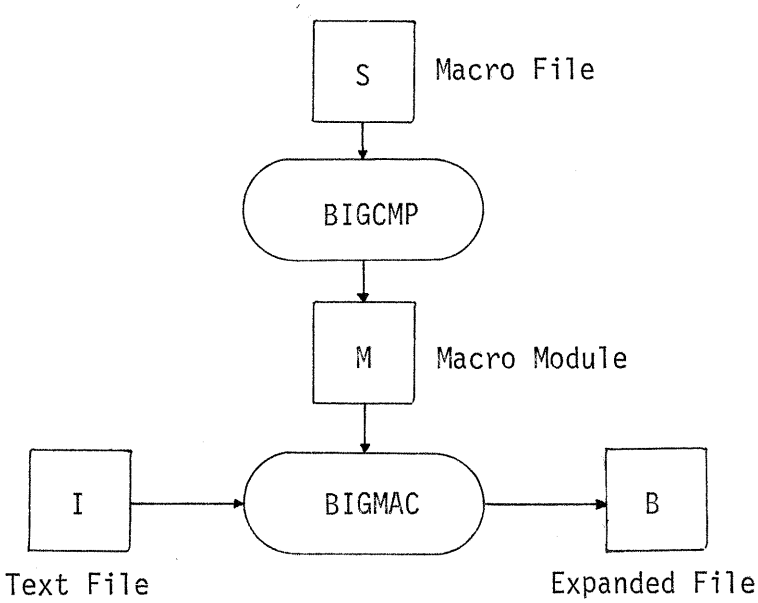
The S parameter is the name of the STREX macro file and the M parameter is the resulting macro module. This step requires 125_8K and depending on how heavily the string and macro features of STREX are used, compiles code at a rate between 5 and 40 lines/second.

The second step is carried out by the procedure BIGMAC which has three parameters --

```
GET BIGMAC/PJ = PAIX,ID = A246)
CALL (BIGMAC, RENAME/M = ?, I = ?, B = ?)
```

The M parameter must be the name of a macro module created by BIGCMP above. The I parameter is the AINSI FORTRAN text to be expanded and the B parameter is the result of the expansion. This step requires 100_8K plus enough extra core to fit the object modules of the macro module. The processing rate is 60 lines/second plus the time spent in the macros themselves.

The diagram below summarizes the steps --



D// SAMPLE APPLICATIONS

Three examples will be given below. The computer printouts for each example consist of three segments in the following order --

1: The macro file for the example.

2: The input text to be expanded by 1.

3: The BIGMAC listing of the expansion of 2 by 1.

The printout for the first example also contains the object code (FORTRAN) for the macros in the macro file and the day file for the job.

The first application illustrates the usage of the parser and environment primitives. An auxillary macro AINSI(INDEX) takes the expression INDEX and determines whether or not it is a legal AINSI subscript expression. A function macro BNS(INDEX) replaces itself with a reference to the INDEXth element of an array B. BNS assigns the index to a temporary variable if and only if AINSI(INDEX) is false.

```
      MACRO
      STRING FUNCTION BNS(IND)
      STRING IND,S
      LOGICAL AINSI
C
C  THIS MACRO ASSIGNS THE INDEX TO A TEMPORARY VARIABLE IFF THE
C    INDEX IS NOT A LEGAL AINSI INDEX.
C
      IF (AINSI(IND)) GO TO 10
        S = GENER(I)
        EXECUTE S . $=$ . IND
        BNS = $B($ . S . $)$
        RETURN
10    BNS = $B($ . IND . $)$
      RETURN
      END
      LOGICAL FUNCTION AINSI(IND)
      FSTRING PARSE
      STRING IND,PS
      INTEGER VTYPE,GETTOK
      LOGICAL LOG
C
C  LET . DENOTE CONCATENATION AND $ DENOTE UNION.  LET C BE AN UN-
C    SIGNED INTEGER CONSTANT AND V BE AN INTEGER VARIABLE.  AN EXPRES-
C    SION IS A LEGAL AINSI SUBSCRIPT IF AND ONLY IF ITS POSTFIX POLISH
C    STRING IS IN THE REGULAR LANGUAGE --
C
C      C$(+$-).(V.C$*.(C.V.C$V.C.C))$(+.C$).(V$*.(V.C$C.V))
C
      PS = PARSE(IND)
      K = 1
      J = GETTOK(PS,K,IL,IR)+1
      GO TO (200,90,200,200,100,200,200,200,200,200,200,40,200,10,20,
     +       200,200,200,200,200,200),J
10    K = K+1
      J = GETTOK(PS,K,IL,IR)
      IF (J.EQ.11) GO TO 30
      IF (J.NE.4) GO TO 50
      K = K+1
      J = GETTOK(PS,K,IL,IR)
      IF (J-11) 80,40,80
20    K = K+1
      J = GETTOK(PS,K,IL,IR)
      IF (J.NE.11) GO TO 50
30    K = K+2
      L = GETTOK(PS,K-1,IP,IQ)
      J = GETTOK(PS,K,IL,IR)
      IF (L.EQ.4) GO TO 50
      IF (L.NE.1) GO TO 200
      IF (VTYPE(IND(IP..IQ),LOG).GT.2) GO TO 200
      IF (J-4) 200,60,200
40    K = K+1
      J = GETTOK(PS,K,IL,IR)
      IF (J.EQ.4) GO TO 70
50    IF (J.NE.1) GO TO 200
      IF (VTYPE(IND(IL..IR),LOG).GT.2) GO TO 200
60    K = K+1
      J = GETTOK(PS,K,IL,IR)
      IF (J-4) 200,100,200
70    K = K+1
      J = GETTOK(PS,K,IL,IR)
80    IF (J.NE.1) GO TO 200
```

```
 90 IF (VTYPE(IND(IL..IR),LOG).GT.2) GO TO 200
100 AINSI = .TRUE.
    RETURN
200 AINSI = .FALSE.
    RETURN
    END
```

```
                INTEGERFUNCTIONBNS(IND)
000003          INTEGERA0(1),A00(2),A01(1),A02(2),A03(1)
000003          INTEGERIND,S
000003          LOGICALAINSI
          C
          C  THIS MACRO ASSIGNS THE INDEX TO A TEMPORARY VARIABLE IFF THE
          C    INDEX IS NOT A LEGAL AINSI INDEX.
          C
000003          CALLMXLSSP(000002)
000005          CALLMXLSLV(BNS)
000007          CALLMXLSPM(IND)
000011          CALLMXLSLV(S)
000013          IF(AINSI(IND))GOTO10
000017          CALLMXASSX(S,MXGENE(2))
000024          CALLMXCBEG(S)
000026          DATAA0/1H=/
000026          CALLMXCMID(MXLITT(1,A0))
000032          I0=MXCEND(IND)
000036          CALLMXEXEC(0,I0)
000040          DATAA00/1HB,1H(/
000040          CALLMXCBEG(MXLITT(2,A00))
000044          CALLMXCMID(S)
000046          DATAA01/1H)/
000046          I00=MXCEND(MXLITT(1,A01))
000053          CALLMXASSX(BNS,I00)
000055          CALLMXLSEP(BNS)
000057          RETURN
000061          DATAA02/1HB,1H(/
000061       10 CALLMXCBEG(MXLITT(2,A02))
000065          CALLMXCMID(IND)
000067          DATAA03/1H)/
000067          I01=MXCEND(MXLITT(1,A03))
000074          CALLMXASSX(BNS,I01)
000076          CALLMXLSEP(BNS)
000100          RETURN
000102          END
```

```
            LOGICALFUNCTIONAINSI(IND)
000003      INTEGERPARSE
000003      INTEGERIND,PS
000003      INTEGERVTYPE,GETTOK
000003      LOGICALLOG
      C
      C  LET . DENOTE CONCATENATION AND $ DENOTE UNION.  LET C BE AN UN-
      C    SIGNED INTEGER CONSTANT AND V BE AN INTEGER VARIABLE.  AN EXPRES-
      C    SION IS A LEGAL AINSI SUBSCRIPT IF AND ONLY IF ITS POSTFIX POLISH
      C    STRING IS IN THE REGULAR LANGUAGE --
      C
      C      C$(+$-).(V.C$*.(C.V.C$V.C.C))$(+.C$).(V$*.(V.C$C.V))
      C
000003      CALLMXLSSP(000001)
000005      CALLMXLSPM(IND)
000007      CALLMXLSLV(PS)
000011      CALLMXASSX(PS,PARSE(IND))
000017      K=1
000020      J=GETTOK(PS,K,IL,IR)+1
000025      GOTO(200,90,200,200,100,200,200,200,200,200,200,40,200,10,20,200,2
           *00,200,200,200,200),J
000056   10 K=K+1
000060      J=GETTOK(PS,K,IL,IR)
000064      IF(J.EQ.11)GOTO30
000066      IF(J.NE.4)GOTO50
000070      K=K+1
000071      J=GETTOK(PS,K,IL,IR)
000075      IF(J-11)80,40,80
000100   20 K=K+1
000102      J=GETTOK(PS,K,IL,IR)
000106      IF(J.NE.11)GOTO50
000110   30 K=K+2
000112      L=GETTOK(PS,K-1,IP,IQ)
000120      J=GETTOK(PS,K,IL,IR)
000124      IF(L.EQ.4)GOTO60
000127      IF(L.NE.1)GOTO200
000130      IF(VTYPE(MXSELT(IND,IP,IQ),LOG).GT.2)GOTO200
000142      IF(J-4)200,60,200
000144   40 K=K+1
000146      J=GETTOK(PS,K,IL,IR)
000152      IF(J.EQ.4)GOTO70
000154   50 IF(J.NE.1)GOTO200
000156      IF(VTYPE(MXSELT(IND,IL,IR),LOG).GT.2)GOTO200
000170   60 K=K+1
000172      J=GETTOK(PS,K,IL,IR)
000176      IF(J-4)200,100,200
000201   70 K=K+1
000203      J=GETTOK(PS,K,IL,IR)
000207   80 IF(J.NE.1)GOTO200
000212   90 IF(VTYPE(MXSELT(IND,IL,IR),LOG).GT.2)GOTO200
000224  100 AINSI=.TRUE.
000225      CALLMXLSEP(0)
000227      RETURN
000231  200 AINSI=.FALSE.
000232      CALLMXLSEP(0)
```

```
000234          RETURN
000236          END
```

```
      SUBROUTINE DUMMY
      INTEGER B(111)
C
      K = BNS(I+J)
      K = BNS(-3*J)
      K = BNS(4)
      K = BNS(4+2)
      K = BNS(3+I)
      K = BNS(4*A)
      K = BNS(I + 3)
      K = BNS(2-3*I)
      K = BNS(3*I-2)
C
      RETURN
      END
$
```

```
      SUBROUTINEDUMMY
      INTEGERB(111)
C
      I0=I+J                                                    + BNS
      K=B(I0)                                                   + (SUBS)
      I00=-3*J                                                  + BNS
      K=B(I00)                                                  + (SUBS)
      ------                                                    + BNS
      K=B(4)                                                    + (SUBS)
      I01=4+2                                                   + BNS
      K=B(I01)                                                  + (SUBS)
      ------                                                    + BNS
      K=B(3+I)                                                  + (SUBS)
      I02=4*A                                                   + BNS
      K=B(I02)                                                  + (SUBS)
      ------                                                    + BNS
      K=B(I+3)                                                  + (SUBS)
      I03=2-3*I                                                 + BNS
      K=B(I03)                                                  + (SUBS)
      ------                                                    + BNS
      K=B(3*I-2)                                                + (SUBS)
C
      RETURN
      END
```

*** SUMMARY OF DUMMY

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
BNS	9	-	0	0	5
	+++++		++++++	++++++	++++++
TOTALS	9	-	0	0	5

EXPANSIONS - 9

*** SUMMARY OF EOF

NO ACTIVATIONS

*** FINAL SUMMARY

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
BNS	9	-	0	0	5
TOTALS	9	-	0	0	5

EXPANSIONS - 9

```
JOB COST  A246CCU  PAIX/A246 78/11/13  14.06.02   $    4.66 (NORMAL)

 PROCESSING                                       $    0.89
   CP                 13  SEC        $    0.33
   PP                 32  SEC        $    0.56
 MEMORY OCCUPANCY                                 $    2.19
   CM                331  WD-HRS     $    2.19
 MASS STORAGE                                     $    1.33
   MSPRU            4336  PRU        $    0.27
   MSACC             571  ACC        $    1.06
 I/O PROCESSING                                   $    0.19
   LP                230  LINES      $    0.19
 I/O MATERIALS                                    $    0.06
   LP                 10  PAGES      $    0.06
                                                  ---------
 TOTAL COST                                       $    4.66

 PF ACCUMULATION RATE PER DAY        $    2.51
 ACCOUNT BALANCE                     $   29.82


        A246CCU.   78/11/13. CU16 KRONOS 2.1 - 9.1


13.55.19. SUBMITTED JOB.
13.55.19.A246,CM=125000,TL=70,DC=HS.
13.55.19. CCID/PW PREVIOUSLY VALIDATED.
13.55.19.PROJECT,PAIX.
13.55.20.GET,AINSI.
13.55.21.GET,BIGCMP.
13.55.22.CALL(BIGCMP,RENAME(S=AINSI,M=MAINSI)
13.55.24.GET,MACPCC/PU=PAIX,ID=A246.
13.55.26.GET,CMPOBJ/PU=PAIX,ID=A246.
13.55.30.GET(SYSDMP,DRIVEA,DRIVEB/PU=PAIX,ID=A246
13.55.30.)
13.55.32.COPYRF,BF,MACPCC,GO.
13.55.33.COPY COMPLETE        1 FILES
13.55.33.REWIND(GO)
13.55.33.OFFSW(1)
13.55.34.LOAD(CMPOBJ)
13.55.57.LDSET(LIB=RUNLIB)
13.55.57.GO(LC=0,AINSI,OBJMAC,MACPCC,SYSDMP,PPRT)
13.56.00.FL TO LOAD  116200   FL TO RUN  122100
13.56.18.STOP
13.56.18.IF(SW1)GOTO,111.
13.56.18.REWIND(AINSI)
13.56.18.COPYRF,SBF,AINSI,OUTPUT.
13.56.19.EOI ENCOUNTERED      0 FILES
13.56.19.COPYRF,CF,DRIVEA,DRIVER.
13.56.20.EOI ENCOUNTERED      0 FILES
13.56.20.COPYRF,CF,MACGLO,DRIVER.
13.56.20.EOI ENCOUNTERED      0 FILES
13.56.20.COPYRF,CF,DRIVEB,DRIVER.
13.56.20.EOI ENCOUNTERED      0 FILES
13.56.21.REWIND(DRIVER)
13.56.21.PACK(DRIVER)
13.56.21. PACK COMPLETE.
13.56.40.RUN(L=0,I=ACTEXC,B=ABB)
13.58.41.  032600  OCTAL REQUIRED
13.58.41.REWIND(ABB)
14.02.01.RUN(L=0,I=DRIVER,B=DBB)
14.04.38.  033000  OCTAL REQUIRED
```

```
14.04.38.REWIND(DBB)
14.05.19.RUN(I=OBJMAC,B=OBB)
14.05.21.  033000  OCTAL REQUIRED
14.05.21.REWIND(OBB)
14.05.22.COPYRF,BF,ABB,LL.
14.05.21.COPY COMPLETE       1 FILES
14.05.22.COPYRF,BF,DBB,LL.
14.05.22.COPY COMPLETE       1 FILES
14.05.22.COPYRF,BF,OBB,LL.
14.05.23.COPY COMPLETE       1 FILES
14.05.23.REWIND(LL)
14.05.24.PACK(LL)
14.05.24.  PACK COMPLETE.
14.05.24.COPYRF,BF,LL,MAINSI,1,CF.
14.05.25.EOI ENCOUNTERED     0 FILES
14.05.25.COPYRF,BF,MACTAB,MAINSI,1,CF.
14.05.26.EOI ENCOUNTERED     0 FILES
14.05.26.GOTO.222.
14.05.27.222,REWIND(DUMFIL)
14.05.27.GET,BIGMAC.
14.05.28.GET,EAINSI.
14.05.29.COPYRF,SBF,EAINSI,OUTPUT.
14.05.30.EOI ENCOUNTERED     0 FILES
14.05.30.REWIND,EAINSI.
14.05.30.CALL(BIGMAC,RENAME(I=EAINSI,M=MAINSI,B=B
14.05.30.B)
14.05.32.REWIND(MAINSI)
14.05.32.COPYRF,BF,MAINSI,OBJ.
14.05.33.COPY COMPLETE       1 FILES
14.05.33.GET,EXPOBJ/PJ=PAIX,ID=A246.
14.05.52.LOAD(EXPOBJ)
14.05.55.OBJ(LC=0,EAINSI,BB,MAINSI)
14.05.56.FL TO LOAD   75600   FL TO RUN  104600
14.06.02.STOP
14.06.05.JOBEND.    $    4.66
```

The second example is a simple implementation of the procedures POP and PUSH. Each macro requires a large number of parameters. The user must be intimately connected with the code produced by the macros in order to use them correctly.

```
      MACRO
      STRING FUNCTION POP(STACK,SINDEX,UNDFLW)
      STRING STACK,SINDEX,UNDFLW
C
C  THIS MACRO GENERATES CODE WHICH BRANCHES TO THE LABEL -UNDFLW- IF
C    THE STACK -STACK- UNDERFLOWS, BUT WHICH POPS THE STACK OTHER-
C    WISE.  -SINDEX- MUST BE THE STACK TOP POINTER.
C
      POP = GENER(I)
      EXECUTE $IF($ . SINDEX . $.LE.0)GOTO$ . UNDFLW
      EXECUTE POP . $=$ . STACK . $($ . SINDEX . $)$
      EXECUTE SINDEX . $=$ . SINDEX . $-1$
      RETURN
      END
C
C
      MACRO
      SUBROUTINE PUSH(STACK,SINDEX,ITEM,LENG,OVRFLW)
      STRING STACK,SINDEX,ITEM,LENG,OVRFLW
C
C  THIS MACRO GENERATES CODE WHICH BRANCHES TO THE LABEL -OVRFLW- IF
C    THE STACK -STACK- OVERFLOWS, BUT WHICH PUSHES THE INTEGER ITEM
C    -ITEM- ONTO THE STACK OTHERWISE.  -SINDEX- MUST BE THE STACK TOP
C    POINTER AND -LENG- MUST BE AN EXPRESSION FOR THE SIZE OF THE
C    STACK.
C
      EXECUTE SINDEX . $=$ . SINDEX . $+1$
      EXECUTE $IF($ . SINDEX . $.GT.$ . LENG . $)GOTO$ . OVRFLW
      EXECUTE STACK . $($ . SINDEX . $)=$ . ITEM
      RETURN
      END
$
```

```
      SUBROUTINE STKADD(LEN,A,B,C,IA,IB,IC)
      INTEGER LEN
C
      INTEGER A(LEN),IA
      INTEGER B(LEN),IB
      INTEGER C(LEN),IC
      INTEGER T(LEN),IT
C
      IC = 0
      IT = 0
C
   10 CALL PUSH(T,IT,POP(A,IA,20)+POP(B,IB,20),LEN,30)
      GO TO 10
C
   20 CALL PUSH(C,IC,POP(T,IT,30),LEN,30)
      GO TO 20
C
   30 RETURN
      END
$
```

```
      SUBROUTINESTKADD(LEN,A,B,C,IA,IB,IC)
      INTEGERLEN
C
      INTEGERA(LEN),IA
      INTEGERB(LEN),IB
      INTEGERC(LEN),IC
      INTEGERT(LEN),IT
C
      IC=0
      IT=0
C
   10 IF(IA.LE.0)GOTO20                                    + POP
      I0=A(IA)
      IA=IA-1
      IF(IB.LE.0)GOTO20                                    + POP
      I00=B(IB)
      IB=IB-1
      IT=IT+1                                              + PUSH
      IF(IT.GT.LEN)GOTO30
      T(IT)=I0+I00
      ------                                               + (SUBS)
      GOTO10
C
   20 IF(IT.LE.0)GOTO30                                    + POP
      I01=T(IT)
      IT=IT-1
      IC=IC+1                                              + PUSH
      IF(IC.GT.LEN)GOTO30
      C(IC)=I01
      ------                                               + (SUBS)
      GOTO20
C
   30 RETURN
      END
```

*** SUMMARY OF STKADD

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
POP	3	-	0	0	9
PUSH	2	-	0	0	6
TOTALS	5	-	0	0	15

EXPANSIONS - 2

*** SUMMARY OF EOF

NO ACTIVATIONS

*** FINAL SUMMARY

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
POP	3	-	0	0	9
PUSH	2	-	0	0	6
	+++++		++++++	++++++	++++++
TOTALS	5	-	0	0	15

EXPANSIONS - 2

The final example is an extension of the second and demonstrates the power of BIGMAC in implementing data abstractions. The data abstraction stack is supported with the macros -

1. SUBROUTINE DECLAR (<type>, <name>, <length>)

   <name> is a variable name.

   <type> is a key word for one of the five standard types.

   <length> is an integer constant or parameter.

   DECLAR declares <name> to be a stack of size <length> and type <type>.

2. SUBROUTINE START (<name>)

   START initializes the top of the stack <name> to the first element of the stack.

3. SUBROUTINE PUSH (<name>, <expression>, <label>)

   <expression> is an expression.

   <label> is a statement label.

   PUSH pushes the expression <expression> onto the top of the stack <name> unless there is an overflow in which case control passes to the statement labelled <label>.

4. FUNCTION POP (<name>, <label>)

   POP pops the top of the stack <name> unless the stack underflows in which case control passes to <label>. The type of the result of the POP is the type given to the stack in its DECLARation.

The required macros are much more complex than the macros of the second example, but much power has been gained. First, stacks of any type are supported. The implementation of POP in the second example constrains stacks to be of INTEGER type. Secondly, less parameters are needed for each invocation of POP and PUSH and the implementation of the abstraction need not be understood by the user. Finally, a fixed amount of storage is needed for temporaries in the code generated by POP. This is not true of the second example. The one drawback is that at most

eight POPs of a given type may occur in a single statement. (See the example of the routine MISTAK.) However, this limit will rarely be exceeded in practice.

Arbitrarily complex data abstractions may be implemented by BIGMAC. The limiting factor is the coding effort.

```
      MACRO
      SUBROUTINE SYSBEG
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
C
C  INITIALIZE THE VECTOR TTYP.  INITIALLY THE MAIN ROUTINE IS NOT
C    BEING SCANNED AND THE FIRST TEMPORARY OF EVERY TYPE IS UP FOR USE.
C
      DO 10 I=1,5
   10    TIND(I)=1
      INMAIN=.TRUE.
      TTYP(1)=$INTEGER$
      TTYP(2)=$REAL$
      TTYP(3)=$LOGICAL$
      TTYP(4)=$DOUBLEPRECISION$
      TTYP(5)=$COMPLEX$
      RETURN
      END
C
C
      MACRO
      SUBROUTINE ROUBEG
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
C
C  AT THE BEGINNING OF EVERY PROGRAM UNIT(P.U.) NO STACKS HAVE BEEN
C    DECLARED, NO DECLARATION ERRORS HAVE OCCURRED, AND NO STACK OF
C    ANY TYPE HAS BEEN DECLARED.
C
      CSTK=0
      DECERR=.FALSE.
      DO 10 I=1,5
   10    TINC(I)=.FALSE.
      RETURN
      END
C
C
      MACRO
      SUBROUTINE MANBEG
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
C
C  INMAIN IS SET WHEN THE MAIN ROUTINE IS ABOUT TO BE SCANNED.
C
      INMAIN=.FALSE.
      CALL ROUBEG
      RETURN
      END
C
C
```

```
      MACRO
      SUBROUTINE PRGEND
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
      STRING S
C
C  IF THE MAIN ROUTINE HAS JUST BEEN SCANNED THEN RESET INAMIN AND
C    DECLARE THE TEMPORARY COMMONS OF THOSE TYPES NOT YET INCLUDED
C    IN THE MAIN ROUTINE.
C
      IF (INMAIN) RETURN
      INMAIN=.TRUE.
      DO 10 I=1,5
         IF (TINC(I)) GO TO 10
            S = GENER(R)
            DECLARE $COMMON/STK$ . TTYP(I)(0..4) . $/$ . S . $(8)$
            DECLARE TTYP(I) . S
   10    CONTINUE
      RETURN
      END
C
C
      MACRO
      SUBROUTINE DECLAR(TYPE,STACK,LEN)
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
      STRING TYPE,STACK,LEN,T,CLST,DLST
C
C  IF A DECLARATION ERROR OR MORE THAN 10 STACKS HAVE BEEN DECLARED
C    IN THE CURRENT P.U. THE RETURN.
C
      IF (DECERR) RETURN
      IF (CSTK.LT.10) GO TO 10
         ERROR $TOO MANY STACKS.$
         DECERR = .TRUE.
         RETURN
C
C  CHECK THAT THE STACK NAME IS UNIQUE AND ASSIGN IT THE NEXT
C    AVAILABLE INTEGER CODE.
C
   10 IF (CSTK.EQ.0) GO TO 12
      DO 11 I=1,CSTK
         IF (.NOT.EQSTR(STACK,NAMS(I))) GO TO 11
            ERROR $DUPLICATE CREATION - $ . STACK . $.$
            RETURN
   11    CONTINUE
   12 CSTK=CSTK+1
C
C  IDENTIFY THE TYPE OF THE STACK.  IF NOT IDENTIFIABLE THEN ISSUE AN
C    ERROR AND ASSUME IT IS INTEGER.
C
      IF (LENSTR(TYPE).LT.4) GO TO 21
      DO 20 L=1,5
         IF (EQSTR(TYPE(0..5),TTYP(L)(0..5))) GO TO 30
```

```
   20    CONTINUE
   21 ERROR $ILLEGAL TYPE - $ . TYPE . $.$
      L=1
C
C  IF THE TEMPORARY COMMON FOR THE TYPE OF THE STACK HAS NOT YET BEEN
C    INCLUDED IN THE CURRENT P.U. THEN DO SO.
C
   30 IF (TINC(L)) GO TO 40
      TINC(L)=.TRUE.
      DLST=TTYP(L)
      CLST=$COMMON/STK$ . TTYP(L)(0..4) . $/$
      DO 31 I=1,8
         T=GENER(R)
         TVAR(L,I)=T
         DLST=DLST . T . $,$
   31    CLST=CLST . T . $,$
      DECLARE CLST(0..LENSTR(CLST))
      DECLARE DLST(0..LENSTR(DLST))
C
C  DECLARE THE STACK AND RETAIN ITS RELEVANT CHARACTERISTICS.
C
   40 DECLARE TTYP(L) . STACK . $($ . LEN . $)$
      NAMS(CSTK)=STACK
      LENS(CSTK)=LEN
      PTRS(CSTK)=GENER(I)
      TYPS(CSTK)=L
      DECLARE $EQUIVALENCE($ . PTRS(CSTK) . $,$ . STACK . $(1))$
      RETURN
      END
C
C
      MACRO
      SUBROUTINE START(STACK)
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
      STRING STACK
      INTEGER WHICH
C
C  DETERMINE THE INTEGER CODE FOR THE STACK WITH THE NAME -STACK- AND
C    IF NON-ZERO GENERATE CODE WHICH INITIALIZES THE STACK TOP TO THE
C    FIRST STACK ELEMENT.
C
      I = WHICH(STACK)
      IF (I.EQ.0) RETURN
      EXECUTE PTRS(I) . $=1$
      RETURN
      END
C
C
      MACRO
      SUBROUTINE PUSH(STACK,ITEM,OVRFLW)
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
      STRING STACK,ITEM,OVRFLW,S
```

```
      INTEGER WHICH
C
C  DETERMINE THE INTEGER CODE FOR -STACK- AND IF NON-ZERO GENERATE
C    CODE WHICH PUSHS -ITEM- ONTO THE TOP OF THE STACK BUT WHICH
C    BRANCHES TO THE STATEMENT LABELLED -OVRFLW- IF AN OVERFLOW
C    OCCURS.
C
      I = WHICH(STACK)
      IF (I.EQ.0) RETURN
      S=PTRS(I)
      EXECUTE S . $=$ . S . $+1$
      EXECUTE $IF($ . S . $.GT.$ . LENS(I) . $)GOTO$ . OVRFLW
      EXECUTE STACK . $($ . S . $)=$ . ITEM
      RETURN
      END
C
C
      MACRO
      STRING FUNCTION POP(STACK,UNDFLW)
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
      STRING STACK,UNDFLW,S
      INTEGER WHICH
C
C  DETERMINE THE INTEGER CODE FOR -STACK- AND IF NON-ZERO THEN GENERATE
C    CODE WHICH POPS THE TOP OF THE STACK INTO THE NEXT AVAILABLE
C    TEMPORARY OF THE APPROPRIATE TYPE, BUT WHICH BRANCHES TO THE
C    STATEMENT LABELLED -UNDFLW- IF AN UNDERFLOW OCCURS.  THE
C    TEMPORARY NAME IS SUBSTITUTED FOR THE FUNCTION REFERENCE.
C
      POP = $**E**$
      I = WHICH(STACK)
      IF (I.EQ.0) RETURN
      L=TYPS(I)
      J=TIND(L)
      POP=TVAR(L,J)
      TIND(L)=J+1
      IF (J.EQ.8) TIND(L)=1
      S=PTRS(I)
      EXECUTE $IF ($ . S . $.LE.1)GOTO$ . UNDFLW
      EXECUTE POP . $=$ . STACK . $($ . S . $)$
      EXECUTE S . $=$ . S . $-1$
      RETURN
      END
C
C
      INTEGER FUNCTION WHICH(STACK)
      COMMON /STKPOL/ DECERR,INMAIN,
     1                TVAR(5,8),TIND(5),TTYP(5),TINC(5),
     2                NAMS(10),LENS(10),PTRS(10),TYPS(10),CSTK
      LOGICAL DECERR,INMAIN,TINC
      STRING TVAR,TTYP,NAMS,LENS,PTRS
      INTEGER TIND,TYPS,CSTK
      STRING STACK
C
C  DETERMINE THE INTEGER CODE OF THE STACK NAMED -STACK-.  IF IT DOES
C    NOT EXIST, 0 IS RETURNED.  IF THERE IS ALSO NO DECLARATION ERROR
C    THEN AN ERROR MESSAGE IS ISSUED.
```

```
C
      IF (CSTK.EQ.0) GO TO 11
      DO 10 I=1,CSTK
         IF (EQSTR(STACK,NAMS(I))) GO TO 20
   10    CONTINUE
      IF (DECERR) GO TO 12
   11 ERROR $STACK - $ . STACK . $ - NOT CREATED.$
   12 WHICH = 0
      RETURN
   20 WHICH = I
      RETURN
      END
$
```

```
      CALL DECLAR(INTEGER,A,30)
      CALL DECLAR(REAL,B,30)
      CALL DECLAR(INTEGER,C,30)
C
      CALL START(A)
      CALL START(B)
C
      DO 10 I=1,30
         CALL PUSH(A,I,20)
   10    CALL PUSH(B,FLOAT(I**2),20)
C
   20 CALL STKADD(30,A,B,C)
C
   30 J=POP(C,40)
      WRITE (6,100) J
  100   FORMAT (1X,I7)
      GO TO 30
C
   40 STOP
      END
      SUBROUTINE STKADD(LEN,A,B,C)
      INTEGER LEN
C
      CALL DECLAR(INTEGER,A,LEN)
      CALL DECLAR(REAL,B,LEN)
      CALL DECLAR(INTEGER,C,LEN)
      CALL DECLAR(INTEGER,T,LEN)
C
      CALL START(C)
      CALL START(T)
C
   10 CALL PUSH(T,POP(A,20)+POP(B,20),30)
      GO TO 10
C
   20 CALL PUSH(C,POP(T,30),30)
      GO TO 20
C
   30 RETURN
      END
      SUBROUTINE MISTAK
      CALL DECLAR(INTEGER,A,30)
      CALL DECLAR(BOOLEAN,R,20)
      CALL DECLAR(INTEGER,A,40)
      CALL START(C)
      CALL DECLAR(COMPLEX,M,1)
      CALL DECLAR(DOUBLE,N,1)
      CALL DECLAR(LOGICAL,O,1)
      CALL DECLAR(INTEGER,P,1)
      CALL DECLAR(INTEGER,Q,1)
      CALL DECLAR(REAL,R,1)
      CALL DECLAR(REAL,S,1)
      CALL DECLAR(REAL,T,1)
      CALL DECLAR(DOUBLE,U,1)
      CALL DECLAR(DOUBLE,V,1)
C
      CALL PUSH(M,POP(C,10),10)
      CALL PUSH(M,POP(M,10)+POP(M,10)+POP(M,10)+POP(M,10)+POP(M,10)+
     1           POP(M,10)+POP(M,10)+POP(M,10)+POP(M,10),10)
   10 RETURN
      END
$
```

```
------
------
```

```
+ SYSBEG
+ (SUBS)
```

*** SUMMARY OF BOF

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
SYSBEG	1	-	0	0	0
TOTALS	1	-	0	0	0

EXPANSIONS - 1

```
      $$$$$$
       COMMON/STKINT/A0,A00,A01,A02,A03,A04,A05,A06
       INTEGERA0,A00,A01,A02,A03,A04,A05,A06
       INTEGERA(30)
       EQUIVALENCE(I0,A(1))
       COMMON/STKREA/A07,A08,A09,A0A,A0B,A0C,A0D,A0E
       REALA07,A08,A09,A0A,A0B,A0C,A0D,A0E
       REALB(30)
       EQUIVALENCE(I00,B(1))
       INTEGERC(30)
       EQUIVALENCE(I01,C(1))
       COMMON/STKLOG/A0F(8)
       LOGICALA0F
       COMMON/STKDOU/A0G(8)
       DOUBLEPRECISIONA0G
       COMMON/STKCOM/A0H(8)
       COMPLEXA0H
      $$$$$$
       ------                                                    + MANBEG
       ------                                                    + (SUBS)
       ------                                                    + DECLAR
       ------                                                    + (SUBS)
       ------                                                    + DECLAR
       ------                                                    + (SUBS)
       ------                                                    + DECLAR
       ------                                                    + (SUBS)
C
       I0=1                                                      + START
       ------                                                    + (SUBS)
       I00=1                                                     + START
       ------                                                    + (SUBS)
C
       DO1000I=1,30
       I0=I0+1                                                   + PUSH
       IF(I0.GT.30)GOTO20
       A(I0)=I
       ------                                                    + (SUBS)
       I00=I00+1                                                 + PUSH
       IF(I00.GT.30)GOTO20
       B(I00)=FLOAT(I**2)
       ------                                                    + (SUBS)
 1000 CONTINUE
C
   20 CALLSTKADD(30,A,B,C)
C
   30 IF (I01.LE.1)GOTO40                                        + POP
       A0=C(I01)
       I01=I01-1
       J=A0                                                      + (SUBS)
       WRITE(6,100)J
  100 FORMAT(1X,I7)
       GOTO30
C
   40 STOP
       ------                                                    + PRGEND
       END                                                       + (SUBS)
```

*** SUMMARY OF MAIN

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
DECLAR	3	-	0	0	0
MANBEG	1	-	0	0	0
POP	1	-	0	0	3
PRGEND	1	-	0	0	0
PUSH	2	-	0	0	6
START	2	-	0	0	2
	+++++		++++++	++++++	++++++
TOTALS	10	-	0	0	11

EXPANSIONS - 10

DECLARATIONS - 16

```
      SUBROUTINESTKADD(LEN,A,B,C)
     $$$$$$
      COMMON/STKINT/A0,A00,A01,A02,A03,A04,A05,A06
      INTEGERA0,A00,A01,A02,A03,A04,A05,A06
      INTEGERA(LEN)
      EQUIVALENCE(I0,A(1))
      COMMON/STKREA/A07,A08,A09,A0A,A0B,A0C,A0D,A0E
      REALA07,A08,A09,A0A,A0B,A0C,A0D,A0E
      REALB(LEN)
      EQUIVALENCE(I00,B(1))
      INTEGERC(LEN)
      EQUIVALENCE(I01,C(1))
      INTEGERT(LEN)
      EQUIVALENCE(I02,T(1))
     $$$$$$
      INTEGERLEN
C
      ------                                                    + ROUBEG
      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)
C
      I01=1                                                     + START
      ------                                                    + (SUBS)
      I02=1                                                     + START
      ------                                                    + (SUBS)
C
   10 IF (I0.LE.1)GOTO20                                        + POP
      A00=A(I0)
      I0=I0-1
      IF (I00.LE.1)GOTO20                                       + POP
      A07=B(I00)
      I00=I00-1
      I02=I02+1                                                 + PUSH
      IF(I02.GT.LEN)GOTO30
      T(I02)=A00+A07
      ------                                                    + (SUBS)
      GOTO10
C
   20 IF (I02.LE.1)GOTO30                                       + POP
      A01=T(I02)
      I02=I02-1
      I01=I01+1                                                 + PUSH
      IF(I01.GT.LEN)GOTO30
      C(I01)=A01
      ------                                                    + (SUBS)
      GOTO20
C
   30 RETURN
      ------                                                    + PRGEND
      END                                                       + (SUBS)
```

*** SUMMARY OF STKADD

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
DECLAR	4	-	0	0	0
POP	3	-	0	0	9
PRGEND	1	-	0	0	0
PUSH	2	-	0	0	6
ROUBEG	1	-	0	0	0
START	2	-	0	0	2
TOTALS	13	-	0	0	17

EXPANSIONS - 10

DECLARATIONS - 12

```
  SUBROUTINEMISTAK
 $$$$$$
  COMMON/STKINT/A0,A00,A01,A02,A03,A04,A05,A06
  INTEGERA0,A00,A01,A02,A03,A04,A05,A06
  INTEGERA(30)
  EQUIVALENCE(I0,A(1))
  INTEGERR(20)
  EQUIVALENCE(I00,R(1))
  COMMON/STKCOM/A07,A08,A09,A0A,A0B,A0C,A0D,A0E
  COMPLEXA07,A08,A09,A0A,A0B,A0C,A0D,A0E
  COMPLEXM(1)
  EQUIVALENCE(I01,M(1))
  COMMON/STKDOU/A0F,A0G,A0H,A0I,A0J,A0K,A0L,A0M
  DOUBLEPRECISIONA0F,A0G,A0H,A0I,A0J,A0K,A0L,A0M
  DOUBLEPRECISIONN(1)
  EQUIVALENCE(I02,N(1))
  COMMON/STKLOG/A0N,A0O,A0P,A0Q,A0R,A0S,A0T,A0U
  LOGICALA0N,A0O,A0P,A0Q,A0R,A0S,A0T,A0U
  LOGICALO(1)
  EQUIVALENCE(I03,O(1))
  INTEGERP(1)
  EQUIVALENCE(I04,P(1))
  INTEGERQ(1)
  EQUIVALENCE(I05,Q(1))
  COMMON/STKREA/A0V,A0W,A0X,A0Y,A0Z,A000,A010,A020
  REALA0V,A0W,A0X,A0Y,A0Z,A000,A010,A020
  REALS(1)
  EQUIVALENCE(I06,S(1))
  REALT(1)
  EQUIVALENCE(I07,T(1))
  DOUBLEPRECISIONU(1)
  EQUIVALENCE(I08,U(1))
 $$$$$$
  ------                                                    + ROUBEG
  ------                                                    + (SUBS)
  ------                                                    + DECLAR
  ------                                                    + (SUBS)

**ERROR(USER)**                                             + DECLAR
  ILLEGAL TYPE - BOOLEAN.

  ------                                                    + (SUBS)

**ERROR(USER)**                                             + DECLAR
  DUPLICATE CREATION - A.

  ------                                                    + (SUBS)

**ERROR(USER)**                                             + START
  STACK - C - NOT CREATED.

  ------                                                    + (SUBS)
  ------                                                    + DECLAR
  ------                                                    + (SUBS)
  ------                                                    + DECLAR
  ------                                                    + (SUBS)
  ------                                                    + DECLAR
  ------                                                    + (SUBS)
  ------                                                    + DECLAR
  ------                                                    + (SUBS)
  ------                                                    + DECLAR
```

```
      ------                                                    + (SUBS)

**ERROR(USER)**                                                 + DECLAR
  DUPLICATE CREATION - R.

      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)
      ------                                                    + DECLAR
      ------                                                    + (SUBS)

**ERROR(USER)**                                                 + DECLAR
  TOO MANY STACKS.

      ------                                                    + (SUBS)
C
      ------                                                    + POP
      I01=I01+1                                                 + PUSH
      IF(I01.GT.1)GOTO10
      M(I01)=**E**
      ------                                                    + (SUBS)
      IF (I01.LE.1)GOTO10                                       + POP
      A07=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A08=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A09=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A0A=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A0B=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A0C=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A0D=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A0E=M(I01)
      I01=I01-1
      IF (I01.LE.1)GOTO10                                       + POP
      A07=M(I01)
      I01=I01-1
      I01=I01+1                                                 + PUSH
      IF(I01.GT.1)GOTO10
      M(I01)=A07+A08+A09+A0A+A0B+A0C+A0D+A0E+A07
      ------                                                    + (SUBS)
10    RETURN
      ------                                                    + PRGEND
      END                                                       + (SUBS)
```

*** SUMMARY OF MISTAK

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
DECLAR	13	-	4	0	0
POP	10	-	0	0	27
PRGEND	1	-	0	0	0
PUSH	2	-	0	0	6
ROUBEG	1	-	0	0	0
START	1	-	1	0	0
	+++++		++++++	++++++	++++++
TOTALS	28	-	5	0	33

EXPANSIONS - 18

DECLARATIONS - 30

*** SUMMARY OF EOF

NO ACTIVATIONS

*** FINAL SUMMARY

MACRO	ACTIVATIONS		ERRORS	COMMENTS	EXECUTABLE STATEMENTS
DECLAR	20	-	4	0	0
MANBEG	1	-	0	0	0
POP	14	-	0	0	39
PRGEND	3	-	0	0	0
PUSH	6	-	0	0	18
ROUBEG	2	-	0	0	0
START	5	-	1	0	4
SYSBEG	1	-	0	0	0
TOTALS	52	-	5	0	61

EXPANSIONS - 39

DECLARATIONS - 58