BIGMAC II: A FORTRAN LANGUAGE AUGMENTATION TOOL*

Eugene W. Myers, Jr. and Leon J. Osterweil

Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

ABSTRACT

This paper describes the motivation, design, implementation, and some preliminary performance characteristics of BIGMAC II, a macro definition capability for creating language enhancers and translators. BIGMAC II enables the user to specify transformations through STREX, a FORTRAN-like language, which enables the specification of macros which are then used to interpretively alter incoming programs. BIGMAC II is specially adapted to the processing of FORTRAN programs. This paper shows how it can be used as a deprocedurizer (or flattener), a dialect-to-dialect translator, a portability and version control aid, and a device for creating language enhancements as sophisticated as new control structures and abstract data types.

BACKGROUND AND MOTIVATION FOR PRODUCING BIGMAC II

Over the past several years a significant number of software analysis tools have been produced by the University of Colorado Software Validation Group. These tools have at least initially been largely directed towards the needs of the Mathematical Software community. Hence they have been designed to be portable over a wide range of machines and to analyze programs written in FORTRAN. These two considerations have led us to code our tools in FORTRAN. As a result we have 1) had good success in readily rehosting our tools on a variety of machines and 2) been able to use our tools to analyze themselves, thereby increasing our confidence in them.

Through this considerable experience with FORTRAN, we have come to deeply understand some of the significant shortcomings of this language. As tool implementors we have chafed at the absence of such features as flexible control constructs and powerful data aggregation capabilities. Yet as creators of portable tools for producing mathematical software we felt a deep commitment to FORTRAN, the lingua franca of that community. In addition we believe that FORTRAN is, perhaps with the exception of COBOL, the most widely available, most nearly standardized of all high level languages, thereby giving it the best prospects as a basis for porting programs. We also found that newer languages, such as Pascal, which offered superior control and data aggregation facilities usually had drawbacks of their own. For example, Pascal's block structure is an obstacle to independent recompilation of subprocedures. This is a serious problem during the development of large tools such as our DAVE system [1,2].

Pulled by these conflicting considerations we have chosen a course taken by many others before us — namely to enhance the FORTRAN language in the ways necessary to facilitate our work. A large number of FORTRAN preprocessors have been built to enhance FORTRAN by the creation of more powerful control constructs. We were concerned more with the need for flexibly defining and accessing powerful data aggregates. Because the preprocessors we studied did not offer significant capabilities of this sort, we constructed a system of our own [3], This capability enabled us to define a rather limited class of structured data types and access them from our standard FORTRAN source text. The implementation details of these data objects were hidden by our data structuring capability, thus enabling us to alter our data structures without the need to alter the source text which accessed them. This capability proved most useful, enabling us to construct our prototype DAVE tool with a minimum amount of trauma.

A serious weakness of this capability is that it conceals the implementation details of the data structures through the use of layers of subroutines. Although quite effective in concealing details, this technique proves quite costly in execution time, necessitating the invocation of numerous subroutines for each data access.

Because of this difficulty we turned to the consideration of a macro capability. Our first goal was to build a capability which could deprocedurize or "flatten" our code by substituting inline the texts of subroutines in place of the subroutine invocations. We conceived of the subroutine text as being a sort of macro definition and the subroutine call as being a parameterized macro invocation. We produced a prototype capability of this sort and successfully used it to flatten the source code for our DAVE system [4].

According to our measurements we gained an improvement in execution time of up to 50% by flattening only a small number of frequently executed

*This work supported in part by U.S. Army Research Office grant #DAAG29-80-C-0094 and National Science Foundation grant #MCS77-02194.

CH1627-9/81/0000/0410$00.75 © 1981 IEEE

subprogram invocations. We then turned our attention to building a more general macro processor to enable substantial, flexible enhancements to FORTRAN. The key to doing this was allowing the user to define new source language statements by declarations in the macro language. Thus, for example, superior control flow constructs such as CASE, IF_THEN_ELSE, and DO_WHILE can readily be added to FORTRAN by declaring them to be new statements whose semantics are established by means of the text of the macros which define them.

The macro processor we created is called BIGMAC II for historical and humorous reasons. Perhaps the most important capability which BIGMAC offers is the capability for creating and accessing powerful data aggregates while hiding their implementation details, thereby adding to FORTRAN true abstract data type capabilities. This is achieved by considering data structure declaration statements to be Fortran language augmentations and data structure accessing constructs to be new language operators. The definitions of the declaration and accessing constructs are made through macro definitions. As shall be shown these definitions can (indeed must) share implementation details among themselves. These details remain invisible to the source code. Thus there is complete freedom to create and alter the implementations of data aggregates beyond the sight and control of the FORTRAN source language coder. This inability to see or access implementation details qualifies this data aggregate capability as an abstract data type definition facility in the classical sense [5].

BIGMAC also facilitates the porting of FORTRAN programs, an important consideration for tool developers. Although, as already noted, FORTRAN is a widespread, standardized language, it is not entirely free of portability problems. Numerous dialects of the language exist, offering a variety of capabilities for handling such constructs as text strings, and multiple precision numeric quantities. These differences are particularly vexing to the writer of analytic tools for mathematical software, as he must be concerned with both the manipulation of source text and the correct analysis of code dealing with all possible numeric data types. A possible, but unsatisfactory, solution is to create a family of versions of the tool program, each intended for a different host compiler. The difficulty here is that each must be maintained and, unless extraordinary care is taken, each assumes a character of its own and incompatibilities between different versions arise. The usual strategy is to write the preponderant majority of the program in a portable subset of FORTRAN, such as PFORT[6] and quarantine compiler dependencies to a very small body of code which must then be recoded for every new host compiler. This approach has been widely used with success but still has its drawbacks. The compiler dependent code is usually written as a set of low level subroutines. Hence it usually is frequently invoked, always at the undesirable cost of subroutine invocations. In addition it still necessitates the creation and maintenance of a family of programs with the attendant multiplication of effort and potential for drift.

Through the use of BIGMAC, a single master copy of program text can be maintained and targeted for the different host machines by translating it with different sets of macros designed to adapt the master copy to the idiosyncrasies of the different host compilers. This approach has been pursued in a limited way [7] with good success. Our intention is to use BIGMAC to combine this notion with the capabilities for creating advanced data and control constructs, thereby enabling us to code in a very powerful, highlevel FORTRAN-like language, yet still maintain a set of completely consistent FORTRAN-source language versions merely by performing macro expansions (see Figure 1).

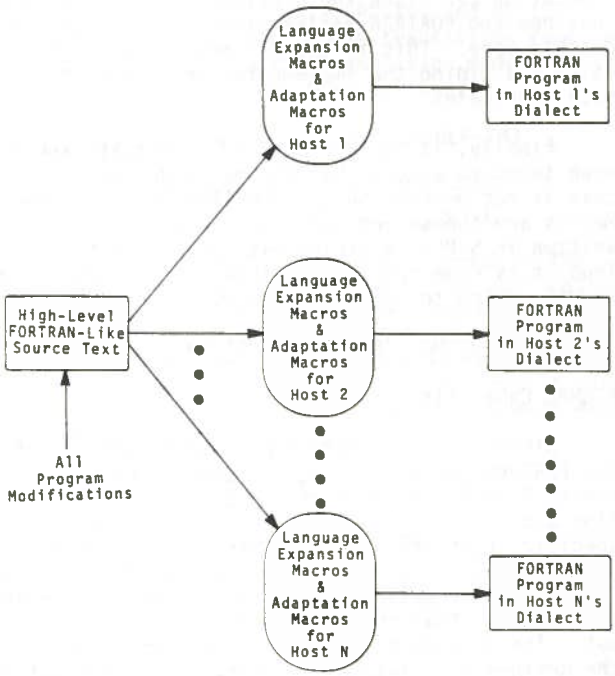

FIGURE 1: BIGMAC AS A PORTING MECHANISM

The remainder of this paper discusses the particular macro capabilities which we have implemented in BIGMAC. The salient features of these capabilities are flexibility, power, and the ease with which BIGMAC can be used by FORTRAN programmers. Much of the flexibility of BIGMAC derives solely from the fact that it is a macro capability. The user is free to declare any macros and define them in any way he chooses. Hence new statement types can be added to the language and arbitrary subroutines can be expanded as in-line code in an arbitrary way. Additional flexibility derives from the fact that existing statements from standard FORTRAN can be redefined according to the dictates of user specified macros. Thus, for example, the syntax and semantics of existing FORTRAN statements can be expanded to encompass new operators. Examples of this will be shown in the following section.

The power of BIGMAC derives in large measure from the fact that it provides for more than a

straightforward in situ text insertion capability. Macro definitions are parameterized, and source text lines are considered to consist of the macro identifier and an argument string. Macro definition bodies can be constructed to analyze the argument strings and to use them as the basis for the construction of appropriate object text. Thus, the object text produced may vary considerably from one invocation to the next. In addition a macro definition body is capable of creating object text and specifying its insertion in any of three places, corresponding to the site of the macro invocation, a location within the declaration block of the invoking program unit, and a location completely outside of the invoking program unit. Hence a macro invocation can cause the creation of entire utility routines and FORTRAN declarations, as well as executable code. This is the primary vehicle for creating and hiding the implementation of powerful data aggregates.

Finally, it is important to note that care has been taken to assure that the macro definition process is not beyond the grasp of FORTRAN programmers. Macros are themselves defined by means of code written in STREX, a string extended FORTRAN dialect. Thus it is expected that FORTRAN programmers could readily learn to produce their own macros.

BIGMAC TECHNICAL DESCRIPTION

BIGMAC Capabilities

BIGMAC is a general purpose translation system for FORTRAN software. The key theoretical concepts behind BIGMAC are those of syntax directed translation and tree transducers [8,9]. To configure a specific translator, one submits a context free grammar [10] and a set of tree macros to the BIGMAC system. The rewrite rules of the grammar determine the language that the translator will accept as input. The tree macros are programs that accomplish the desired translation by modifying the derivation tree [10] of the input. Each rewrite rule in the grammar is coupled with a tree macro. The grammar and tree macros that configure a translator are termed a translation specification.

Once a translator has been configured, it will transform a segment of text as follows. If the submitted text is not in the language generated by the translator's grammar then errors are issued where appropriate and the text is output unmodified. Otherwise a derivation tree for the text is built where each vertex in the tree corresponds to a rewrite rule and hence a tree macro. These tree macros are invoked when their associated vertices are reached in a traversal of the derivation tree. This activity results in a modification of the derivation tree. The string derived by this modified tree is then output as the desired translation.

It is important to note that the idea of producing a source code transformation system is not new with this effort. Boyle has produced a system, TAMPR [7], based on the notion of supporting user specified, correctness-preserving tree transformations. Our own work differs from TAMPR most strikingly in that it is specially adapted to the needs of the FORTRAN community. Thus the user specifies translations in STREX, a language comfortably close to FORTRAN. In addition, many primitives are supplied to facilitate the transformation of FORTRAN programs. We conjecture that these primitives also give to the BIGMAC system more power than is available through TAMPR.

BIGMAC has a built-in FORTRAN grammar. Thus BIGMAC translators will always accept FORTRAN software as input. This built-in grammar can be augmented at the statement and operator levels by the placement of additional rewrite rules in a translation specification. Tree macros must accompany these additional rewrite rules but their presence is optional for the built-in FORTRAN rewrite rules. Whenever a tree macro is not present in a translation specification it is assumed to be the null program. Thus the translator configured from the empty translation specification will accept FORTRAN code as input and return the same code as output. This augmentation approach to translation specifications is in keeping with BIGMAC's role as a FORTRAN software tool. BIGMAC translators will accept a specified superset of FORTRAN as input and perform a translation based on a selected subset of the rewrite rules.

The target or output code of a BIGMAC translation is expected to be FORTRAN, although any other language (e.g., assembler, Pascal) could be output. To facilitate the production of FORTRAN as output, there are a number of special primitives available to the tree macro writer for this purpose. The efficiency of translated code is dependent on the coding effort put into the tree macros. Good but not optimal results can usually be obtained with little effort.

As already noted, the BIGMAC system can be used to configure translators for a variety of tasks. In the paragraphs below, we describe in a general way how BIGMAC can be used to create some of these different types of translators.

At the highest level BIGMAC can be used as a compiler-compiler [10]. In order to do so, one must compose a translation specification encompassing the desired language and transformation. In this case, the built-in FORTRAN grammar is ignored. As an example of this, BIGMAC is currently being used to build a compiler for a PASCAL-like language which produces FORTRAN as target code.

BIGMAC can be used to extend the FORTRAN language in several ways. For example, new data abstractions [11,5] such as a STRING or BIT VECTOR data type can be added to the language. This requires the introduction of at least one new declarative statement and a number of operators and executable statements for manipulating the data type. The rewrite rules for these new forms and tree macros which translate them into standard FORTRAN are all that is needed to accomplish the extension. New control structures such as IF_THEN_ELSE and DO_WHILE can be added by the introduction of rewrite rules and tree macros for these new statements. The tree programs necessary for these forms

are particularly simple and thus a preprocessor having the power of IFTRAN [12], for example, can readily be configured.

The need for a program flattener for a program coded in FORTRAN has already been discussed. A BIGMAC translator for performing this flattening can be configured simply by specifying the appropriate tree programs for those FORTRAN rewrite rules which correspond to the subroutine calls and function references which are to be flattened.

The utility of a macro processor as a portability aid has also been discussed. The use of BIGMAC to accomplish this will be discussed in detail shortly.

The Use of BIGMAC

The design of BIGMAC envisions two classes of users. A small group of BIGMAC experts, called macro writers, are responsible for configuring translators. Only macro writers need to know how to compose translation specifications. The second user class is called the using community and consists of those groups that use any of the translators provided by the macro writers. The members of the software using community need only know how to program code which utilizes any of a translator's enhancements.

The structure of BIGMAC is depicted in Figure 2. The system consists of a translation compiler

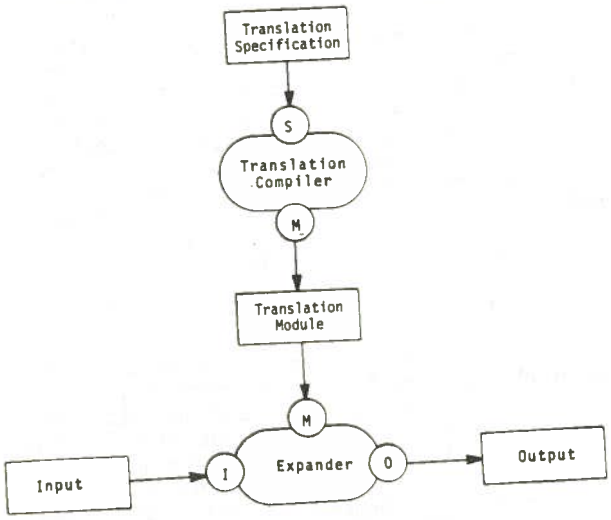

FIGURE 2: BIGMAC STRUCTURE

and an expander. A translation specification is submitted to the translation compiler which results in the creation of a translation module. This module consists of parsing tables and object modules for the tree macros. The expander when coupled with a translation module, forms a translator which takes a user's input and transforms it accordingly.

The translation specification and the input have been separated in the BIGMAC system to accomodate the distinction between macro writers and the using community. Furthermore, a translation specification is compiled because it is expected to be used repeatedly. To interpret complex specifications for each translation would be extremely inefficient.

Consider as an example the approach to porting illustrated in Figure 1. This approach can be implemented by using BIGMAC as follows. A group of macro writers must first compose a translation specification to support a high-level FORTRAN dialect as requested by the using community. They also write specifications that configure porting translators for host machines 1, 2, and 3. Call these specifications HLF-TS, P1_TS, P2-TS, and PS-TS respectively. Using the BIGMAC translation compiler, the macro writers form the translation modules HLF-TM, P1-TM, P2-TM, and P3-TM as follows.*

```
Translation_Compiler(S=HLF-TS,M=HLF-TM)

Translation_Compiler(S=Pi-TS,M=Pi-TM)
                        for i = 1, 2, 3.
```

Now suppose that a development group in the using community has written code for an application system in the Fortran dialect they specified previously. Call the code SC-HLF. To run this code on the local machine requires invoking the appropriate BIGMAC translator and compiling the result with the local FORTRAN compiler. This procedure is expressed in functional notation as:

```
Expander(I=SC-HLF,M=HLF-TM,O=SC-FOR)

Fortran_Compiler(I=SC-FOR,O=SC-BIN)
```

Once the development group's software is ready for porting to machine i the appropriate porting translator is used.

```
Expander(I=SC-FOR,M=Pi-TM,O=SC-FORi)
```

The output, SC-FORi, is then sent to host machine i, where it is subsequently compiled by that machine's FORTRAN compiler. Thus by writing the four translation specifications above, the macro writers have created an environment for the software using community in which a high-level FORTRAN can be employed and in which porting to three additional machines is automatic.

Implementation of BIGMAC

In Figure 3, the structure and interaction of a translation module and the BIGMAC expander are shown in detail. The expander performs a translation in three passes. In the first pass, the front-

*The functional notation, 'Translation_Compiler (S=<file$_1$>,M=<file$_2$>)' denotes an application of the translation compiler, where the translation specification is <file$_1$> and the resulting translation module is <file$_2$>. Similarly, in the statement 'Expander(I=<file$_1$>,M=<file$_2$>,O=<file$_3$>)',<file$_1$> is the input, <file$_2$> the translation module, and <file$_3$> the resulting output.

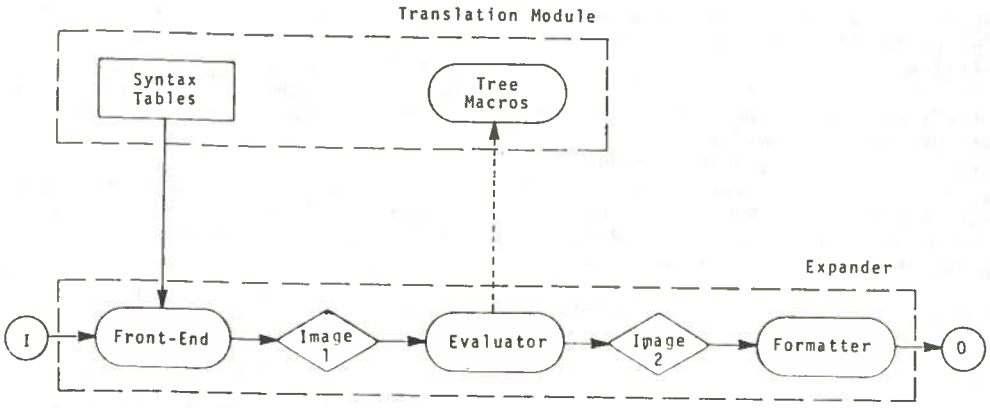

FIGURE 3: BIGMAC TRANSLATOR IN DETAIL

end builds a derivation tree of the input by consulting the syntax tables of the translation module. This derivation tree is output as the first image. In the second pass, the evaluator traverses the derivation tree calling the tree macros of the translation module when appropriate. This activity results in a modification of the derivation tree which constitutes the second image. In the last pass, the formatter produces as the final output the string yielded by the modified derivation tree.

Up to this point a rather abstract model of BIGMAC's operation has been given. BIGMAC's behavior has been described in terms of context-free grammars,derivation trees, and tree programs. A system which tries to embrace the full generality of these concepts, would undoubtedly be highly inefficient. One of the main design aims of BIGMAC was to simplify these concepts in such a way that they still provide a powerful translation capability for which an efficient implementation is, nevertheless, possible. These simplifications are described in the paragraphs below.

Front End & Syntax Tables

A grave space problem arises if one attempts to build a derivation tree for the entire input stream in a straightforward manner. The BIGMAC system simplifies matters by processing the input text and inter-pass images in sequential forward scans, one statement at a time. A statement is defined in the FORTRAN sense of the word, that is, as a 72-character line, optionally followed by up to nineteen 66-character continuation lines. There are assumed to be four types of statements — header, specification, executable, and tail — which must be grouped into program units according to the following syntax.

<Input> ← <Program_Unit>*

<Program_Unit> ← <Header> <Specifications>*
<Executable>* <Tail>

There is also a neutral type statement whose occurrence in the input is unrestricted. This organization supports the FORTRAN notion of program units or modules, but other organizations are possible by declaring all statements to be neutral and placing the burden of sequence checking on the tree programs.

The range of syntactic augmentation allowed in a translation specification is restricted to the extent that a simple LL-parsing [10] scheme suffices in syntactically decomposing the input. In this scheme, look-ahead is very rarely required and usually involves the scanning of just one or two characters. The front-end builds a derivation tree for each statement and outputs these in sequence. The front-end also detects any syntactic errors.

For each program unit, the front-end builds a symbol table of the labels and identifiers that occur in the unit. The front-end also builds a global symbol table of all those identifiers whose context determines that their scope is global (e.g., subroutine names, common block name, etc.). These tables are used to generate unique identifiers and labels by the tree macros in the evaluation pass. They are also used for associating attributes or descriptors with each identifier.

Evaluator

The BIGMAC evaluator processes the derivation tree of each statement in a postorder traversal. As each vertex is traversed, its associated tree macro, if present, is invoked. Conceptually, tree macros can be thought to transform the derivation tree directly. However, in the BIGMAC system the mechanisms for text generation (modification) are quite simple. A tree macro can append statements, semantic error messages, and comments to the text stream at several strategic locations. In this way, a tree macro can generate side effects, document them, and report semantic errors. A tree macro can also modify the derivation tree containing the invoking vertex, v, as follows. The tree macro can replace the string currently yielded by v in the derivation tree by any string it chooses to produce. The tree macro receives as arguments, the strings currently yielded by each son of v in the derivation tree. Figure 4 gives an example of this text substitution process. In Figure 4, the operator subscripts are shown to indicate the order of evalua-

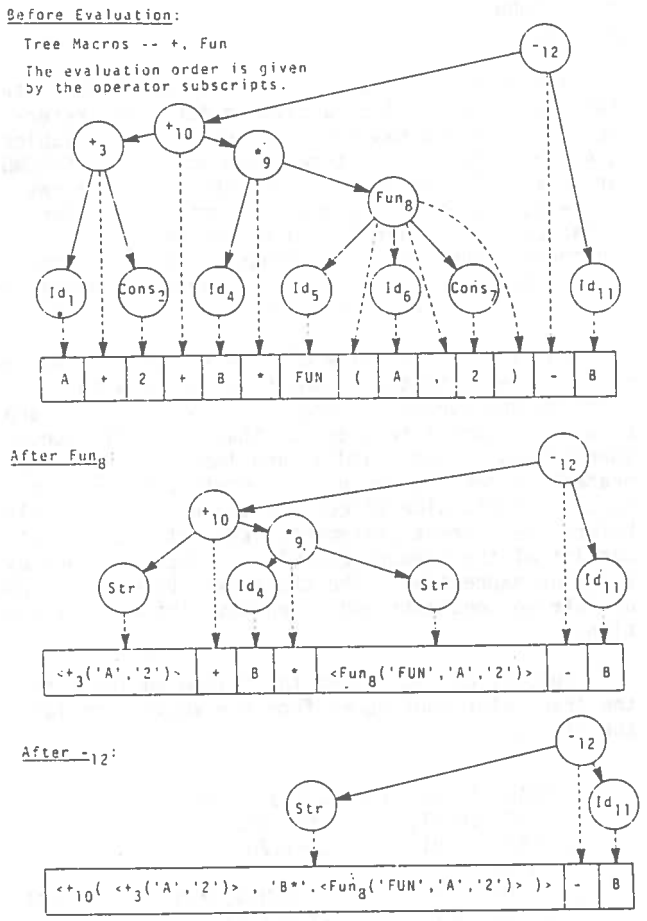

FIGURE 4: TEXT SUBSTITUTION PROCESS

tion followed. Also note only the operators Fun and + have defining tree macros. The other operators are evaluated according to the usual rules of FORTRAN.

Since the simple text substitution scheme employed in the BIGMAC system deals only with the strings yielded by derivation tree vertices, a derivation tree need never be built. A simple text substitution algorithm used in many of the older macro processors [13] suffices to perform the desired transformation and only requires the post-order sequence of invoking vertices. This algorithm provides increased space and time efficiency especially when the translation only operates on a small subset of the grammar rewrite rules.

Tree Macros

In BIGMAC translation specifications, a tree macro is a program unit written in an extension of FORTRAN called STREX (String Extended) FORTRAN. A compiler for this extension of FORTRAN was produced by a bootstrap of the BIGMAC utility. The STREX language supports a subset of FORTRAN and has the following additional features —

1. A STRING data type.
2. A HEAP data type.
3. Text generation primitives.
4. Symbol table primitives.
5. Global and local macro communication primitives.

STREX FORTRAN supports all FORTRAN constructs except for those involving I/O or Hollerith, Real, Double Precision, or Complex data types [14]. The additional features are described below.

STREX supports the notion of a string data type in full generality [15]. Simple variables, arrays, and functions can all be declared to be of type string. The value of a string variable may vary arbitrarily in length at runtime. At the expression level strings can be concatenated, substrings can be selected, the current length can be queried, and strings can be tested for equality. String constants are denoted by a sequence of characters enclosed in dollar-signs($). At the statement level string variables can be assigned (by value) and the replacement of a substring of a string variable is also possible. Strings are used in features 3, 4, and 5.

The STREX heap data type is a vector of storage of arbitrary length. The elements of the vector may be any data type including heap. In this way, recursive structures such as lists and trees can be readily implemented. The type of a vector element is latent; that is, it may vary at runtime. Whereas assignment is by value for all other data types, assignment to a heap variable changes the vector the variable refers to. Thus an assignment between two heap variables, say H1 = H2, causes H1 to refer to the vector referred to by H2, and H2 becomes undefined. There are primitives to determine the length of a variable's vector and whether a variable is undefined. There are also statements for creating and freeing heap vectors. Heap variables are used in features 4 and 5.

STREX language contains several statements which append strings to the output text stream at any of three locations. The string expressions used in these primitives are assumed to represent either statements, comments, labels, or error messages depending on the primitive. The three locations at which a string may be appended are:

a/ immediately before the statement currently being scanned,

b/ immediately before the first executable statement of the program unit currently being scanned,

c/ immediately before the first statement of the current program unit.

These locations allow one to generate side-effects, additional specification statements, and additional program units, respectively. There are expression level primitives for generating identifiers and labels which are unique with respect to either the global symbol table or the local symbol table of the current program unit. These primitives return the labels and identifiers as strings and are used in constructing branches and tempory variables.

The symbol tables produced by BIGMAC's front end are available to the macro writer. A heap variable, presumably containing a descriptor, is associated with each symbol. One can look up a symbol, thereby accessing the symbol's heap variable. This variable may be accessed or modified according to the intentions of the macro writer. New symbols may be added dynamically.

Tree macros can communicate to each other in one of two ways. Common blocks can be included in any tree macro. The extents of the variables in such a common block are global to the evaluation process. In this way, global communication is possible — any tree macro may modify or access these global variables. In the event that a son of an invoking vertex is also an invoking vertex, the tree macro associated with the son may pass a heap vector to the tree macro of its father. This local communication mechanism is accomplished with a pair of "pass-catch" primitives. This mechanism allows for the bottom-up propagation of synthesized attributes (heap vectors) [16].

Formatter

The BIGMAC formatter preserves the spacing and indentation found in the input stream. This formatter has two output ports. The primary output contains the transformed text less any comments or error messages. The other port is to the printer. This output contains the comments and error messages and in addition contains summaries of the tree macros' activities.

SOME SIMPLE BIGMAC EXAMPLES

Two BIGMAC examples are given in the following paragraphs. For each example the BIGMAC translation specification is listed and discussed first. The discussion is then followed by a sample translation.

The first example generates in-line code for the following function which computes the sum of the elements of an array.

```
      REAL FUNCTION SUM(A,N)
      REAL A(N)
      SUM = 0.
      DO 10 I = 1,N
   10    SUM = SUM + A(I)
      RETURN
      END
```

The translation specification which will perform this in-lining consists of one tree macro —

```
1.  REFERENCE B = SUM(A,N)
2.  STRING IL, I, L
3.  B = .GENR.
4.  IL = .GENI.
5.  I = .GENI.
6.  L = .GENL.
7.  EXECUTE   B*$=0.$
8.  EXECUTE   IL*$=$*N
9.  EXECUTE   $DO $*L*$ $*I*$=1,$*IL
10. LABEL   L
11. EXECUTE   $  $*B*$ = $*B*$+$*A*$($*I*$)$
12. RETURN
13. END
```

The header of the tree macro (line 1) indicates that the macro is to be activated for every reference to SUM which has two arguments. The variables B, A, and N in this statement are implicitly STRING variables. The value of B when the macro returns is the string which replaces the reference. The variables A and N are passed to the macro on entry and their values are the strings representing the two arguments to SUM. Line 2 declares the variables IL, I, and L to be of type STRING.

In line 3, the value of B is assigned to an implicitly real identifier which is unique with respect to the current program unit. Lines 4, 5, and 6 have the same effect except that implicit integer identifiers (lines 4 and 5) and labels (6) are generated. Lines 7 through 11 generate code for the function SUM's side-effect and place it immediately before the current statement. Each of these lines consist of the keyword EXECUTE or LABEL followed by a string expression. The characters between $-signs are string constants and * denotes string concatenation.

Suppose one submitted the following input to the translator configured from the above translation specification.

```
      REAL  A(15), B(20), T(5), SUM
      READ (INPUT) (A(I),I=1,15)
      READ (INPUT) (B(I),I=1,20)
      DO 10 I = 1,5
   10    IF(I.NE.5) T(I) = SUM(A,3*I) + SUM(B,20)
      WRITE (OUTPUT) (T(I),I=1,4)
      STOP
      END
```

The output would be as follows

```
      REAL A(15), B(20), T(5), SUM
      READ (INPUT) (A(I), I=1,15)
      READ (OUTPUT) (B(I),I=1,15)
      DO 10 I = 1,5
         IF(.NOT.(I.NE.5)) GOTO 10
         A00000 = 0.
         I00000 = 3*I
         DO 10000 I00001 = I,I00000
10000       A00000 = A00000 + A(I00001)
         A00001 = 0.
         I00002 = 20
         DO 10001 I00003 = 1,I00002
10001       A00001 = A00001 + B(I00003)
         T(I) = A00000 + A00001
   10    CONTINUE
      WRITE (OUTPUT) (T(I),I=1,4)
      STOP
      END
```

Note that the meaning of the DO label 10 was preserved. The BIGMAC front-end automatically preconditions the input code so that the meaning of labels is preserved. It also splits the logical if statement so that any text placed "immediately" before the second clause of the IF statement is guaranteed to be executed just before the execution of this

clause. The code produced is not optimal. A slightly more sophisticated tree macro would not generate I00001 or I00003 or the assignment to I00001.

The second example illustrates the addition of an IF-THEN-ELSE control structure to the FORTRAN language. This requires the addition of three statements with the syntax —

```
<statement> ← 'IF' <expression> 'THEN'
            ← 'ELSE'
            ← 'ENDIF'
```

The proper nesting of these statements is enforced. An ELSE-statement refers to the most recent IF-statement. The translation specification consists of four tree macros and a global block.

```
 1.        GLOBAL PSHDWN
 2.           HEAP LABSTK
 3.        END
 4.        STATEMENT/E/ $IF$-.EXP./A-$THEN$
 5.        INCLUDE PSHDWN
 6.        HEAP    H
 7.        STRING  LAB
 8.        LAB = .GENL.
 9.        EXECUTE $IF (.NOT.($*A*$)) GOTO $*LAB
10.        NEW H(2)
11.        H(1,.HEP.) = LABSTK
12.        H(2,.STR.) = LAB
13.        LABSTK = H
14.        RETURN
15.        END
16.        STATEMENT/E/ $ELSE$
17.        INCLUDE PSHDWN
18.        STRING LAB
19.        IF (.UNDEF.LABSTK) GOTO 10
20.           LAB = .GENL.
21.           EXECUTE   $GOTO$ * LAB
22.           LABEL     LABSTK(2,.STR.)
23.           LABSTK(2,.STR.) = LAB
24.           RETURN
25.     10    ERROR     $IMPROPER NESTING$
26.           RETURN
27.        END
28.        STATEMENT/E/ $END$-$IF$
29.        INCLUDE PSHDWN
30.        IF (.UNDEF.LABSTK) GOTO 10
31.           LABEL LABSTK(2,.STR.)
32.           LABSTK = LABSTK(1,.HEP.)
33.           RETURN
34.     10    ERROR $IMPROPER NESTING$
35.           RETURN
36.        END
37.        END OF UNIT
38.        INCLUDE PSHDWN
39.        IF(.UNDEF.LABSTK) RETURN
40.        ERROR $IMPROPER NESTING$
41.        FREE LABSTK
42.        RETURN
43.        END
```

In lines 1 through 3 a global block called PSHDWN is declared to consist of the single HEAP variable LABSTK. LABSTK will be used to implement a pushdown stack of the labels being used in the transformation. The INCLUDE statements (lines 5, 17, 29, and 38) in each of the tree macros indicate that the variables in the PSHDWN global block are to be used by these macros.

The first tree macro (lines 4 to 15) corresponds to the IF-THEN statement. In lines 8 and 9 the appropriate conditional branch is generated. The label used in this branch is pushed onto the LABSTK push down stack in lines 10 through 13.

The ELSE tree macro (lines 16 to 27) checks the nesting of statements (line 19) and produces an error message if appropriate (line 25). If the nesting is correct then a branch is generated to tie off the IF-THEN clause and this clause's target label is appended to output (lines 20-22). The ELSE label replaces the IF-THEN label in the top element of the push-down stack (line 23).

The END-IF tree macro (lines 28-36) also checks the nesting of statements (line 30). If there are no errors then the current target label is appended to the output and the push-down stack is popped (lines 31 and 32).

The last tree macro (lines 37 to 43) is a special macro which is activated whenever the end of a program unit is reached. This macro checks the nesting of statements and frees the push-down stack.

Suppose one submitted the following input to the translator configured from this translation specification.

```
      .
      .
      .
      IF I.NE.Ø THEN
         IF J.NE.Ø THEN
               A = A + I + J
         ELSE
               A = A + I
         ENDIF
      ELSE
         IF J.NE.Ø THEN
               A = A + J
         ENDIF
      ENDIF
```

The output would be as follows —

```
      .
      .
      .
      IF (.NOT.(I.NE.Ø)) GOTO 10000
         IF(.NOT.(J.NE.Ø)) GOTO 10001
            A = A + I + J
         GOTO 10002
10001       A = A + I
10002 GOTO 10003
10000    IF (.NOT.(J.NE.Ø)) GOTO 10004
            A = A + J
10004    CONTINUE
10003 .
      .
      .
```

Once again the semantics of the transformation is correct but not optimal. The use of .NOT. could be folded in the three IF statements. The statement "GOTO 10003" is spurious. A more sophisticated translation specification could avoid these failings.

This concludes the technical description of BIGMAC. The reader interested in an example illustrating the addition of a new data type is referred to Appendix A which describes an ambitious translator for a bit vector data type. [17].

EXPERIENCES AND FUTURE WORK

BIGMAC is written in a portable subset of FORTRAN 66, in which dependencies have been quarantined to a small set of subprograms. BIGMAC consists of 10,000 lines of source text. The system requires 45K words of memory plus whatever space the macros of a particular translation specification require. The following timing figures were obtained on a CDC 6400.

Front-end	:	70	source lines/sec.
Evaluator	:	90	"
Formatter	:	90	"
Total Throughput	:	27	"

These times were obtained with the empty translation specification (identity translator) and thus provide an upper bound on the performance of BIGMAC.

As indicated earlier, we produced the STREX compiler module of BIGMAC by a bootstrapping operation utilizing a prototype version of BIGMAC. The translation specification needed for this bootstrap involved a macro for every operator and statement in the language and required 1000 lines of STREX code. The timing figures for the STREX compiler on the CDC 6400 are as follows —

Front-end	:	60	source lines/sec.
Evaluator	:	50	"
Formatter	:	90	object lines/sec.
Total Throughput	:	17	source lines/sec.
Expansion Factor	:	2.2	object lines/source line
Macro Rate	:	75	macros/sec.

The STREX compiler is an example of a heavily enhanced FORTRAN dialect. Thus the above figures provide an approximate lower bound on BIGMAC's performance, although more complex enhancements are conceivable.

One of the nice features of BIGMAC is illustrated by the STREX compiler. The utilization of a complete set of macros allowed us to write macros that perform a complete semantic check of STREX code. Coupled with BIGMAC's inherent syntactic checking, this means that the FORTRAN object code produced by STREX is guaranteed to be compileable FORTRAN. Thus error reporting is confined to the source level, a feature not found in many preprocessing systems.

We are currently in the late stages of using BIGMAC to define a translator capable of accepting as input essentially a large subset of PASCAL and producing as output essentially FORTRAN 66. Simultaneously, we are using BIGMAC to write macro sets which will allow us to port this FORTRAN code to a modest range of host compilers. The completion of these translators will mark the beginning of our efforts to use BIGMAC as a production tool. We expect to use these translators to simultaneously target a large PASCAL program to a range of host FORTRAN compilers. This attempt will do much to help us evaluate the practicality of this macro approach as a portability and version control aid.

ACKNOWLEDGMENTS

We wish to thank our colleagues Lloyd Fosdick, Dan Ruegg, Becky Jones, and Randy Levine for their substantial contributions to the design and implementation of the system described here. In addition we gratefully acknowledge the support of the U. S. Army Research Office through grant DAAG 29-80-C-0094 and National Science Foundation grant MCS 77-02194.

REFERENCES

[1] Osterweil, L.J. and Fosdick, L.D. "DAVE — A Validation Error and Detection and Documentation System for Fortran Programs," Software Software-Practice and Experience 6, pp. 473-486.

[2] Fosdick, L.D. and Osterweil L.J. "Data Flow Analysis in Software Reliability," ACM Computing Surveys 8, pp. 305-330.

[3] Osterweil, L.J., Clarke, L.A. and Smith, D.W. "A Data Base System Designed for Flexibility and Usability for Fortran," Tech. Rep. CU-CS-072-75, Dept. of Computer Science, Univ. of Colo. at Boulder, Boulder, Colo., July 1975.

[4] Myers, E.W. "The BIGMAC User's Manual," Tech. Rep. CU-CS-145-78, Dept. of Computer Science, Univ. of Colo. at Boulder, Boulder, Colo., Nov. 1978.

[5] Liskov, B.H. and Zilles, S.N. "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering SE-1, pp.7-19, (1975).

[6] Ryder, B.G. "The PFORT Verifier," Software-Practice and Experience 4, pp. 359-377, 1974.

[7] Boyle, J. and Matz, M. "Automating Multiple Program Realizations." MRI Conf. Rec. XXIV Symp. on Computer Software, Polytechnic Press (1976), 421-456.

[8] Baker, B.S. "Generalized Syntax Directed Translation, Tree Tranducers, and Linear Space." Siam J. Computing 7, 3 (1978), pp. 376-391.

[9] Krishnaswamy, R. and Pyster, A.B. "On the Correctness of Semantic-Syntax-Directed Translations." J. ACM 27, 2 (1980), pp. 338-355.

[10] Aho, A.V. and Ullman, J.D. The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing. Prentice-Hall (1972).

[11] Morris, J.B. "Data Abstraction: A Static Implementation Strategy," Conf. Rec. SIGPLAN Symp. on Compiler Construction, (1979),1-7.

[12] Melton, R.A. "Automatically Translating Fortran to IFTRAN," Proc. 8th Annual Symp. on Interface, UCLA Health Sci. Comp.Facility, pp. 291-297, (1975).

[13] Cole, A.J. Macro Processors. Cambridge University Press (1976).

[14] American National Standards Institute, FORTRAN, ANSIX3.9 (1966).

[15] Elson, M. Concepts of Programming Languages, Science Research Associates, Inc. (1973).

[16] Knuth, D.E. "Semantics of Context-Free Languages," Math Systems Theory 2, 2 (1968), pp. 127-145.

[17] Myers, E.W. "An Introduction to FLAT," Tech. Rep. CU-CS-179-80, Dept. of Computer Science, Univ. of Colo. at Boulder, Boulder, Colo. June 1980.

APPENDIX A

A BIGMAC TRANSLATOR FOR A BIT-VECTOR DATA TYPE

Many algorithms employ the notion of a set whose underlying implementation is assumed to utilize bit vectors. This provides the motivation for extending the FORTRAN language to incorporate a BIT VECTOR data type via a BIGMAC translator. Such an extension is described in this appendix.

We begin by informally describing the syntax and semantics of the augmentation. Simple variables and arrays may be declared to be of type BIT VECTOR. For example, the statement 'BIT VECTOR(180) A,B(10)' declares A to be a bit vector of 180 bits, and B to be a 10 element array whose elements are bit vectors of 180 bits. Bit vector variables may be formal parameters but not function names. It was also thought to be convenient to allow one to declare integer constants. For instance,

```
CONSTANT  NPROC = 100
CONSTANT  NVAR  = 1000
```

declares NVAR to be the constant 1000 and NPROC to be the constant 100. These statements can occur anywhere within the input and apply in all subsequent code. For example,

```
BIT VECTOR(NVAR) LOCAL(NPROC)
```

declares LOCAL to be a 100 element array of 1000 bit bit vectors.

Bit vector arithmetic is performed with the aid of a number of binary, unary, and nullary operators with which one can build expressions. These bit vector operations are sketched in the table below.

Op	Precedence	Type of Result	Meaning
A.NE.B	400(lowest)	LOGICAL	A ≠ B
A.EQ.B	400	LOGICAL	A = B
A.UNION.B	300	BIT VECTOR	A ∪ B
A.INTER.B	200	BIT VECTOR	A ∩ B
A.DIFF.B	100(highest)	BIT VECTOR	A - B
.COMP.A	-	BIT VECTOR	~ A
.EMPTY.	-	BIT VECTOR	Ø
.UNIV.	-	BIT VECTOR	~ Ø

For all binary operators, the arguments must be bit vectors of the same length. Additional operators can be supported as needed by adding to the translation specification which supports this extension. A specific bit of a bit vector may be tested by referencing the bit's position. For example, the expression '(A.INTER.B(1))(I*3)' returns a logical value which is true if and only if the $I*3^{th}$ bits of both A and B(1) are set. Bit vector expressions may be actual arguments to a procedure invocation.

Assignment is extended to include the BIT VECTOR data type. For example, the statement 'B(3) = .COMP.A' assigns the bit vector value of the expression .COMP.A to the variable B(3). In addition, individual bits can also be assigned. For example, 'B(3)(2) = .TRUE.' sets the second bit of B(3) and 'B(3)(2) = A(2)' assigns the second bit of B(3) to the value of the second bit of A.

The translation specification which supports this extension is quite lengthy (approximately 400 lines) and thus will not be discussed in detail. Instead, the nature of the transformation this translation specification performs will be discussed using the sample translation in Figure A as an example.

All occurrences of a constant identifier are replaced with the constant assigned to the identifier. For example, in Figure A, 'COMMON /LIST/ LLST, LARR(NVAR)' becomes 'COMMON /LIST/ LLST, LARR(1000)'. All bit vector variables are turned into integer arrays by the addition of an extra dimension. The size of this dimension is the number of words required by a bit vector. The specification in Figure A is for a CDC 6600 which has 60 bit words. Thus the declaration, 'BIT VECTOR(NVAR) TV' becomes 'INTEGER TV(17)' where 17 equals (NVAR-1)/60+1.

Any reference to a bit vector variable within the executable part of a program unit, is transformed into an augmented array reference where the extra dimension is a unique free variable. For example, a reference to 'TP' becomes 'TP(I00000)' and a reference to 'AT(SN)' becomes 'AT(I00000,SN)'. The free variable, I00000, is used as required by the context of the reference. For example, the assignment 'AT(SN) = TP' is transformed into the two lines of code —

```
                DO 10004 I00000 = 1,2
10004             AT(I00000,SN) = TP(I00000)
```

By constructing a do loop in which the free variable is the index, the assignment takes place word by word. As another example of the context sensitive usage of the free variable, consider the code generated for the bit selection in the statement TP(P) = .TRUE. —

```
I00000 = (P-1)/60+1
I00000 = P-(I00000-1)*60
TP(I00000) = TP(I00000).OR.I00000(I000001)
```

The free variable pair (I00000,I000001) determine the word and index within the word of the selected bit P. I00002(I00001) is a constant word whose I00001th bit is set in the block data unit initializing I00002.

The free variable is not bound, however, when forming bit vector expressions. For example, the expression 'LOCAL(P).DIFF.FORMAL(P).INTER.OPT(P)' is transformed into 'LOCAL(I00000,P).AND..NOT. FORMAL(I00000,P).AND.OPT(I00000,P)'. The free variable is not bound until a context like the ones above is reached. In Figure A, the expression is passed by value to the subroutine LIST in the statement 'CALL LIST(LOCAL(P).DIFF.FORMAL(P).INTER.OPT(P), NVAR)'. The code generated for this statement is —

```
      DO   10003 I00000 = 1,17
10003      I00003(I00000) = LOCAL(I00000,P).AND..NOT.
     *     FORMAL(I00000,P).AND.OPT(I00000,P)
      CALL LIST(I00003,1000)
```

A unique bit vector variable I00003 is generated and the value of the expression is assigned to it. I00003 is then passed to LIST as an actual argument.

Some features of the translator are not illustrated in Figure A. The constants .EMPTY. and .UNIV. are folded whenever possible. That is, an expression like (A.INTER..EMPTY.).DIFF.B is simplified to '.COMP.B'. Semantic errors such as type incompatibilities are detected and reported. For example, the expression 'TP(SNP)(P).AND.AT' is transformed into '**ERROR**.AND.**ERROR**' as TP(SNP) is not a bit vector expression and AT is not a logical expression.

The code generated is quite efficient but not optimal. In Figure A, only one temporary variable is generated and occupied only 17 words of storage. A small amount of superfluous code is present. For example, the word-index pair for bit position V is computed twice in the sequence of statements —

```
TV(V) = .TRUE.
CALL LIST(LOCAL(P).DIFF.FORMAL(P).INTER.OPT(P),NVAR)
TV(V) = .FALSE.
```

when it only needed to be computed once.

FIGURE A: SAMPLE TRANSFORMATION

INPUT:

```
      CONSTANT NPROC = 100
      CONSTANT NVAR  = 1000
      CONSTANT NSET  = 3000
        ..
        ..
      SUBROUTINE ALIAS(LOCAL,FORMAL,OPT,NUMP,AT)
C
C     BIT VECTOR(NVAR) LOCAL(NPROC),FORMAL(NPROC),OPT(NPROC)
C
C     BIT VECTOR(NPROC) AT(NSET)
C
C     BIT VECTOR(NVAR)  TV
C
C     BIT VECTOR(NPROC) TP
C
C     COMMON /LIST/ LLST,LARR(NVAR)
C
      INTEGER V,P,
     *        SN,HASH,
     *        WKP,WLIST(NSET)
C
C
      CALL INTHSH
      WKP=0
      TP=.EMPTY.
      TV=.EMPTY.
      DO 30 P=1,NUMP
        TP(P)=.TRUE.
        CALL LIST(LOCAL(P).DIFF.FORMAL(P).INTER.OPT(P),NVAR)
        DO 20 I=1,LLST
          V=LARR(I)
          TV(V)=.TRUE
          SN=HASH(TV,SET,NUMS)
          TV(V)=.FALSE.
          AT(SN)=TP
          WKP=WKP+1
   20     WLIST(WKP)=SN
   30   TP(P)=.FALSE.
C
      END
```

FIGURE A: (continued)

```
      BLOCKD DATA A00001
      COMMON /A00000/ I00000(60)
      DATA I00000(1)/4000000000000000000000B/,I00000(2)/2000000000000000
     *0000B/,I00000(3)/1000000000000000000000B/
        ..
        ..
      DATA I00000(58)/0000000000000000000004B/,I00000(59)/00000000000000
     *000000B/,I00000(60)/0000000000000000000001B/
      END
        ..
        ..
      SUBROUTINE ALIAS(LOCAL,FORMAL,OPT,NUMP,AT)
C
      INTEGER LOCAL(17,100),FORMAL(17,100),OPT(17,100)
C
      INTEGER AT(2,3000)
C
      INTEGER TV(17)
C
      INTEGER TP(2)
C
      COMMON /LIST/ LLST,LARR(1000)
C
      INTEGER V,P,
     *        SN,HASH,
     *        WKP,WLIST(3000)
C
C
      COMMON /A00000/ I00002(60)
      INTEGER I00003(17)
C
      CALL INTHSH
      WKP=0
      DO 10000 I00000=1,2
10000   TP(I00000)=000000000000000000000B
      DO 10001 I00000=1,17
10001   TV(I00000)=000000000000000000000B
      DO 10002 P=1,NUMP
        I00000=(P-1)/60+1
        I00001=P-(I00000-1)*60
        TP(I00000)=TP(I00000).OR.I00002(I00001)
        DO 10003 I00000=1,17
10003     I00003(I00000)=LOCAL(I00000,P).AND..NOT.FORMAL(I00000,P).AND.
     *    OPT(I00000,P)
        CALL LIST(I00003,1000)
        DO 20 I=1,LLST
          V=LARR(I)
          I00000=(V-1)/60+1
          I00001=V-(I00000-1)*60
          TV(I00000)=TV(I00000).OR.I00002(I00001)
          SN=HASH(TV,SET,NUMS)
          I00000=(V-1)/60+1
          I00001=V-(I00000-1)*60
          TV(I00000)=TV(I00000).AND..NOT.I00002(I00001)
          DO 10004 I00000=1,2
10004       AT(I00000,SN)=TP(I00000)
          WKP=WKP+1
   20     WLIST(WKP)=SN
   30   I00000=(P-1)/60+1
        I00001=P-(I00000-1)*60
10002   TP(I00000)=TP(I00000).AND..NOT.I00002(I00001)
C
      END
```