NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A PRELIMINARY SKETCH OF FORMULA ALGOL

by

Alan J. Perlis
Renato Iturriaga
Thomas Standish

Revised July 21, 1965

The research reported here was supported by the Advanced Research Projects Agency of the Department of Defense under the Contract SD-146 to the Carnegie Institute of Technology.

INTRODUCTION

In the information processing sciences a central role is played by programming languages and by the complex information processing programs that we can write from them. The power of programming languages available to the programmer often determines whether or not a given programming task can be accomplished without prohibitive expense. As the power of available programming languages increases both the variety of information processing tasks that can be programmed and the ease with which they can be programmed increase correspondingly.

In earlier years algebraic languages, list processing languages and string manipulating languages have existed separately from one another. Recently, formula manipulating languages have evolved, and, in addition, there have been efforts to combine various different kinds of processing in one language.* The design of Formula Algol represents an effort in this direction. Specifically, Formula Algol is an extension to Algol providing formula manipulating, list processing, and limited string processing capabilities. Thus, Formula Algol is a language in which the advantages of these various different kinds of processing are combined, but we anticipate that Formula Algol will be particularly well adapted to algebraic symbol manipulating processes.

* For example: FLPL [1] , FORMAC [2], ALPAK [3], the earlier version of FORMULA ALGOL [4], and AMBIT [5].

TABLE OF CONTENTS

I. EXPRESSIONS

There are seven different kinds of expressions in Formula Algol. The first three kinds of expressions, namely designational expressions, arithmetic expressions, and Boolean expressions are those of Algol.

In addition, there are four other kinds of expressions in Formula Algol which are not in Algol. These are:

(1) Formula Expressions,

(2) Pattern Expressions,

(3) Symbolic Expressions, and

(4) Tree Expressions.

Every expression has a value. These values have associated types. Types INTEGER, REAL, and BOOLEAN, are already defined in Algol. In addition to these Formula Algol has two new types, FORM and SYMBOL. A variable declared of type FORM will take on two properties. First, formulas and patterns may be stored into it. Second, it may be used within formula and pattern expressions. Likewise, a variable declared of type SYMBOL may contain lists and may occur in symbolic expressions and tree expressions.

Whereas arithmetic and Boolean expressions have values whose types are always arithmetic and Boolean respectively, such is not the case for formula expressions, pattern expressions, symbolic expressions, and tree expressions. By contrast, formula expressions and pattern expressions may take on not only values of type FORM but also any one of the others. Likewise, symbolic expressions and tree expressions may take on not only values of type SYMBOL but also any one of the others. [Note: This means that formula expressions and pattern expressions may take on values of type SYMBOL, and that symbolic expressions and tree expressions may take on values of type FORM.]

II. FORMULA MANIPULATION IN FORMULA ALGOL

A. Data Structures for Formulas

In addition to the data structures already in Algol, Formula Algol introduces some new data structures. Formulas are one of these additional new data structures.

A formula is represented as a list structure composed ✓ as follows: The representation is either

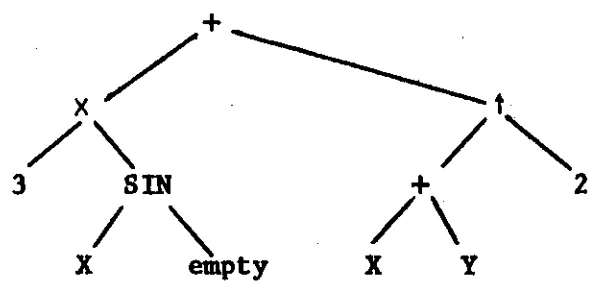
(1) an atomic formula variable, or

(2) an atomic constant, or else it is

(3) a combination, consisting of a binary operator, a left half, which is a formula, and a right half, which is a formula (or which is empty in the case of a unary operator which is a binary operator with an empty right half)

For example:

If X and Y have been declared of type FORM, then the expression 3 × SIN (X) + (X + Y) ↑ 2 is represented by the following list structure:



Formulas need not be only arithmetic and Boolean. In addition, there are four other types of formulas:
(1) conditional formulas (2) assignment formulas (3) procedure formulas, and (4) array formulas. The internal computer representation of all of these formulas is the same. In analogy to the internal representation of arithmetic and Boolean formulas, additional types of binary operators are introduced allowing us to represent conditionals, assignment statements, procedures, and array elements. This is done as follows:

(1) Conditional Formulas

Suppose we have the statement

```
F ← .IF B THEN G ELSE H;
```

where F is of type FORM and where the type of B, G and H is immaterial. This creates a formula which we may represent by

$$\begin{tikzcd}
 & \rightarrow \arrow[dl, no head] \arrow[dr, no head] & & \\
B & & \text{EITHER} \arrow[dl, no head] \arrow[dr, no head] & \\
 & G & & H
\end{tikzcd}$$

in which $\rightarrow$ and EITHER are internal binary operators (meaning that these operators are not part of the alphabet of the source language). The formula is stored into F, causing the value of F to become a conditional formula. Later, when we discuss the evaluation of formulas, we will see that the conditional action represented by this formula can be executed when the EVAL operator is applied to F. Here G will become the value of F if B is found to be true and H will be the value of F otherwise.

(2) Assignment Formulas

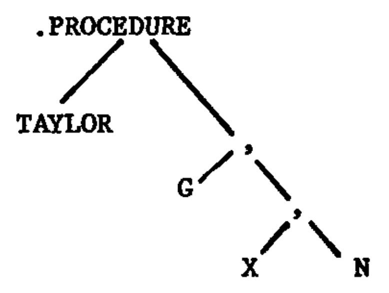
Suppose we have the statement F $\leftarrow$ G. $\leftarrow$ A + B; where F is of type FORM and the types of G, A and B are any type other than symbol. This creates a formula, which we may represent by

$$\begin{tikzcd}
& .\leftarrow \arrow[dl, dash] \arrow[dr, dash] & & \\
G & & + \arrow[dl, dash] \arrow[dr, dash] & \\
& A & & B
\end{tikzcd}$$

in which .← is a binary operator, and this formula is stored into F. Hence, the value of F is an assignment formula. If, at a later time, we apply the EVAL operator to F, the assignment of A + B to be the contents of G takes place. Thus, at one point in a program, we may describe the skeleton of an assignment, and at a later time, as a result of some unpredictable running experience, the program may fill in the details and carry out the assignment. The essential theme of this is that we may specify at any point to any desired degree of detail a partial schema for a computation, postponing until later the specification of details dependent on the outcome of further computation which details we cannot predict.

(3) Procedure Formulas

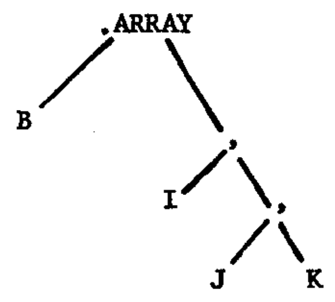
Suppose we have the statement F ← TAYLOR.(G,X,N); where F,G,X, and N are of type FORM. Executing this statement creates a formula which we may represent by



in which .PROCEDURE is a binary operator, and this formula is stored in F. As in the previous case, applying the EVAL operator to F causes the actual execution of the procedure TAYLOR.

(4) Array Formulas

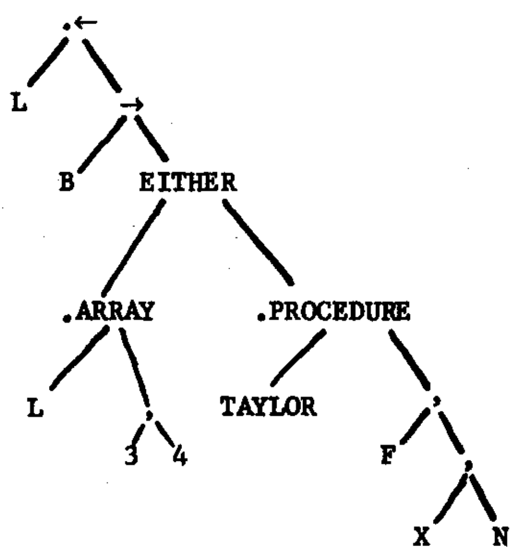
Executing the statement F ← B.[I,J,K]; causes the formula



to be stored in F, where .ARRAY is a binary operator. F must be of type FORM, B can be of any type, but it must be the identifier of a previously declared array. Applying the EVAL operator to F with values for I, J, and K causes the execution of the array access. For example, executing the statements

```
A ← L.[3,4];
P ← TAYLOR. (F,X,N);
G ← .IF B THEN A ELSE P;
F ← L.←G;
```

causes the following formula to be assigned the value of F



The value of F thus represents a postponed assignment to L of a postponed conditional which if B is true represents the postponed array access L[3,4], and if B is false the postponed procedure execution TAYLOR (F,X,N).

B. Operators for Formulas

Having added formulas as a new type of data structure certain classes of operators are immediately suggested.

(1) Constructors

A formula is always created by writing directly the linearized algebraic expression which represents it. For example, if the formula, $3 \sin x + (x + y)^2$, is going to be stored in the formula variable Z, we shall write:

```
Z ← 3 * SIN (X) + ( X + Y ) ↑ 2 ;
```

After the execution of this statement, Z is a formula variable having a formula value represented by the list structure described on Page 3.

As we will see later the EVAL operator is also capable of being used as a constructor.

(2) Predicates and Selectors

If as a result of some computation a formula has been built up whose structure cannot be predicted in advance, we may precisely determine its structure by the use of predicates. These predicates are <u>sufficient</u> in the sense that, whatever constructions are used to create a formula, the process may be reversed by the choice of a sequence of predicates. In the case of formulas, we call these predicates "patterns". Patterns are defined by the following BNF syntax equations:

```
<pattern>::= <formula expression> == <pattern structure>|
     <formula expression> >> <pattern structure>|
     <identifier>:  <formula expression > >>
          <identifier>:  <pattern structure>
```

In this definition, a formula expression is compared with a "pattern structure" to determine whether the expression is an exact instance of the pattern structure or whether the expression contains an instance of the pattern structure. Thus, we write <formula expression> == <pattern structure> to mean "Is the formula expression an instance of the pattern structure?", and we write <formula expression> >> <pattern structure> to mean "Does the formula expression contain an instance of the pattern structure?" The use of the <identifier>'s in the definition of a pattern will be explained later. Pattern structures are defined as follows:

```
<pattern structure>::=
<a formula in which some of the primaries may have
  been replaced by pattern primaries and some of the
  operators may have been replaced by |<variable>| >
```

<pattern primary>::= <unlabelled pattern primary>|

<variable> : <unlabelled pattern primary>

<unlabelled pattern primary> ::= <type> | OF (<variable>) |ATOM

The colon used in the definition of pattern primaries is an extraction operator. It extracts the part of the formula matching the corresponding unlabelled pattern primary. A copy of the extracted part of the formula is assigned as the value of the identifier found to the left of the colon.

Suppose R is a variable declared of type SYMBOL for which the following assignment statement has been executed:

R ← / [index: J] [operator: +,-,/] [comm:true,false,false];

The effect of executing this statement (as will be explained later when we introduce the list processing features of Formula Algol) is to assign R a description list whose attribute value list follows the first occurrence of the mark / in the above assignment statement. Here, each attribute precedes the colon inside each pair of square brackets and the value list associated with that attribute follows the colon.

Consider now the pattern F == A : INTEGER |R| B:FORM. This pattern will be true in any of the two following cases:

(1) The first operand of F is an integer <u>and</u> the second is a formula <u>and</u> the main operator is either +, -, or /.

(2) The first operand of F is a formula <u>and</u> the second operand of F is an integer <u>and</u> the operator is +.

In case 1, assuming there was a match, A is assigned the value of the integer, B is assigned the value of the formula, the index J is set to 1, 2, or 3 according to whether the main operator was +, -, or / respectively, and this main operator is stored as a data term in the contents of R. In case 2, since the value of the attribute "comm" on the description list of R is the list [true, false, false], the pattern also stands for commutative instances of the right and left operands about any of the operators +, -, or / whose corresponding Boolean values following the attribute "comm" are set to <u>true</u>. Thus, in this case, + is a commutative operator and - and / are not. Therefore, only commutative instances about the operator + are considered. We note that [comm: true, false, false] need not appear on the description list of R at all, in which case no commutative instances of any operator will be considered. Later we can use the construction $|\langle R\rangle|$ in an expression in place of an operator. The operator that $|\langle R\rangle|$ stands for is exactly the one which was extracted during previous pattern matching. Alternatively, R may be assigned any operator by the assignment statement $R \leftarrow +$; and $|\langle R\rangle|$ may be used in the same fashion.

The following are examples of patterns:

Example 1. Let A,B,X,Y, and Z be declared of type FORM, and let R be declared of type REAL. Suppose the statement

```
X ← 3 * SIN ( Y )  +  ( Y - Z ) / R + 2 * R ;
```

has been executed.

Consider now the statement:

```
IF  X  >>  A : INTEGER * B: SIN(FORM)
     THEN Z ← 2 * B + A ;
```

Since the pattern X>> A:INTEGER * B:SIN(FORM) is true, the assignment Z ← 2 * B + A will be executed, assigning as the value of Z the formula 2 * SIN (Y) + 3 because A has the value 3 and B the value SIN(Y).

Example 2. Let X be of type SYMBOL, A,B,Y,M,T,G, and P be of type FORM, and let D be of type BOOLEAN. Then executing the statements:

```
X ← [REAL,INTEGER,BOOLEAN];
G ← Y + 8 * ( M - T ) ;
P ← FORM + A : OF ( X ) * B: FORM ;
D ← G == P ;
```

causes D to be set to <u>true</u> because the pattern G == P is true, and causes A to be set to 8 and B to be set to M - T. In this example, we observe that in the definition of a pattern primary a type may be replaced by a symbol having a list of types as contents.

Suppose that we wish to extract either the name or the parameters of a procedure formula or that we wish to extract either the name or the subscripts of an array formula. To do these, we may write patterns of the form

```
F ==  X: FORM. (P1:FORM,P2:FORM)
F ==  X: FORM. [X1:INTEGER,X2:INTEGER]
```

Here X will contain the name of the procedure or array and P1,P2,X1, and X2 will contain the parameters or subscripts in the event that there is a match. Extraction from conditional formulas and assignment formulas is entirely analogous.

We now explain by means of an example the function of the colon in patterns of the form

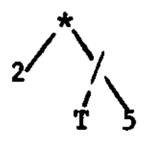
```
<identifier> : <formula expression> >>
     <identifier> : <pattern structure>
```

Suppose we have executed the statements

F ← 2* (SIN(X↑2 + Y↑2) + COS(X↑2 - Y↑2)) /5;

G ← SIN(FORM) + COS(FORM),

where all variables used are of type FORM. Then A: F >> T: G is a pattern with value <u>true</u>. The value of T will replace the first instance of G in F, i.e., the expression SIN(X↑2 + Y↑2) + COS(X↑2 - Y↑2) [this being the first sub-expression of F matching the pattern G] and A is assigned the expression 2* T /5. In this case, the atomic symbol T is contained in the formula



stored in A.

Thus, A is the same as F with the first sub-expression of F matching G replaced by the value of T.

3. Evaluation Rules

We may think of formulas as abstractions of computations. By manipulating formulas we alter the computations they represent. At some point in the execution of a program, we may wish to carry out the computation represented by a formula. To do this, we could substitute values for those variables whose values are not assigned, and those values will be combined according to the computation expressed by the formula resulting in an evaluated formula. In order to accomplish the above we have the EVAL operator.

If we have a formula consisting of formula variables joined by arithmetic operators, then, if we assign each of the formula variables a numerical value, the result of evaluation of the formula will be a number. Hence, the evaluation of an arithmetic formula by complete substitution of numbers for formula variables is a computation carrying the set of numbers substituted into a number. Analagously, substitution of Boolean values for formula variables in a Boolean formula produces a Boolean value.

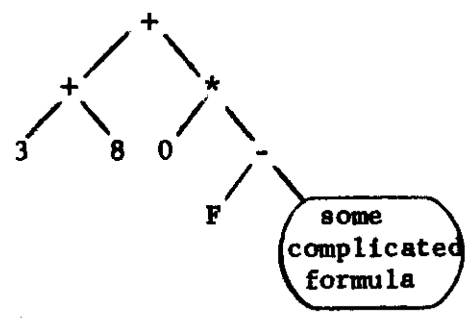
On the other hand, we need not substitute arithmetic or Boolean values for formula variables, but rather, we can substitute other formulas. Thus, in this case, evaluation of the formula, instead of collapsing it to a single value, expands it to an enlarged formula. Hence, EVAL may be used as a constructor.

A third use of EVAL is that of producing trivial simplifications in a formula without altering its value and without substitution.

A final use of EVAL is to carry out the array access or procedure call indicated by a "dot array formula" or a "dot procedure formula", or to carry out the assignment of a value or the choice of a value indicated by a "dot assignment formula" or a "dot if formula".

For example: If the formula X has the value

X =



Then the expression EVAL X will produce the number 11 as value, as in the example V ← 3 + EVAL X, after which V has the value 14. X is not altered by EVAL X.

These uses of EVAL represent extreme cases. In a given application they may be combined. Thus evaluation of a formula may produce partial collapsing, partial expansion, and some trivial algebraic simplification simultaneously. [Note: substitution is always simultaneous.]

The syntax and interpretation of evaluation formulas is as follows:

```
<EVAL formula> ::= EVAL  <variable> |
    EVAL <bound variables> <expression> <list of values>
<list of values> ::= (<actual parameter list>) | ([<variable>])
<bound variables> ::= (<variable list>) | ([<variable>])
<variable list> ::= <variable> | <variable list> , <variable>
```

Consider a statement of the form:

$$D \leftarrow \text{EVAL } (X_1, X_2, \ldots, X_m) \text{ F } (Y_1, Y_2, \ldots, Y_n) \qquad [1]$$

where $n \geq 1$, and $m \geq 1$.

Then it is the case that:

(a) F must be a variable declared of type FORM; nevertheless at execution time it may become any type.

(b) If the current value of F is not an expression, then the effect of [1] is precisely that of $D \leftarrow F$.

(c) If the current value of F is an expression, then D will have the value obtained by the execution of the piece of code produced by the interpretation of F as an arithmetic or Boolean expression, in which were substituted Y_i for all of the corresponding occurrences of X_i in F, for all $i \leq m$. The substitution is made in accordance with the following rules. If

X_i is not of type FORM, X_i is ignored; or if

X_i is of type FORM, but does not occur in F, X_i is is ignored, or if

X_i is of type FORM, and it occurs in F, but $i > n$, X_i is ignored, or if

X_i is of type FORM, and it occurs in F, with $i \leq n$, then X_i is replaced by Y_i.

(d) Y_i may be an expression of any type.

(e) The expression F is not changed as a consequence of the execution of EVAL.

(f) D must be of type FORM, unless the evaluation of F produces a numerical or logical value.

The evaluation process creates a new expression which may be ultimately of any type, depending on current values.

Executing EVAL F where F is an assignment formula, a procedure formula or an array formula, respectively, causes the assignment to be executed, the procedure to be evaluated, and the array element to be accessed respectively.

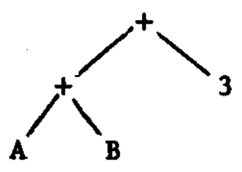
When evaluating a conditional formula, only if the Boolean formula in the if clause collapses to a Boolean value will the conditional formula be executed. This process requires substitutions. If, on the other hand, the Boolean formula does not collapse to a single value, then another conditional formula is constructed with the corresponding substitutions.

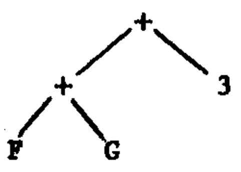
C. Remarks

When the value of a formula is given by an assignment statement later assignment statements <u>do not</u> alter the originally assigned formula. Assignments are <u>not retroactive</u>. For example,

```
Y ← A + B;
X ← Y + 3;
Y ← F + G;
```

After the execution of these statements X has as its value the formula and not





In symbol manipulation the reverse is true. As we will see later, during the discussion of the list processing features of the language, after the execution of the following three statements:

```
Y ← [ A,B ];
X ← [ Y,3 ];
Y ← [ F,G ];
```

the value of the expression <FIRST OF <X> >, which means "the contents of the first element of the list stored as the contents of X", will be the list [F,G] and not the list [A,B].

Experience indicates that those with a background in list processing languages confuse the above mentioned point when dealing with Formula Algol for the first time.

III. LIST PROCESSING IN FORMULA ALGOL

A. Symbolic Data Structures

There are three kinds of symbolic data structures: symbols, list structures, and description lists.

(1) Symbols

A variable, S, in Formula Algol, which is declared to be a symbol acquires the following properties:

(a) S names a storage location into which symbolic data structures may be stored.

(b) S may have a description list attached to it into which attributes and values may be entered and retrieved.

(c) S may be used as an atom in constructing symbolic data structures.

Note: Unlike other types of variables in Formula Algol, S does not have a value. The contents of S is not the value of S. Instead, the contents must be accessed by applying an operator to S, namely, by enclosing S in contents brackets, <S>. Whereas writing the real variable R in a statement such as $T \leftarrow 2 * R$; causes the value of R to be used, writing the symbol S in the list [A, B, C, S] causes S itself to be entered into the list, and not some value or structure which S stands for.

(2) List Structures

Symbols may be concatenated into a list by writing them one after another, and by separating them with commas. This list may be assigned as the contents of another symbol by executing an assignment statement. E.g. VOWEL ← [A, E, I, O, U];

An item concatenated into a list need not be a symbol. It may be any expression legal in Formula Algol. For example: let X, Y, and Z be formulas, let A, B, and C be Boolean, let U, V, and W be real, and let R, S, and T be symbols. Then the following assignment statement is legal.

```
S ← [ X + SIN (Y), 3 + 2*U, IF B THEN R ELSE T, [R,T,R], -36];
```

At the time this statement is executed, each expression on the right is evaluated, and the list of values is stored into the contents of S. In effect, automatic data term declaration results from storing non-symbolic values into list structures. Note that the second from the last item in the above expression is the quantity [R, T, R]. This becomes a sublist of the list stored into S. Hence, the expression stored into S is, in reality, a list structure. For this reason, the expression on the right hand side of the above statement is called a tree expression. Tree expressions represent list structures into which values of any type may be entered and retrieved. The operators for manipulating tree expressions will be introduced subsequently.

(3) Description Lists

A description list is a sequence of attributes and values.

Each attribute is followed by a list of values associated with it. This value list may contain more than one member, it may contain only one member, or it may be empty. A description list is always attached to a symbol and becomes permanently bound to it. Any symbol may become an attribute, and, in addition to symbols, the value of any expression legal in Formula Algol may become a value. Let A_1, A_2, and A_3 be symbols used as attributes, and let V_{mn}, for m and n integers, stand for values. Then an example of a description list is

$$/ [A_1 : V_{11}, V_{12}, V_{13}] [A_2 : V_{21}] [A_3 : V_{31}, V_{32}, V_{33}, V_{34}]$$

The operators for entering and retrieving attributes and values in description lists will be introduced subsequently.

B. Operators for Accomplishing List Processing

The introduction of symbolic data structures into Formula Algol makes mandatory the existence of certain classes of operators among which are first, constructors, to create them; second, selectors, to gain access to various parts of them; and third, predicates; to determine the structure of those whose structure is unknown.

1. Constructors

The most elementary and direct method of creating both tree expressions and description lists is to write them out linearly and to store them into the contents of some symbol. E.g. Assume that all identifiers in the following statement are symbols:

S ← [A,B,C,D] ;

S ← /[types: mu, pi, rho] [ancestors: orthol, para5] [color: green] ;

The first statement creates the list [A,B,C,D] and stores it as the contents of S. The second creates the description list / [types: mu, pi, rho] [ancestors: orthol, para5] [color: green] and attaches it to the symbol S.

Methods of creating and altering both tree expressions and description lists by means of editing statements and value entry statements will be introduced after the introduction of selectors.

2. Selectors

Symbolic expressions may, upon evaluation, yield both symbols and lists of symbols as values. They may also, upon evaluation, yield arithmetic, Boolean, and formula values. Anywhere a symbol appears it may be replaced by a symbolic expression, which, upon evaluation, yields a symbol as a result. Likewise, anywhere a list of symbols appears it may be replaced by a symbolic expression, which, upon evaluation, yields a list of symbols as a result. Selectors are operators which are applied to a symbolic data structure to gain access to a part of that data structure. The following classes of selectors are available:

(a) Retrieval of the contents of a symbol.

Suppose S ← [A,B,C] ; has been executed. Then S contains the list [A,B,C] . To access the contents of S we form the expression < S >. This is read "the contents of S". It is a symbolic expression whose value is, in this case, the list [A,B,C] . If we execute the statements: T ← S; and S ← [A,B,C] ; then < T > is a single valued symbolic expression with value S, and < < T > > is a list valued symbolic expression with value [A,B,C] . The angular brackets, < >, may be nested arbitrarily many times to give arbitrarily many levels of indirection.

(b) Retrieval of values from a description list.

Suppose the statement

```
S ← /[types:  mu, pi, rho] [ancestors: orthol, para5]
     [color:  green] ;
```

has been executed causing a description list to have been created and to have been attached to S. Suppose we wish to determine the values of the attribute "ancestors" on the description list of S. To do this we write "THE ancestors OF S". This is a list valued symbolic expression having the list [orthol, para5] as value. The expression "THE color OF S" is a single valued symbolic expression having the symbol "green" as value.

(c) Retrieval of elements and sublists of a list.

Suppose the statement S ← [X, A, X, A, A, X, A, A, A, X] ; has been executed causing the list shown to be stored as the contents of S. We may access the various symbols on this list by means of "selection expressions". Selection expressions consist of selector operators applied to list valued symbolic expressions. We know already that < S > is a list valued symbolic expression having the list [X, A, X, A, A, X, A, A, A, X] as a value. Hence, the expression 3RD OF < S > is a single valued symbolic expression having the value X. Likewise, LAST OF < S > has the value X, whereas 2ND BEFORE LAST OF < S > has the value A, and whereas 2ND BEFORE 3RD X OF < S > has the value A. Selection expressions need not have single values. For example: the expression FIRST 4 OF < S > has the list [X, A, X, A] as value, and the expression BETWEEN FIRST X AND 1ST BEFORE LAST X OF < S > has the list [A, X, A, A, X, A, A] as value. Selectors may be compounded to access sublists and their elements. Suppose the statement S ← [A, [X, X, [A, A], X], A] has been executed. Then the expression 2ND OF < S > is a list valued symbolic expression with the list [X, X, [A,A], X] as value, whereas the expression 3RD OF 2ND OF < S > has the list [A,A] as value, and whereas the expression LAST OF 3RD OF 2ND OF < S > is a single valued symbolic expression with the value A.

3. Predicates

Predicates for determining the structure of lists and list structures are of two kinds: "list patterns" and "relations". List patterns use the mechanisms found in COMIT to test whether a linear list is an instance of a certain linear pattern. The constituent selector list describes the pattern being tested for, and is composed of a sequence of constituent selectors separated by commas. The symbols $, and $n may be used as constituent selectors with the same significance as in COMIT [viz. $ stands for any arbitrary number of successive constitutents, and $n stands for n consecutive arbitrary constituents]. If a symbolic expression is used as a constituent selector, its value is first computed, and if that value is a list, each element of the list becomes one of the consecutive constituent selectors in the constituent selector list. Other kinds of elements, to be introduced later, may also become constituent selectors.

A list pattern compares the list determined by a list valued symbolic expression to a linear pattern described by a constituent selector list to see if the list is an instance of the pattern. The list pattern is a Boolean primary with values _true_ and _false_ and thus may be combined with other Boolean expressions by means of logical connectives.

A list pattern has the syntax

```
<list pattern> ::= <list valued symbolic expression> HAS
                                        <constituent selector list>
```

By looking at two examples these concepts will become clear.

Example 1. Suppose the statement S ← [A,B,C,D] has been executed, where all variables involved have been declared to be symbols. Consider now the statement

```
IF < S > HAS $1, B, $ THEN T ← [ < T >, B ] ELSE
                T ← [ < T >, LAST OF < S > ];
```

Since the contents of S, which is the list [A,B,C,D], is an instance of the pattern $1, B, $ (which is read " a single arbitrary constituent, followed by a B, followed by any number of arbitrary constituents ") the predicate " < S > HAS $1, B, $ " is true. Therefore, T ← [< T >, B] is executed, which has the effect of appending a B to the end of the list stored as the contents of T.

As with the pattern expressions used to determine the structure of formulas, list patterns may also function as selectors. The same mechanism is used to accomplish this. If any constituent selector in a constituent selector list is preceded by a variable declared of type SYMBOL followed by a colon, then the corresponding constituent in the list being tested, in the event that there is a match, is inserted into the contents of that symbol variable. The contents may be accessed at any later point in the program.

Example 2. As in the previous example, suppose the statement S ← [A,B,C,D] has been executed where all variables are symbols. Then executing the statement

```
     IF < S> HAS T: $2, V: $2 THEN S ← [ < V >, < T > ];
```

changes the contents of S to be the list [C,D,A,B]. Furthermore, < T > is the list [A,B], and < V > is the list [C,D].

Relations constitute a second class of predicates. The following kinds of tests are available:

(a) Equality Relations

If we have two symbolic expressions we may test whether their values are equal by means of the relation <symbolic expression> = <symbolic expression>. The values of the symbolic expressions may be single symbols, lists of symbols, formulas, or values of any other type. Naturally if the values of the two symbolic expressions are non-conformable data structures the result of the predicate will be false.

(b) Testing for types

A single valued symbolic expression having a value whose type is unknown may be used in the relation <symbolic expression> IS <restricted type> in order to determine the type. A restricted type is either REAL, INTEGER, BOOLEAN, HALF, LOGIC, FORMULA, SYMBOL, or SUBLIST.

For example: Suppose R is REAL, B is BOOLEAN, F is FORM, and A,B,S, and T are SYMBOL. Suppose further that the statement S ← [R,B,F, [A,B], T]; has been executed. Then

the relation 1ST OF < S > IS REAL is <u>true</u>
the relation 2ND OF < S > IS REAL is <u>false</u>
the relation 3RD OF < S > IS FORM is <u>true</u>
the relation 4TH OF < S > IS SUBLIST is <u>true</u>, and
the relation LAST OF < S > IS SYMBOL is <u>true</u>.

(c) Testing for membership in a class

Formula Algol permits sets to be defined by means of class definitions. For example, suppose the statement V ← [A,E,I,O,U] has been executed. Then the statement

LET (| VOWEL |) = [X | AMONG (X,V)];

defines the set of all vowels, (| VOWEL |), where AMONG (P,Q) is a Boolean Procedure which is true if P is an element of the list contained in Q, and false otherwise. Let us now suppose that, having sometime previously executed the statement S ← [A,B,C], we execute the statement

IF 1 ST OF < S > IN (| VOWEL |) THEN GO TO exit;

The relation 1 ST OF < S > IN (| VOWEL |) will be evaluated by first computing the value of the expression 1 ST OF < S >, which is the symbol A, and second by substituting A for the formal parameter X in the class definition of (| VOWEL |). This results in the Boolean procedure AMONG (A,V) being executed, the value of which is <u>true</u>. Thus A is <u>in</u> the class (| VOWEL |), and the relation 1 ST OF < S > IN (| VOWEL |) is <u>true</u>. This causes us to GO TO exit in the above statement.

Class definitions may consist of Boolean combinations of other defined classes. E.g. LET (|A|) = (|B|) $\wedge$ (|C|); is legal provided (|B|) and (|C|) are elsewhere defined. Another example of a class definition would be

LET (|EMPTY|) = [X | <u>false</u>];

This defines the empty set.

Class definitions may be used as constituent selectors in list patterns. When this is done, the constituent matching the class definition is tested for membership in the class. If the result is true the list pattern continues to be matched against the list being tested. If the result is false, the list pattern fails to match the list being tested. E.g.

< S > HAS D, (| VOWEL |), $ is a legal list pattern which tests the list contained in S to see if it is of the form D, followed by a vowel, followed by any arbitrary number of arbitrary constituents.

4. Other Operators and Statements

There are four species of statements which remain to be introduced. These are "value entry statements", "editing statements", "push down and pop up statements", and some additional new types of "for statements". We will discuss them in the order given.

(a) Value entry statements

Value entry statements enter values on description lists. They supplement the role performed by assignment statements in this regard. Suppose that S ← /[types: mu, pi, rho] [color: red] has been executed. Then if we execute the statement: THE color OF S IS green; we replace the value of the attribute "color" on the description list of S with the new value "green". This yields the altered description list /[types: mu, pi, rho] [color: green] as a result. On the other hand, we could have executed the statement: THE color OF S IS ALSO green. Instead of replacing the color "red" with the value "green" the latter statement appends the value "green" to the value list following the attribute "color". This yields /[types: mu, pi, rho] [color: red, green] as a result. Finally, we may use value entry statements to delete values from value lists of specific attributes. Executing the statement: THE types OF S IS NOT pi; alters the above description list to be of the form /[types: mu, rho] [color: green].

(b) Editing statements

Editing statements are used to transform, permute, alter, and delete elements in lists. Suppose S ← [X,A,A,X] has been executed. Then the statement INSERT Y BEFORE LAST OF < S > changes < S > to look like [X,A,A,Y,X]. Similarly, the statement INSERT [Y,Z] (AFTER 1ST OF, BEFORE LAST OF)< S > changes < S > to look like [X, [Y,Z], A,A, [Y,Z], X]. The statement DELETE 3RD BEFORE LAST OF < S > alters < S > to look like [A,A,X], and DELETE ALL A OF < S > causes < S > to be changed to [X,X]. In a similar vein, the statement ALTER ALL A OF < S > TO [C,C] changes < S > to look like [X, [C,C], [C,C], X]. Finally, the statement COPY FIRST 3 OF < S > AFTER LAST OF < T > would have the effect of appending [X,A,A] to the tail of the list given by < T >. These examples do not exhaust all possible syntactic combinations permissible in editing statements, rather, they exemplify some of the editing operations that are possible.

(c) Push down and Pop up Statements

The contents of any symbol may be regarded as a push down stack. If S ← [A,B,C] has been executed. Then ↓S is a statement, which when executed, pushes the entire contents of S down one level. Picturing the contents of S as a description list / [CONT: A,B,C], we may also picture the effect of ↓S, which is to insert a stack marker on top of the contents, transforming the above picture to / [CONT: |A,B,C]. If we now execute S ← [C,D], the latter picture transforms to / [CONT: C,D | A,B,C]. Here, the value of < S > is the list [C,D]. The lower levels of the push down stack are inaccessible to the operation of extracting contents until a pop up statement has been performed. Executing the statement ↑S then transforms

/ [CONT: C,D | A,B,C] into / [CONT: A,B,C], and now the value of < S > is [A,B,C]. Whenever the contents of S is empty, the expression < S > has the value NIL. Thus, should S be popped too many times, nothing will remain in the push down stack and the contents of S will be NIL. The push down operator, ↓, and the pop up operator, ↑, may be applied any number of times in succession, as in the examples ↑↑↑S, and ↓↓S. There is no limit to the number of levels a push down stack may have.

(d) Additional Types of For Statements

We may wish to generate the elements of a list one by one in order to assign them to the controlled variable in a for-statement. For this purpose the for-list elements, ATTRIBUTES OF S, and ELEMENTS OF < S > are introduced. Here attributes on the description list of S are generated in the order they occur by ATTRIBUTES OF S, and ELEMENTS OF < S > generates the successive elements of the list < S >. In the latter case < S > may be replaced with any list valued symbolic expression.

Parallel generation is also permissible. If S ← [A, B,C], T ← [D,E], and U ← [F,G,H,I] have been executed, then executing the statement

```
PARALLEL FOR (I,J,K) ← ELEMENTS OF (<S>, <T>, <U>) DO
          L ← [ <I>, <I>, <J>, <K> ];
```

causes the following to happen. First, all first elements of the lists contained in S, T, and U respectively are generated and placed in the contents of the controlled variables I, J, and K respectively. Control then passes to the statement following the DO, and when finished with the execution of the statement found there returns. On the second cycle, all second elements of S, T, and U are generated and placed in I, J, and K respectively. Control then passes to the statement following the DO and returns. On the third cycle all third elements are generated, on the fourth cycle all fourth elements are generated, and so on. If any list runs out of elements before any of its neighbors, NIL keeps getting generated as the Nth element of that list whenever N exceeds the number of elements on the list. The parallel generation stops on the first cycle before a NIL would be generated from all lists.

C. Remarks

It is legal to declare FORMULA arrays and SYMBOL arrays as well as FORMULA procedures and SYMBOL procedures. In the case of the first two, the array elements are of types FORM and SYMBOL respectively.

In the case of the last two, the procedures have FORM and SYMBOL values respectively.

There will be available to the programmer a number of standard list processing library functions such as COUNT(L), which counts the number of elements on the list L and gives the result as its value, EMPTY(L) which is true if <L> is empty and false otherwise, AMONG (X,L) which is true if X is an element of the list contained in L and false otherwise, CREATE(n,L) which inserts a list of n created internal symbols as the contents of L, these symbols being taken from the list of available space, ERADL(S) which erases the description list of S, and several others.

In addition to having values, formulae may have description lists attached. This is done by assignment statements. E.g.

F ← / [properties: continuous, differentiable].

As in the case of symbols, value retrieval statements and value entry statements may be used to alter and retrieve attributes and values from such description lists.

```
FORM PROCEDURE  DERV(G,X); FORM G,X;
COMMENT: THIS PROCEDURE COMPUTES THE DERIVATIVE OF G WITH RESPECT
TO X ITERATIVELY. ( IN REFERENCE[4] A RECURSIVE DIFFERENTIATION
ROUTINE WAS GIVEN). THE BASIC STRATEGY OF THE PROCEDURE IS THIS.
WE HAVE TWO FORMULA ARRAYS, F AND D.  F[I] CONTAINS A FORMULA
TO BE DIFFERENTIATED, AND D[I] WILL CONTAIN THE EXPRESSION OF
ITS DERIVATIVE. THIS DERIVATIVE IS CONSTRUCTED, IN GENERAL,
IN TERMS OF THE DERIVATIVES OF THE OPERANDS OF F[I]. THESE
OPERANDS WILL BE STORED IN SOME OTHER F[K] AND F[J] WITH K,J>I.
IN THE EXPRESSION OF D[I] WE DO NOT USE THE VALUES OF D[K]
AND D[J] RATHER WE USE THE ACCESSING FUNCTIONS (FORMULA ARRAY
EXPRESSIONS). ULTIMATELY THE PROCESS OF DIFFERENTIATION REDUCES
TO DIFFERENTIATION OF X AND OF CONSTANTS, THUS CAUSING IT TO TERMINATE.
WHEN THIS HAPPENS WE COLLECT BACKWARDS BY MEANS OF EVAL;

BEGIN SYMBOL T,ANY; INTEGER I,J,K; FORM ARRAY F,D[1:100];
SWITCH L ← L1,L2,L3,L4,L5;
T ← / [INDEX:K][OPERATOR:+,-,*,/,↑];
ANY ← [REAL,INTEGER,HALF,FORM];
F[1]←G;  J←1;
FOR I ← 1 STEP 1 UNTIL J DO
   BEGIN
      IF F[I] IS ATOM THEN D[I] ← IF F[I]=X THEN 1 ELSE 0 ELSE
         BEGIN
            IF F[I] == F[J+1]: OF(ANY) |T| F[J+2]:OF(ANY)  THEN
            GO TO L[K] ELSE GO TO ERROR;
         END;
      GO TO CONTINUE;
      L1: D[I] ← D.[J+1] + D.[J+2]; GO TO NEXT;
      L2: D[I] ← D.[J+1] - D.[J+2]; GO TO NEXT;
      L3: D[I] ← D.[J+1]*F[J+2]  +  D.[J+2]*F[J+1]; GO TO NEXT;
      L4: D[I] ← (F[J+2]*D.[J+1] - F[J+1]*D.[J+2])/F[J+2]↑2; GO TO NEXT;
      L5: D[I] ← F[J+2]*F[J+1]↑(F[J+2] - 1 )  + LN(F[J+1])*F[I]*D.[J+2];
      NEXT: J ← J + 2;
      CONTINUE:  ;
   END OF LOOP;
FOR I ← J STEP -1 UNTIL 1 DO
D[I] ← EVAL D[I];
DERV ← D[1];
END PROCEDURE;
```

FORMULA PROCEDURE TO COMPUTE THE GENERAL TAYLOR SERIES
EXPANSION OF A FORMULA WITH RESPECT TO
N VARIABLES

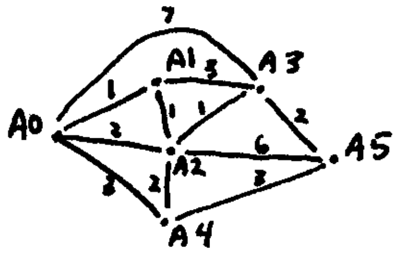
```
FORM PROCEDURE TAYLOR(F,U,V,M,N);
   FORM F; SYMBOL U,V; INTEGER M,N;
   COMMENT: LET  F BE THE FORMULA TO BE EXPANDED, LET U BE A SET OF INITIAL
VALUES, LET V BE A SET OF VARIABLES IN F: X1,X2,...,ETC., LET M
BE THE NUMBER OF TERMS DESIRED, AND LET N BE THE NUMBER OF VARIABLES
IN F.

   BEGIN SYMBOL W,S,R; FORM Z,G; FORM ARRAY T,H[1:N]; INTEGER I,J;
      FOR I ← 1 STEP 1 UNTIL N DO BEGIN
         INSERT  I TH OF V ← I TH OF U AFTER LAST OF <W>;
         INSERT H.[I] AFTER LAST OF <S>;
         INSERT T.[I] AFTER LAST OF <R>; END;
      Z← EVAL(<V>)F(<R>); FACT ← 1;
      TAYLOR ← EVAL(<R>)Z(<U>);
      FOR I ← 1 STEP 1 UNTIL M DO BEGIN
         G←0; FACT ← FACT*I ;END;
      FOR J ← 1 STEP 1 UNTIL N DO
        BEGIN   G ← G + H.[J]*DERV(Z,T.[J]);  Z←G; END;
      TAYLOR ← TAYLOR + EVAL(<S>,<R>)Z(<W>,<V>)/FACT;
END PROCEDURE;
```

PROGRAM TO COMPUTE THE PATH OF MINIMUM LENGTH
IN A CONNECTED GRAPH WITH EDGES OF POSITIVE LENGTH

```
BEGIN
COMMENT: THIS PROGRAM FINDS THE MINIMUM PATH AND PRINTS THIS PATH
TOGETHER WITH ITS LENGTH. THE GRAPH IS ENTERED AS A SET OF NODES
WITH DESCRIPTION LISTS ATTACHED TELLING THE OTHER NODES TO WHICH
EACH NODE IS IMMEDIATELY CONNECTED (ATTRIBUTES) AND TELLING THE
LENGTHS OF THE EDGES THAT FORM THOSE CONNECTIONS (VALUES). FURTHER,
AN INDEX IS ATTACHED TO EACH NODE WHICH IS ZERO FOR THE ORIGIN
AND A NUMBER GREATER THAN THE SUM OF ALL EDGE LENGTHS IN THE
GRAPH FOR THE OTHER NODES. AN ALGORITHM, GIVEN IN BERGE ''GRAPH
THEORY'' IS USED WHEREBY IF  INDEX[A] IS THE INDEX OF NODE A, IF
L[A,B] IS THE LENGTH OF THE EDGE FROM A TO B, AND IF
INDEX[A] + L[A,B] < INDEX[B]  THEN THE INDEX OF B IS REPLACED
BY THE NUMBER  INDEX[A] + L[A,B]  AND THIS PROCESS IS CONTINUED
UNTIL NO NODE HAS AN INDEX THAT CAN BE FURTHER DIMINISHED. THE
INDEX OF THE TERMINUS HAS THEN BEEN DECREASED MONOTONICALLY AND
THERE MUST HAVE BEEN A NODE LAST USED FOR THIS PURPOSE. THIS
IS THE SECOND TO LAST NODE IN THE MINIMUM PATH.  LIKEWISE, THE
INDEX OF THE SECOND TO LAST NODE HAS BEEN DECREASED MONOTONICALLY
AND THERE MUST HAVE BEEN SOME NODE LAST USED FOR THIS PURPOSE.
THIS IS THE THIRD TO LAST NODE IN THE MINIMUM PATH.  ITERATING WE
FIND THE MINIMUM PATH CONNECTING THE ORIGIN TO THE TERMINUS
[ FOR A PROOF SE E  BERGE ,OP.CIT.]. HERE WE TAKE AS AN EXAMPLE
THE FOLLOWING GRAPH WITH ORIGIN  A0  AND TERMINUS  A5;

RE
```



```
REAL N; SYMBOL A0,A1,A2,A3,A4,A5,INDEX,LASTNODE,NODELIST,MINPATH,S,T;
BOOLEAN ITERATE; LABEL AGAIN,ALPHA,BYPASS;

NODELIST ← [ A0,A1,A2,A3,A4,A5];
A0 ← / [ A1:1][A2:2][A3:7][A4:3][INDEX:0];
A1←  /  [ A0:1][A2:1][A3:3][INDEX:100];
A2 ← / [ A0:2][A1:1][A3:1][A4:2][A5:6][INDEX:100];
A3 ← / [ A1:1][A2:1][A5:2][INDEX:100];
A4 ← / [A0:3][A2:2][A5:3][INDEX:100];
A5 ← / [ A3:2][A2:6][A4:3][INDEX:100];
AGAIN:  ITERATE ← FALSE;
FOR  S ← ELEMENTS OF  <NODELIST>  DO
   FOR T ← ATTRIBUTES OF <S> DO
      BEGIN
         IF <T> = INDEX ∨ <T> = LASTNODE  THEN GO TO BYPASS;
         N ← THE INDEX OF <T>  + THE <T> OF <S> ;
         IF  N < THE INDEX OF <S>  THEN
            BEGIN
               ITERATE ← TRUE;  THE INDEX OF <S> IS N;
               THE LASTNODE OF <S> IS <T> ;
            END;
         BYPASS:  ;
      END FOR LOOP;
IF ITERATE = TRUE THEN GO TO AGAIN;
COMMENT: HERE WE PRINT THE RESULTS:
```

```
PRINT( <'THE LENGTH OF THE MINIMUM PATH IS'> );
N ← THE INDEX OF A5;
 NAME(A5); PRINT( < 2R,1D> );
MINPATH ← [ A0,A5 ] ;
X ← A5;
ALPHA:  IF THE LASTNODE OF <X> ¬= NIL THEN
  BEGIN     INSERT THE LASTNODE OF <X> 1ST BEFORE LAST OF <MINPATH>;
       X ← THE LASTNODE OF <X>; GO TO ALPHA;   END
PRINT ( <E>, <'THE MINPATH IS '> ) ;
NAME(MINPATH); PRINT( < E, 1LIST>) ;
END PROGRAM
```

PROGRAM TO TRANSLATE ARITHMETIC EXPRESSIONS IN
INFIX NOTATION INTO ARITHMETIC EXPRESSIONS IN POLISH PREFIX NOTATION

```
BEGIN
COMMENT: ASSUME THAT WE ARE GIVEN A CLASS OF EXPRESSIONS DEFINED BY
THE FOLLOWING BACKUS NORMAL FORM SYNTAX EQUATIONS.

   <ARITHMETIC EXPRESSION> ::= <TERM> | <+-> < TERM> | <ARITHMETIC EXPRESSION>
                <+-> < TERM >
   < TERM > ::= <FACTOR> | <TERM> < */ > <FACTOR >
   < FACTOR > ::= <PRIMARY > | <FACTOR> ↑ <PRIMARY>
   < PRIMARY > ::= < IDENTIFIER > | (  <ARITHMETIC EXPRESSION> )

WE ARE TO TRANSLATE THIS CLASS OF EXPRESSIONS INTO CORRESPONDING INFIX
EXPRESSIONS IN THE CLASS DEFINED BY THE FOLLOWING BACKUS NORMAL FORM
EQUATIONS.

   < PREFIX A E > ::= < TERM > | NEG < TERM > | <+-> <PREFIX A E > < TERM >
   < TERM > ::= < FACTOR > | < */ > < TERM > < FACTOR >
   < FACTOR > ::= < PRIMARY > | ↑ < FACTOR > < PRIMARY >
   < PRIMARY > ::= < IDENTIFIER > | < PREFIX A E >

HERE  NEG  IS A UNARY NEGATION OPERATOR. WE SEE THAT A  UNARY MINUS
PRECEDING A TERM IN AN INFIX EXPRESSION IS PERMITTED AND MUST BE
TRANSLATED INTO THE PREFIX UNARY OPERATOR NEG FOLLOWED BY THE TERM.
THUS THE EXPRESSION  -(A+B)*C↑2 + E*D WOULD BE TRANSLATED AS
+ NEG * + A B ↑ C 2 * E D. THE PROGRAM FIRST READS A SOURCE STRING
TO BE TRANSLATED AND STORES THIS EXPRESSION ITEM BY ITEM IN A LIST
FOUND AS THE CONTENTS OF THE SYMBOL  ''INPUT''. A SIMULATION OF A
FLOYD-EVANS PRODUCTION TRANSLATOR  IS THEN USED TO ACCOMPLISH THE
TRANSLATION;

SYMBOL S,STACK,INPUT,X,L1,L2,L3,U,V,W,Y,PM,TD,IDENT,LFTPAREN,P,F,T,AE;
LABEL AE0,AE1,P1,F1,T1;
PROCEDURE  REVERSE; SYMBOL Q;
   BEGIN Q ← < STACK > ; ↑ STACK; INSERT <Q> AFTER LAST OF <STACK> END;
PROCEDURE SCAN;
COMMENT: THIS PROCEDURE SCANS THE NEXT CHARACTER FROM THE LIST STORED IN THE
CONTENTS OF THE SYMBOL ''INPUT'' AND STACKS IT LAST IN THE WORKSPACE S;
   BEGIN INSERT 1ST OF <INPUT> AFTER LAST OF <S>; DELETE 1ST OF <INPUT>;END;
PROCEDURE APPENDOPERATOR;
   BEGIN INSERT <X>  BEFORE 1ST OF <STACK>; END;
COMMENT: CLASS DEFINITIONS;
   L1 ← [ +,- ];
   LET (|PM|) = [ U| AMONG(U,L1) ];
   L2 ← [ *,/ ];
   LET (|TD|) = [ V| AMONG(V,L2) ];
   LET (|IDENT|) = [ W | <W> IS SYMBOL ];
   LET (|LFTPAREN|) = [ Y| <Y> = '(' ];
COMMENT: HERE WE READ AND STORE THE EXPRESSION TO BE TRANSLATED INTO THE
CONTENTS OF THE SYMBOL INPUT. THE CONTENTS OF INPUT ARE READ OFF CONSECUTIVELY
ELEMENT BY ELEMENT EACH TIME THE PROCEDURE SCAN IS CALLED DURING THE
PROGRAM;
NAME(INPUT);  READ( < E > , < 1 LIST > ) ;
SCAN; COMMENT: INITIALLY WE START THE TRANSLATION BY SCANNING THE FIRST
   CHARACTER IN THE INPUT STRING;
GO TO AE0;

COMMENT: ENTER THE TRANSLATOR;
AE0: IF LAST OF <S> IN (|PM|) THEN SCAN AND GO TO AE0;
     IF LAST OF <S> IN (|IDENT|) THEN BEGIN
          ↓STACK; STACK ← LAST OF <S>;
          DELETE LAST OF <S>; INSERT P AFTER LAST OF <S>;
          SCAN; GO TO AE0; END;
     IF LAST OF <S> IN (|LFTPAREN|) THEN SCAN AND GO TO AE0 ELSE GO TO ERROR;
P1:  IF LAST 4 OF <S> HAS  F,X:↑,P,$1 THEN BEGIN
          REVERSE; APPENDOPERATOR;
          DELETE BETWEEN 3RD BEFORE LAST AND LAST OF <S>; GO TO F1;END;
     IF LAST 2 OF <S> HAS  P,$1 THEN DELETE 1ST BEFORE LAST OF <S> AND
          INSERT F 1ST BEFORE LAST OF <S> AND GO TO F1 ELSE GO TO ERROR;
F1:  IF LAST 2  OF <S> HAS F,↑ THEN SCAN AND GO TO AE0;
     IF LAST 4 OF <S> HAS T,X:(|TD|),F,$1 THEN BEGIN
          REVERSE;APPENDOPERATOR; DELETE BETWEEN 3RD BEFORE LAST
          AND LAST OF <S>; GO TO T1; END;
     IF LAST 2 OF <S> HAS F,$1 THEN DELETE 1ST BEFORE LAST OF <S>
          AND INSERT T 1ST BEFORE LAST OF <S> AND GO TO T1 ELSE GO TO ERROR;
T1:  IF LAST 2 OF <S> HAS T,(|TD|) THEN SCAN AND GO TO AE0;
     IF LAST 4 OF <S> HAS AE,X:(|PM|),T,$1 THEN BEGIN
          REVERSE; APPENDOPERATOR;  DELETE BETWEEN 3RD BEFORE LAST AND
          LAST OF <S>; GO TO AE1; END;
     IF LAST 3 OF <S> HAS  X:(|PM|),T,$1 THEN BEGIN
          IF <X> = '-' THEN  X ← NEG; APPENDOPERATOR;
          DELETE BETWEEN 3RD BEFORE LAST AND LAST OF <S>;
          INSERT  AE 1ST BEFORE LAST OF<S>; GO TO AE1; END;
     IF LAST 2 OF <S> HAS T,$1, THEN DELETE 1ST BEFORE LAST IN <S>
          AND INSERT  AE 1ST BEFORE LAST IN <S> AND GO TO AE1 ELSE GO TO ERROR;
AE1: IF LAST 3 OF <S> HAS (,AE,) THEN DELETE LAST 3 OF <S> AND
           AND INSERT P 1ST AFTER LAST OF <S> AND SCAN AND GO TO P1;
     IF LAST 2 OF <S> HAS AE,(|PM|) THEN SCAN AND GO TO AE0;
      IF LAST 2 OF <S> HAS AE,$1 THEN GO TO EXIT ELSE GO TO ERROR

ERROR: PRINT(<'ERRONEOUS SOURCE STRING',E>);
EXIT: NAME(<STACK>); PRINT(<E>,<1LIST>,<E>);
END OF PROGRAM;

- ( A + B ) * C ↑ 2 + E * D  ;    COMMENT: ← INPUT TEXT;

EXPECTED OUTPUT IS   + NEG * + A B ↑ C 2 * E D  .
```

NOTE: THE WORD ''AND'' HAS BEEN USED IN PLACE OF ; IN THE ABOVE TEXT TO CONCATENATE SEVERAL STATEMENTS INTO A COMPOUND STATEMENT WITHOUT THE USE OF THE BRACKETS BEGIN--END.

THE WANG ALGORITHM

```
BEGIN
   COMMENT: THIS ALGORITHM OF HAO WANG [CF.IBM JOURNAL,JAN '60,PP 2-22]
DETERMINES THE VALIDITY OF WELL FORMED FORMULAS OF PROPOSITIONAL
CALCULUS. THE FORMULA TO BE PROVED OR DISPROVED ENTERS AS A LIST
CONSISTING OF THE MARK →, FOLLOWED BY THE FORMULA (EXPRESSED IN
FORMULA ALGOL AS A BOOLEAN FORMULA), FOLLOWED BY AN OCCURRENCE OF
THE SYMBOL NIL.  THIS LIST IS STORED AS THE CONTENTS OF THE
SYMBOL SEQUENT. WE SEARCH FOR THE FIRST LOGICAL CONNECTIVE IN THE
SEQUENT.  IF THERE IS NONE WE TRANSFER TO RULE0 WHICH DETERMINES
THE VALIDITY OF THE FORMULA ACCORDING TO A TERMINAL RULE GIVEN BY
WANG. IF THERE WAS A LOGICAL CONNECTIVE, THEN, HAVING NOTED THE POSITION
OF THE ARROW → (ALL THAT IS NECESSARY IS TO KNOW WHETHER THE ARROW
CAME BEFORE THE FIRST LOGICAL CONNECTIVE), WE TRANSFER TO THE
APPROPRIATE RULE TO TRANSFORM THE SEQUENT IN ORDER TO ELIMINATE
THE LOGICAL CONNECTIVE.  IF THE RULE SELECTED PRODUCES TWO SEQUENTS
AS A RESULT OF ELIMINATING THE CONNECTIVE, ONE OF THEM IS ENTERED
INTO THE SECOND LEVEL OF THE PUSH DOWN STACK CONTAINED IN THE
SYMBOL SEQUENT. THE OTHER IS PLACED ON THE TOP LEVEL AND IS
PROCESSED NORMALLY. AFTER THE TOP LEVEL HAS BEEN PROCESSED THE LOWER
LEVELS ARE PROCESSED PROVIDED THE PROCESS HAS NOT BEEN HALTED BY
THE DISCOVERY OF AN INVALID SEQUENT.  THE ALGORITHM STOPS EITHER
WHEN AN INVALID SEQUENT HAS BEEN FOUND OR WHEN , ALL SEQUENTS HAVING
BEEN SHOWN VALID, THE PUSH DOWN STACK IS EMPTY.

FORMULA A,B,F; SYMBOL SEQUENT T,P,Q,I; INTEGER N,ARROWPOSITION;BOOLEAN VALID;
LABEL ITERATE,AGAIN,RULE0,RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULE7,RULE8,
RULE9,RULE10,TEST1,HALT;  SWITCH SWITCH1←RULE1,RULE2,RULE3,RULE4,RULE5,
RULE6,RULE7,RULE8,RULE9,RULE10;

PROCEDURE  EXTRACTMAINOP(G,S); FORMULA G; SYMBOL S;
   COMMENT: THIS PROCEDURE FINDS THE MAIN OPERATOR OF THE BOOLEAN
FORMULA G AND STORES THIS OPERATOR IN THE CONTENTS OF THE SYMBOL S.
IT ALSO EXTRACTS THE LEFT AND RIGHT HAND SUBEXPRESSIONS, IF ANY,
AND STORES THEM IN A AND B RESPECTIVELY;
   BEGIN
      IF  G == ATOM THEN  S ← NIL  ELSE
      IF  G ==  ¬ A:OF(FORM) THEN  S ← '¬' ELSE
      IF  G ==  A:OF(FORM) ∧ B: OF(FORM) THEN S ← '∧' ELSE
      IF  G ==  A:OF(FORM) ∨ B: OF(FORM) THEN S ← '∨' ELSE
      IF  G ==  A:OF(FORM) EQL B:OF(FORM) THEN S ← 'EQL' ELSE
      IF  G ==  A:OF(FORM) IMP B:OF(FORM) THEN S ← 'IMP' ELSE
      PRINT( <'ERROR MALFORMED EXPRESSION',E>);
   END PROCEDURE;

INTEGER PROCEDURE WHICHRULE; SYMBOL L; INTEGER K; LABEL ALPHA;
   BEGIN
      COMMENT: THIS PROCEDURE DETERMINES AN INTEGER TELLING WHICH
         RULE TO TRANSFER TO DEPENDING ON THE LOGICAL CONNECTIVE GIVEN
         AND WHETHER IT OCCURRED BEFORE OR AFTER THE ARROW →;
      L ← [ ¬,∧,∨,IMP,EQL];
      FOR K ← 1 STEP 1 UNTIL 5 DO
         IF K TH OF <L> = <S> THEN GO TO ALPHA;
      ALPHA:  WHICHRULE ← 2 * K - ARROWPOSITION;
   END PROCEDURE;
```

```
BEGIN
<SEQUENT> ← [ < SEQUENT>,NIL];
ITERATE: N←1; ARROWPOSITION ← 0; NAME(SEQUENT);PRINT(<1LIST,E>);
AGAIN:  IF N TH OF <SEQUENT> = '→' THEN BEGIN ARROWPOSITION ← 1;
        N←N+1; GO TO AGAIN; END ELSE
        IF N TH OF <SEQUENT> = NIL THEN GO TO RULE0 ELSE
        F ← N TH OF <SEQUENT>; EXTRACTMAINOP(F,T);
        IF <T> = NIL THEN BEGIN N←N+1; GO TO AGAIN; END ELSE
        GO TO SWITCH1(WHICHRULE);

RULE1: DELETE N TH OF < SEQUENT>; INSERT A BEFORE 1ST OF <SEQUENT>;
       GO TO ITERATE;
RULE2: DELETE N TH OF <SEQUENT>; INSERT A AFTER LAST OF <SEQUENT>;
       GO TO ITERATE;
RULE3: DELETE N TH OF <SEQUENT>; Q←<SEQUENT>; INSERT A AFTER (N-1)ST
       OF <SEQUENT>; ↓ SEQUENT; SEQUENT ← <Q>; INSERT B AFTER (N-1)ST
       OF < SEQUENT>; GO TO ITERATE;
RULE4: DELETE N TH OF <SEQUENT>; INSERT A AFTER (N-1)ST OF <SEQUENT>;
       INSERT B AFTER N TH OF <SEQUENT>; GO TO ITERATE;
RULE5: COMMENT: SAME AS RULE 4; GO TO RULE4;
RULE6: COMMENT: SAME AS RULE 3; GO TO RULE3;
RULE7: DELETE N TH OF <SEQUENT>; INSERT A 1ST BEFORE 1ST '→' OF <SEQUENT>;
       INSERT B AFTER N TH OF <SEQUENT>; GO TO ITERATE;
RULE8: DELETE N TH OF <SEQUENT>; Q ← <SEQUENT>; INSERT A AFTER LAST OF <SEQUENT>;
       ↓ SEQUENT; SEQUENT←<Q>; INSERT B AFTER (N-1)ST OF <SEQUENT>;
       GO TO ITERATE;
RULE9: DELETE NTH OF <SEQUENT>; Q←<SEQUENT>; INSERT B BEFORE 1ST OF <SEQUENT>;
       INSERT A AFTER NTH OF <SEQUENT>; ↓SEQUENT; SEQUENT←<Q>;
       INSERT A BEFORE 1ST OF <SEQUENT>; INSERT B AFTER N TH OF <SEQUENT>;
       GO TO ITERATE;
RULE10:DELETE N TH OF <SEQUENT>; Q← <SEQUENT>; INSERT A,B AFTER LAST OF
       <SEQUENT>; ↓SEQUENT; SEQUENT←<Q>; INSERT A,B BEFORE 1ST OF <SEQUENT>;
       GO TO ITERATE;

COMMENT: RULE0 CHECKS TO SEE IF SOME ATOM ON THE RIGHT IS ALSO ON THE LEFT
IN THE SEQUENT;

RULE0: IF <SEQUENT> HAS P:$,→,Q:$ THEN GO TO TEST1 ELSE PRINT(<'ERROR
          MALFORMED SEQUENT',E>);
TEST1: VALID ← FALSE;
         FOR I ← ELEMENTS OF <P> DO
            IF AMONG(<I>,Q) THEN VALID ← TRUE;
         NAME(SEQUENT); PRINT(<1LIST,E>);
         IF VALID = FALSE THEN BEGIN PRINT(<'THEOREM NOT VALID',E>);
            GO TO HALT;END;
         ↑SEQUENT; IF ¬ EMPTY(SEQUENT) THEN GO TO ITERATE ELSE
         PRINT(<'THEOREM VALID',E>);
HALT:     ;
END PROGRAM;
```

IV. THE BACKUS NORMAL FORM SYNTAX FOR FORMULA ALGOL

Add to the Revised Algol Report:

```
<formula expression> ::= <arithmetic expression>|<Boolean expression>|
          <an arithmetic expression (Boolean expression) in which some
          of the primaries (Boolean primaries) have been replaced
          by "procedure formula", "array formula", or "eval formula">|
          <substitution formula>
<conditional formula> ::= .IF <formula expression> THEN <expression>
          ELSE <expression>
<procedure formula> ::= <function designator>.<actual parameter part>
<assignment formula> ::= . ← <expression>
<array formula> ::= <identifier> . [<subscript list>]
<eval formula> ::= EVAL␣<variable> | EVAL <bound variables>
          <expression> <list of values>
<list of values> ::= (<actual parameter list>) | ([<variable>])
<bound variables> ::= (<variable list>) | ([<variable>])
<variable list> ::= <variable> | <variable list> , <variable>
<expression> ::= <arithmetic expression> | <Boolean expression> |
          <designational expression> | <formula expression> |
          <pattern expression> | <symbolic expression> |
          <tree expression>
<unlabelled pattern primary> ::= <type> | OF (<variable>) | ATOM
```

```
<pattern primary> ::= <unlabelled pattern primary> |
          <variable> : <unlabelled pattern primary>
<pattern structure> ::= <a formula expression in which some of the
          primaries may have been replaced by pattern primaries and
          some of the operators may have been replaced by
          |<variable>|>
<pattern> ::= <formula expression> == <pattern structure>|
          <identifier> :: <formula expression>  >>
          <identifier> :: <pattern expression> |
          <formula expression>  >>  <pattern structure>
<variable> ::= <simple variable> | <subscript variable> |
          . <identifier>
```

```
<symbolic expression> ::= <variable> | <function designator>|

          <selection expression> | <value retrieval expression>|

          '<'  <symbolic expression> '>'

<list valued symbolic expression> ::= <symbolic expression>

          having a list of symbols as value.
```

In the following syntax equations the syntactic class <symbolic expression> refers to that subclass of symbolic expressions, as defined above, which have single values. The syntactic class <list valued symbolic expression> refers to that class of symbolic expressions, as defined above, which have lists of symbols as values.

```
<symbolic expression list> ::= <symbolic expression> |

          <symbolic expression list> , <symbolic expression>

<tree expression> ::= [ <tree expression list> ]

<tree expression list element> ::= <expression> |

          [<tree expression list element>] <description list> |

          <symbolic expression> <description list> |

<tree expression list> ::= <tree expression list> ,

          <tree expression list element> |

          <tree expression list element>

<description list> ::= / <attribute value list>

<attribute value list> ::= <attribute value segment> |

          <attribute value list> <attribute value segment>

<attribute value segment> ::=

          [<symbolic expression> : <expression list> ]

<value retrieval expression> ::= THE <symbolic expression>

          OF <symbolic expression>
```

```
<value entry statement> ::= THE <symbolic expression> OF
         <symbolic expression> <is phrase> <expression>
<is phrase> ::= IS | IS NOT | IS ALSO
<list pattern> ::= <list valued symbolic expression> = =
         [<constituent selector list>] |
         <list valued symbolic expression> = =
         <list valued symbolic expression> | <symbolic expression> = =
         <symbolic expression>
<constituent selector list> ::= <constituent selector> |
         <constituent selector list> , <constituent selector>
<constituent selector> ::= $ | $ <unsigned integer> |
         <class name> | <symbolic expression> | <augmented type>
         <list valued symbolic expression> |
         <symbolic expression> <description list> |
         <label> : <constituent selector>
<class name> ::= (| <symbolic expression> |)
<class primary> ::= <class name> | [<class expression>]
<class secondary> ::= <class primary> | ¬ <class primary>
<class factor> ::= <class secondary> | <class factor> ∧
         <class secondary>
<class expression> ::= <class factor> | <class expression> ∨
         <class factor>
<class definition> ::= LET <class name> = [<formal parameter> '|'
         <Boolean expression> ] | LET <class name> =
         <class expression>
```

```
<assignment statement> ::= <symbolic expression> ←<description list>|
          <symbolic expression> ←<expression>|
          <symbolic expression> ←<tree expression list>
<for list element> ::= ... | <symbolic expression>|
          ELEMENTS OF <list valued symbolic expression> |
          ATTRIBUTES OF <symbolic expression>
<for clause> ::= ...| FOR <symbolic expression> ←<for list> DO |
          PARALLEL FOR [<formal parameter list>] ←
          ELEMENTS OF [<symbolic expression list>] DO |
          PARALLEL FOR <list valued symbolic expression> ←
          ELEMENTS OF <list valued symbolic expression> DO
<unlabelled basic statement> ::= ... | <class definition> |
          <value entry statement> | <push down statement> |
          <pop up statement> | <editing statement>
<push down operator> ::= ↓ | <push down operator> ↓
<pop up operator> ::= ↑ | <pop up operator> ↑
<push down statement> ::= <push down operator> <symbolic expression>
<pop up statement> ::= <pop up operator> <symbolic expression>
<relation> ::= ... | <symbolic expression> = = <class name> |
          <symbolic expression> = <symbolic expression> |
          <list valued symbolic expression> =
          <list valued symbolic expression> | <list pattern>
<augmented type> ::= REAL | INTEGER | BOOLEAN | FORMULA
          SYMBOL | SUBLIST | LOGIC | HALF | TEXT | ATOM |
<selection expression> ::= <selector> OF
          <list valued symbolic expression>
<ordinal suffix> ::= ST | ND | RD | TH
<ordinal selector> ::= <arithmetic primary>␣<ordinal suffix> |
          FIRST | LAST
<elementary position> ::= <ordinal selector> | <ordinal selector>
          <class name> | <ordinal selector> <expression> |
          <ordinal selector> <augmented type>
<position> ::= <elementary position> | <arithmetic primary>␣
          <ordinal suffix> BEFORE <elementary position> |
          <arithmetic primary>␣<ordinal suffix> AFTER
          <elementary position>
<selector> ::= BETWEEN <position> AND <position> |
          ALL AFTER <position> | ALL BEFORE <position> |
          FIRST <unsigned integer> | LAST <unsigned integer> |
          <position> | ALL <expression> | ALL <augmented type> |
          ALL <class name> |
<insertion locator> ::= BEFORE <position> OF |
          AFTER <position> OF
<insertion locator list> ::= <insertion locator> |
          <insertion locator list> , <insertion locator>
<insertion locator part> ::= <insertion locator> |
          ( <insertion locator list> )
```

```
<selector list> ::= <selector> | <selector list> , <selector>
<selector part> ::= <selector> | ( <selector list> )
<editing statement> ::= INSERT <tree expression list>
          <insertion locator part> <list valued symbolic expression>|
          DELETE <selector part> OF <list valued symbolic expression>|
          ALTER <selector part> OF <list valued symbolic expression>
          TO <expression> |  DELETE <symbolic expression>
<expression> ::= ... | <arithmetic expression> | <Boolean expression>|
          <designational expression> | <formula expression>|
          <symbolic expression> | <tree expression>|
          <pattern expression>
<expression list> ::= <expression> | <expression list> , <expression>
```

REFERENCES

1. Gelernter, H., Hansen, R.R., Gerberich, C.L.: "A Fortran compiler List processing Language", Journal ACM, 7, 87 - 101 (April 1960).

2. Bond, E., et al: "FORMAC, An Experimental Formula Manipulation Compiler", Proc 19th National Conference ACM 1964, K2-1.

3. Brown, W.S., "The ALPAK System for Non-numerical Algebra on a Digital Computer", Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, internal publication.

4. Perlis, A.J., and Iturriaga, R.: "An Extension to Algol for Manipulating Formulae", Communications of the ACM, 7, 127 - 130, (February 1964).

5. Christensen, Carlos: "AMBIT, A Programming Language for Algebraic Symbol Manipulation", Paper CA-64-4-R, Computer Associates, Inc., Wakefield, Massachusetts.

<u>MTHAT ASSEMBLER FOR THE CDC G-21</u>

Daniel Ross

(Computation Center, Carnegie Institute of Technology)

This manual is a reference guide for MTHAT. As reference material, the various features of MTHAT are organized by their functions. An alphabetical listing of sudos is also included.

This manual is designed to be used in coordination with:

<u>Bendix G-20 Central Processor Machine Language Manual</u>, available through Control Data Corporation.

The following papers are also needed as references for the Assembly Language Code on the Carnegie Tech, CDC G-21. The papers may be obtained by writing to the Documentation Office, PH118-P, Department of Computer Sciences, Carnegie Institute of Technology, Pittsburgh, Pa., 15217.

"Specifications for the Use of Routines in the Carnegie Tech Monitor 'THEM THINGS'", code COO-42.1 (The 'THEM THINGS' write-up is also included in the 1965 User Manual, page 291.)

"Monitor References for Staff Members", code CID-47

"Execute OPCODE" (December 11, 1963)

"Scatter Repeat and Indirect Block-Length Addressing" (January 15, 1964)

"Special Registers" (March 26, 1964)

"CC-11" (March 30, 1964)

The research reported here was supported by the Advanced Research Projects Agency of the Department of Defense under the Contract SD-146 to the Carnegie Institute of Technology.

MTHAT Assembler for the CDC G-21

CONTENTS

MTHAT

1. INTRODUCTION

MTHAT is a one-pass symbolic assembler for the CDC (formerly Bendix) G-21 computer. It is designed to be used in conjunction with the Carnegie Tech G-21 monitor system.

The input to MTHAT is a set of punched cards, or the images of punched cards obtained from either the G-21 control console or the remote teletype units. The outputs are G-21 machine code in the computer memory, usually one word of code for each card input, and a printed assembly listing. There also are provisions for communication between MTHAT and an operator or programmer at the G-21 control console.

MTHAT is called a one-pass assembler because usually each input card is processed once only.

2. INPUT CARDS

The input cards or card images to MTHAT may be classified into four categories.

1. Cards to be listed only. The images of these cards are printed on the assembly listing, but the cards are otherwise ignored by MTHAT. For more details, see the LIS sudo.
2. G-21 instruction words. MTHAT translates the card images into the proper machine code, and stores the code in the computer memory for execution after the MTHAT assembly has been completed.
3. MTHAT pseudo-instructions, called "sudos." These sudos are executed immediately by the MTHAT system. However, some of them result in the storage of G-21 machine code, for use after the MTHAT assembly has been completed.
4. MTHAT macro-instructions. The appearance of a single macro card may result in the assembly of several other cards. The choice of which other cards are assembled is determined by the user when he writes the macro declaration. More details appear in the discussion of macros.

3. CARD FORMAT

The columns on the input cards are grouped into fields, which contain specific types of information. The most important fields are as follows:

Columns 1 to 2: Language field. This field is ignored by MTHAT, but is used by the Carnegie Tech G-21 monitor system.

Columns 4 to 12: Label field. If the name of an undefined region appears in this field, then the region is assigned the value of the Current Location Counter, A. If the region is given a nonzero subscript, then the regional base is assigned the value: A - the value of the subscript. If the name of an undefined label appears in this field, then the label is assigned the value of A. If a defined value (such as a constant, or the name of a defined region or label, or an expression) appears in this field, then a check is made to verify that the defined value equals the value of A. A blank label field is ignored.

Column 13: Flag field. Usually blank. When nonblank, this column usually contains the digits 0, 1, 2, or 3. Other characters may appear in this column for various special purposes.

Columns 15 to 17: Opcode field. This field contains the three characters of a G-21 instruction, an MTHAT sudo, or an MTHAT macro. If this field is blank, the card is treated as a comment card (See COM sudo, Section 7).

Column 20: Mode field. This column usually is blank or contains the digits 0, 1, 2, or 3. Other characters may appear in this column for various special purposes.

Columns 24 to 67: Parameters field. The parameters appropriate to the opcode appear here. The parameters are separated by commas, and terminated by a semicolon. Everything to the right of the semicolon is treated as comments. If a semicolon is used to terminate the parameters before the specified number of parameters have been supplied, then the remaining parameters are treated as blanks. Blank parameters are interpreted as zeroes, unless specified otherwise.

All other columns: Immaterial

Some opcodes do not use all the fields listed above. Also, some opcodes use other fields which are not listed above. All opcodes use the label and opcode fields. Those opcodes which require parameters use the parameters field, unless specified otherwise. The other fields are used only where specified.

MTHAT processes cards by scanning first the label field, then the opcode field, then the other fields as appropriate. Thus the label field is processed even on cards which do not cause the assembly of G-21 code, such as COM sudo cards.

4. INPUT PARAMETERS

Unless specified otherwise, parameters are expressions consisting of constants and variables, and operators on the constants and variables. The types of constants are as follows:

1. Blank. Blank parameters are interpreted as zeroes, unless specified otherwise.
2. Decimal integer. One or more decimal digits.
3. Octal integer. A slash, /, followed by one or more octal digits.
4. Power of 2. A dollar sign, $, followed by a decimal integer. The allowable range is from $0=1 to $31=/20000000000.
5. The numeric value of alphanumeric characters. A greater than, >, followed by 4 characters in the next 4 columns of the card. The internal representations of the characters (See Section 29) are concatenated in the standard 4-character-per-word, 8-bit format. The value of the constant is taken to be the value of the resulting 32-bit integer. Blank characters in the specified 4 columns are significant. If the 4 columns would extend past the end of the field on the card, the trailing characters are treated as blanks.
6. The contents at assembly time of a specified location. An expression whose value equals the desired address is surrounded by parentheses or by square brackets. If surrounded by parentheses, a numeric access is made of the named location. If surrounded by brackets, a logic access is made of the named location. It is not possible to nest sets of parentheses or brackets within each other.

The types of variables are as follows:

7. Region. Described in Section 5.

8. Label. Described in Section 5.

9. Free name. Described in Section 5.

10. Greater variable. There are six greater variables: A>, B>, C>, D>, E>, and F>. They are used primarily as formal parameters for macros, although they may be used elsewhere if desired. They are described in Section 17.

The operators used in expressions are as follows:

+	add
-	subtract
*	multiply
:	divide.... by convention, 0:0=0
$\wedge$	Boolean bit intersection

The character $\vee$ has a special use in both constants and variables, as described in Section 19. Blank spaces between nonblank characters forming a constant or variable are ignored completely, except in item 5 above.

Expressions are evaluated strictly from left to right, with no heirarchy among the operators. Thus 1+2*3=9, and 5: -7 produces an error because 5 is divided by 0.

Expression evaluation is performed in double precision floating point arithmetic until a final value is obtained. The type of storage used for the final value depends upon the specific use of the parameter. If an integer is required, the floating point number is converted to an integer by truncation.

All the variables in an expression must have defined values at the time the expression is evaluated, unless specified otherwise.

Some sudos have listable parameters, or listable sets of parameters. Sudos with listable sets of parameters allow the parameters to be repeated several times on a single card, separated by commas. The action of the sudo is performed once for each set of parameters.

5. REGIONS, LABELS, AND FREE NAMES

A region is a sequential area within the computer memory. The base of the region is the address of the first word in the region. If, for example, B is the name of a region, the B15=B0+15.

A label is a name given to some particular location or value. The values of adjacent labels are not necessarily related. For example, if L0 to L20 are labels, it is possible for L12=5 and L13=200.

The syntax of regions and labels is the same for both: a letter or one of the other characters listed below (called "identifiers") followed by a decimal integer (called the "subscript"). If no subscript follows, then a subscript of 0 is assumed. The possible identifiers are:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z | ← → ¬ ≠ < ↓ ↑ ' .

Regions and labels are distinguished from each other by their first use in a program. If an identifier first appears in the label field of a card (See Section 3) or in a DEF sudo (See Section 9), then the identifier names a region. If the identifier first appears in a LBL sudo (See Section 9), then the identifier is used to name labels.

The identifier A is predefined by the MTHAT system to be a region, whose initial value is /30000. A is used as the Current Location Counter for MTHAT; the value of A automatically is incremented by 1 for each word assembled.

Labels |0 to |100 are predefined by MTHAT. Except for |36 and |37, their values are the locations of subroutines and tables in the monitor. Labels '0 to '24 and |36 and |37 are predefined as the locations of subroutines and tables in MTHAT, or other references within MTHAT. A complete list is given in Section 33.

In some circumstances a label may be used whose value has not yet been defined at the time the card is processed. The use of undefined labels is described in detail where it applies. Regions must always be defined before they are used. However, the use of regions is discouraged. No concordance is available for regions (See Section 15), and everything that can be done with regions can be done just as well with labels. For example, the regional reference D10 could just as well be written D0+10, where D0 is a label.

Any combination of identifiers, digits, and the characters / $ ∨ which do not fit the syntax of constants or regions or labels may be used as a free name. The first 8 nonblank characters of a free name are significant; the remaining characters (if any) are ignored. Free names are put into a table (See Section 16) in the order in which they are encountered. The position of a free name in the table determines a subscript for the "period" identifier. For example, the 5th free name encountered is processed in exactly the same manner as though .5 had been written instead. If .0 to .200 have been declared as labels by the LBL sudo, and if the LFN sudo has been used, then 200 free names may be used instead of labels.

Regions and labels may be defined to have integer values between 0 and /177777 inclusive.

6. G-21 INSTRUCTIONS

When a G-21 instruction is assembled, the opcode field of the card (See Section 3) may contain either the 3 characters of the opcode mnemonic (See Section 28) or the octal integer value of the opcode, with the slash (indicating an octal constant) omitted. For each G-21 instruction, MTHAT has defined a standard mode (See Section 28). If the opcode field contains a mnemonic and the mode field is blank, then the standard mode is assembled. Any punching in the mode field overrides the use of the standard mode. If the opcode field contains an octal integer, the standard mode always is 0.

There are two parameters to an instruction. The first is the address, whose value must be between 0 and /177777 inclusive. A single undefined label may be used for the address, if desired. When the value of the label is later defined, that value automatically will be stored in the instruction word that had the undefined label. No operators or constants or other variables may be used in the expression if an undefined label is used. The second parameter is the index, whose value must be between 0 and /77 inclusive. All variables used in the index expression must be defined when the card is processed.

The flag field is processed, but it is an error for a G-21 instruction to have a flag 1 or flag 3. A blank flag field is treated as flag 0.

If anything is punched in columns 22 or 23 of the card, then the G-21 internal representation of the characters in these two columns (See Section 29) is used as the address for the instruction word, and the index is set =0. The parameters field of the card, where address and index usually are found, is ignored.

The G-21 instructions which use hardware registers or line commands are processed by MTHAT as special sudos, so that register mnemonics and line command mnemonics may be used. These instructions normally are of use only when programming a monitor system for the G-21.

The sudo ADC is processed like an OCA instruction, with standard mode =0.

7. CONTROLLING THE ASSEMBLY LISTING

The following sudos determine what is printed on the assembly listing, but do not affect the assembled code.

- - -

COM sudo - comment

no parameters

The contents of the card, except for the label field, are treated as comments only.

- - -

TOP sudo - type or print

2 parameters, nonlistable

If the first parameter =0, subsequent assembly continues with all printing of card images, octal dumps, etc., suppressed. If the first parameter =1, subsequent assembly continues with all printing enabled. Initially MTHAT has printing enabled.

The second parameter controls the printing of error messages, as follows:

parameter value	error message appearance
0	unchanged from previous state
1	assembly listing only
2	assembly listing and control console
3	control console only

If an error message appears on the console, then a single card image is expected to be typed in by the operator at the control console to correct the error, before normal assembly resumes. The first TOP parameter does not affect the printing of error messages. Initially MTHAT error messages appear on the assembly listing only.

TOP card images always are printed.

- - -

LIN sudo - line

1 parameter, nonlistable

The printer is upspaced the designated number of blank lines. The LIN sudo card image never prints. A blank parameter is treated as LIN 1.

- - -

PAG sudo - page

no parameters

The printer is upspaced to the top of the next page before the card image is printed. Page titling and numbering initiated by the TOF sudo is turned off.

- - -

TOF sudo - top of form

alphanumeric string in columns 25 to 72

The printer is upspaced to the top of the next page. The TOF card image is not printed, but the string on the TOF card is printed as a page title at the top of this and subsequent pages. The pages are sequence numbered. This process is terminated by a PAG sudo or another TOF sudo.

- - -

LIS sudo - list

no parameters

The LIS card image is not printed. Columns 5 to 80 of subsequent card images are printed on the assembly listing, but are not processed by MTHAT in any other way. Listing terminates and normal processing resumes when a card is encountered with the single word THAT in columns 5 to 8. The THAT card image is not printed.

The LIS sudo terminates macro declarations in the same manner as the FIN sudo.

- - -

DMP sudo - octal dump

2 parameters, listable; mode

The first parameter is the starting location and the second parameter is the ending location for an octal dump of the contents of memory. The dump may start slightly before the specified starting location, and may end slightly after the specified ending location. Lines of dump which would contain all zeroes are not printed. A blank line is printed after the dump.

The value of RXA is added to the specified starting and ending locations, unless some nonblank character is punched in the mode field.

- - -

OCT sudo - octal listing

no parameters

The value of A printed to the left of subsequent card images will be in octal. Initially A is printed in octal.

- - -

DEC sudo - decimal listing

no parameters

The value of A printed to the left of subsequent card images will be in decimal.

- - -

PVE sudo - print value of expression

1 parameter, listable

The value of the parameter is printed. It is not an error if the parameter is a single undefined label. If the PVE card image came from the control console, then the value of the parameter is typed back on the console, as well as being printed on the assembly listing.

- - -

TIM sudo - time

no parameters

The time is printed since the last occurrence of a TIM sudo, or since the last job card if there were no preceding TIM sudos.

- - -

OPM sudo - operator message

no parameters

The standard monitor operator message is printed.

- - -

ERR sudo - errors

See Section 18.

8. LEAVING MTHAT TO EXECUTE ASSEMBLED CODE

OUT usually is used to terminate assembly. If MTHAT is retained in the computer memory after the end of an assembly, and is called again as a subroutine in the user's program, then either OUT or RET may be used to return control to the user's program. MTT usually is used to execute user-written subroutines during the course of assembly.

- - -

OUT sudo - leave MTHAT

1 parameter, nonlistable

Control is transferred to the location named. Before doing so, various assembly statistics are printed. This includes printing the concordance table, if one was generated (See Section 15), and printing the free name table, if free names were used (See Sections 5 and 16).

If error messages appear on the control console (See TOP sudo, Section 7), then the type-in task OK is required before OUT is executed (See Section 25).

Upon leaving MTHAT, all user index registers are cleared to zero.

In order to safeguard against executing code in which assembly errors have occurred, the user should check all his labels using either the CHK or PRT sudos (See Section 9), and exit from assembly with the following code:

```
PIE
OUT      |0; halt
OUT      -----; desired location
```

- - -

MTT sudo - mark transfer to

1 parameter, nonlistable

MTHAT executes a TRM at assembly time to the location named. If the user wishes to force the appearance of an assembly error, his subroutine should return to the mark. Otherwise his subroutine should return to the mark +1.

The user's subroutine should not use the same index registers as are used by MTHAT. Currently MTHAT uses index registers /30 to /55, but this is subject to (gradual) change.

The MTT card image is printed on the assembly listing after execution of the MTT.

- - -

RET sudo - return

no parameters

MTHAT returns control to the mark of the routine which called it. MTHAT must have been called by a routine by executing TRM |37.

9. CONTROLLING REGION AND LABEL DEFINITIONS

See Section 5 for the distinction between regions and labels. The values of regions and labels may be defined either by the DEF sudo, or by the name of the region or label appearing in the label field of a card (See Section 3). Redefinition of the value of an already defined region or label can be done only by the DEF sudo, unless the region or label has been released by REL or LBL. All labels that are used in assembling code must be defined before they are released or before the end of assembly, but it is not an error for an unused label to remain undefined.

- - -

DEF sudo - define

1 parameter, listable

The parameter is not just the usual expression, but instead is of the form:

variable = expression

where the variable is either a region or label (or free name). The variable is defined to have the value of the expression, which must not itself contain undefined variables. The value of the definition, when truncated to an integer, must be between 0 and /177777 inclusive. If the variable is a label which was used prior to its definition, the value is filled in to the assembled words which used the label.

The location at which code is assembled may be altered by redefining A.

- - -

CHK sudo - check labels

1 parameter, listable

The parameter is a single label. All the labels with the named identifier, from subscript 0 to the named subscript, are checked. An error occurs if any of these labels has been used but not defined. If the named subscript =0 or if there is no subscript, all the declared labels of the named identifier are checked. Example:

```
                    CHK     L5, M;
```

Labels L0 to L5 and all the M labels are checked.

- - -

LBL sudo - declare or release labels

1 parameter, listable

The parameter is a single label. If the named identifier is not a region, and if the identifier has not previously appeared in a LBL sudo, then the identifier is declared to name labels with subscripts from 0 to the named subscript. Example:

```
                       LBL     T7;
```

New Labels T0 to T7 are declared.

On subsequent use of the identifier in a LBL sudo, the labels with subscripts from 0 to the named subscript are checked as in the CHK sudo, then released to the undefined state for reuse. If the named subscript =0 or if there is no subscript, all the declared labels of the named identifier are checked and released. An error occurs if the named subscript is higher than the original subscript used to declare the labels. Example:

```
                       LBL     T7;
```

Labels T0 to T7 are checked and released.

See the CON sudo, Section 15, for a discussion of the effect of the LBL sudo on the concordance.

- - -

REL sudo - release

1 parameter, listable

The parameter is a single region or label. If the parameter is a region, the named identifier is released for reuse either as a region or for labels. If the parameter is a label, the single named label is checked as in the CHK sudo, and then released to the undefined state for reuse.

See the CON sudo, Section 15, for a discussion of the effect of the REL sudo on the concordance.

- - -

PRT sudo - print labels

1 parameter, listable

The parameter is a single label. All the labels with the named identifier, from subscript 0 to the named subscript, are checked as in the CHK sudo. The values of all defined labels are printed. If the PUL switch is on, a special notation is made for those labels which are both undefined and unused. If the named subscript =0 or if there is no subscript, all the declared labels of the named identifier are checked and printed. Example:

```
                    PRT       L5, M;
```

Labels L0 to L5 and all the M labels are checked, and their values are printed.

- - -

PUL sudo - print undefined and unused labels

no parameters

Each use of the PUL sudo reverses the state of the PUL switch. Initially the PUL switch is off. The PUL switch is used by the PRT sudo.

- - -

LEN sudo - label and enter

1 parameter, partially listable

The parameter is a single label. The named labels are checked and released as in the LBL sudo (listable), then a single word of all zeroes is assembled as in the ENT sudo (nonlistable). The printer is upspaced two blank lines before the LEN card image is printed on the assembly listing.

- - -

OUI sudo - label, yes concordance

See Section 15.

- - -

NON sudo - label, no concordance

See Section 15.

10. ASSEMBLING G-21 INSTRUCTIONS

The following sudos assemble G-21 instructions for later execution. See also Section 6 and Section 26.

- - -

ENT sudo - enter

no parameters

A word of all zeroes is assembled. The printer is upspaced two blank lines before the ENT card image is printed on the assembly listing.

By convention, ENT is used at the beginning of a closed subroutine. The assembled word holds the return mark of the TRM calling the subroutine.

- - -

LEN sudo - label and enter

See Section 9.

- - -

SCP sudo - set character pointer

1 parameter, nonlistable

Two LXP instructions are assembled which will set the monitor character pointer to the named column number at run time.

- - -

*** sudo - space for generated instruction

no parameters

A word of all zeroes is assembled. By convention, this word will be written over by an instruction that is generated at run time.

11. ASSEMBLING INTEGERS, ADDRESSES, AND BOOLEAN DATA

LWD sudo - logic word

1 or 2 parameters, nonlistable; flag; mode

The first parameter is assembled and stored with an STL instruction. The first parameter may be an expression or a single undefined label. If the first parameter is an expression with value $<$ \$16, or if it is an undefined label, then a second parameter may appear at the user's option. The second parameter must be an expression with value between 0 and \$16-1 inclusive. The second parameter automatically is multiplied by \$16 and united with the first parameter.

The flag and mode fields are scanned after the parameters have been evaluated. The flag, if specified, is united with whatever is in the flag field of the assembled word due to the parameter evaluation. The mode, if specified, overrides whatever is in the mode field due to the parameter evaluation.

- - -

WRD sudo - word

1 parameter, listable

The parameter may be an expression, a single undefined label, or a minus sign followed by a single undefined label. If the resulting value is ≥ 0, it is assembled and stored with an STL instruction. If the resulting value is <0, it is assembled and stored with an STI instruction.

- - -

CLW sudo - complement logic word

1 parameter, nonlistable; flag; mode

The parameter must be an expression with all variables defined. The parameter is evaluated, then the flag and mode fields processed as in the LWD sudo. The bit-complement of the result is assembled and stored with an STL instruction. A FLG sudo (See Section 18) preceding this will be treated in the usual manner, not complemented.

- - -

ADC sudo - address constant

2 parameters, nonlistable; flag; mode

This sudo is processed as an OCA instruction with standard mode =0. By convention, this sudo is used to assemble addresses in the 15+1 bit pattern of G-21 instruction word format.

- - -

CKS sudo - checksum

See Section 18.

12. ASSEMBLING FLOATING POINT NUMERIC DATA

The parameters to the sudos in Section 12 are not expressions, but instead are signed floating point numbers of the form: signed mantissa, followed by a signed integer power of 8 or 10. (8 for HPL and FPL, 10 for HPC and FPC.) The radix used in evaluating the mantissa and exponent is 8 for HPL and FPL, and 10 for HPC and FPC. The exponent begins with the character $_{10}$ for all four sudos. Examples:

+25.31$_{10}$-7

.54

-.66$_{10}$14

1$_{10}$+3 (omitting the mantissa would result in a value =0)

2.9 (HPC and FPC only....the octal number system does not include the digit 9)

- - -

HPL sudo - half precision octal

1 parameter, listable

The mantissa is evaluated as an octal number. If an exponent appears, it is evaluated as an octal integer and 8 is raised to that power. The resulting number is stored with an STS instruction.

- - -

FPL sudo - full precision octal

1 parameter, listable

The parameter is evaluated as in the HPL sudo and stored with an STD instruction.

- - -

HPC sudo - half precision decimal

1 parameter, listable

The mantissa is evaluated as a decimal number. If an exponent appears, it is evaluated as a decimal integer and 10 is raised to that power. The resulting number is stored with an STS instruction.

- - -

FPC sudo - full precision decimal

1 parameter, listable

The parameter is evaluated as in the HPC sudo and stored with an STD instruction.

13. ASSEMBLING ALPHANUMERIC DATA

ALF sudo - alphanumeric

digit in card column 24; alphanumeric string starting in column 25

The digit determines how many words there are in the string, at 4 characters per word. A blank in column 24 means 1 word, a 0 in column 24 means 10 words. The words are assembled in the standard 8-bit, 4-character-per-word format.

- - -

NAM sudo - name

5-character string starting in card column 24; flag; mode

The standard 6-bit representation of the 5 characters is assembled into bits 0 to 29 of a word. Then the flag, if specified, is united with the word. The mode, if specified, overrides whatever is already in the mode field of the assembled word.

14. ALTERING THE LOCATION OR CONTENTS OF ASSEMBLY

Usually the location of assembly is altered by redefining the value of the Current Location Counter, A. See the DEF sudo, Section 9.

- - -

RXA sudo - relocator

1 parameter, nonlistable

This sudo is used to assemble code which will eventually be transferred to some different locations for execution. The Current Location Counter, A, should be defined as the location at which the code will ultimately be executed. The RXA sudo sets the value of RXA, which initially is zero. Assembled code is stored at A+RXA.

The CPY, DMP, and SXX sudos automatically compensate for nonzero RXA. The CKS and PBC sudos do not compensate for nonzero RXA. It probably is an error to execute an OUT or MTT sudo to code that was assembled with nonzero RXA, and which has not been transferred to its ultimate location.

All code assembled with nonzero RXA has the value of A+RXA and the value of RXA printed at the left edge of the assembly listing.

- - - -

COA sudo - continue on assembly

1 parameter, nonlistable

Two operations result from this sudo. First, RXA←A+RXA-parameter. Then A← parameter. The result is that A is redefined to the value of the parameter, but RXA is adjusted so that storage at A+RXA continues in sequence from the previous value of A+RXA.

- - -

SXX sudo - set storage extracter

1 parameter, nonlistable

During assembly, the previously existing contents of A+RXA are accessed with a CAL instruction and extracted with the storage extracter. The resulting bit pattern is united with the assembled word and the result is stored at A+RXA. The SXX sudo sets the storage extracter, which initially is zero.

- - -

FLG sudo - flag

See Section 18.

15. CONCORDANCE

The concordance is a cross-reference showing the card sequence numbers of every card that referred to a label, named a macro, or caused an assembly error. The concordance is generated during the assembly, and printed at the end of assembly as part of the OUT sudo (See Section 8). One word of memory is required to store each concordance reference. There is approximately one concordance reference per card input, on the average. The concordance is most useful if it is made on all labels, but it can be made on selected labels only, if storage space is scarce, by appropriate use of the OUI and NON sudos.

- - -

CON sudo - concordance

1 parameter, nonlistable

Concordance generation is initiated by the first appearance of the CON sudo. The parameter names the starting location for the table of references. Table entries continue in sequence unless the table is repositioned by another use of the CON sudo. If the parameter =0, concordance generation ceases. The OUT sudo prints whatever concordance has been generated.

An initial LBL sudo which declares new labels (See Section 9) is not included in the concordance. Labels declared by the LBL sudo are concorded, unless this state is changed by the OUI and NON sudos. Subsequent use of the LBL or REL sudos for releasing labels does not alter their current state of whether or not they are concorded.

- - -

REF sudo - reference

1 parameter, listable

This sudo creates a reference in the concordance table to every label in the parameter, whether or not the label currently is defined. Regions and constants are ignored, and the expression is not evaluated. The sudo is used for documentation purposes only.

- - -

OUI sudo - label, yes concordance

1 parameter, listable

The paramenter is a label, which is declared or released for re-use as in the LBL sudo (See Section 9). Subsequent references to the labels designated will be concorded, if a concordance is being generated at all.

- - -

NON sudo - label, no concordance

1 parameter, listable

The parameter is a label, which is declared or released for re-use as in the LBL sudo (See Section 9). Subsequent references to the labels designated will not be concorded.

16. FREE NAMES

See Section 5 for a discussion of what free names are and how they are used.

- - -

LFN sudo - locate free name table

1 parameter, nonlistable

Free names may not be used until a table for their storage has been located. The parameter is the base address of the table. Free names of 2, 3, or 4 characters length require 1 word of storage. Free names of 5, 6, 7, or 8 characters length require 2 words of storage. The LFN sudo may be used only once in a program.

- - -

PFN sudo - print the free name table

no parameters

The free name table is printed in both numeric and alphabetic order. It is not an error if no free name table exists. The free name table also is printed automatically by the OUT sudo (See Section 8).

17. MACROS

Each macro consists of several card images that are stored in the computer memory during the macro declaration. All the card images of the macro may subsequently be read back and processed by MTHAT, at each appearance of a macro call card. The macro call card has the name of the macro in the opcode field (See Section 3), and may have several actual parameters to the macro in the parameters, flag, and mode fields. The actual parameters in the parameters field of the macro call card correspond to the formal parameters A>, B>, C>, D>, E>, and F> that may be used in the coding of the macro declaration. The "greater" variables may also be used in other parts of an MTHAT program, but their primary use is as formal parameters to macros.

The greater variables are loaded by the up-to-6 actual parameters appearing on a macro call card or VAR sudo card. The actual parameters are separated by commas, and are loaded into A>, B>, C>, D>, E>, and F> in that order. Any actual parameters omitted at the end of the card, or left blank between commas, leave the corresponding greater variables unchanged. If an actual parameter consists of an expression other than a single region or label, then the expression is evaluated immediately and its value is loaded into the proper greater variable. In subsequent use, the greater variable is treated as a constant. If the actual parameter consists of a single region or label, then the name of the region or label is loaded into the proper greater variable. In subsequent use, the actual parameter is treated as though it were the name of that region or label, and is given whatever meaning is currently assigned to that name.

It also is possible to parameterize the flag and mode fields for use in macros or elsewhere. There exist within MTHAT two variables called the "flag temp" and the "mode temp". These variables are loaded whenever a macro call card of VAR sudo card is processed, and are read whenever any other card is processed.

Each of these variables may be in any one of five states:

0

1

2

3

BLANK

The rules for loading the flag and mode temps are summarized in the following table.

FLAG PUNCH	MODE PUNCH	PREVIOUS STATE OF TEMP	NEW STATE OF TEMP
0, 1, 2, 3	0, 1, 2, 3	immaterial	0, 1, 2, 3
A, B, C, D, E, F	A, B, C, D, E, F	immaterial	A> ∧3, B> ∧3, C> ∧3, D> ∧3, E> ∧3, F> ∧3
*	*	0, 1, 2, 3, BLANK	0, 1, 2, 3, BLANK
illegal	+	0, 2	2
		1, 3	3
		BLANK	BLANK
illegal	-	0, 2	0
		1, 3	1
		BLANK	BLANK
blank	blank	immaterial	BLANK

The rules for reading the flag and mode temps are summarized in the following table.

FLAG PUNCH	MODE PUNCH	STATE OF TEMP	EFFECTIVE FLAG OR MODE
0, 1, 2, 3	0, 1, 2, 3	immaterial	0, 1, 2, 3
A, B, C, D, E, F	A, B, C, D, E, F	immaterial	A> ∧3, B> ∧3, C> ∧3, D> ∧3, E> ∧3, F> ∧3
*	*	0, 1, 2, 3	0, 1, 2, 3
		BLANK	0 flag, standard mode
illegal	+	0, 2	2
		1, 3	3
		BLANK	standard mode ∨2
illegal	-	0, 2	0
		1, 3	1
		BLANK	standard mode ∧1
blank	blank	immaterial	0 flag, standard mode

where "standard mode" refers to the standard mode of the card being processed.

- - -

VAR sudo - greater variable

6 parameters, nonlistable; flag; mode

The greater variables and flag and mode temps are loaded as described above.

- - -

MAD sudo - macro administration

3 parameters, nonlistable

This sudo may be used at most once in a program, which must be before the first MAC sudo. The first parameter specifies the maximum number of macros to be declared, and must be ≤ 300. The size of the MTHAT label table is incremented by twice this value. The second parameter is the location where the card images that comprise the macros are to be stored. Each card image requires 21 words of storage. The third parameter, if it appears, is the maximum number of blocks that can be pushed at any given time by the PSH sudo. These blocks each require 20 words of storage, and are stored at the location designated by the second parameter. Storage of macro card images starts after the end of these blocks.

- - -

MAC sudo - declare a macro

macro name in card columns 26 to 28

This sudo starts a macro declaration. Images of the following cards are stored in core, to be processed when the macro is called. Macro calls may be stored within the declaration of another macro, in which case the macro call card and all the card images of the macro being called are stored again as part of the macro being declared. This allows nesting of macros to indefinite depth. Macros may not call themselves.

Any cards may appear in a macro declaration except the following: LFN, MAC, RET, TB1, TB2, ZRO. Each macro may consist of a maximum of 255 card images.

- - -

FIN sudo - finish macro declaration

character in card column 24

This sudo terminates the declaration of a macro. Until this card is processed, the macro may not be called. If the character in column 24 is the digit 0, then all the greater variables are cleared to zero. Otherwise the greater variables remain unchanged. The FIN card image is printed twice on the assembly listing, because the card is both stored as part of the macro declaration, and processed to terminate the declaration. However, the label field of the FIN card is processed only during a macro call. The LIS sudo also terminates macro declarations, but it is not recommended for this purpose.

- - -

The remaining sudos in this section are designed to simplify the task of using MTHAT as a macro assembler. The BRA, BRV, and BRD sudos may appear only inside macros. These three sudos cause branching of the card processing by MTHAT. That is, after processing one of these sudos, the next card processed by MTHAT may not be the card that appears immediately after the sudo. The next card processed is called the "destination card." Destination cards are marked by having a "destination number" punched in column 3. The destination number may be a digit, 0,1,2,3,4,5,6,7,8,9, or it may be the character $_{10}$. Only cards within macros, including the FIN card, may be destination cards.

If several macros are nested together, the card branching may go in and out of the inner macros, but may not leave the outermost macro. In other words, the destinations must be nested within the outermost macro. The destination numbers are treated as local to the outermost macro, and may be reused in any other disjoint macro. Special precautions must be observed when a destination lies within the scope of an ITR sudo. (See Section 19)

BRA sudo - unconditional branch

1 parameter, nonlistable

This sudo may appear only inside macros.

The value of the parameter is the destination number of the next card to be processed by MTHAT. If the parameter is blank, the next card processed will be the card appearing immediately after the BRA card.

- - -

BRV sudo - conditional branch on value

4 parameters, nonlistable

This sudo may appear only inside macros.

The first parameter is evaluated and truncated to a 32-bit signed integer. The resulting value determines which of the three possible destinations, in the three following parameters, will be branched to by MTHAT. If the value is positive, MTHAT will branch to the first destination. If the value is zero, MTHAT will branch to the second destination. If the value is negative, MTHAT will branch to the third destination.

Only the actual destination parameter is evaluated by MTHAT. If the parameter is blank or absent, the next card processed will be the card appearing immediately after the BRV card.

If the value of the destination parameter is not between 0 and 10, inclusive, then an assembly error will result. This may be used to force the appearance of an error message as the result of an assembly-time test.

- - -

BRD sudo - conditional branch on definition

3 parameters, nonlistable

This sudo may appear only inside macros.

The first parameter must be one of the terms listed in Section 4. However, the BRD sudo is of use only if the first parameter is a term of Type 7,8,9, or 10.

If the term currently has a defined value, then MTHAT will branch to the destination indicated by the second parameter. If the term currently is undefined, then MTHAT will branch to the destination indicated by the third parameter.

Only the actual destination parameter is evaluated by MTHAT. If the parameter is blank or absent, the next card processed will be the card appearing immediately after the BRD sudo.

If the value of the destination parameter is not between 0 and 10, inclusive, then an assembly error will result. This may be used to force the appearance of an error message as the result of an assembly-time test.

- - -

PSH sudo - push down macro parameters

no parameters

The greater variables, flag temp, and mode temp are pushed down, in a block. The maximum number of blocks that can be pushed at any given time is determined by the third parameter to the MAD sudo.

- - -

POP sudo - pop up macro parameters

character in column 24 of card

The last block pushed by the PSH sudo is popped up. The flag and mode temps are restored, and some of the greater variables are restored. A letter in column 24 of the card indicates the first of the greater variables to be restored. For example, if column 24 contains the letter C, then C>,D>,E>, and F> are restored, and A> and B> are left alone. If column 24 is blank, then all the greater variables are restored. If column 24 contains the digit 0, then none of the greater variables are restored (but the flag and mode temps are restored).

- - -

SSC sudo - subscript

1 parameter, nonlistable

The parameter must be one of the terms listed in Section 4, except not a term of Type 6. However, the SSC sudo is of use only if the parameter is a term of Type 9 or 10.

If the parameter is a constant, then the subscript variable V (see Section 19) is set to the value of the parameter. If the parameter is a variable, then V is set to the subscript of the variable, whether or not the variable currently has a defined value. This sudo may be used to find the subscript of a label corresponding to a free name.

18. ERROR DETECTION AND CORRECTION

Assembly errors detected by MTHAT immediately terminate processing of the card on which they were found. The error message may be printed on the assembly listing, or typed on the control console for correction by the operator, or both, depending on the second parameter to the TOP sudo (See Section 7). If error output appears on the control console, then the card column number at which the error was detected, and the current value of A, also are typed out. A single card image is requested from the operator, to correct the error. If more than one card is required to correct the error, then the first card image typed in may be the TYP sudo (See Section 25). If a concordance is being generated, a reference is created for each detected assembly error.

In order to prevent a waste of computer time in the assembly of "garbage", MTHAT assembly automatically halts if an average error rate of one error for every four cards is maintained.

- - -

DBG sudo - debug

no parameters

If this sudo appears anywhere in an MTHAT assembly, the monitor trace subroutine |72 is enabled. This subroutine will trace all 2-flagged instructions in the assembled code.

- - -

FLG sudo - flag

1 parameter, nonlistable

The parameter must evaluate to 0, 1, 2, or 3. This value will be united with the flag field of the next word assembled. A blank parameter is given the value 2. An error occurs if FLG 1 or FLG 3 precedes a G-21 instruction word, or if any nonzero flag precedes the CKS or LC8 sudos.

Because operand assembly (OA) instructions are not traced, placing a FLG sudo before an OA instruction causes the next non-OA instruction to be traced.

- - -

ERR sudo - errors

no parameters

A count of all the errors which have not been corrected at the control console is printed. ERR card images always are printed, independent of the parameters to the TOP sudo (See Section 7).

- - -

ZEC sudo - zero the error counter

no parameters

This card is processed in the same manner as ERR sudo cards, and then the counter of uncorrected errors is set to zero.

- - -

PIE sudo - process if errors

no parameters

This card is processed in the same manner as ERR sudo cards. If the counter of uncorrected errors is nonzero, the next card following is processed in the normal manner. If the counter of uncorrected errors is zero, processing of the next card is bypassed. The card image is printed on the assembly listing with arrows to indicate bypassing. Normal MTHAT processing resumes with the second card after the PIE sudo.

The PIE sudo is used to prevent execution of code that contained assembly errors. For further discussion, see the OUT sudo, Section 8.

- - -

CKS sudo - checksum

2 parameters, partially listable

The two parameters are the starting and ending locations for a checksum (listable) formed using ADL. A single word is assembled (nonlistable) which contains the checksum.

- - -

CSR sudo - check status report

1 parameter, nonlistable

Changes made to the MTHAT system are summarized in a one-line status report. The CSR sudo causes printing of the status report on the assembly listing. MTHAT also has a "change number", which is incremented whenever a significant change is made to the system. If a parameter appears with the CSR sudo, the parameter is tested against the change number. An error occurs if the parameter is less than the change number, which indicates that some of the user's code may be incorrect because of a change in the system. If the parameter is blank, no test is made.

19. ASSEMBLY-TIME ITERATION

The character V, called the subscript variable, serves a special function in the evaluation of constants and variables. Its primary use is for assembly-time iteration, but it may be used elsewhere if so desired. V is a double precision variable that may be intermixed with digits in a constant or in the subscript of a variable. Evaluation is performed from left to right, with the evaluated portion to the left of V being multiplied by the value of V, and then the portion to the right of V being evaluated and added to form the final result. In the following examples, assume that the current value of V is 5.

PARAMETER	VALUE
V	5
V + 1	6
LV	L5
LV + 1	L5 + 1
$V	$5
V1	6
/V 22	/27 = 5 + /22
LV1	L6
2V	10
VV	25
2V1	11
2V1V	55
2V1V1	56
L2V1V1	L56
/10 V 17	/67 = /50 + /17

- - -

SET sudo - set V

1 parameter, nonlistable

The double precision subscript variable V is set to the value of the parameter.

- - -

ITR sudo - iterate

5 parameters, nonlistable

This sudo allows for iterative processing of card images during assembly time. Call the parameters N_1, N_2, N_3, N_4, N_5. N_1 is the value which V is assigned before iteration begins. N_1 is evaluated and stored into V before the other parameters are evaluated. N_2 is the value of the increment to V which automatically occurs at the end of each iteration cycle. N_1 and N_2 may be any value, positive or negative, and are stored in double precision. N_3 is the number of cycles of iteration to be performed. N_3 must be a non-negative integer. N_4 is the number of cards composing a cycle of the iteration. N_4 must be a strictly positive integer. N_5 is the base address of a region in memory where N_4 card images may be stored until the interation is completed. Each card image requires 21 words of storage space.

The following cards may not appear within the scope of an iteration (that is, among the N_4 cards following the ITR card): ITR, LIS, MAC, TBl, or macro call cards.

The parameters on the ITR card are evaluated only once, before iteration begins.

If the ITR sudo appears inside a macro, then the 5th parameter, N_5, must be omitted, since the card images will be stored in the computer memory as part of the macro declaration. The scope of the iteration must be properly nested within the outermost macro which contains the ITR card as part of its declaration. However, the scope of the iteration may extend past the end of a macro nested within the outermost macro. For example, a piece of (useless but) valid code might be:

```
MAC       ABC
COM      COMMENT 1 OF ABC
ITR      0,0,6,3; USELESS ITERATION
COM      COMMENT 2 OF ABC
FIN
MAC       XYZ
COM      COMMENT 1 OF XYZ
ABC
COM      COMMENT 2 OF XYZ
COM      COMMENT 3 OF XYZ
FIN
```

because the declaration of macro XYZ is expanded by MTHAT to:

```
MAC       XYZ
COM      COMMENT 1 OF XYZ
ABC
COM      COMMENT 1 OF ABC
ITR      0,0,6,3; USELESS ITERATION
COM      COMMENT 2 OF ABC
FIN
COM      COMMENT 2 OF XYZ
COM      COMMENT 3 OF XYZ
FIN
```

Caution must be exercised when using the branch sudos BRA, BRV, and BRD (see Section 17) in conjunction with ITR inside a macro. Do not branch into the scope of the iteration from outside the scope unless you previously branched out from inside while the iteration still was in progress. Otherwise erratic card processing might occur.

20. PARALLEL TABLES

Parallel tables are two or more tables for which the entries correspond, such as a table of task names and a table of corresponding subroutine locations for executing the tasks. It is possible to put the corresponding entries for two tables on a single card, both to help make the tables self-commenting and to prevent accidental displacement of the entries in one of the tables. If more than two parallel tables are desired, entries for the first two tables are put on one card and corresponding entries for the remaining tables are put on another card that follows immediately.

The two sudos TB1 and TB2 are intended to precede and follow the cards which contain the table entries, in order to delimit the scope of parallel table processing. The sudo TB3 is used on data cards appearing between TB1 and TB2, only if there are three or more parallel tables.

- - -

TB1 sudo - table 1

1 parameter, nonlistable

All cards following the TB1 card, except TB3 cards, will be processed in the normal manner, like any other MTHAT cards. Also, the card images will be stored in the computer memory, starting at the location named in the parameter. Each card image requires 21 words of storage space. The following cards may not appear between TB1 and TB2 sudos: ITR, LIS, MAC, OUT, RET, TB1, and macro call cards.

The label field of TB3 cards will be processed as usual, but otherwise TB3 cards will be treated as comment cards.

- - -

TB2 Sudo - table 2

2 parameters, listable

AND

TB3 sudo - table 3

variable format

The card images that have been stored in core are read back and processed again, with print off, in order to form the second parallel table:

1. TB3 cards are ignored entirely.
2. Only a special field is examined on the cards that are processed. This field starts with a control column (col. 68) and extends to the last column (col. 80). The remainder of the card is ignored.
3. If the control column contains a + or -, then a single expression is expected in the field. The + or - in the control column is considered part of the expression. The expression is assembled as is done by the WRD sudo. The expression may have all variables defined, or it may consist of a + or - in the control column and a single undefined label.
4. If the control column contains a digit, then that number of alphanumeric words are assembled. The characters for the words are taken from the field, starting in the first column after the control column, and packed 4 characters per word. The number of words specified by the digit in the control column must not require more than 3 characters past the last column in the field. That is, $1 \le \text{digit} \le$ $\downarrow [[\text{number of last column} + 3 - \text{number of control column}]/4]$.

Characters taken from beyond the last column of the field are assembled as blanks.

5. If the control column contains the letter A, then a single expression is expected in the remainder of the field, starting in the first column after the control column. The Current Location Counter, A, is redefined to have the value of this expression.

6. If the control column contains any character except the characters listed in (3), (4), and (5) above, then the card image is ignored.

Processing continues on these cards until the TB2 sudo card is read again. Columns 68 to 80 of the TB2 card are not processed. At this point, columns 24 to 67 of the TB2 card are scanned for a pair of parameters. If the parameters are not found, parallel table processing terminates.

If the parameters are found, then the first parameter is interpreted to be the number of a control column, and the second parameter is interpreted to be the number of the last column of the field. The card images that have been stored in memory are read back and processed again, with print off, in order to form a third parallel table. This time only the TB3 cards are processed; all other cards are ignored. Processing follows conventions (2) to (6) above, where the number of the control column and the number of the last column are determined by the expressions on the TB2 card.

If the TB2 card contains more pairs of expressions, then 4th, 5th, etc. parallel tables are formed from the data on the TB3 cards.

21. CONTROLLING INPUT CARDS

Card images may be obtained from the control console (See Section 25), the card reader, disc, tape, or the computer memory (for macros, assembly-time iteration, or parallel tables). Each card image read by MTHAT is sequence numbered. The MTHAT sequence number is printed on the assembly listing to the left of the card image, and to the left of the current value of A. Sequence numbers appearing to the right of the card image were assigned by the AND system.

- - -

CSS sudo - card source switch

3 parameters, nonlistable

Call the parameters N_1, N_2, N_3. N_1 and N_2 are monitor logical file table entry numbers or else blank; N_3 may be any expression or blank. The CSS sudo operates as follows:

(1) Record the source for subsequent card images in logical file table entry N_1 (if the parameter is blank, no write occurs).

(2) Read from logical file table entry N_2 the source for subsequent card images (if blank, no read).

(3) Reset the source for subsequent card images as follows:

N_3 blank, no reset. Subsequent card images come from the source read in (2) above, or from the previous source if N_2 is blank.

$N_3<0$, subsequent card images come from the beginning of AND scratch. The information obtained from the read operation in (2) above is lost.

$N_3=0$, subsequent card images come from the card reader.

$N_3>0$, subsequent card images come from disc or tape, starting from the source point read in (2) above, or from the previous disc or tape source if N_2 is blank.

In order to understand the operation of CSS, it is necessary to know that the monitor has a single switch which determines whether cards are read from the card reader or from either disc or tape. The monitor also has a single pointer to the location of the next card image coming from disc or tape. This pointer is not altered when cards are read from the card reader. Both the switch and the pointer may be saved by storing their values in any one of the logical file table entries.

- - -

BYP sudo - bypass

3 parameters, nonlistable

Call the parameters N_1, N_2, N_3. Normal MTHAT processing of the card following the BYP card is bypassed if the logical proposition $[N_1 > N_2] \equiv [N_3 \neq 0]$ is true. If the proposition is false, the following card is processed as usual. Arrows printed on the assembly listing indicate that the card has been bypassed. The parameters are evaluated as signed integers.

- - -

BNC sudo - bypass N cards

1 parameter, nonlistable

The parameter specifies the number of cards following the BNC card whose processing is to be bypassed. Arrows printed on the assembly listing indicate that the cards have been bypassed.

- - -

PIE sudo - process if errors

See Section 18.

- - -

CRD sudo - card input

See Section 25.

- - -

TYP sudo - control console input

See Section 25.

- - -

BRA sudo – unconditional branch

See Section 17.

- - -

BRV sudo - conditional branch on value

See Section 17.

- - -

BRD sudo - conditional branch on definition

See Section 17.

22. SAVING ASSEMBLED CODE

Assembled code may be saved on row-binary cards by the PBC sudo, or on AND records by MTHAT subroutine '22 (See Section 24). There is no sudo for executing '22, but the proper calling sequence may be set up and executed by an MTT (See Section 8) if desired.

- - -

PBC sudo - punch binary cards

3 parameters, nonlistable

The first two parameters are the starting and ending locations of a region of the computer memory whose contents are punched on row-binary cards. If a third parameter =1 appears, the MTHAT symbol table also is punched. Punching the symbol table allows later alteration of the program using the original regions and labels, when the program is read back by RBC.

The use of a concordance, free names, or macros affects the use of the symbol table. If the symbol table is punched by PBC when any of these features are in use, the user must also include additional PBC sudos which punch the concordance table, free name table, and macro card images.

- - -

RBC sudo - read binary cards

no parameters

The cards punched by the PBC sudo are read back into the locations from which they were punched. If the MTHAT symbol table was punched by PBC, it is read back and completely replaces the current MTHAT symbol table. One RBC sudo is required for each use of the PBC sudo.

The row-binary cards that were punched by PBC should follow immediately after the RBC card, with no blank cards intervening. Two blank cards should be placed after the row-binary deck before the remaining MTHAT cards.

23. USER-DECLARED SUDOS

Usually any code the user wishes executed at assembly time can be written so that it is entered by the MIT sudo (See Section 8). However, MTHAT has a sudo trap feature which allows the user to write his own sudos. Sudo processing is so intimately connected with the internal mechanism of MTHAT, that the user is strongly urged to contact the person maintining the MTHAT system before attempting to write his own sudos.

24. RUN-TIME FEATURES

Some of the subroutines and tables used by MTHAT during assembly may also be of use to the user during run-time. Switch '21 and subroutines '18, '19, '22, '23, and '24 are independent of the presence of the remainder of the MTHAT system. They are located at the bottom of user memory, and extend up to location '20-1. The remaining tables and subroutines are scattered throughout the MTHAT system. See Section 33 for a complete list of available tables and subroutines.

- - -

Subroutine '17: Convert (ACC) to label for symbolic disassembly.

Input in ACC: A number, presumably an address.

Output in ACC: The label corresponding to that address.

Operation: If $1 \le$ input $\le \$16$, then a search is made through the label table for the first label whose value = $\downarrow$ [input]. If found, the subroutine exits to the mark + 1 with the label in the ACC as follows:

Bits 0 to 7: alphanumeric representation of the identifier

Bits 8 to 21: subscript stored as an integer

The search is performed in the same order as labels are printed in the concordance. If no such label is found, or if the input is not within the specified range, the subroutine exits to the mark with garbage in the ACC.

Storage: Some parts of MTHAT, the label table, and index registers /30 and /31.

- - -

Subroutine '18: Write a logical file table entry.

Input in index register /30: A logical file table entry number.

Operation: The contents of the switch and pointer in the monitor which determine the source of subsequent card images are written into the logical file table entry (See CSS sudo, Section 21).

Exits: RED exit to mark, error, no write performed

GREEN exit to mark + 1

Storage: Below location '20, and index register /30

- - -

Subroutine '19: Read a logical file table entry.

Input in index register /30: A logical file table entry number.

Operation: The contents of the logical file table entry are read into the monitor switch and pointer which determine the source of subsequent card images (See CSS sudo, Section 21).

Exits: RED exit to mark, error, no read performed

GREEN exit to mark + 1

- - -

Switch '21: Upper core request.

This switch is set by the 64K sudo (See Section 27) and interrogated by subroutine '24. Its value is 0 if upper core memory has not been requested or if the request was denied. Its value is 1 if the request was granted.

- - -

Subroutine '22: Write a disc or tape file.

Several disjoint regions of memory are written to disc or tape. They may be read back by '23. The file must have been set up and named in one of the monitor logical file table entries. A buffer region of length =1 block [320 words] must be provided. This subroutine uses index registers /30, /31, and /32 for temporary storage. Calling sequence code:

Word 0: TRM '22

Word 1: logical file table entry number

Word 2: location of buffer region

Following pairs of words [at least 1 pair]:

1st word: starting location of data region to be written

2nd word: ending location +1 of data region to be written

Next word: RED exit error instruction

Next word: GREEN exit instruction and subsequent code

The word pairs designating regions to write are assumed to continue until a word is found with some nonzero bits in bit positions 16 to 31. This word is assumed to be the RED exit error instruction. Therefore, the RED exit error instruction may not be an OCA mode 0. If the subroutine returns through the RED exit, the accumulator contains an integer indicating the error, as follows:

0	error in calling sequence
1	error in logical file table entry
2	attempt to write into protected area
3	insufficient space in file

The pairs of words in the calling sequence which designate regions are altered by the subroutine.

- - -

Subroutine '23: Read a disc or tape file.

A file written by '22 is read back into the memory locations from which it was written. The file must have been set up and named in one of the monitor logical file table entries. A buffer region of length =1 block [320 words] must be provided. This subroutine uses index registers /30, /31, and /32 for temporary storage. Calling sequence code:

Word 0: TRM '23

Word 1: logical file table entry number

Word 2: location of buffer region

Word 3: RED exit error instruction

Word 4: GREEN exit instruction and subsequent code

If the subroutine returns through the RED exit, the accumulator contains an integer indicating the error, as follows:

0	error in calling sequence
1	error in logical file table entry
2	data in file was not written by '22
3	read operation completed, but checksum failed

- - -

Subroutine '24: Test if (ACC) is within user memory.

Input in ACC: A number, presumably an address.

Exits: Mark, (ACC) is not within user memory.

Mark +1, (ACC) is within user memory.

Output: Input (ACC) is undisturbed.

25. CONTROL CONSOLE INTERACTION

System programmers who use the control console for on-line debugging can take advantage of the control console features of MTHAT. The TOP sudo (See Section 7) allows error messages to be typed automatically on the control console, and the errors to be corrected by typed-in card images. The OPA sudo allows predetermined messages to be typed out, and the PFC sudo allows octal parameters to be typed in. Card source switching may be accomplished by the CRD and TYP sudos when MTHAT is processing cards, or by the ,CRD and ,TYP type-in tasks in the extended monitor type-in task table. Preprogrammed halts for execution of type-in tasks may be accomplished by the MON sudo. Unscheduled halts for execution of type-in tasks may be accomplished by the ,MON type-in task.

Card images typed in from the control console may consist of up to 40 characters, including blanks and commas. The first 3 commas scanned act as field delimiters, like the TAB keys on a typewriter. The characters typed in before the first comma are shifted to card columns starting at column 4, the beginning of the label field on the card. The first comma typed in does not appear as part of the card image, but instead acts as a tab to column 15, the beginning of the opcode field. The second comma acts as a tab to column 20, the mode field. The third comma acts as a tab to column 24, the beginning of the parameters field. All subsequent commas typed in are processed the same as any other character, and appear in the card image. For example, to type in the card image:

```
   L3          CLA  0   5,X7;
```

the characters to type in are:

```
L3,CLA,0,5,X7;
```

When card images are being typed in from the control console for listing only (See the LIS sudo, Section 7), commas are not given the special function described above. All commas appear as part of the card image.

See Section 18 for a discussion of error messages on the control console.

During an MTHAT assembly, the standard monitor type-in task table is extended to include the tasks described below. The task names must not be typed in after the end of assembly unless the MTHAT system remains undisturbed in the computer memory. The tasks are:

,CRD	Enter MTHAT, read cards from the card reader, disc, or tape.
,TYP	Enter MTHAT, get card images from the control console.
,PAG	Page the line printer, enter MTHAT, get card images from the control console.
,MON RET	Leave a note for MTHAT to enable monitor type-in when MTHAT has completed processing the current card.
OK	Process the current card (See the OUT sudo, Section 8).

- - -

CRD sudo - card input

no parameters

Subsequent card images are taken from the card reader, disc, or tape.

- - -

TYP sudo - control console input

no parameters

Subsequent card images are taken from the control console.

- - -

MON sudo - monitor

no parameters

Normal card processing is halted. When the control console is free from any input or output tasks initiated previously, the monitor type-in is enabled.

- - -

OPA sudo - operator alert

alphanumeric string in card columns 25 to 80

The string is typed out on the control console on a full interrupt basis, while other card processing by MTHAT continues.

- - -

PFC sudo - parameters from console

2 parameters, nonlistable

The first parameter, whose value must be either 1 or 2, is the number of octal parameters to be typed in from the control console and stored in |62 and |62+1. The second parameter is the location of a closed subroutine to be executed automatically when type-in has been completed. If the second parameter is absent, blank, or zero, no subroutine will be executed when type-in has been completed.

The type-in is done on a full interrupt basis, while other card processing by MTHAT continues. It is the responsibility of the user to check that type-in has been completed before attempting to utilize the typed-in parameters. This may be done by an MTT to a subroutine which in turn calls on |56, the control console busy test in the monitor. An MTT directly to |56 would cause an assembly error, because |56 would return to the RED exit of the MTT sudo (See Section 8).

- - -

TOP sudo - type or print

See Section 7.

- - -

PVE sudo - print value of expression

See Section 7.

- - -

OUT sudo - leave MTHAT

See Section 8.

26. HARDWARE REGISTERS AND LINE COMMANDS

MTHAT has a single table '11 of register and line command mnemonics. It is the responsibility of the user to guard against conflict between these mnemonics and the free names in his code, when writing code that utilizes any of the sudos in this Section.

- - -

ERA, ERO, EXR, and LDR sudos

2 parameters, nonlistable; flag; mode

These sudos assemble the G-21 instructions ERA, ERO, EXR, and LDR. The standard mode is mode 2. The first parameter, which is assembled into the address field of the G-21 instruction word, is processed in a manner identical to other G-21 instructions (See Section 6). The second parameter, which is assembled into the index field of the G-21 instruction word, is compared with the mnemonics in MTHAT table '11. If it is a mnemonic, the corresponding value is loaded into the index field. If it is not a mnemonic, the second parameter is then processed in a manner identical to other G-21 instructions.

- - -

TC8, TD8, and RD8 sudos

2 parameters, nonlistable; flag; mode

These sudos assemble the G-21 instructions TC8, TD8, and RD8. The instructions are used as the second word of block input/output commands. The standard mode is mode 0. All other comments pertaining to the ERA, ERO, EXR, and LDR sudos also apply to the TC8, TD8, and RD8 sudos.

- - -

TLC sudo

2 parameters, nonlistable; flag; mode

This sudo assembles the G-21 instruction TLC. The first parameter, which is assembled into the address field of the G-21 instruction word, is compared with the mnemonics in MTHAT table '11. If it is a mnemonic, the corresponding value is loaded into the address field, and the standard mode of the instruction is mode 0. If it is not a mnemonic, the first parameter is then processed in a manner identical to other G-21 instructions, and the standard mode of the instruction is mode 2. The second parameter, which is assembled into the index field of the G-21 instruction word, is processed in a manner identical to other G-21 instructions (See Section 6).

- - -

LC8 sudo - 8-bit line commands

1 parameter, listable

This sudo assembles line commands and numeric data in 8-bit characters for transmission over the communication line. If the parameter is a mnemonic from MTHAT table '11, the corresponding value is packed in a single 8-bit character. If the parameter is an expression, the value of the expression is packed in two characters and numeric flags added. An error occurs if the value of the expression is not between 0 and /7777 inclusive.

The resulting characters are assembled four per G-21 word. If the number of characters generated is not an integer multiple of 4, the remaining characters are packed left-justified and the rest of the last word made zero.

27. OTHER SUDOS

ZRO sudo - zero memory

All of user memory past the current end of the MTHAT label table is set to zero. If an upper core request has been granted, upper core user memory also is set to zero. In particular, this sudo will destroy the data stored in a concordance table, free name table, or stored macro card images.

- - -

CPY sudo - copy

2 parameters, listable

Call the parameters N_1, N_2. The N_2 words last assembled are repetitively copied into the next N_1 locations. If N_2=0, then zeroes are stored into the next N_1 locations. N_2 must be nonnegative. Both parameters must appear on the CPY card. If the last N_2 words contain any undefined labels, these will not later be defined in the copies.

Example:

```
        L1  WRD    5, 7;
            CPY    9, 2;
```

is the same as;

```
        L1  WRD    5,7;
            WRD    5,7,5,7,5,7,5,7,5;
```

- - -

SIZ sudo - size of MTHAT

no parameters

The subscript variable V (See Section 19) is set to the current size of MTHAT, including the MTHAT label table. After executing the SIZ sudo, '16 + V = the address of the first location after the current end of MTHAT and the label table.

- - -

64K sudo - request use of upper core memory

no parameters

Before the 64K sudo is executed, switch '21 contains the value 0. If the request is granted, the contents of '21 are set to 1. If the request is denied, the contents of '21 remain at 0 and an assembly error results.

- - -

OPT sudo - option

3 parameters, nonlistable

This sudo just saves an MTT to the monitor option subroutine, |34. The three parameters are the same as in the option writeup.

28. LIST OF G-21 INSTRUCTIONS AND STANDARD MODES

145	ADA	2		131	IEC	2	
000	ADC	0	(See Section 11)	111	IEZ	2	
045	ADD	2		011	IOZ	2	
055	ADL	2		051	ISN	2	
105	ADN	2		171	IUC	2	
002	ADX	2		071	IUO	2	
006	AXT	2		151	IUZ	2	
033	BTR	0		056	LDR	2	(See Section 26)
015	CAL	2		032	LXM	2	
035	CCL	2		012	LXP	2	
005	CLA	2		077	MPY	2	
025	CLS	2		140	OAA	2	
053	DIV	2		040	OAD	2	
135	ECL	2		100	OAN	2	
072	ERA	2	(See Section 26)	000	OCA	2	
052	ERO	2	(See Section 26)	020	OCS	2	
115	EXL	2		160	OSA	2	
076	EXR	2	(See Section 26)	120	OSN	2	
061	FGO	2		060	OSU	2	
121	FLO	2		057	RDV	2	
021	FOM	2		120	RD8	0	(See Section 26)
001	FOP	2		013	REP	0	
101	FSM	2		137	SKP	0	
141	FSN	2		153	STD	0	
041	FSP	2		133	STI	0	
161	FUO	2		173	STL	0	
031	ICZ	2		113	STS	0	

073	STZ	0
165	SUA	2
065	SUB	2
075	SUL	2
125	SUN	2
022	SUX	2
026	SXT	2
140	TC8	0 (See Section 26)
117	TDC	2
100	TD8	0 (See Section 26)
157	TLC	0 or 2 (See Section 26)
017	TRA	0
037	TRE	0
177	TRM	0
175	UCL	2
155	UNL	2
010	XEQ	2
036	XMT	2
016	XPT	2

29. LIST OF ALPHANUMERIC CHARACTERS

G-21 Internal Representation	G-21 Character	Punch Card Hole Pattern	Model 29 Keypunch Character
00	Space	No Punch	Space
01	A	+1	A
02	B	+2	B
03	C	+3	C
04	D	+4	D
05	E	+5	E
06	F	+6	F
07	G	+7	G
10	H	+8	H
11	I	+9	I
12	J	-1	J
13	K	-2	K
14	L	-3	L
15	M	-4	M
16	N	-5	N
17	O	-6	O
20	P	-7	P
21	Q	-8	Q
22	R	-9	R
23	S	02	S
24	T	03	T
25	U	04	U
26	V	05	V
27	W	06	W

G-21 Internal Representation	G-21 Character	Punch Card Hole Pattern	Model 29 Keypunch Character
30	X	07	X
31	Y	08	Y
32	Z	09	Z
33	|	28	:
34	←	68	=
35	→	-78	¬
36	¬	+58	(
37	,	038	,
40	0	0	0
41	1	1	1
42	2	2	2
43	3	3	3
44	4	4	4
45	5	5	5
46	6	6	6
47	7	7	7
50	8	8	8
51	9	9	9
52	$_{10}$	078	?
53	.	+38	.
54	+	+	&
55	-	-	-
56	*	-48	*
57	/	01	/
60	=	38	#
61	∨	+78	|
62	≠	-28	!

G-21 Internal Representation	G-21 Character	Punch Card Hole Pattern	Model 29 Keypunch Character
63	∧	+68	+
64	<	-58	)
65	$	-38	$
66	>	-68	;
67	;	48*	@
70	(	048	%
71	[	058	_
72	]	068	>
73	)	+48	<
74	↓	78	"
75	↑	+28	¢
76	:	028	0-2-8
77	'	58	'
160	Tab	+-2	
161	Car Ret	+-3	
166	Bksp	-02	
167	Unlock	-03	
170	EOM	-04	
	job card $	+-89	

* Model 26 keypunch character ' .

Note: The CC-10 and teletypes do not have the character ≠, so /62 may be used as a no-op.

30. LIST OF G-21 SHIFT MULTIPLIERS

Left Shift	Number	Right Shift
1	0	000 00 00001
2	1	101 00 00004
4	2	101 00 00002
10	3	101 00 00001
20	4	102 00 00004
40	5	102 00 00002
100	6	102 00 00001
200	7	103 00 00004
400	8	103 00 00002
1000	9	103 00 00001
2000	10	104 00 00004
4000	11	104 00 00002
10000	12	104 00 00001
20000	13	105 00 00004
40000	14	105 00 00002
05 00 00001	15	105 00 00001
05 00 00002	16	106 00 00004
05 00 00004	17	106 00 00002
06 00 00001	18	106 00 00001
06 00 00002	19	107 00 00004
06 00 00004	20	107 00 00002
07 00 00001	21	107 00 00001
07 00 00002	22	110 00 00004
07 00 00004	23	110 00 00002
10 00 00001	24	110 00 00001
10 00 00002	25	111 00 00004
10 00 00004	26	111 00 00002
11 00 00001	27	111 00 00001
11 00 00002	28	112 00 00004
11 00 00004	29	112 00 00002
12 00 00001	30	112 00 00001
12 00 00002	31	113 00 00004

31. OCTAL-DECIMAL CONVERSION TABLES

Decimal	Octal
10	12
20	24
30	36
40	50
50	62
60	74
70	106
80	120
90	132
100	144
200	310
300	454
400	620
500	764
600	1 130
700	1 274
800	1 440
900	1 604
1 000	1 750
2 000	3 720
3 000	5 670
4 000	7 640
5 000	11 610
6 000	13 560
7 000	15 530
8 000	17 500
9 000	21 450
10 000	23 420
20 000	47 040
30 000	72 460

Octal	Decimal
10	8
20	16
30	24
40	32
50	40
60	48
70	56
100	64
200	128
300	192
400	256
500	320
600	384
700	448
1 000	512
2 000	1 024
3 000	1 536
4 000	2 048
5 000	2 560
6 000	3 072
7 000	3 584
10 000	4 096
20 000	8 192
30 000	12 288
40 000	16 384
50 000	20 480
60 000	24 576
70 000	28 672
100 000	32 768

32. INDEX OF SUDOS

Sudo	Section	Sudo	Section	Sudo	Section
ADC	11	LDR	26	SET	19
ALF	13	LEN	9	SIZ	27
BNC	21	LFN	16	SSC	17
BRA	17	LIN	7	SXX	14
BRD	17	LIS	7	TB1	20
BRV	17	LWD	11	TB2	20
BYP	21	MAC	17	TB3	20
CHK	9	MAD	17	TC8	26
CKS	18	MON	25	TD8	26
CLW	11	MTT	8	TIM	7
COA	14	NAM	13	TLC	26
COM	7	NON	15	TOF	7
CON	15	OCT	7	TOP	7
CPY	27	OPA	25	TYP	25
CRD	25	OPM	7	VAR	17
CSR	18	OPT	27	WRD	11
CSS	21	OUI	15	ZEC	18
DBG	18	OUT	8	ZRO	27
DEC	7	PAG	7	64K	27
DEF	9	PBC	22	***	10
DMP	7	PFC	25		
ENT	10	PFN	16		
ERA	26	PIE	18		
ERO	26	POP	17		
ERR	18	PRT	9		
EXR	26	PSH	17		
FIN	17	PUL	9		
FLG	18	PVE	7		
FPC	12	RBC	22		
FPL	12	RD8	26		
HPC	12	REF	15		
HPL	12	REL	9		
ITR	19	RET	8		
LBL	9	RXA	14		
LC8	26	SCP	10		

33. LIST OF PREDEFINED LABELS

|0 to |100	Monitor references, except:
|36	Base address of MTHAT symbol table
|37	Subroutine entry point to MTHAT
'0	Base address of a table of left shift multipliers, from ←15 to ←31, inclusive
'1	Base address of a table of right shift multipliers, from →0 to →31, inclusive
'2	Base address of a table of single bits, from $0 to $31, inclusive
'3	Base address of a table of flag bits, from F0 to F3, inclusive
'4	Base address of the alphanumeric representations of all the identifiers used by MTHAT, stored one character per word in the same order as the MTHAT identifier table
'5	Size of table '4
'6	Base address of a table of G-21 instruction mnemonics, the 3 characters of each mnemonic stored right-justified in the word
'7	Size of table '6
'8	Base address of a table of G-21 opcodes and standard modes, appearing in the same order as table '6
'9	Base address of a table of MTHAT sudo mnemonics, the 3 characters of each mnemonic stored right-justified in the word
'10	Size of table '9
'11	Base address of a table of G-21 register and line command mnemonics, stored right-justified in the word

'12	Size of table '11
'13	Base address of a table of register numbers and line commands, appearing in the same order as table '11
'14	Sudo trap transfer location
'15	Base address of MTHAT card image, stored four characters per word into 21 words
'16	Base address of MTHAT
'17	Subroutine: Convert (ACC) to label for symbolic dis-assembly
'18	Subroutine: Write a logical file table entry
'19	Subroutine: Read a logical file table entry
'20	First location after the end of storage used by the MTHAT run-time subroutines and tables
'21	Switch: Upper core request
'22	Subroutine: Write a disc or tape file
'23	Subroutine: Read a disc or tape file
'24	Subroutine: Test if (ACC) is within user memory