THE TREE-META COMPILER-COMPILER SYSTEM:

A Meta Compiler System for the Univac 1108 and the General Electric 645

C. Stephen Carr
David A. Luther
Sherian Erdmann

University of Utah

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by David A. Luther, RADC, GAFB, N.Y. 13440 under Contract No. AF30(602)-4277.

FOREWARD

This interim report describes research accomplished by Computer Science of the University of Utah, Salt Lake City, Utah, for the Advanced Research Projects Agency, administered by Rome Air Development Center, Griffiss Air Force Base, New York under Contract No. AF30(602)-4277. Secondary report number is TR 4-12. Mr. David A. Luther (EMIIO) is the RADC Project Engineer.

This technical report has been reviewed and is approved.

David A. Luther

Approved: DAVID A. LUTHER
Project Engineer

TREE META

ABSTRACT

Tree Meta is a compiler-compiler system for context-free languages. Parsing statements of the metalanguage resemble Backus-Naur Form with embedded tree-building directives. Unparsing rules include extensive tree-scanning and code-generation constructs. Examples in this report are drawn from algebraic and special-purpose languages. The process of bootstrapping from a simpler metalanguage is explored in detail.

This report is based on an earlier one by D.I. Andrews and J.F. Rulifson of Stanford Research Institute which described the SDS 940 version of Tree Meta. The Tree Meta system described in this report was bootstrapped from the SDS 940 with a minimum of hand coding.

TABLE OF CONTENTS

APPENDICES

TREE META

INTRODUCTION

1. Some Definitions

Terms such as "metalanguage" and "metacompiler" have a variety of meanings. In this report "Language," without the prefix "meta," means any formal computer language. These are generally languages like ALGOL or FORTRAN. Any metalanguage is also a language.

> A compiler is a computer program which reads a formal-language program as input and translates that program into instructions which may be executed by a computer. The term "compiler" also means a listing of the instructions of the compiler.
>
> A language which can be used to describe other languages is a metalanguage. English is an informal, general metalanguage. Backus-Naur Form or BNF (NAUR1) is a formal metalanguage used to define ALGOL. BNF is weak, for it described only the syntax of ALGOL, and says nothing about the semantics or meaning. English, on the other hand, is powerful, yet its informality prohibits its translation into computer programs.
>
> A metacompiler, in the most general sense of the term, is a program which reads a metalanguage program as input and translates that program into a set of instructions. If the input program is a complete description of a formal language, the translation is a compiler for the language.

2. Design Standards and Scope

The broad meaning of the word "metacompiler," the strong, divergent views of many people in the field, and our restricted use of the word, necessitate a formal statement of the design standard and scope of Tree Meta.

Tree Meta is built to deal with a specific set of languages; namely, those which are strictly context free in the formal sense. There is no attempt to design universal languages, or machine independent languages, or any other goals of many compiler-compiler systems.

Compiler-compiler systems may be rated on two almost independent features: the syntax they can handle and the features within the system which ease the compiler-building process.

Tree Meta parses context-free languages in a top down fashion using limited backup. Some context sensitive constructs can also be handled; i.e., flags and values and block structure in symbol tables. There is little interest, however, in dealing with such problems as the FORTRAN "continue" statement, the PL/1 "enough ends to match," or the ALGOL "is it procedure or is it a variable" question. Tree Meta is only one part of a system-building technique. There is flexibility at all levels of the system, and the design philosophy has been to reap maximum payoff rather than fight old problems.

Many of the features considered necessary for a compiler-compiler system are absent in Tree Meta. There are no features for handling multi-dimensional subscripts or higher-level macros. These features are not present because the users have not needed them. None, however, would be difficult to add.

3. Compiler Writing Schemes

There are two classes of formal-definition compiler-writing schemes.

In terms of usage, the productive or synthetic approach to language definition is the most common. A productive grammar consists primarily of a set of rules which describes a method of generating all the possible strings of the language.

The reductive or analytic technique states a set of rules which describe a method of analyzing any string of characters and deciding whether that string is in the language. This approach simultaneously produces a structure for the input string so that code may be generated.

The metacompilers are a combination in both schemes. They are neither purely productive nor purely reductive, but merge both techniques into a single system. These compilers are expressible in their own language, hence the prefix "meta."

4. Top-Down Parsing

The following is a formal discussion of top-down parsing algorithms. It relies heavily on definitions and formalisms which are standard in the literature and may be skipped by the lay reader. For a language L, with vocabulary V, non-terminal vocabulary N, productions P, and head S, the top-down parse of a string u in L starts with S and looks for a sequence of productions such that S => u (S produces u).

```
Let     V = {E, T, F, +, *, 9, ), X}
        N = {E, T, F}
        P = {E ::= T / T + F
             T ::= F / F * T
             F ::= X / ( E )}
        L = (V,N,P,E)
```

The following intentionally incomplete ALGOL procedures will perform a top-down analysis of strings in L.

a. boolean procedure E; E := if T then (if is symbol ('+') then E else true) else false; comment is symbol (arg) is a boolean procedure which compares the next symbol in the input string with its argument, arg. If there is a match, the input stream is advanced;

b. boolean procedure T; T := if F then (if is symbol ('*') then T else true) else false;

c. boolean procedure F; F := if is symbol ('X') then true else if is symbol ('(') then (if E then (if is symbol (')') then true else false) else false) else false;

Practical recognizers, as opposed to abstract systems, such as BNF, can get into infinite loops in a manner know as left recursion. The left-recursion problem can readily be seen by a slight modification of L. Change the first production to

```
E ::=T/E + T
```

and the procedure for E in the corresponding way to

E : = if T then true else if E . . . .

Parsing the string "X+X", the procedure E will call T, which calls F, which tests for "X" and gives the result "true." E is then true but only the first element of the string is in the analysis, and the parse stops before completion. If the input string is not a member of the language, T is false and the alternative E is called, which, of course, calls T again, and E loops infinitely.

The solution to the problem in Tree Meta is the repetition operator. In Tree Meta, the first production could be

```
E = T$("+" T)
```

where the dollar sign-parentheses indicate that the quantity inside the parentheses can be repeated any number of times, including zero times.

Tree Meta makes no check to ensure that the compiler it is producing lacks syntax rules containing left recursion. The use of left recursion is one of the more common mistakes made by inexperienced metalanguage programmers.

5. Tree Meta Input Language

The input lnaguage to the metacompiler closely resembles BNF. The primary difference between a BNF rule

```
<go to> :: = go to <label>
```

and a metalanguage rule

```
GOTO = "GO" "TO" .ID:
```

is that the metalanguage has been designed to use a computer-oriented character set and predefined basic entities. The REPETITION (arbitrary-number) operator and parenthesis construct of the metalanguage are lacking in BNF. For example,

```
TERM = FACTOR $(("*" / "/" / "↑") FACTOR);
```

is a metalanguage rule that would replace 3 BNF rules.

The ability of the compilers to be expressed in their own language has resulted in the proliferation of metacompiler systems. Each one is easily bootstrapped from a more primitive version, and complex compilers are built with little programming or debugging effort.

BASIC SYNTAX

CHAPTER 2

1. Syntax Rules

A metaprogram is a set of metalanguage rules. Each rule has the form of a BNF rule, with output instructions embedded in the syntactic description.

The Tree Meta compiler converts each of the rules to a set of Fortran statements.

As the rules (acting as instructions) compile a program, they read an input stream of characters one character at a time. Each new character is subjected to a series of tests until an appropriate syntactic description is found for that character. The next character is then read and the rule testing moves forward through the input.

The following four rules illustrate the basic constructs in the system. They will be referred to later by the reference numbers R1A through R4A.

```
R1A     EXP = TERM ("+" EXP / "-" EXP / .EMPTY);
R2A     TERM = FACTOR $("*" FACTOR / "/" FACTOR);
R3A     FACTOR = "-" FACTOR / PRIM;
R4A     PRIM = .ID / .NUM/ "(" EXP ")";
```

The identifier to the left of the initial equal sign names the rule. This name is used to refer to the rule from other rules. The name of rule R1A is EXP.

The right part of the rule--everything between the initial equal sign and the trailing semicolon--is the part of the rule which effects the scanning of the input.

Five basic types of entities may occur in a right part. Each of the entities represents some sort of a test which results in setting a general flag to either "true" or "false."

a. A string of characters between quotation marks (") represents a literal string. These literal strings are tested against the input stream as characters are read.

b. Rule names may also occur in a right part. If a rule is processing input and a name is reached, the named rule is invoked. R3A defines a FACTOR as being either a minus sign followed by a FACTOR, or just a PRIM.

c. The right part of the rule FACTOR has just been defined as "a string of elements," or "another string of elements." The "or's" are indicated by slash marks (/) and each individual string is called an alternative. Thus, in the above example, the minus sign and the rule name FACTOR are two elements in R3A. These two elements make up an alternative of the rule.

d. The dollar sign is the repetition operator in the metalanguage. A $ must be followed by an STEST element, and it indicates that this element may occur an arbitrary number of times (including zero). Parentheses can be used to group a set of elements into a single STEST element to be repeated. This is shown in rules R1A and R2A above.

e. In Tree Meta, three basic recognizers are "identifier" as .ID, "number" as .NUM, and "string" as .SR. Other basic recognizers are described in Section 4 on page 217. Another basic entity which is treated as a recognizer, but does not look for anything, is .EMPTY. It always returns a value of "true." Two basic entities may be seen in rule R4A. A basic recognizer is a program in Tree Meta that may be called upon to test the input stream for an occurrence of a particular entity; i.e., .ID checks for any combinations of letters and digits starting with a letter; .NUM checks for any combination of digits; and .SR checks for any combinations of letters enclosed in double quotes.

As an example, suppose that the input stream contains the string X*Y when the rule EXP is invoked during a compilation. EXP first calls rule TERM, which calls FACTOR, which tests for a minus sign. This test fails and FACTOR then tests for a plus sign and fails again. Finally, FACTOR calls PRIM, which tests for an identifier. The character X is an identifier; it is recognized and the input stream advances one character.

PRIM returns a value of "true" to FACTOR, which in turn returns to TERM. TERM tests for an asterisk and fails. It then tests for a slash and fails. The dollar sign in front of the parenthesized group of TERM, however, means that the rule has succeeded because TERM has found a FACTOR followed by zero occurrences of "* FACTOR" or "/ FACTOR." Thus, TERM returns a "true" value to EXP. EXP now tests for plus sign and finds it. The input stream advances another character.

EXP now calls on itself. All necessary information is saved so that the return may be made to the right place. In calling on itself, it goes through the sequence just described until it recognizes the Y.

Thinking of the rules in this way is confusing and tedious. It is best to think of each rule separately. For example, one should think of R2A as defining a TERM to be a series of FACTORS separated by asterisks and slashes and not attempt to think of all the possible things a FACTOR could be.

2. Parse Trees

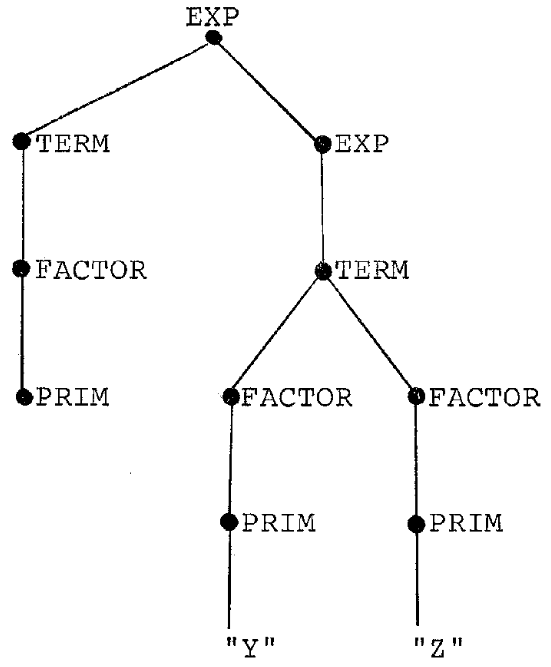
Tree Meta builds a parse tree of the input stream before producing any output. Before we describe the syntax of node generation, let us first discuss parse trees.

A parse tree is a structural description of the input stream in terms of the given grammar.

Using the four rules above, the input stream

X+Y*Z

has the following parse tree:

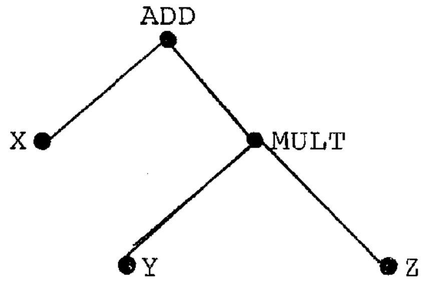


In this tree, each node is either the name of a rule or one of the primary entities recognized by the basic recognizer routines.

Also, there is a great deal of subcategorization. For example, Y is a PRIM which is a FACTOR which is the left number of a TERM. This degree of subcategorization is generally undesirable.

The tree produced by the metacompiler program is simpler than the one above, yet it contains sufficient information to complete the compilation.

The parse tree actually produced is:



In this tree, the nodes are the names of output rules which generate code.

The parse rules which produce the above tree are the same as the four previous rules with new syntax additions to perform the appropriate node generation. A colon followed by an output rule name is used in a parse rule to build a tree node. The complete rules are:

```
R1B     EXP = TERM ("+" EXP :ADD{2}/"-" EXP :SUB{2}/.EMPTY);
R2B     TERM = FACTOR $("*" FACTOR :MULT{2}/ "/" FACTOR :DIVD{2});
R3B     FACTOR = "-" FACTOR :MINUS{1} / PRIM;
R4B     PRIM = .ID / .NUM / "(" EXP ")";
```

As these parse rules scan an input stream, they perform just like the first set. As the entities are recognized, however, they are stored on a push-down stack until the node-generation element of the parse rule removes them to make trees. As an example, consider how the input stream X+Y*Z is analyzed.

EXP calls TERM, which calls FACTOR, which calls PRIM, which recognizes the X. The input stream moves forward and the X is put on a stack.

PRIM returns to FACTOR, which returns to TERM, which returns to EXP. The plus sign is recognized and EXP is again called. This is an example of a recursive call. Again EXP calls TERM, which calls FACTOR, which calls PRIM, which recognizes the Y. The input stream is advanced, and Y is put on the push-down stack. The stack now contains Y,X, and the next character on the input stream is the asterisk.

PRIM returns to FACTOR, which returns to TERM. The asterisk is recognized, and the input is advanced another character.

The rule TERM now calls FACTOR again, which calls PRIM, which recognizes the Z, advances the input stream, and puts the Z on the push-down stack. PRIM returns to FACTOR. FACTOR returns to its second call from TERM.

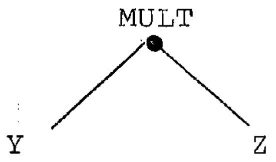
The construct :MULT is now processed. This names the next node to be put in the tree. Later we will see that, in a complete metacompiler program, there will be a rule named MULT which will be processed when the time comes to produce code from the tree. Next, the {2} in the rule TERM is processed. This tells the system to construct a portion of a tree. The branch is to have two nodes, and they are to be the last two entities recognized (they are on the stack). The name of the branch is to be MULT, since that was the last name given. The branch is constructed and the top two items of the stack are replaced by the new node of the tree.

The stack now contains:

MULT

X

The parse tree is now



Notice that the nodes are assembled in a left-to-right order, and that the original order of recognition is retained.

Rule TERM now returns to EXP, and EXP returns to the previous call on itself. The next node is named by executing the :ADD; i.e., names the next node for the tree. The{2} in rule EXP is now executed. A branch of the tree is generated which contains the top two items of the stack and whose name is ADD. The top two items of the stack are removed, leaving it as it was initially, empty. The tree is now complete, as first shown, and all the input has been passed over.

3. Unparse Rules

Now a second set of rules, the unparse rules, are applied to the tree to generate code. The unparsing rules have two functions: they produce output and they test the tree in much the same way as the parsing rules test the input stream. This testing of the tree allows the output to be based on the deep structure of the input, and, hence, better output may be produced.

3a. Output

Before we discuss the node-testing features, let us first describe the various types of output that may be produced. The following list of output-generation features in the meta-compiler system is enough for most examples.

1. The output is line-oriented, and the end of a line is determined by a carriage return. To instruct the system to produce a carriage return, one writes a backslash as an element of an unparse rule.

2. To put a tab character into the output stream, one writes a comma as an element of an output rule.

3. A literal string can be inserted in the output stream by enclosing the literal string in quotes in the unparse rule. Notice that, in the unparse rule, a literal string becomes output; while, in the parse rules, it becomes an entity to be tested for in the input stream. To output a Fortran continuation statement which has 100 as a label, one would write the following string of elements in an unparse rule:

`"100", "CONTINUE" \`

4. As can be seen in the last example of a tree, a node of the tree may be either the name of an unparse rule, such as ADD, or one of the basic entities recognized during the parse, such as the identifier X.

5. Suppose that the expression X+Y*Z has been parsed and the program is in the ADD unparse rule processing the ADD node (later we will see how this state is reached). To put the identifier X into the output stream, one writes "*1" (meaning "the first node below") as an element. For example, to generate an output line with fixed and variable parts, one would write:

`,"CALL ("*1")"\`

6. To generate the code for the left-hand node of the tree one merely mentions "*1" as an element of the unparse rule. Caution must be taken to ensure that no attempt is made to append a nonterminal node to the output stream; each node must be tested to be sure that it is the right type before it can be evaluated or output.

Generated labels are handled automatically. A label is referred to by a number sign followed by a number. Every time a label is mentioned during the execution of a rule, a label is generated, and then appended to the output stream. If one output rule calls another output rule, all the labels are saved, and new ones generated. When a return is made, the previous labels are restored.

As trees are being built during the parse phase, a time comes when it is necessary to generate code from the tree. To do this, one writes an asterisk as an element of a parse rule; for example,

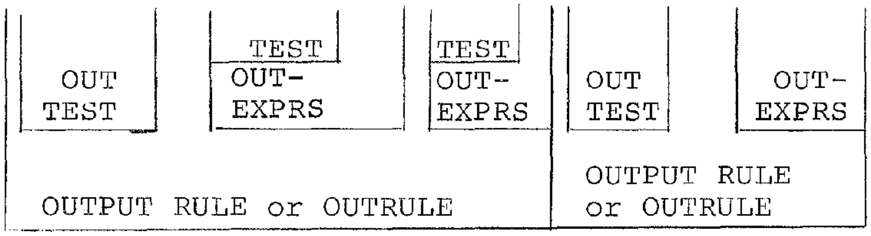
```
R5B     PROGRAM = ".PROGRAM" $(ST *) ".END";
```

which generates code for each statement (ST) after it has been entirely parsed. When the asterisk is executed, control of the program is transferred to the rule whose name is the foot (top node or last generated node) of the tree. When return if finally made to the rule which initiated the output, the entire tree is cleared and the generation process begins anew.

3b. <u>Node Testing</u>

Structurally, an unparse rule is a rule name followed by a series of output rules. The diagram of an unparse rule may be referenced while reading the following section.

```
MDX {-,.ID} => IT{*1} 'A / *2 'B  {-}  =>  *1:S;
```



Each output rule begins with a test of nodes. The series of output rules make up a set of highest-level alternatives. When an unparse rule is called, the test for the first output rule is made. If it is satisfied, the remainder of the alternative is executed; if it is false, the next alternative output rule test is made. This process continues until either a successful test is made or all the alternatives have been tried. If a test is successful, the alternative is executed and a return is made from the unparse rule with the general flag set "true." If no test is successful, a return is made with the general flag "false."

Suppose a translator is to be constructed for a language with arbitrary expressions as subscripts. For example:

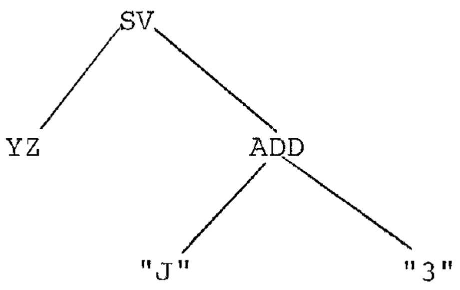
```
X(I*J - 3)
YZ(3 * K / J)
```

The target language (Fortran, for example) usually does not allow this. Fortran subscripts are normally a simple integer variable or constant optionally followed by a signed constant. For example:

```
X(I)
YZ(J + 3)
```

By building a tree before generating any output code, it is possible to detect special cases and take appropriate action. Suppose, during the parse phase, the following tree is built.

Subscripted variable



An unparse rule with four alternatives could be used to detect special cases.

```
SV{-, ADD {.ID, .NUM}} =>   (special case)   /
  {-, SUB {.ID, .NUM}} =>   (special case)   /
  {-, .ID} =>               (special case)   /
  {-, -} =>                 (general case)   ;
```

The simplest test that can be made is the test to ensure that the correct number of nodes emanate from the node being processed. The ADD rule may begin

```
ADD{-,-} =>
```

The string within the brackets is known as an out-test. The hyphens are individual items of the out-test. Each item is a test for a node. All that the hyphen requires is that a node be present. The name of a rule need not match the name of the node being processed.

1. If one wishes to eliminate the test at the head of the out-rule, one may write a slash instead of the bracketed string of items. The slash, then, takes the place of the test and is always true. Thus, a rule which begins with a slash immediately after the rule name may have only one out-rule. The rule

   ```
   MT / => .EMPTY;
   ```

   is frequently used to flag the absence of an optional item in a list of items. It may be tested in other unparse rules, but it itself always sets the general flag true and returns.

2. The nodes emanating from the node being evaluated are referred to as *1, *2, etc., counting from left to right.

To test for equality between nodes, one merely writes *i for some i as the desired item in an out-test. For example, to see if node 2 is the same as node 1, one could write either {-,*1} or {*2,-}. To see if the third node is the same as the first, one could write {-,*2,*1}. In this case, the *2 could be replaced by a hyphen.

3. One may test to see if a node is an element which was generated by one of the basic recognizers by mentioning the name of the recognizer. Thus to see if the node is an identifier one writes.ID; to test for a number one writes .NUM. To test whether the first node emanating from the ADD is an identifier and if the second node exists, one writes {.ID,-}.

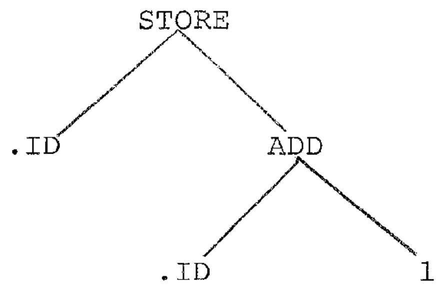
4. To check for a literal string on a node, one may write a string as an item in an out-test. The construct {-,"1"} tests to be sure that there are two nodes and that the second node is a 1. The second node will have been recognized by the .NUM basic recognizer during the parse phase.

5. A generated label may be inserted into the tree by using it in a call to an unparse rule in another unparse rule. This process will be explained later. To see if a node is a previously generated label, one writes a number sign followed by a number. If the node is not a generated label the test fails. If it is a generated label, the test is successful and the label is associated with the number following the number sign. To refer to the label in the unparse rule, one writes the number sign followed by the number.

6. Finally, one may test to see if the name matches a specified name. Suppose that one had generated a node named STORE. The left node emanating from it is the name of the variable and on the right is the tree for an expression. An unparse rule may begin as follows:

```
STORE{-,ADD{*1,"1"}} => , "MIN " *1
```

The *1 as an item of the ADD refers to the left node of the STORE. Only a tree such as



would satisfy the test, where the two identifiers must be the same or the test fails. An expression such as X ← X + 1 meets all the requirements.

3c. Out-Expressions

Each out-rule, or highest-level alternative, in an unparse rule is also made up of alternatives. These alternatives are spearated by slashes, as are the alternatives in the parse rules.

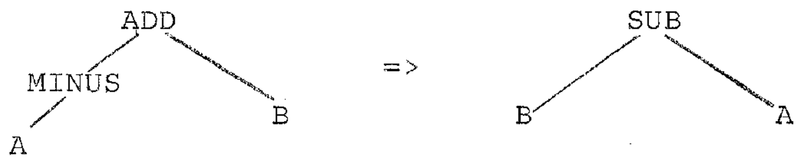
The alternatives of the out-rule are called "out-exprs." The out-expr may begin with a test, or it may begin with instructions to output characters. If it begins with a test, the test if made. If it fails, the next out-expr in the out-rule is tried. If the test is successful, control proceeds to the next element of the out-expr. When the out-expr is done, a a return is made from the unparse rule.

The test in an out-expr resembles the test for the out-rule. There are two types of these tests.

1. Any non-terminal node in the tree may be transferred to by its position in the tree rather than its name. For example, *2 would invoke the second node from the right. This operation not only transfers control to the specific node, but it makes that node the one from which the next set of nodes tested emanate. After control is returned to the position immediately following the *2, the general flag is tested. If it is "true" the out-expr proceeds to the next element. If it is "false" and the *2 is the first element of the out-expr the next alternative of the out-expr is tried. If the flag is "false" and the *2 is not the first element of the out-expr, a compiler error is indicated and the system stops.

2. The other type of test is made by invoking another unparse rule by name and testing the flag on the completion of the rule. To call another unparse rule from an out-expr, one writes the name of the rule followed by an argument list enclosed in brackets. The argument list is a list of nodes in the tree. Copies of these nodes are put on the node stack, and when the call is made, the rule being called sees the argument list as its set of nodes to analyze. For example:

```
ADD{MINUS{-},-} =>  SUB{*2,*1:*1}
```



This tree building feature maintains the substructure of the nodes being transferred, such as the structure beneath A and B.

Only nodes and generated labels can be written as arguments. Nodes are written as *1, *2, etc. To reach other nodes of the tree, one may write such things as *1:*2, which means "the second node emanating from the first node emanating from the node being evaluated." Referring to the tree for the expression X+Y*Z on page 203, if ADD is being evaluated, *2: *1 is Y. To go up the tree, one may write an "uparrow" (↑) followed by a number before the asterisk-number-colon sequence. The uparrow means to go up that many levels before the search is made down the tree. If MINUS were being evaluated, ↑1*2 would be the B.

If a generated label is written as an argument, it is generated at that time and passed to the called unparse rule so that that rule may use it or pass it on to other rules. The generated label is written just as it is in an output element; i.e., a number sign followed by a number.

The calls on other unparse rules may occur anywhere in an output expression (out-expr). If they occur in a place other than the first element, they are executed in the same way, except that after the return, the flag is tested; if it is false a compiler error is indicated. This use of extra rules helps in making the output rules more concise.

The rest of an out-expr is made up of output elements appended to the output stream, as discussed above.

Sometimes, it is necessary to set the general flag in an out-expr, just as it is sometimes necessary in the parse rules. .EMPTY may be used as an element in an out-expr at any place.

Out-exprs may be nested, using parentheses, in the same way as the alternatives of the parse rules.

4. Additional Features

Some additional features of Tree Meta make programming easier for the user.

If a literal string is but one character, one may write an apostrophe followed by the character rather than writing a quotation mark, the character, and another quotation mark. For example: 'S and "S" are interchangeable in either a parse rule or an unparse rule.

As the parse rules proceed through the input stream, they may come to a point where they are in the middle of a parse alternative and there is a failure. This may happen for two reasons: backup is necessary to parse the input, or there is a syntax error in the input. Backup will not be covered in this introductory chapter. If the syntax error occurs, the system prints out the line in error with an arrow pointing to the character which cannot be parsed. The system then stops. To eliminate this, one may provide for an error message by writing a "?" followed by a rule name. The error construct may appear after any test except the first in the parse equations. For example,

```
        ST = .ID'= $2 EQERR EXP ?3 EXERR'; ?4 SYNERR : STORE{2};
```

Suppose this rule is executing and has called rule EXP, and EXP returns with the flag false. Instead of stopping, Tree Meta prints the line in error with an arrow pointing to the offending character and an error comment which contains the number 3. The compiler then transfers control to the parse rule EXERR.

Comments may be inserted anywhere in a metalanguage program where blanks may occur. A comment begins and ends with a "%" and may contain any character except, of course, a "%."

In addition to the basic recognizers .ID, .NUM, and .SR, three others are occasionally very useful.

> The symbol .LET tests for the occurrence of a single letter, and the symbol .DIG tests for the occurrence of a single number. Also, .CHR tests for the occurrence of any single character (letter, digit, or special character).

The recognizers .CHR, .LET, and .DIG, if successful, store away a character in a special way; hence, references to it are not exactly the same as for other basic recognizers. In all three cases, the octal representation of the characters is put directly in KSTACK. In node testing, if one wishes to check for the particular occurrence of a character that was recognized by .CHR, .LET, or .DIG, one uses the single quote - character construct. If one wishes to test what <u>rule</u> recognized a character, use .CHR, .LET, or .DIG. When outputting a node which is a character, letter, or digit, one adds :C to the node indicator. For example, *1:C outputs all characters, whether recognized by .CHR, .LET, or .DIG.

When a compilation is very simple, it may be cumbersome to build a parse tree and then output from it; hence, the ability to output directly from parse rules is available.

The syntax for direct output from parse rules is generally the same as for unparse rules. The output expression is written within square brackets. (See formal description, p. 312.) The items from the input stream which normally are put in the parse tree may be copied to the output stream by referencing them in the output expression. The most recent item recognized is referenced as * or *S0. Items recognized previous to that are *S1, *S2, etc., counting in reverse order--that is, counting down from the top of the KSTACK in which they are kept. Other characteristics of the items such as length, number, character may be put in the output stream as in unparse rules by L, N, C, respectively; i.e., *S1L will output the length of the item S1.

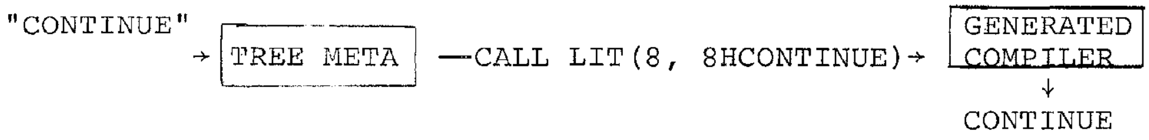
Normally, the items are removed from the stack and put into the tree; however, if they are just copied directly to the output stream, they remain in the KSTACK. They are removed by writing an "&" at the end of the parse rule (just before the ;). This causes all input items added to the KSTACK by that rule to be removed. The input stack is, thus, the same as it was when the rule was called.

In addition to the previous means of outputting code, another exists which permits output in a more immediate sense into the body of code which is the generated compiler. Remember that the basic function of Tree Meta is to output a body of code (symbolic Fortran statements) which acts as a compiler for some user-defined language.

As a specific example of this process, consider the parse construct "CONTINUE" which generates for the user's compiler: CALL TST(8, 8HCONTINUE).

The same construct in an unparse rule generates the Fortran output: CALL LIT(8, 8HCONTINUE).

Both Fortran statements are executed when the generated compiler is running. The LIT subroutine, for example, outputs the eight characters CONTINUE in the output stream.



Sometimes, the Tree-Meta user wants to output code immediately from Tree Meta having the result executed in the generated compiler instead of being executed as code that the generated compiler has output. Thus, for example,

```
!(,"REWIND"/)
```

causes a Fortran rewind statement to be inserted directly into the generated compiler. This statement would be executed "immediately" as the generated compiler is being executed instead of being "deferred" for execution one step later in the program the generated compiler generates.

FORMAL DESCRIPTION

CHAPTER 3

This chapter is a formal description of the complete Tree Meta language. It is designed as a reference guide, not as a training manual.

1. Programs and Rules

Syntax

```
program = (".META"  .ID  size/".CONTINUE"  .ID)
                 (".LIST"/.EMPTY) $(rule)".END";

size = '(siz $(', siz) / .EMPTY;

siz = .CHR '= .NUM ;

rule = .ID  ('= exp('&/ .EMPTY) / '/"=  "genl /
outrul) '; ;
```

Semantics

A file of symbolic Tree Meta code may be either an original main file or a continuation file. A compiler may be composed of any number of files, but there may be only one main file.

The mandatory identifier fillowing the string .META in a main file names the rule at which the parse will begin, and is also the name of the Fortran symbolic element produced.

The optional .LIST refers to a listing of

(1) code - output code from TREE META (the generated compiler)

(2) source - the input to TREE META (compiler

specifications)

The options are:

.LIST OFF :no listing
.LIST :list both source and code
.LIST SOURCE :list source, no code
.LIST CODE :list code, no source

If not specified, TREE META lists code and source. The list option can be used anyplace an NTEST is used.

> The size construct sets the allocation parameters for the three stacks and string storage for use by the generated compiler. The default sizes are those used by the Tree Meta compiler. M,K,N, and S are the only valid characters; the size is something which must be determined by experience. The maximum number of cells used during each compilation is printed out at the end of the compilation.

When a file begins with .CONTINUE, no initialization or storage-allocation code is produced.

There are three different kinds of rules in a Tree Meta program. All three begin with the identifier which names the rule.

1. Parse rules are distinguished by the = following the identifier. If all the elements which generate possible nodes during the execution of a parse rule are not built into the tree, they must be popped from the kstack by writing an ampersand immediately before the semicolon.
2. Rules with the string /= following the identifier may be composed only of elements which produce output. There is

no testing of flags within a rule of this type.

3. Unparse rules have a left bracket following the identifier. This signals the start of a series of node tests.

2. Expressions

Syntax

```
exp = '  suback ('/ exp / EMPTY) / subexp ('/ exp / EMPTY);
suback = ntest (suback / .EMPTY) / stest (suback / .EMPTY) ;
subexp = (ntest / stest)  (noback / .EMPTY) ;
noback = (ntest / stest ('? .num (.id / '?) / .EMPTY) )
         (noback /.EMPTY) ;
```

Semantics

The expressions in parse rules are composed entirely of ntest, stest, and error-recovery constructs. The four rules above, which define the allowable alternation and concatenation of the test, are necessary to reduce the instructions executed when there is no backup of the input system. Tree Meta users can control the use of back up in their generated compilers on a subexpression by subexpression basis.

An expression is essentially a series of subexpressions separated by slashes. Each subexpression is an alternative of the expression. The alternatives are executed in a left-to-right order until a successful one is found. The rest of that alternative is then executed and the rule returns to the rule which invoked it.

The subexpressions are series of tests. Only subexpressions which being with a left arrow are allowed to back up the input stream and rescan it.

If any STEST other than the first within the subexpression fails, three possibilities exist. The course of action is determined by the following syntax for the error code:
" ' ? NUM (.ID / '?)".

(1) If one question mark is present, the system prints the number following the "?" in the error code.
If the optional identifier is given eg: "? 21 RULE 1", the system then transfers control to that rule; if another "?" is given instead of the optional identifier eg: "?21?, the system stops.

(2) If a backup arrow is used ("<-"), the input stream is backed up to try another subexpression.

(3) If both error code and back-up arrow are absent, the system prints an error comment and stops. Thus, both error codes and back-up arrows may only be used with subexpressions of more than _one_ STEST. (i.e., the two rules below are _not_ valid):

```
RULE 1 = '* ?21E ;
RULE 2 = ← STEST ;
```

If the test fails, the input stream is restored to the position it had when the subexpression began to test the input stream and the next alternative is tried. The input stream may never be moved back more characters than are in the ring buffer (5000). Normally, backup is over identifiers or words and the buffer is long enough.

3. Elements of Parse Rules

Syntax

```
ntest = ': .ID/'[ (.NUM '] / genp'] /'*/ list/ " = >
```

```
list = ".LIST"("SOURCE"/"CODE"/"OFF" / .EMPTY) ;

genp = genpl / .empty;

genpl = genp2  (genpl / .empty);

genp2 = '* ('S.num / .empty) ('L / 'C / 'N / .empty / genu;

comm = ".EMPTY" / '! (.SR / 'itst'));

itst = (.SR/'\/',/''+.CHR/"#1"/"#2"/ "#3"/ '$ .ID)

       itst / .EMPTY);

stest = '. .ID ( '((INSIDEPAREN/.EMPTY)')/.EMPTY)/

    .ID/

    .SR/

    '( EXP ') /

    '+ stest/

    (.NUM/.EMPTY) '$ (.NUM/.EMPTY) stest/

    '-stest/

    "--" stest ;
```

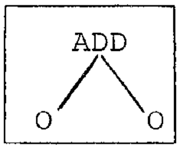
Semantics

The ntest elements of a parse rule cannot change the value of the general flag and, therefore, need not be followed by flag-checking code in the compiler.

The ': .ID construct (:XX) creates a new node in the tree with the name XX. The identifier used must be the name of an unparse rule.

example: :ADD{2}

:ADD creates a new node called ADD.{2} grabs two items off the kstack and attaches them to the above node; as



The {nnn} construct grabs nnn number of items off the kstack, and attaches them to the node last created.

The {genp} is used to write output into the normal output stream during the parse phase of the compilation without building trees. For description of GENU, see section 6.

An asterisk causes the rule currently in execution to perform a subroutine call to the rule named by the top of the tree.

The " =>" STEST construct will cause the input stream to be scanned to the occurrence of any STEST.

The comm elements are common to both parse and unparse rules.

The .EMPTY in any rule sets the general flag true.

The exclamation point construct "!" preceding a string or any ITST can be used to insert code directly into the compiler being produced. ITST provides for the insertion of a string, a comma, a single character, generated labels, or !($ .ID) inserts the statement label of the rule named by .ID. An example of several of the construct is:

```
!(."GO TO " #1\#1, "CONTINUE\)
```

This will output:

```
      GO TO 1023
 1023 CONTINUE
```

Stests always test the input stream for a literal string or a basic entity. If the entity is found, it is removed from the input stream and stored in string storage. Its position in string storage is saved on a push-down stack so that the entity may later be added as a terminal node to the tree.

An .ID construct provides a standard subroutine call to the identifier. Supplied with the Tree Meta library are subroutines for .ID, .NUM, .SR, .CHR, .LET, and .DIG which check for identifier, number, string, character, letter, and digit respectively.

To generate a call to a subroutine other than the ones above, the '. .ID must be followed immediately by an argument list in parentheses. The argument list may be empty (i.e. .COL(72) and .BLANKC() would generate CALL COL (72) and CALL BLANKC, respectively).

An identifier by itself produced a call to the rule with the name of the identifier via the MCALL subroutine.

A literal string merely tests the input stream for the string. If it is found, it is discarded. The apostrophe-character construct functions like the literal string, except that the test is limited to one character. The apostrophe construct will examine the input stream for the first non-blank character and test it with the character immediately following the apostrophe.

A "+" before any STEST item prevents skipping leading spaces. For example, 'A +.CHR will pick up the next character following the "A" in the input, even if it is a space. Notice that + ɓ will test the next character in the input stream for a blank.

The number-$-number construct is the repetition operator of Tree Meta. m$n preceding an stest element in a parse rule means that there must be between m and n occurrences of the next element coming up in the input. The default options for m and n are zero and infinity, respectively.

The hyphen ("-") construct before any STEST item tests to see if the STEST items is <u>not</u> in the input stream. For example, -'* .CHR will pick up any character except *. Any items that are put on the kstack during the test are removed after the test. Thus, -('* .ID) would not leave the identifier on the kstack. The pointers are restored after the test has been completed. The "-" test may be nested to any level: -('*-('* .ID)). The construct "--" before any STEST item tests to see if the STEST item is there, without moving the input pointer. Thus, --'* .CHR will pick up only an *.

4. Unparse Rules

Syntax

```
    outrul = '{outr (outrul / .EMPTY);
    outr = items '} "=>" outexp;
    items = item (', items /.EMPTY);
    item = '- / .ID '{items}'   / nsimpl / '. .ID / .SR/
        ''+ .CHR/'#.NUM;
```

Semantics

The unparse rules are similar to the parse rules in that they test something and return a true or false value in the general flag. The difference is that the parse rules test the input stream, delete characters from the input stream, and build a tree, while the unparse rules test the tree, collapse sections of the tree, and write output.

There are two levels of alternation in the unparse rules. The highest level is not written in the normal style of Tree Meta as a series of expressions separated by slashes; rather, it is written in a way intended to reflect the matching of nodes and structure within the tree. Each unparse rule is a series of these highest-level alternations. The tree-matching parts of the alternations are tried in sequence until one is found that successfully matches the tree. The rest of the alternation is then executed. There may be further tests within the alternation, but not complete failure as with the parse rules.

The syntax for a tree-matching pattern is a left bracket, a series of items separated by commas, and a right bracket. The items are matched against the branches emanating from the current top node. The matching is done in a left-to-right order. As soon as a match fails, the next alternation is tried.

If no alternation is successful, a false value is returned.

Each item of an unparse alternation test may be one of seven different kinds of test.

1. A hyphen is merely a test to be sure that a node is there. This sets up appropriate flags and points so that the node may be referred to later in the unparse expression if the complete match is successful.

2. The name of the node may be tested by writing an identifier which is the name of a rule. The identifier must then be followed by a test on the subnodes.

3. A nonsimple construct, primarily an asterisk-number-colon sequence, may be used to test for node equivalence. Note that this does not test for complete substructure equivalence, but merely to see if the node being tested has the same name as the node specified by the construct.

4. The .ID, .NUM, .CHR, .LET, .DIG, or .SR checks to see if the node is terminal and was put on the tree by an identifier recognizer, number recognizer, etc., during the parse phase. This test is very simple, for it merely checks a flag in the upper part of a word.

5. If a node is a terminal node in the tree, and if it has been recognized by one of the basic recognizers, it may be tested against a literal string. This is done by writing the string as an item. The literal string does not have to be put into the tree with an .SR recognizer; it can be any string in string storage, put in with .SR, .NUM, or .ID.

6. If the node is terminal and was generated by the .CHR, .LET, or .DIG recognizers, it may be matched against another specific character by writing the apostrophe-character construct as an item.

7. Finally, the node may be tested to see if it is a generated label. The labels may be generated in the unparse expressions and then passed down to other unparse rules. The test is made writing a "#"-number construct as an item. If the node is a generated label, not only is this match successful, but the label is made available to the elements of the unparse expression as the number following the "#."

5. Unparse Expressions

Syntax

```
outexp = subout ('/outexp / .empty);
subout = outt (rest / .empty) /rest;
```

```
rest = outt (rest / .empty) / gen (rest / .empty);
outt = .id '{arglst '} / '(outexp ') / nsimpl (':
     (S / L / N / C) / empty);
arglst = argmnt (', arglst / .empty) /.empty;
argmnt = nsimp / '# .num;
nsimpl = '↑ .NUM nsimp / nsimp;
nsimp = '* .num ( ': nsimp / .empty);
genl = (out / comm) (genl / .empty);
gen = comm /genu / '< / '> ;
```

Semantics

The rest of the unparse rules follow more closely the style of the parse rules. Each expression is a series of alternations separated by slash marks.

Each alternation is a test followed by a series of output instructions, calls of other unparse rules, and parenthesized expressions. Once an unparse expression has begun executing calls on other rules, elements may not fail; if they do a compiler error is indicated and the system stops.

The first element of the expression is the test. This element is a call on another rule, which returns a true or false value. The call is made by writing the name of the rule followed by a series of nodes. The nodes are put together to appear as part of the tree, and when the call is made, the unparse rule called views the nodes specified as the current part of the tree, and thus the part to match against and process.

Two kinds of things may be put in as nodes for the calls. The simplest is a generated label. This is done by writing a "#" followed by a number. Only the numbers

1, 2, and 3 may be used in the current system. If a label has not yet been generated, one is made up. This label is then put into the tree.

Any already constructed node may also be put into the the tree in this new position. The old node is not removed--rather, a copy is made. The substructure of the nodes being transferred is maintained. An asterisk-number construct refers to nodes in the same way as the highest-level alternation.

This process of making new structures from the already-existing tree is a very powerful way of optimizing the generated compiler and condensing the number of rules needed to handle compilation.

The rest of the unparse expression is made up of output commands, and more calls on unparse rules. As noted above, if any except the first call of an expression fails, a compiler error is indicated and the system stops.

The asterisk-number-colon construct is used frequently in the Tree Meta system. It appears in the node-matching syntax as well as in the form of an element in the unparse expressions. When it is in an expression, it must specify a node which exists in the tree.

If the node specified is the name of another rule, then control is transferred to that node by the standard subroutine linkage.

If the node is terminal, then the terminal string associated with the node is copied onto the output stream.

The simplest form of the construct is an asterisk followed by a number, in which case the node is found by counting the appropriate number of nodes from left to right. This may be followed by a colon-asterisk-number construct which means to go down one level in the tree after performing the asterisk-number choice and count over the number of nodes specified by the number following the colon. This process may be repeated as often as desired, and one may therefore go as deep as one wishes. All of this specification may be preceded by an ↑-number construct which means to go up in the tree, through parent nodes, a specified number of times before starting down.

After the search for the node has been completed, a number of different types of output may be specified if the node is terminal. There is a compiler error if the node is not terminal.

:s puts out the literal string.

:L puts out the length of the string as a decimal number.

:N puts out the string-storage index pointer if the node is a string-storage element; otherwise, it puts out the decimal code for the node if it is a .CHR node. The 1108 version adds 1000 to the number before it is output.

:C puts out the character if the node was constructed with a .CHR, .LET, or .DIG recognizer.

6. Output

Syntax

```
    genu = out / '. .ID '( ( INSIDEPAREN / .EMPTY) ') /
       ' # .NUM (':/ .EMPTY);
    out = ('\/ ', / .SR / ''+ .CHR / "+w" / "-w" /
       ".w" / "Δw";
```

Semantics

The standard primitive output features include the following:

1. Write a carriage return with a backslash.

2. Write a tab with a comma.

3. Write a literal string by giving the literal string.

4. Write a single character using the apostrophe-character construct.

5. Write references to temporary storage by using a working counter. Three types of action may be performed with the counter. +W adds one to the counter, -W subtracts one from the counter, and .W writes the current value of the counter onto the output stream without changing it. Finally, ΔW writes the maximum value that the counter ever reached during the compilation.

The : .ID '((INSIDEPAREN/.EMPTY') is used to generate a call to a subroutine. For example, .CERR (5(X,Y)) generates a call to the subroutine CERR with the argument 5(X,Y).

#N means "define generated label N at this point in the program being compiled." (N may be 1,2, or 3). If a colon is written directly after the generated label (#2:), Tree Meta writes the generated label in the output stream followed by a CONTINUE statement. This construct is added only to save space and writing.

PROGRAM ENVIRONMENT

CHAPTER 4

When a Tree Meta program is compiled by the metacompiler, a Fortran version program is generated. However, it is not a complete program since several routines are missing. All Tree Meta programs have common functions such as reading input, generating output, and manipulating stacks. It would be cumbersome to have the metacompiler duplicate these routines for each program, so they are contained in a library package for all Tree Meta programs. The library of routines must be loaded with the compiled Fortran version of the Tree Meta program to make it complete.

The environment of the Tree Meta program, as it is running, is the library of routines plus the various data areas.

This section describes the environment in its three logical parts: input, stack organization, and output.

1. Input Machinery

The input stream of text is broken into lines and put into an input buffer. Carriage returns in the text are used to determine the ends of lines. Any line longer than 72 characters is broken into two lines. This line orientation is necessary for syntax error reporting, a possible anchor mode, and a source listing option.

It is the job of routine RLINE to fill the input line buffer. If the listing flag is on, RLINE copies the new line to the output file. There is a buffer pointer which indicates which character is to be read from the line buffer next, and RLINE resets the pointer to the first character of the line.

Input characters for the Tree Meta program are not obtained from the input line buffer, but from an input window, which is actually a character ring buffer. Such a buffer is necessary for backup. There are three pointers into the input window. NCCP points to the next character to be read by the program. This may be moved back by the program to effect backup. MCCP is never changed except by a library routine when a new character is stored in the input window. NCCP is used to compute the third pointer, the input-window pointer IW. Actually, NCCP and MCCP are counters, and only IW points into the array, which is the character ring buffer. MCCP is never backed up and always indicates the next position in the window where a new character must be obtained from the input line buffer. Backup is registered in IBCK and is simply the difference between NCCP and MCCP. IBCK is always negative or zero.

There are several routines which deal directly with the input window.

> The routine PUTIN takes the next character from the input line buffer and stores it at the input-window position indicated by IW. This involves incrementing the input-buffer pointer, or calling RLINE if the buffer is empty. PUTIN does not change IW.
>
> The routine INC is used to put a character into the input window. It increases IW by one by calling a routine, UPIWP,

which makes IW wrap around the ring buffer correctly. If there is backup (i.e., if IBCK is less than 0), IBCK is increased by one and INC returns, since the next character is in the window already. Otherwise, MCCP is increased by one, and PUTIN is called to store the new character.

A routine called INCS is similar to INC except that it skips all blanks or comments which may be at the current point in the input stream. This routine implements the comment and blank deletion for .ID, .NUM, .SR, and other basic recognizers. INCS first calls INC to get the next character and increment IW. From then on, INC is called successively until INC returns with a non-blank character. The nonblank character is then compared with a comment character. When the end of the comment is located, INCS returns to its blank-skipping loop.

Note that comments do get into the input window, but the printer IW skips past them.

Before beginning any input operation, the IW pointer must be reset, since the program may have set NCCP back. The routine WPREP computes the value of IBCK from NCCP-MCCP. This value must be between 0 and the negative of the window size. IW is then computed from NCCP modulo the window size.

The program-library interface for inputting items from the input stream consists of the routines, ID, NUM, SR, LET, DIG, and CHR. The first three are quite similar. ID is typical of them, and works as follows: First MFLAG is set false. WPREP is called to set up IW, then INCS is called to get the first character. If the character at IW is not a letter, ID returns (MFLAG is still false); otherwise, a loop to input over letter-digits is executed. When the letter-digit test fails, the flag is set true, and the identifier is stored in the string storage area. The class of characters is determined by an array (indexed by the character itself) of integers indicating the class. Before returning, ID calls the routine, STORE which updates NCCP to the last character read in (which was not part of the identifier). That is, NCCP is set to MCCP + IBCK - 1.

The occurrence of a given literal string in the input stream is tested for by calling routine TST. The character count and the string are passed as arguments. TST deletes leading blanks and inputs characters, comparing them one at a time with the characters of the literal string. If at any point the match fails, TST returns false. Upon reaching the end of the string, TST sets the flag true, sets NCCP to MCCP - 1 + IBCK, and returns. In addition to TST, there is a simple routine to test for a single character string (TCH). It inputs one character (deleting blanks), compares it to the given character and returns false, or adjusts NCCP and returns true.

2. Stacks and Internal Organization

Three stacks are available to the program. A stack called MSTACK (MARK-STACK) is used to hold return locations and generated labels for the program's recursive routines. Another stack, called KSTACK (KEEP-STACK), contains references to input items. When a basic recognizer is executed, the reference to that input item is pushed into KSTACK. The third stack is called NSTACK (NODE-STACK), and contains the actual tree. The three stacks are declared in the Tree Meta program rather than the library: the program determines the size of each.

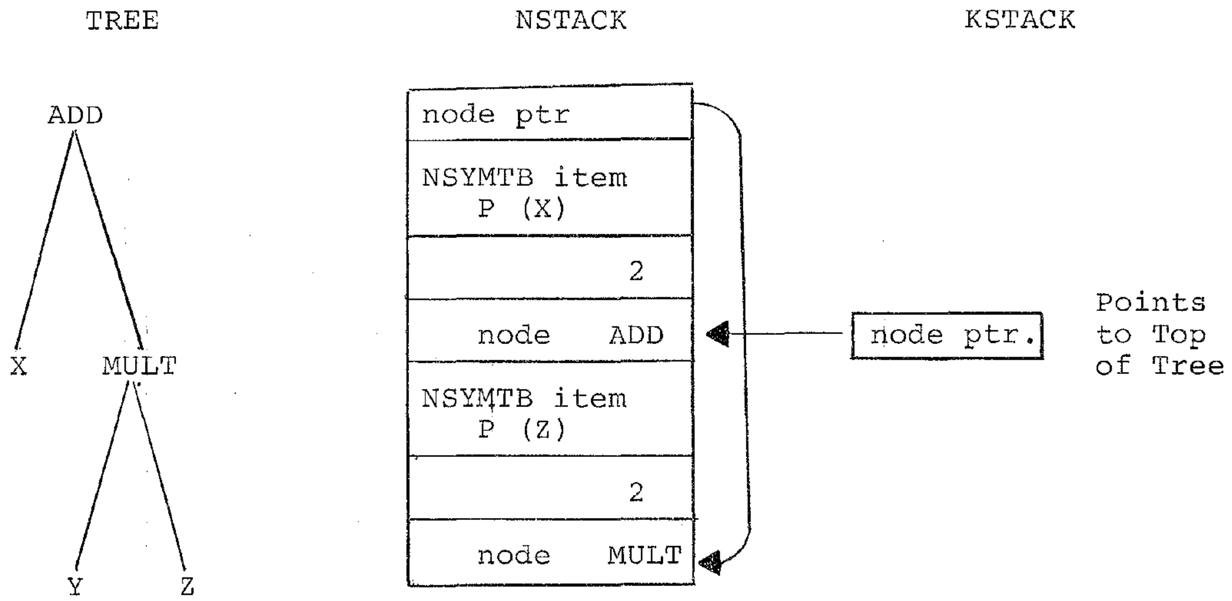
At the beginning of each routine, the location that the routine was called from and space for the generated labels are placed on the MSTACK. The routine is then free to generate labels or call other routines. The routine ends by popping up the generated labels from MSTACK and returning to the location on the top of MSTACK.

KSTACK contains single-word entries. Each entry will eventually be placed in NSTACK as a node in the tree. The format of the node words is as follows: There are two kinds of nodes, terminal and nonterminal. Terminal nodes are references to input items. Nonterminal nodes are generated by the parse rules, and have names which are names of output rules.

A terminal node is a 36-bit word with either a string-storage index or a character in the address portion of the word, and a flag in the top part of the word. The flag indicates which of the basic recognizers (.ID, .NUM, .SR, .LET, .DIG, or .CHR) read the item from the input stream.

A nonterminal node consists of a word with the address of an output rule in the address portion, and a flag in the top part which indicates that it is a nonterminal node. A node pointer is a word with an NSTACK index in the address and a pointer flag in the top part of the word. Each nonterminal node in NSTACK consists of a nonterminal node word followed by a word containing the number of subnodes on that node, followed by a terminal node word or node pointers for each subnode.

For example,



KSTACK contains terminal nodes (input items) and nonterminal node pointers which point to nodes already in NSTACK. NSTACK contains nonterminal nodes.

String Storage is another stack-like area. All the items read from the input stream by the basic recognizers (except .CHR, .LET, .DIG) are stored in the string-storage area NSYMTB. An index into NSYMTB points to the character count for a string.

All the items read from the input stream by the three basic recognizers, .ID, .NUM, .SR are stored in the string storage area NSYMTB. As well as the character string which was recognized, three other items make up the entire entry for any string. They are a value entry, a flag entry, and an entry indicating the string character count.

declared in the Tree Meta program rather than the library: the program determines the size of each.

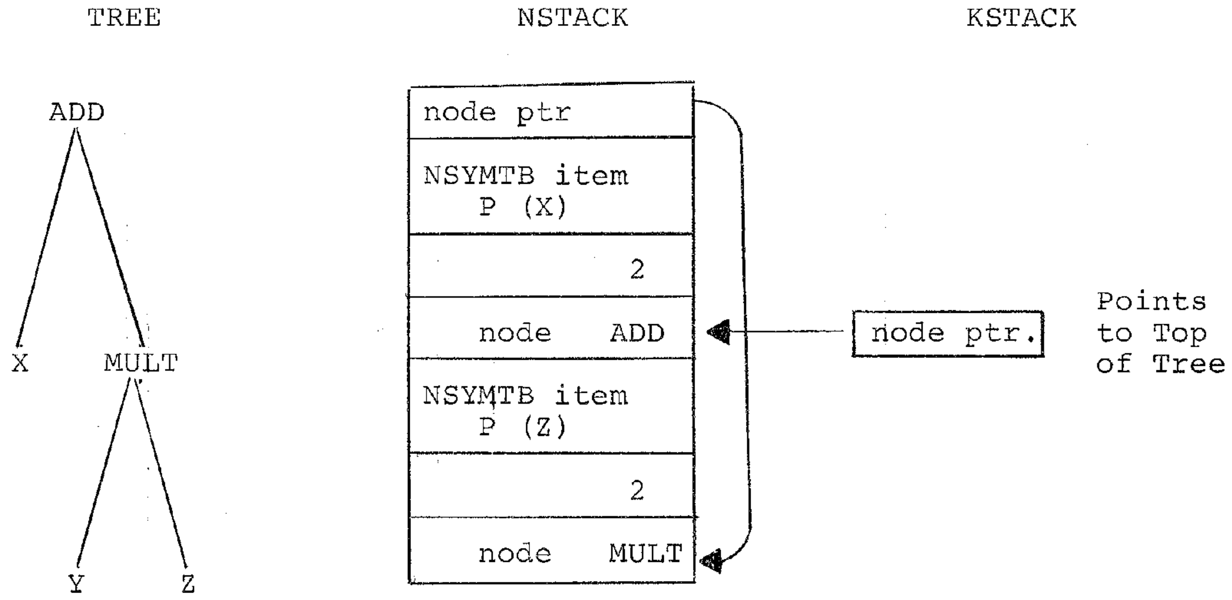
At the beginning of each routine, the location that the routine was called from and space for the generated labels are placed on the MSTACK. The routine is then free to generate labels or call other routines. The routine ends by popping up the generated labels from MSTACK and returning to the location on the top of MSTACK.

KSTACK contains single-word entries. Each entry will eventually be placed in NSTACK as a node in the tree. The format of the node words is as follows: There are two kinds of nodes, terminal and nonterminal. Terminal nodes are references to input items. Nonterminal nodes are generated by the parse rules, and have names which are names of output rules.

A terminal node is a 36-bit word with either a string-storage index or a character in the address portion of the word, and a flag in the top part of the word. The flag indicates which of the basic recognizers (.ID, .NUM, .SR, .LET, .DIG, or .CHR) read the item from the input stream.

A nonterminal node consists of a word with the address of an output rule in the address portion, and a flag in the top part which indicates that it is a nonterminal node. A node pointer is a word with an NSTACK index in the address and a pointer flag in the top part of the word. Each nonterminal node in NSTACK consists of a nonterminal node word followed by a word containing the number of subnodes on that node, followed by a terminal node word or node pointers for each subnode.

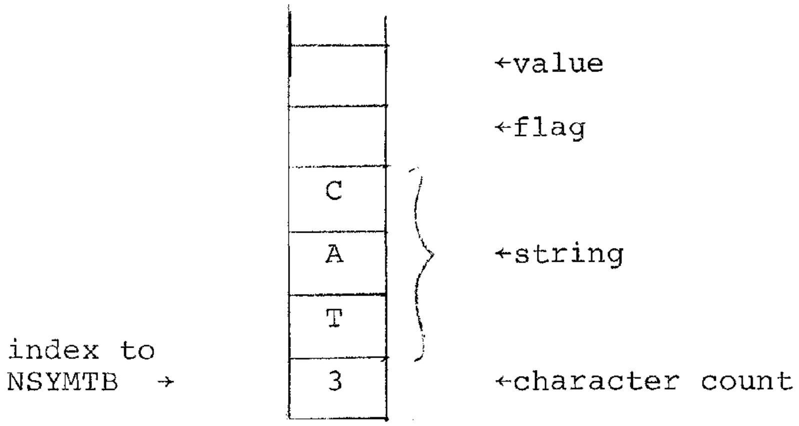
For example,



KSTACK contains terminal nodes (input items) and nonterminal node pointers which point to nodes already in NSTACK. NSTACK contains nonterminal nodes.

String Storage is another stack-like area. All the items read from the input stream by the basic recognizers (except .CHR, .LET, .DIG) are stored in the string-storage area NSYMTB. An index into NSYMTB points to the character count for a string.

All the items read from the input stream by the three basic recognizers, .ID, .NUM, .SR are stored in the string storage area NSYMTB. As well as the character string which was recognized, three other items make up the entire entry for any string. They are a value entry, a flag entry, and an entry indicating the string character count.



A search of NSYMTB proceeds from the bottom up--or the last word stored. The bottom up search combined with the appropriate settings of the flag entries facilitates block storage of variables, as in ALGOL.

Tree Meta provides two routines for setting and testing the flag-word in NSYMTB. TURN is used to set the bit pattern of the flag; for example, TURN(40,-40) would turn "on" the 30th bit of the flag word in the last string referenced by ISTAR. TEST is used to test for a particular flag; for example, TEST(40,-60) returns MFLAG = 1 if the 30th bit of the flag word is "1" or "on," and the 31st is "0" or "off."

TEST and TURN are implemented as follows: if B is the first argument and M the second (the mask), then:

TURN: FLAG = OR(B,AND(M,FLAG))

TEST: compare B with AND(B,OR(M,FLAG))

Thus, TURN(40,-40) considers the arguments as octal numbers, which when converted to binary and arranged as above, leave:

```
BIT     0                30    35
M       0 . . . . . . . 011111
FLAG    0 . . . . . . . 000000
     ______________________________
        0 . . . . . . . 000000
B       0 . . . . . . . 100000
FLAG    0 . . . . . . . 100000
```

The test routine works in similar fashion as arranged in the way described.

Other routines perform housekeeping functions like packing and unpacking strings, etc. There are three error-message writing routines to write the three types of error messages (syntax, system, and compiler). The syntax error routine copies the current input line to the output and gives the line number. A routine called FINISH closes the files, writes the number of cells used for each of the three stack areas (KSTACK, MSTACK, NSTACK) and the number of characters read, and terminates the program.

> At many points in the library routines, parameters are checked, and if they are out of bounds, the system error routine is called. This routine writes a number indicating what the error is and terminates the program. The error codes are listed in Appendix C.

Additional library subroutines generate labels, save and restore labels and return addresses on MSTACK, compare flags in NSTACK, generate nodes on NSTACK, etc.

3. Output Facilities

The output from a Tree Meta program consists of a string of characters. The output facilities available to the program consist of a set of routines to append characters, strings, and numbers to the output stream.

A string in NSYMTB can be written on the output stream by calling the routine OUTS with the NSYMTB index for that string in ISTAR. OUTS checks the NSYMTB index and generates a system-error message if it is not reasonable.

A literal string of characters is written by calling the routine LIT. The arguments are of the same form as TST.

A number is written using the routine OUTN. The binary representation is given and is written as a signed decimal integer.

All of the above routines keep track of the number of characters written on the output stream NU. Based on this count, a routine called TAB will output enough spaces to advance the current output line to the next tab stop. Tabs are set at 10-character intervals. The routine CRLF will affect a carriage return and a line feed and CIO will reset NU.

The Tree Meta system provides a routine that is very convenient for debugging. This routine, METSTA, will print out the state of the system at the point of being called. METSTA will print the information in the three internal stacks, the line currently in the input buffer and output buffer, the values of the character pointers, the symbol table, MFLAG, ISTAR, and several other items.

A DETAILED EXAMPLE

CHAPTER 5

```
.META EXAMPL  (name of element on unit D)
% TREE-META PROGRAM EXAMPLE %
.LIST SOURCE

EXAMPL = !"MFLAG=1" !"ID1ST=2" % IN 1108 TRMETA, THIS FLAG (ID1ST)
        MUST BE SET SO THAT THE PRE-DATA WILL NOT BE TRANSFERRED
        TO THE OUTPUT STREAM BY THE 'EXAMPLE' COMPILER.  THE MFLAG IS
        SET SO THAT TRMETA WILL NOT FAIL AFTER THE FIRST STEST.
        LOOK AT GENERATED CODE FOR EXAMPLE OF THIS %
        "EXAMPLE" .ID % ID WILL BE THE NAME OF THE PROGRAM %
        (FLAG/.EMPTY)
        NEXTGF ;

%******************************************************************%
% THE FOLLOWING PARSE RULES PROCESS THE BEGINNING OF THE CARDS %

NEXTGF = $(STATEMENT *)
         "END" ?1? {\, "END" } .STPMTA() :

STATEMENT = -"END" {\} =>.COL(1)
            (COMMENTCARD/LABEL (GFSTA/FLAG/FORSTA)?2E ;

COMMENTCARD = +'C {'C} .SET() => .BLANKC() .COPY() ;
         %  -SET AND COPY PASS STRINGS THROUGH TM DIRECTLY TO OUTPUT STREA
            -BLANKC TESTS FOR THE REST OF THE CARD BLANK. %

LABEL = .NUM {*S0} 1$(+' {'  }) ?3E / 7$(+'   ;   ) ;

FORSTA = .CHR {*S0C} .SET() => .BLANKC() .COPY() ;

GFSTA = MAKEBREAK/LOCAL/GFOR:

E = .EMPTY .RESET() =>'; $(STATEMENT *) "END" ?99E {\. "END"}.STPMTA();

%******************************************************************%
            % THESE PARSE RULES BUILD A TREE %

MAKEBREAK = "MAKE " ASSEXP ?10E  :MAKE{1}/
            "BREAK " ASSEXP ?11E :BREAK{1} ;

ASSEXP = ITEM '* ?12E ITEM ?13E '= ?14E ITEM ?15E :TRIPLE{3} ;

ITEM = '? :FLAGWORD{0} / PRIM ;

PRIM = <- .NUM '. .NUM :REALNUM{2} /
       .NUM /
       VAR /
       '' +.CHR ?20E $(-'' +.CHR :DO{2})  '' ?21E :LIT{1} /
       ;( EXP ?22E ') ?23E :PAR{1} ;
```

```
VAR = .ID ('( EXP ?30E $(', EXP ?31E :COMMA{2}) ') :SUBVAR{2}/.EMPTY);

EXP = TERM ( '+ EXP ?40E :PLUS{2}/'- EXP ?41E :MINUS{2}/.EMPTY) ;

TERM = FACTOR $('* FACTOR ?42E :MULT{2}/'/ FACTOR ?43E :DIVIDE{2}) ;

FACTOR = '- FACTOR ?40E :NEG{1} / PRIM ;

LOCAL = "LOCAL" .ID :LOC{1} ;

%*******************************************************************%
          % UNPARSE RULES FOR THE ABOVE PARSE RULES %

BREAK{-} => "CALL BREAK (" *1 ') ;
 MAKE{-} => "CALL MAKE(" *1 ') ;
TRIPLE{-,-,-} => *1 ', *2 ', *3 ;
    DO{-,-} => DOTST{*1} DOTST{*2} ;
DOTST{.CHR} => *1:C
      {-} => *1 ;
DIVIDE{-,-} => *1 '/ *2 ;
COMMA {-,-} => *1 ', *2 ;
PLUS{-,-} => TNEG{*2} MINUS{*1;*2:*1}/
             *1 '+ *2 ;
MINUS{-,NEG{}} => PLUS{*1,*2:*1}
     {-,-} => *1 '- *2 ;
TNEG{NEG{}} => .EMPTY ;
  MULT{-,-} => *1 '* *2 ;
REALNUM{.NUM,.NUM} => *1 '. *2 ;
NEG{-} => '- *1 ;
LIT{-} => '' *1 '' ;
SUBVAR{.ID,-} => *1 PAR *2  ;
PAR -  => '( *1 ') ;
FLAGWORD /  => "'$$$'" ;
LOC{-} => "INTEGER " *1  , "DATA " *1 "/'*LOCAL'/";

%*******************************************************************%
            % THESE PARSE RULES DO NOT BUILD A TREE, BUT OUTPUT DIRECTLY

GFOR = "FOR " ("EACH "/"ALL "/"EVERY "/.EMPTY)
       FORASSEXP ?60E "DO" ?61E .NUM ?62E .IDGOT()
       %IDGOT PUSHES A COPY OF THE NUMBER ONTO A SPECIAL STACK%
       {"CALL GFOR(LOC: +W .W ', *S3 ', *S2 ', *S1 ')\ }
       { #1 , "CALL GINC(LOC: .W ",$" *S0 ', *S3 ', *S2 ', *S1 ')}
       $(-.IDTST() STATEMENT * ) % IDTST TESTS FOR THE NUMBER PICKED UP%
       { \ , "GO TO " #1 -W }
       .IDBK(): % IDBK POPS THE NUMBER OFF THE SPECIAL STACK %

FORITEM = '? .IPUT(5,5H'$$$')/ % IPUT PUTS A STRING INTO
          STRING-STORAGE AS IF IT HAD BEEN PICKED UP BY .SR %
          .ID /
          .NUM ;

FORASSEXP =  ITEM '* ?65E ITEM ?66E '= ?67E ITEM ?68E ;

%*******************************************************************%
```

```
% THE FOLLOWING RULES DO NOT BUILD A TREE OR OUTPUT DIRECTLY:  THEY ARE
  USED FOR FLAG-SETTING IN THE 'EXAMPLE' COMPILER. %

FLAG = DEBUG/LIST ;

DEBUG = "DEBUG" ( "ON" !"DBGFLG=1"/ "OFF" !"DBGFLG=0" ) ;

LIST = "LIST" ("SOURCE" !"LSTSRC=1" !"LSTCOD=0" /
                "CODE" !"LSTSRC=0" !"LSTCOD=1" /
                 "OFF" !"LSTSRC=0" !"LSTCOD=0" /
                .EMPTY !"LSTSRC=1" !"LSTCOD=1"  ) ;

%*****************************************************************%
.END EXAMPLE
```

```
C This is the actual "EXAMPL" compiler generated by TREE-META
C on the 1108.
C****************************
C                1108  TREE - META
C****************************
C FLAGS AND FLAG CONSTANTS
        INTEGER LSTCOD,LSTSRC
        INTEGER IMDFLG
        INTEGER PTRFLG
        DATA PTRFLG/0200000000000/
        INTEGER ADRFLG
        DATA ADRFLG/0100000000000/
        INTEGER CHRFLG
        DATA CHRFLG/0040000000000/
        INTEGER SRFLG
        DATA SRFLG/0020000000000/
        INTEGER GENFLG
        DATA GENFLG/0010000000000/
        INTEGER IDFLG
        DATA IDFLG/0004000000000/
        INTEGER NUMFLG
        DATA NUMFLG/0002000000000/
        INTEGER MFLAG
        INTEGER LLNFLG
        INTEGER OBGFLG
        DATA OBGFLG/0/
        PARAMETER CRNG=64
        INTEGER GNLB1,GNLB2,GNLB3
        INTEGER TRMCHR
        DATA TRMCHR/0770505050505/
        INTEGER NCMNT
        DATA NCMNT/0520505050505/
        INTEGER ATCHR
        DATA ATCHR/0760505050505/
        INTEGER BLKCHR
        DATA BLKCHR/0050505050505/
        INTEGER BLANK5
        DATA BLANK5/00505050505/
        INTEGER LETFLG
        DATA LETFLG/0001000000000/
        INTEGER DIGFLG
        DATA DIGFLG/0000200000000/
        INTEGER OCTFLG
        DATA OCTFLG/0000100000000/
        INTEGER LSS
        INTEGER KSP
        INTEGER KTX
        INTEGER MSP
C       BITS/CHAR, CHARACTER RANGE
        INTEGER NW(5000),IW,MW(5000)
```

```
         INTEGER NS(80),LS
         INTEGER NV(72),IV
         INTEGER NU(100),IU
         INTEGER GET
         INTEGER INCFLG
         INTEGER wRK,XWRK
C********************************************************
         COMMON /LETFLG/LETFLG
         COMMON /OCTFLG/OCTFLG
         COMMON /INCFLG/INCFLG
         COMMON /DIGFLG/DIGFLG
         COMMON /CC/MCCP,NCCP,IBCK
         COMMON /CNT/ICNT,NCNT
         COMMON /MAXMIN/KSPMAX,MSPMAX,NSPMAX,KSPI,MSPI,NSPI
         COMMON /MASKS/ADRMSK
         COMMON /IME/IME,ME
         COMMON /IERR/IERR
         COMMON /DBGFLG/DBGFLG
         COMMON /LSTFLS/LSTCOD,LSTSRC
         COMMON /FLGCNS/PTRFLG,ADRFLG,CHRFLG,SRFLG,GENFLG,IDFLG,NUMFLG
         COMMON /GNLB/GNLB1,GNLB2,GNLB3,MAXGLB,IGN
         COMMON /CHRS/ TRMCHR,NCMNT,QTCHR,BLKCHR,BLANK5,BLANKS
         COMMON /NTAB/NTAB
         COMMON /NMW/NW,IW,IXW,MW
         COMMON /NS/NS,LXS
         COMMON /NU/NU,IXU,IU
         COMMON /NV/NV,IXV,IV
         COMMON /FLAGS/IMDFLG,LLNFLG
         COMMON /MISC/MARK,CIW,SNSP,MSPLN1,ID1ST
         COMMON/LSSAVE/LSSAVE
         COMMON /METAS/MFLAG,ISTAR,LS
         COMMON /KT/KT,KTX
         COMMON /IGLAB/IGLAB
         COMMON /NCLASS/NCLASS
C**********************************THESE WERE PARAMETER STATEMENTS*****************
         INCFLG=0
         MSPLN1=131
         NTAB=10
         MAXGLB=32767
         KSPI=1
         MSPI=2
         NSPI=1
         PBC=6
         IXW=5000
         LXS=80
         IXV=72
         IXU=100
C*****************************************************************************
```

```
          INTEGER BLANKS
          DATA BLANKS/1H /
C MASKS
           INTEGER ADRMSK
           DATA ADRMSK/0777777/
C******************************
          INTEGER ICNT
          INTEGER NCNT
          INTEGER ME
          INTEGER IME
          INTEGER ISTAR
          INTEGER MCCP,IBCK
          INTEGER NCCP
          INTEGER CIW
          INTEGER KSPMAX
          INTEGER MSPMAX
          INTEGER NSPMAX
           INTEGER MARK
          INTEGER IGLAB
          INTEGER IERR
          INTEGER IGN
          INTEGER KT
          INTEGER NSP
          INTEGER SNSP
          INTEGER IDIST
          INTEGER ISTUPO
C******************************
C         1=BLANK,2=LETTER,3=DIGIT,4='',5=OTHER
          INTEGER NCLASS(CRNG)
          DATA(NCLASS(I),I=1,CRNG)
     *    /5,5,5,5,5,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
     *    2,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,3,3,3,3,3,3,3,3,3,3,
     *    5,5,5,5,4,5/
C******************************
C******************************
           COMMON /KSTACK/KSTACK(500),KXSP,KSP
            KXSP=500
           COMMON /MSTACK/MSTACK(500),MXSP,MSP
            MXSP=500
           COMMON /NSTACK/NSTACK( 700),NXSP,NSP
            NXSP=700
           COMMON /NSYMTB/NSYMTB(3500),LXSS,LSS
            LXSS=3500
           CALL INIT
           WRK=1
           XWRK=0
           CALL RLINE
           CALL NCALL($1017,$32766)
```

```
32766      CALL FINISH
           CALL LIMITS
           STOP ENDCMP
C EXAMPL
1017       CONTINUE
           MFLAG=1
           IF(MFLAG),32765,
           ID1ST=2
           CALL TST(7,7HEXAMPLE)
           IF(MFLAG.EQ.0)CALL BIGERR
           CALL ID
           IF(MFLAG.NE.0)CALL KPUSH(ISTAR+IDFLG)
           IF(MFLAG.EQ.0)CALL BIGERR
           CALL MCALL($1059,$32764)
32764      CONTINUE
           MFLAG=1
           CALL MCALL($1068,$32763)
32763      CONTINUE
           IF(MFLAG.EQ.0)CALL BIGERR
32765      CONTINUE
           CALL MRTN
C NEXTGF
1068       CONTINUE
32762      CONTINUE
           CALL MCALL($1080,$32761)
32761      CONTINUE
           IF(MFLAG),32760,
           CALL OUTREE
           IF(MFLAG.EQ.0) CALL CERR(2)
32760      CONTINUE
           IF(MFLAG),,32762
           MFLAG=1
           IF(MFLAG),32759,
           CALL TST(3,3HEND)
           IERR=(1)
           IF(MFLAG.EQ.0)CALL ERR(0)
           CALL CRLF
           CALL TAB
           CALL LIT(3,3HEND)
           CALL STPMTA
           IF(MFLAG.EQ.0)CALL BIGERR
32759      CONTINUE
           CALL MRTN
C STATEMENT
1080       CONTINUE
           CALL MPUSH(KT)
           CALL MPUSH(KSP)
           CALL MPUSH(NCCP)
```

```
          CALL TST(3,3HEND)
          MFLAG=MOD(MFLAG+1,2)
          NCCP=MPOP(ISTUPD)
          KSP=MPOP(ISTUPD)
          KT =MPOP(ISTUPD)
          IF(MFLAG),32758,
          CALL CRLF
32757     CONTINUE
          CALL COL(1)
          IF(MFLAG),,32756
          NCCP=NCCP+1
          GOTO 32757
32756     CONTINUE
          CALL MCALL($1119,$32755)
32755     CONTINUE
          IF(MFLAG),,32754
          CALL MCALL($1127,$32753)
32753     CONTINUE
          IF(MFLAG),32752,
          CALL MCALL($1135,$32751)
32751     CONTINUE
          IF(MFLAG),,32750
          CALL MCALL($1059,$32749)
32749     CONTINUE
          IF(MFLAG),,32748
          CALL MCALL($1144,$32747)
32747     CONTINUE
32748     CONTINUE
32750     CONTINUE
          IERR=(2)
          IF(MFLAG.EQ.0)CALL ERR($1152)
32752     CONTINUE
32754     CONTINUE
          IF(MFLAG.EQ.0)CALL BIGERR
32758     CONTINUE
          CALL MRTN
C COMMENTCARD
1119      CONTINUE
          INCFLG=1
          CALL  TCH(1HC)
          INCFLG=0
          IF(MFLAG),32746,
          CALL  CIO(1HC)
          CALL SET
          IF(MFLAG.EQ.0)CALL BIGERR
32745     CONTINUE
          CALL BLANKC
          IF(MFLAG),,32744
```

```
              NCCP=NCCP+1
              GOTO 32745
 32744        CONTINUE
              CALL COPY
              IF(MFLAG.EQ.0)CALL BIGERR
 32746        CONTINUE
              CALL MRTN
C LABEL
 1127         CONTINUE
              CALL NUM
              IF(MFLAG),32743,
              ISTAR = ISTAR+NUMFLG
              CALL KPUSH(ISTAR)
              IME=KT
              ISTAR=AND(IME,ADRMSK)
              IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)
              CALL OUTS
              CALL SAV
              CALL MPUSH(-1)
 32742        CONTINUE
              INCFLG=1
              CALL  TCH(1H )
              INCFLG=0
              IF(MFLAG),32741,
              CALL  CIO(1H )
 32741        CONTINUE
              MSTACK(MSP)=MSTACK(MSP)+1
              IF(MSTACK(MSP)-32767),,32740
              IF(MFLAG),,32742
              IF(MSTACK(MSP).GE.1)MFLAG=1
              GO TO 32739
 32740        MFLAG=0
 32739        ISTUPD=MPOP(ISTUPD)
              CALL RSTR
              IERR=(3)
              IF(MFLAG.EQ.0)CALL ERR(41152)
              GO TO 32738
 32743        CONTINUE
              CALL SAV
              CALL MPUSH(-1)
 32737        CONTINUE
              INCFLG=1
              CALL  TCH(1H )
              INCFLG=0
              IF(MFLAG),32736,
              CALL  CIO(1H )
 32736        CONTINUE
              MSTACK(MSP)=MSTACK(MSP)+1
```

```
            IF(MSTACK(MSP)-32767),,32735
            IF(MFLAG),,32737
            IF(MSTACK(MSP).GE.5)MFLAG=1
            GO TO 32734
32735       MFLAG=0
32734       ISTUPD=MPOP(ISTUPD)
            CALL RSTR
32738       CONTINUE
            CALL MRTN
C FORSTA
1144        CONTINUE
            CALL CHR
            IF(MFLAG.NE.0)CALL KPUSH(ISTAR+CHRFLG)
            IF(MFLAG),32733,
            IME=KT
            ISTAR=AND(IME,ADRMSK)
            IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)
            FLD(0,6,ISTUPD)=FLD(30,6,IME)
            CALL CIO(OR(ISTUPD,BLANK5))
            CALL SET
            IF(MFLAG.EQ.0)CALL BIGERR
32732       CONTINUE
            CALL BLANKC
            IF(MFLAG),,32731
            NCCP=NCCP+1
            GOTO 32732
32731       CONTINUE
            CALL COPY
            IF(MFLAG.EQ.0)CALL BIGERR
32733       CONTINUE
            CALL MRTN
C OFSTA
1135        CONTINUE
            CALL MCALL($1198,$32730)
32730       CONTINUE
            IF(MFLAG),,32729
            CALL MCALL($1206,$32728)
32728       CONTINUE
            IF(MFLAG),,32727
            CALL MCALL($1213,$32726)
32726       CONTINUE
32727       CONTINUE
32729       CONTINUE
            CALL MRTN
C E
1152        CONTINUE
            MFLAG=1
            IF(MFLAG),32725,
```

```
                CALL RESET
                IF(MFLAG.EQ.0)CALL BIGERR
32724           CONTINUE
                CALL  TCH(1H;)
                IF(MFLAG),,32723
                NCCP=NCCP+1
                GOTO 32724
32723           CONTINUE
32722           CONTINUE
                CALL MCALL($1080,$32721)
32721           CONTINUE
                IF(MFLAG),32720,
                CALL OUTREE
                IF(MFLAG.EQ.0) CALL CERR(2)
32720           CONTINUE
                IF(MFLAG),,32722
                MFLAG=1
                IF(MFLAG.EQ.0)CALL BIGERR
                CALL TST(3,3HEND)
                IERR=(99)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL CRLF
                CALL TAB
                CALL LIT(3,3HEND)
                CALL STPMTA
                IF(MFLAG.EQ.0)CALL BIGERR
32725           CONTINUE
                CALL MRTN
C MAKEBREAK
1198            CONTINUE
                CALL TST(5,5HMAKE )
                IF(MFLAG),32719,
                CALL MCALL($1238,$32718)
32718           CONTINUE
                IERR=(10)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL NDLB($1250)
                CALL MKND(1)
                GO TO 32717
32719           CONTINUE
                CALL TST(6,6HBREAK )
                IF(MFLAG),32716,
                CALL MCALL($1238,$32715)
32715           CONTINUE
                IERR=(11)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL NDLB($1272)
                CALL MKND(1)
```

```
32716   CONTINUE
32717   CONTINUE
        CALL MRTN
C ASSEXP
1238    CONTINUE
        CALL MCALL($1279,$32714)
32714   CONTINUE
        IF(MFLAG),32713,
        CALL  TCH(1H*)
        IERR=(12)
        IF(MFLAG.EQ.0)CALL ERR($1152)
        CALL MCALL($1279,$32712)
32712   CONTINUE
        IERR=(13)
        IF(MFLAG.EQ.0)CALL ERR($1152)
        CALL  TCH(1H=)
        IERR=(14)
        IF(MFLAG.EQ.0)CALL ERR($1152)
        CALL MCALL($1279,$32711)
32711   CONTINUE
        IERR=(15)
        IF(MFLAG.EQ.0)CALL ERR($1152)
        CALL NDLB($1308)
        CALL MKND(3)
32713   CONTINUE
        CALL MRTN
C ITEM
1279    CONTINUE
        CALL  TCH(1H?)
        IF(MFLAG),32710,
        CALL NDLB($1319)
        CALL MKND(0)
        GO TO 32709
32710   CONTINUE
        CALL MCALL($1330,$32708)
32708   CONTINUE
32709   CONTINUE
        CALL MRTN
C PRIM
1330    CONTINUE
        CALL SAV
        CALL NUM
        IF(MFLAG.NE.0)CALL KPUSH(ISTAR+NUMFLG)
        IF(MFLAG),32707,
        CALL  TCH(1H.)
        IF(MFLAG),32706,
        CALL NUM
        IF(MFLAG.NE.0)CALL KPUSH(ISTAR+NUMFLG)
```

```
         IF(MFLAG),32705,
         CALL NDLB($1340)
         CALL MKND(2)
32705    CONTINUE
32706    CONTINUE
32707    CONTINUE
         CALL RSTR
         IF(MFLAG),,32704
         CALL NUM
         IF(MFLAG.NE.0)CALL KPUSH(ISTAR+NUMFLG)
         IF(MFLAG),,32703
         CALL MCALL($1346,$32702)
32702    CONTINUE
         IF(MFLAG),,32701
         CALL  TCH(1H')
         IF(MFLAG),32700,
         INCFLG=1
         CALL CHR
         IF(MFLAG.NE.0)CALL KPUSH(ISTAR+CHRFLG)
         INCFLG=0
         IERR=(20)
         IF(MFLAG.EQ.0)CALL ERR($1152)
32699    CONTINUE
         CALL MPUSH(KT)
         CALL MPUSH(KSP)
         CALL MPUSH(NCCP)
         CALL  TCH(1H')
         MFLAG=MOD(MFLAG+1,2)
         NCCP=MPOP(ISTUPD)
         KSP=MPOP(ISTUPD)
         KT =MPOP(ISTUPD)
         IF(MFLAG),32698,
         INCFLG=1
         CALL CHR
         IF(MFLAG.NE.0)CALL KPUSH(ISTAR+CHRFLG)
         INCFLG=0
         IF(MFLAG.EQ.0)CALL BIGERR
         CALL NDLB($1356)
         CALL MKND(2)
32698    CONTINUE
         IF(MFLAG),,32699
         MFLAG=1
         IF(MFLAG.EQ.0)CALL BIGERR
         CALL  TCH(1H')
         IERR=(21)
         IF(MFLAG.EQ.0)CALL ERR($1152)
         CALL NDLB($1367)
         CALL MKND(1)
```

```
           GO TO 32697
32700      CONTINUE
           CALL  TCH(1H()
           IF(MFLAG),32696,
           CALL MCALL($1373,$32695)
32695      CONTINUE
           IERR=(22)
           IF(MFLAG.EQ.0)CALL ERR($1152)
           CALL  TCH(1H))
           IERR=(23)
           IF(MFLAG.EQ.0)CALL ERR($1152)
           CALL NDLB($1389)
           CALL MKND(1)
32696      CONTINUE
32697      CONTINUE
32701      CONTINUE
32703      CONTINUE
32704      CONTINUE
           CALL MRTN
C VAR
1346       CONTINUE
           CALL ID
           IF(MFLAG.NE.0)CALL KPUSH(ISTAR+IDFLG)
           IF(MFLAG),32694,
           CALL  TCH(1H()
           IF(MFLAG),32693,
           CALL MCALL($1373,$32692)
32692      CONTINUE
           IERR=(30)
           IF(MFLAG.EQ.0)CALL ERR($1152)
32691      CONTINUE
           CALL  TCH(1H,)
           IF(MFLAG),32690,
           CALL MCALL($1373,$32689)
32689      CONTINUE
           IERR=(31)
           IF(MFLAG.EQ.0)CALL ERR($1152)
           CALL NDLB($1407)
           CALL MKND(2)
32690      CONTINUE
           IF(MFLAG),,32691
           MFLAG=1
           IF(MFLAG.EQ.0)CALL BIGERR
           CALL  TCH(1H))
           IF(MFLAG.EQ.0)CALL BIGERR
           CALL NDLB($1416)
           CALL MKND(2)
32693      CONTINUE
```

```
                MFLAG=1
32694           CONTINUE
                CALL MRTN
C EXP
1373            CONTINUE
                CALL MCALL($1423,$32688)
32688           CONTINUE
                IF(MFLAG),32687,
                CALL   TCH(1H+)
                IF(MFLAG),32686,
                CALL MCALL($1373,$32685)
32685           CONTINUE
                IERR=(40)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL NDLB($1435)
                CALL MKND(2)
                GO TO 32684
32686           CONTINUE
                CALL   TCH(1H-)
                IF(MFLAG),32683,
                CALL MCALL($1373,$32682)
32682           CONTINUE
                IERR=(41)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL NDLB($1448)
                CALL MKND(2)
32683           CONTINUE
                MFLAG=1
32684           CONTINUE
                IF(MFLAG.EQ.0)CALL BIGERR
32687           CONTINUE
                CALL MRTN
C TERM
1423            CONTINUE
                CALL MCALL($1457,$32681)
32681           CONTINUE
                IF(MFLAG),32680,
32679           CONTINUE
                CALL   TCH(1H*)
                IF(MFLAG),32678,
                CALL MCALL($1457,$32677)
32677           CONTINUE
                IERR=(42)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL NDLB($1469)
                CALL MKND(2)
                GO TO 32676
32678           CONTINUE
```

```
              CALL  TCH(1H/)
              IF(MFLAG),32675,
              CALL MCALL($1457,$32674)
32674         CONTINUE
              IERR=(43)
              IF(MFLAG.EQ.0)CALL ERR($1152)
              CALL NDLB($1483)
              CALL MKND(2)
32675         CONTINUE
32676         CONTINUE
              IF(MFLAG),,32679
              MFLAG=1
              IF(MFLAG.EQ.0)CALL BIGERR
32680         CONTINUE
              CALL MRTN
C FACTOR
1457          CONTINUE
              CALL  TCH(1H-)
              IF(MFLAG),32673,
              CALL MCALL($1457,$32672)
32672         CONTINUE
              IERR=(40)
              IF(MFLAG.EQ.0)CALL ERR($1152)
              CALL NDLB($1489)
              CALL MKND(1)
              GO TO 32671
32673         CONTINUE
              CALL MCALL($1330,$32670)
32670         CONTINUE
32671         CONTINUE
              CALL MRTN
C LOCAL
1206          CONTINUE
              CALL TST(5,5HLOCAL)
              IF(MFLAG),32669,
              CALL ID
              IF(MFLAG.NE.0)CALL KPUSH(ISTAR+IDFLG)
              IF(MFLAG.EQ.0)CALL BIGERR
              CALL NDLB($1495)
              CALL MKND(1)
32669         CONTINUE
              CALL MRTN
C BREAK
1272          CONTINUE
              CALL BEGN
              ICNT=1
              IF(NCNT.NE.ICNT) MFLAG=0
              IF(MFLAG),32668,
```

```
              CALL LIT(11,11HCALL BREAK()
              IME=KT
              IME=GET(1)
              CALL DOIT($32667)
32667         CONTINUE
              IF(MFLAG.EQ.0) CALL CERR(1)
              CALL  CIO(1H))
32668         CONTINUE
              CALL MRTN
C MAKE
1250          CONTINUE
              CALL BEGN
              ICNT=1
              IF(NCNT.NE.ICNT) MFLAG=0
              IF(MFLAG),32666,
              CALL LIT(10,10HCALL MAKE()
              IME=KT
              IME=GET(1)
              CALL DOIT($32665)
32665         CONTINUE
              IF(MFLAG.EQ.0) CALL CERR(1)
              CALL  CIO(1H))
32666         CONTINUE
              CALL MRTN
C TRIPLE
1308          CONTINUE
              CALL BEGN
              ICNT=3
              IF(NCNT.NE.ICNT) MFLAG=0
              IF(MFLAG),32664,
              IME=KT
              IME=GET(1)
              CALL DOIT($32663)
32663         CONTINUE
              IF(MFLAG),32662,
              CALL  CIO(1H,)
              IME=KT
              IME=GET(2)
              CALL DOIT($32661)
32661         CONTINUE
              IF(MFLAG.EQ.0) CALL CERR(1)
              CALL  CIO(1H,)
              IME=KT
              IME=GET(3)
              CALL DOIT($32660)
32660         CONTINUE
32662         CONTINUE
32664         CONTINUE
```

```
      CALL MRTN
C DO
1356  CONTINUE
      CALL BEGN
      ICNT=2
      IF(MCNT.NE.ICNT) MFLAG=0
      IF(MFLAG),32659,
      CALL OTCLL1($1503)
      IME=ME
      IME=GET(1)
      CALL KPUSH(IME)
      ICNT=ICNT+1
      CALL OTCLL2($32658)
32658 CONTINUE
      IF(MFLAG),32657,
      CALL OTCLL1($1503)
      IME=ME
      IME=GET(2)
      CALL KPUSH(IME)
      ICNT=ICNT+1
      CALL OTCLL2($32656)
32656 CONTINUE
32657 CONTINUE
32659 CONTINUE
      CALL MRTN
C DOTST
1503  CONTINUE
      CALL BEGN
      IF(AND(NSTACK(KTX),CHRFLG).NE.CHRFLG) MFLAG=0
      ICNT=ICNT+1
      IF(MCNT.NE.ICNT) MFLAG=0
      IF(MFLAG),32655,
      IME=KT
      IME=GET(1)
      ISTAR=AND(IME,ADRMSK)
      IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)
      FLD(0,6,ISTUPD)=FLD(30,6,IME)
      CALL CIO(OR(ISTUPD,BLANK5))
      GO TO 32654
32655 CONTINUE
      CALL BEGN
      ICNT=1
      IF(MCNT.NE.ICNT) MFLAG=0
      IF(MFLAG),32653,
      IME=KT
      IME=GET(1)
      CALL DOIT($32652)
32652 CONTINUE
```

```
32653      CONTINUE
32654      CONTINUE
           CALL MRTN
C DIVIDE
1483       CONTINUE
           CALL BEGN
           ICNT=2
           IF(NCNT.NE.ICNT) MFLAG=0
           IF(MFLAG),32651,
           IME=KT
           IME=GET(1)
           CALL DOIT($32650)
32650      CONTINUE
           IF(MFLAG),32649,
           CALL  CIO(1H/)
           IME=KT
           IME=GET(2)
           CALL DOIT($32648)
32648      CONTINUE
32649      CONTINUE
32651      CONTINUE
           CALL MRTN
C COMMA
1407       CONTINUE
           CALL BEGN
           ICNT=2
           IF(NCNT.NE.ICNT) MFLAG=0
           IF(MFLAG),32647,
           IME=KT
           IME=GET(1)
           CALL DOIT($32646)
32646      CONTINUE
           IF(MFLAG),32645,
           CALL  CIO(1H,)
           IME=KT
           IME=GET(2)
           CALL DOIT($32644)
32644      CONTINUE
32645      CONTINUE
32647      CONTINUE
           CALL MRTN
C PLUS
1435       CONTINUE
           CALL BEGN
           ICNT=2
           IF(NCNT.NE.ICNT) MFLAG=0
           IF(MFLAG),32643,
           CALL OTCLL1($1510)
```

```
              IME=ME
              IME=GET(2)
              CALL KPUSH(IME)
              ICNT=ICNT+1
              CALL OTCLL2($32641)
      32641   CONTINUE
              IF(MFLAG),32642,
              CALL OTCLL1($1448)
              IME=ME
              IME=GET(1)
              CALL KPUSH(IME)
              ICNT=ICNT+1
              IME=ME
              IME=GET(2)
              IF(AND(IME,PTRFLG).NE.PTRFLG)CALL CERR(3)
              IME=GET(1)
              CALL KPUSH(IME)
              ICNT=ICNT+1
              CALL OTCLL2($32640)
      32640   CONTINUE
              GO TO 32639
      32642   CONTINUE
              IME=KT
              IME=GET(1)
              CALL DOIT($32638)
      32638   CONTINUE
              IF(MFLAG),32637,
              CALL  CIO(1H+)
              IME=KT
              IME=GET(2)
              CALL DOIT($32636)
      32636   CONTINUE
      32637   CONTINUE
      32639   CONTINUE
      32643   CONTINUE
              CALL MRTN
      C MINUS
      1448    CONTINUE
              CALL BEGN
              ICNT=ICNT+1
              KTX=KTX-1
              CALL RI1($1489,$32634)
              IF(NCNT.NE.ICNT) MFLAG=0
              CALL RI2
      32634   CONTINUE
              ICNT=ICNT+1
              IF(NCNT.NE.ICNT) MFLAG=0
              IF(MFLAG),32635,
```

```
          CALL GTCLL1($1435)
          IME=NE
          IME=GET(1)
          CALL KPUSH(IME)
          ICNT=ICNT+1
          IME=NE
          IME=GET(2)
          IF(AND(IME,PTRFLG).NE.PTRFLG)CALL CERR(3)
          IME=GET(1)
          CALL KPUSH(IME)
          ICNT=ICNT+1
          CALL CTCLL2($32633)
32633     CONTINUE
          GO TO 32632
32635     CONTINUE
          CALL BEGN
          ICNT=2
          IF(NCNT.NE.ICNT) MFLAG=0
          IF(MFLAG),32631,
          IME=KT
          IME=GET(1)
          CALL DOIT($32630)
32630     CONTINUE
          IF(MFLAG),32629,
          CALL  CIO(1H-)
          IME=KT
          IME=GET(2)
          CALL DOIT($32628)
32628     CONTINUE
32629     CONTINUE
32631     CONTINUE
32632     CONTINUE
          CALL MRTN
C TNEG
1510      CONTINUE
          CALL BEGN
          CALL RI1($1489,$32627)
          IF(NCNT.NE.ICNT) MFLAG=0
          CALL RI2
32627     CONTINUE
          ICNT=ICNT+1
          IF(NCNT.NE.ICNT) MFLAG=0
          IF(MFLAG),32626,
          MFLAG=1
32626     CONTINUE
          CALL MRTN
C MULT
1469      CONTINUE
```

```
      CALL BEGN
      ICNT=2
      IF(NCNT.NE.ICNT) MFLAG=0
      IF(MFLAG),32625,
      IME=KT
      IME=GET(1)
      CALL DOIT($32624)
32624 CONTINUE
      IF(MFLAG),32623,
      CALL  CIO(1H*)
      IME=KT
      IME=GET(2)
      CALL DOIT($32622)
32622 CONTINUE
32623 CONTINUE
32625 CONTINUE
      CALL MRTN
C REALNUM
1340  CONTINUE
      CALL BEGN
      IF(AND(NSTACK(KTX),NUMFLG).NE.NUMFLG) MFLAG=0
      ICNT=ICNT+1
      KTX=KTX-1
      IF(AND(NSTACK(KTX),NUMFLG).NE.NUMFLG) MFLAG=0
      ICNT=ICNT+1
      IF(NCNT.NE.ICNT) MFLAG=0
      IF(MFLAG),32621,
      IME=KT
      IME=GET(1)
      CALL DOIT($32620)
32620 CONTINUE
      IF(MFLAG),32619,
      CALL  CIO(1H.)
      IME=KT
      IME=GET(2)
      CALL DOIT($32618)
32618 CONTINUE
32619 CONTINUE
32621 CONTINUE
      CALL MRTN
C NEG
1489  CONTINUE
      CALL BEGN
      ICNT=1
      IF(NCNT.NE.ICNT) MFLAG=0
      IF(MFLAG),32617,
      CALL  CIO(1H-)
      IME=KT
```

```
            IME=GET(1)
            CALL DOTT($32616)
32616       CONTINUE
32617       CONTINUE
            CALL MRTN
C LIT
1367        CONTINUE
            CALL BEGN
            ICNT=1
            IF(NCNT.NE.ICNT) MFLAG=0
            IF(MFLAG),32615,
            CALL  CTO(1H')
            IME=KT
            IME=GET(1)
            CALL DOTT($32614)
32614       CONTINUE
            IF(MFLAG.EQ.0) CALL CERR(1)
            CALL  CTO(1H')
32615       CONTINUE
            CALL MRTN
C SUBVAR
1416        CONTINUE
            CALL BEGN
            IF(AND(NSTACK(KTX),IDFLG).NE.IDFLG) MFLAG=0
            ICNT=ICNT+1
            KTX=KTX-1
            ICNT=ICNT+1
            IF(NCNT.NE.ICNT) MFLAG=0
            IF(MFLAG),32613,
            IME=KT
            IME=GET(1)
            CALL DOTT($32612)
32612       CONTINUE
            IF(MFLAG),32611,
            CALL GTCLL1($1389)
            IME=ME
            IME=GET(2)
            CALL KPUSH(IME)
            ICNT=ICNT+1
            CALL GTCLL2($32610)
32610       CONTINUE
32611       CONTINUE
32613       CONTINUE
            CALL MRTN
C PAR
1389        CONTINUE
            CALL BEGN
            ICNT=1
```

```
          IF(NCNT.NE.ICNT) MFLAG=0
          IF(MFLAG),32609,
          CALL  CIO(1H()
          IME=KT
          IME=GET(1)
          CALL DOIT($32608)
32608     CONTINUF
          IF(MFLAG.EQ.0) CALL CERR(1)
          CALL  CIO(1H))
32609     CONTINUE
          CALL MRTN
C FLAGWORD
1319      CONTINUE
          CALL LIT(5,5H'$$$')
          CALL MRTN
C LOC
1495      CONTINUE
          CALL BEGN
          ICNT=1
          IF(NCNT.NE.ICNT) MFLAG=0
          IF(MFLAG),32607,
          CALL LIT(8,8HINTEGER )
          IME=KT
          IME=GET(1)
          CALL DOIT($32606)
32606     CONTINUF
          IF(MFLAG.EQ.0) CALL CERR(1)
          CALL CRLF
          CALL TAB
          CALL LIT(5,5HDATA )
          IME=KT
          IME=GET(1)
          CALL DOIT($32605)
32605     CONTINUF
          IF(MFLAG.EQ.0) CALL CERR(1)
          CALL LIT(10,10H/'*LOCAL'/)
32607     CONTINUE
          CALL MRTN
C GFOR
1213      CONTINUE
          CALL TST(4,4HFOR )
          IF(MFLAG),32604,
          CALL TST(5,5HEACH )
          IF(MFLAG),,32603
          CALL TST(4,4HALL )
          IF(MFLAG),,32602
          CALL TST(6,6HEVERY )
          MFLAG=1
```

```
32602     CONTINUE
32603     CONTINUE
          IF(MFLAG.EQ.0)CALL BIGERR
          CALL MCALL($1553,$32601)
32601     CONTINUE
          IERR=(60)
          IF(MFLAG.EQ.0)CALL ERR($1152)
          CALL TST(2,2HDO)
          IERR=(61)
          IF(MFLAG.EQ.0)CALL ERR($1152)
          CALL NUM
          IF(MFLAG.NE.0)CALL KPUSH(ISTAR+NUMFLG)
          IERR=(62)
          IF(MFLAG.EQ.0)CALL ERR($1152)
          CALL IDGOT
          IF(MFLAG.EQ.0)CALL BIGERR
          CALL LIT(13,13HCALL GFOR(LOC)
          WRK=WRK+1
          XWRK=MAX0(WRK,XWRK)
          CALL OUTN(WRK)
          CALL  CTO(1H,)
          IME=ISTACK(KSP+1-4)
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(0)
          CALL OUTS
          CALL  CTO(1H,)
          IME=ISTACK(KSP+1-2)
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(0)
          CALL OUTS
          CALL  CTO(1H,)
          IME=ISTACK(KSP+1-1)
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(0)
          CALL OUTS
          CALL  CTO(1H))
          CALL CRLF
          CALL CEN(GNLB1)
          CALL TAB
          CALL LIT(13,13HCALL GINC(LOC)
          CALL OUTN(WRK)
          CALL LIT(2,2H,$)
          IME=1
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(0)
          CALL OUTS
          CALL  CTO(1H,)
          IME=ISTACK(KSP+1-3)
```

```
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)
          CALL OUTS
          CALL  CIO(1H,)
          IME=KSTACK(KSP+1-2)
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)
          CALL OUTS
          CALL  CIO(1H,)
          IME=KSTACK(KSP+1-1)
          ISTAR=AND(IME,ADRMSK)
          IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)
          CALL OUTS
          CALL  CIO(1H))
32600     CONTINUE
          CALL MPUSH(KT)
          CALL MPUSH(KSP)
          CALL MPUSH(NCCP)
          CALL IOTST
          MFLAG=MOD(MFLAG+1,2)
          NCCP=MPOP(ISTUPD)
          KSP=MPOP(ISTUPD)
          KT =MPOP(ISTUPD)
          IF(MFLAG),32599,
          CALL MCALL($1020,$32598)
32598     CONTINUE
          IF(MFLAG.EQ.0)CALL BIGERR
          CALL OUTREE
          IF(MFLAG.EQ.0) CALL CERR(2)
32599     CONTINUE
          IF(MFLAG),,32600
          MFLAG=1
          IF(MFLAG.EQ.0)CALL BIGERR
          CALL CRLF
          CALL TAB
          CALL LIT(6,6HGO TO )
          CALL CEN(GNLB1)
          WRK=WRK-1
          CALL IDRK
          IF(MFLAG.EQ.0)CALL BIGERR
32604     CONTINUE
          CALL MRTN
C FORITEM
1651      CONTINUE
          CALL  TCH(1H?)
          IF(MFLAG),32597,
          CALL IPUT(5,5H'$$$')
          IF(MFLAG.EQ.0)CALL BIGERR
```

```
                GO TO 32596
32597           CONTINUE
                CALL TO
                IF(MFLAG.NE.0)CALL KPUSH(ISTAR+IDFLG)
                IF(MFLAG),,32595
                CALL NUM
                IF(MFLAG.NE.0)CALL KPUSH(ISTAR+NUMFLG)
32595           CONTINUE
32596           CONTINUE
                CALL MRTN
C FORASSEXP
1553            CONTINUE
                CALL MCALL($1279,$32594)
32594           CONTINUE
                IF(MFLAG),32593,
                CALL  TCH(1H*)
                IERR=(65)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL MCALL($1279,$32592)
32592           CONTINUE
                IERR=(66)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL  TCH(1H=)
                IERR=(67)
                IF(MFLAG.EQ.0)CALL ERR($1152)
                CALL MCALL($1279,$32591)
32591           CONTINUE
                IERR=(68)
                IF(MFLAG.EQ.0)CALL ERR($1152)
32593           CONTINUE
                CALL MRTN
C FLAG
1659            CONTINUE
                CALL MCALL($1666,$32590)
32590           CONTINUE
                IF(MFLAG),,32589
                CALL MCALL($1673,$32588)
32588           CONTINUE
32589           CONTINUE
                CALL MRTN
C DEBUG
1666            CONTINUE
                CALL TST(5,5HDEBUG)
                IF(MFLAG),32587,
                CALL TST(2,2HON)
                IF(MFLAG),32586,
                DBGFLG=1
                GO TO 32585
```

```
32586       CONTINUE
            CALL TST(3,3HOFF)
            IF(MFLAG),32584,
            DBGFLG=0
32584       CONTINUE
32585       CONTINUE
            IF(MFLAG.EQ.0)CALL BIGERR
32587       CONTINUE
            CALL MRTN
C LIST
1673        CONTINUE
            CALL TST(4,4HLIST)
            IF(MFLAG),32583,
            CALL TST(6,6HSOURCE)
            IF(MFLAG),32582,
            LSTSRC=1
            LSTCOD=0
            GO TO 32581
32582       CONTINUE
            CALL TST(4,4HCODE)
            IF(MFLAG),32580,
            LSTSRC=0
            LSTCOD=1
            GO TO 32579
32580       CONTINUE
            CALL TST(3,3HOFF)
            IF(MFLAG),32578,
            LSTSRC=0
            LSTCOD=0
            GO TO 32577
32578       CONTINUE
            MFLAG=1
            IF(MFLAG),32576,
            LSTSRC=1
            LSTCOD=1
32576       CONTINUE
32577       CONTINUE
32579       CONTINUE
32581       CONTINUE
            IF(MFLAG.EQ.0)CALL BIGERR
32583       CONTINUE
            CALL MRTN
         END
```

```
              EXAMPLE TEST (name of element on unit D)
              LIST SOURCE
              INTEGER PART,COST
              DATA /PART,COST/'PART','COST'/
C
C
C      MAKE ASSOCIATIONS
C
              MAKE COST*'HOUSE'=0
              MAKE PART*'HOUSE'='WALL1'
              MAKE PART*'HOUSE'='WALL2'
              MAKE PART*'HOUSE'='WALL3'
              MAKE PART*'HOUSE'='WALL4'
              MAKE PART*'HOUSE'='ROOF'
              MAKE PART*'HOUSE'='FLOOR'
              MAKE COST*'WALL1'=200
              MAKE COST*'WALL2'=300
              MAKE COST*'WALL3'=200
              MAKE COST*'WALL4'=300
             MAKE COST*'ROOF'=295
              MAKE COST*'FLOOR'=300
              MAKE PART*'WALL1'='WINDOW'
              MAKE PART*'WALL1'='WINDOW'
              MAKE PART*'WALL1'='WINDOW'
              MAKE COST*'WINDOW'=50
              MAKE PART*'WALL2'='DOOR'
              MAKE COST*'DOOR'=75
             MAKE PART*'WALL3'='FIRPL'
              MAKE COST * 'FIRPL' = 200
C
C
C      CALL COSTS FOR ANY ITEM IN HOUSE.
              CALL GTCOST('HOUSE')
              PRINT 333,ICOST
              CALL GTCOST('WINDOW')
              PRINT ccc,ICOST
333           FORMAT(' COST=',I6)
C
C             BREAK TREE
C
              BREAK COST*?=?
              BREAK PART*?=?
              STOP
C
C-----------------------------------------
C SUBROUTINE FOR COMPUTING COST OF ITEMS
C
              SUBROUTINE GTCOST(ITEM)
              INTEGER ITEM
              LOCAL X
              LOCAL Y
              LOCAL Z
              ICOST=0
              FOR EACH COST*ITEM=X DO 100
```

```
              ICOST=ICOST+X
100           CONTINUE
              FOR EACH PART*ITEM=Y DO 200
              FOR EACH COST*Y=X DO 300
              ICOST=ICOST+X
300           CONTINUE
              FOR EACH PART*Y=Z DO 400
              FOR EACH COST*Z=X DO 500
              ICOST=ICOST+X
500           CONTINUE
400           CONTINUE
200           CONTINUE
              RETURN
              END
```

APPENDIX A

UTAH TREE-META CONTROL CARDS

```
@A RUN ARCHIT,496802,2,98,,12 SHERIAN U **ARPA** TRMETA 'EXAMPLE'
@ DPR
@ HDG        GENERATE 'EXAMPLE' COMPILER
@ ASG A=$CSC3$
@ ASG C=$CSC1$
@ ASG D
@ XQT CUR
 IN A
 IN C
 TRW A
 TRW C
A XQT TRMETA
         *
         COMPILER SPECIFICATIONS
         (PRODUCES A COMPILER ON UNIT D WITH THE NAME AS SPECIFIED:
         IN THIS CASE, 'EXAMPL'.
         *
@ XQT CUR
 TRW D
 IN D
TRW D
@IA FOR,* EXAMPL,EXAMPL
@ XQT EXAMPL
         *
         LANGUAGE STATEMENTS
         PUTS OUTPUT OF COMPILER ON UNIT D WITH NAME SPECIFIED:
         IN THIS CASE, 'TEST'.)
         *
@ XQT CUR
 TRW D
 IN D
 TRW D
@IA FOR,* TEST,TEST
```

APPENDIX B

RADC TREE-META

Several of the special characters used in the Tree-Meta metalanguage used at RADC differ from those used at Utah and/or as used in this paper. The "@" replaces the "?", the "←" replaces the "!", and the "↑" replaces the "Δ". The new IBM keypunch referred to as the "GE keypunch" should be used to punch all Tree Meta programs for ease of punching and reading.

At present the RADC version of Tree Meta is available in card form only as a set of binary element decks. This set of decks includes the main program and all supporting subroutines. To generate a compiler using Tree-Meta, the following deck arrangement is suggested:

```
$       IDENT (regular format)
$       OPTION FORTRAN

          TREE-META BINARY DECKS

$       EXECUTE DUMP
$       LIMITS 99,40000,0,10000
$       TAPE   03,A1R,,XXXXX,,'NAME'03

          COMPILER SPECIFICATIONS

$       FORTRAN LSTOU,COMDK
$       TAPE    S*,A1D,,YYYYY,,'NAME'03
$       ENDJOB
***EOF
```

where XXXXX and YYYYY are five-digit magnetic tape numbers and 'NAME' is programmer's identification.

To execute the generated compiler, the deck arrangement would be exactly the same as above with a binary deck of the generated compiler replacing the binary deck of the main program of Tree-Meta (the new compiler uses the same support routines as Tree-Meta) and the new language statements replacing the compiler specifications

APPENDIX C

ERROR CODES

The following list is a collection of error code numbers printed by the Tree-Meta system. An error number is printed by the system when an error occurs while processing the compiler specifications, and an error number is provided in the Tree-Meta specifications. Therefore, the following list is a reference from the error number to the Tree-Meta rule name which was processing the user's metalanguage statements when the error occurred. Hopefully, then, the particular cause of the error in question can be determined by comparing the metalanguage statement with the requirements of the rule which printed the error.

ERR - refers to source language error

1.	TRMETA
2.	TRMETA
3.	RULE
5.	RULE
6.	RULE
7.	EXP
8.	EXP
9.	EXP
10.	NOBACK
11.	NOBACK
12.	NTEST
13.	NTEST
14.	NTEST
15.	NTEST
16.	NTEST
19.	STEST
20.	STEST
21.	STEST
23.	STEST
24.	STEST
25.	STEST
26.	STEST
27.	OUTRUL
29.	OUTR
30.	OUTR
32.	ITEMS
33.	ITEM
34.	ITEM
35.	ITEM
36.	ITEM
39.	OUTT
40.	OUTT
41.	OUTT
42.	GENU
43.	GENU
44.	GENU
50.	SIZE
51.	GENP2
52.	COMM
53.	COMM
54.	SIZ
55.	SIZ
60.	CLOSEPAREN
61.	CLOSEPAREN
62.	CLOSEPAREN
98.	ERROR1
99.	E
191.	STEST
192.	STEST

The following is a collection of error codes generated by the Tree-Meta support subroutines.

CERR - refers to compiler error

SERR - refers to system error

CERR(10) OPPS . . . FAILURE OF STEST AND NO ERROR PATH SPECIFIED

SERR(113) REFERENCE TO KSTACK IS LESS THAN 0

SERR(13) REFERENCE TO KSTACK IS GREATER THAN MAXIMUM DIMENSIONS

SERR(18) REFERENCE TO MSTACK IS LESS THAN 0 (GENERATED EITHER THROUGH AN ATTEMPT TO POP MSTACK OR THROUGH AN MONITOR RETURN TO ITEM ON MSTACK)

SERR(12) REFERENCE TO NSTACK IS GREATER THAN MAXIMUM DIMENSIONS

CERR(2) THE REFERENCE TO TREE IN NSTACK DOES NOT FIND A POINTER

TREER - REFERENCE TO TREE POINTS BEYOND BOTTOM OF NSTACK (OUTREE)
SSERR - ATTEMPT TO OUTPUT A LITERAL THAT STARTS LESS THAN 1, OR ENDS GREATER THAN 120 (SSERR)
NSPERR - PREMATURE END OF FILE BEING CURRED IN (OCUROF)

APPENDIX D: TREE-META SPECIFICATIONS

```
TRMETA=(".META" .ID?1?SIZE :BEGIN[2]/".CONTINUE"!"ID1st=2".ID?1? "MT[0])
     (LIST/.EMPTY)  :SETUP[1] * $(  RULE * !"LSS=LSSAVE")
                                                     ".END" ?2E  :ENDN[0] * ;

SIZE  = '(  SIZ $(', SIZ :DO[2]) ') ?50E /  .EMPTY :SIZDF[02;

SIZ   = .CHR '= ?54E  .NUM ?55E :SIZS[2];

RULE = .ID
           ( '= EXP ?3E("&&" :USERID[1]/ '& :KPOPK[1]/.EMPTY):OUTPT[2]/
       ' / "=>" ?3E GEN1 :OUTPT[2]  /
       OUTRUL  :OUTPT[2]) ?5E '; ?6E ;

EXP = "<-" SUBACK ?7E  ('/ EXP ?8E :BALTER[2] / .EMPTY :BALTER[1] /
       SUBEXP ('/ EXP  ?9E :ALTER[2]/ .EMPTY);

SUBACK = NTEST  (SUBACK  :DO[2] /.EMPTY) /
       STEST (SUBACK  :CONCAT[2] / .EMPTY);

SUBEXP = (NTEST / STEST)  (NOBACK :CONCAT[2] / . EMPTY);

NOBACK = (NTEST / STEST  ('? .NUM ?10E :LOAD[1]  (.ID / '? ;ZRO[0]) ?11E
       :ERCOD[3] / .EMPTY :ER[1])    )
       (NOBACK :DO[2] / .EMPTY;

NTEST =  ': .ID ?12E :NDLB[1] /
       '[ ( .NUM   '] ?14E :MKNODE[1] /
              GENP '] ?52E ('Δ/.EMPTY :MT[0] :DO[2])  ) /
       '< GENP '> ?53E ('Δ /.EMPTY :OUTCR[0] :DO[2])  :TTY[1] /
       '* :GO[0] /
                 LIST /
       "=>" STEST ?15E :SCAN[1] /




       COMM:

LIST =   ".LIST"       ("SOURCE" !"LSTSRC=1" !"LSTCOD=0"
                      /"CODE"    !"LSTSRC=0" !"LSTCOD=1"
                      /"OFF "    !"LSTSRC=0" !"LSTCOD=0"
                      /.EMPTY    !"LSTSRC=1" !"LSTCOD=1") ;

GENP = GENP1 / .EMPTY :MT[0];

GENP1 = GENP2 (GENP1 :DO[2] / .EMPTY);

GENP2 = '* ('S .NUM ?51E :PAROUT[1] / .EMPTY :ZRO[0] :PAROUT[1])
       ('L :OL / 'C :OC / 'N :ON / .EMPTY :OS)[0] :NOPT[2]/ GENU;

COMM = ".EMPTY" :SET[0] /
       '! (.SR :IMED[1] / '(ITST?52E') :IMED[1])?53E ;

ITST = (    .SR/'\:ICR[0]'' :ITB[0]/ ''+.CHR / "#1":ILB1[0] /
```

```
                                           "#2":ILB2[0]/ "#3":ILB3[0]/
                      '$.ID :IN[1] ) (ITST :DO[2]/.EMPTY :MT[0] :DO[2])

STEST= '. .ID?19E((+'(( '):MT[0]/INSIDEPAR:LOAD[1]')?191E)?192E):CALL[
                                                       /.EMPTY:PRIM[1])
      .ID :CALL[1]/
       .SR :STST[1] /
       '( EXP ?20E ') ?21E /
      '+ STEST ?25E :INS[1] /
       '' +.CHR :CTST[1]/
    (.NUM'$ ?23E/'$ :ZRO[0])(.NUM/.EMPTY :IFIN[0]) STEST ?24E :ARB[3] /
       "--" STEST                                ?26E :MNTST[1]   /
       '-   STEST                                ?26E :NTST[1];

INSIDEPAR = !"CALL IDSET" CLOSEPAREN !"CALL IDGET" ;

CLOSEPAREN = => (.COL(72) ERROR1/ --')/'( CLOSEPAREN ?10E ') ?11E
                CLOSEPAREN ?12E ) ;
ERROR1= !('"PRINT "#1/#1,"FORMAT(' NESTING OF PARENTHESES IS WRONG')"\)
           !"CALL RESET" => ';  $( RULE * ) ".END" ?99E !"CALL STPMTA";

OUTRUL = '[ OUTR ?27E (OUTRUL :ALTER[2] / .EMPTY) :OSET[1];

OUTR = OUTEST  "=>" ?29E OUTEXP ?30E :CONCAT[2];

OUTEST = ( (‘] :MT / "-]" :ONE / "-,-]" :TWO / "-,-,-]" :THRE) [0] /
               ITEMS '] )  :CNTCK[1];

ITEMS = ITEM (', ITEMS ?32E :ITMSTR[2] / .EMPTY :LITEM[1]) ;

ITEM = '- :MT[0] /
       .ID '[ ?33E OUTEST ?34E :RITEM[2]/
       NSIMP1 :NITEM[1] /
       '. .ID ?35E :FITEM[1] /
       .SR :TTST[1] /
       ''+.CHR  :CHTST[1] /
       '# .NUM ?37E :GNITEM[1];

REST = OUTT (REST :OER[2]/ .EMPTY) / GEN (REST :DO[2]/ .EMPTY);

OUTT = .ID '[ ?39E ARGLST '] ?40E :OUTCLL[2] / '( OUTEXP ') ?41E /
       NSIMP1 (': ('S :OS / 'L :OL / 'N :ON/ 'C :OC)[0] :NOPT[2] /
               .EMPTY :DOIT[1]);

ARGLST = ARGMNT :ARG[1] (', ARGLST :DO[2] / .EMPTY) / .EMPTY :MT[0];
```

```
ARGMNT = NSIMP :ARGLD[1] / '* .NUM :GENARG[1] ;

NSIMP1 = <- 'Δ .NUM NSIMP :UP[2] / NSIMP :LKT[1];

NSIMP = '* .NUM (<-                ': NSIMP :CHASE[2] / .EMPTY :LCHASE[1]);

GEN1 = (OUT/COMM) (GEN1 :DO[2] / .EMPTY);

GEN = COMM / GENU / '< :TTY[0] / '> :FILE[0];

GENU =  OUT /
        '. .ID?42E((+'(( ')!MT[0]/IASICPAR:LOAD[1]')?43E ?44E ):CALL[;
                              /.EMPTY:CALL[1])/             .ID :CALL[1]/
        '# .NUM :GNL[L[1] (':  :DEF[1] / .EMPTY) ;

OUT = ('\ :OUTCR / ', :OUTAB) [0] /
        .SR :OUTSRL[1] /
        ''+.CHR :OUTCH[1] /
        ¤+W¤ :UPWRK[0]  /
        ¤-W¤ :DWNWRK[0] /
        )(.W¤ :OUTWRK[0] /
        'AW¤  :MAXWRK[0]);

E = .EMPTY .RESET() => ';  %(RULE * ¤LSS=LSSAVE¤) ¤.END¤ ?99E.STPMTA(


%OUT RULES%

SETUP[-] => *1 ;

BEGIN[-,-] => *2, ,CALL INIT¤\ ,¤WRK=0¤\, ¤XWRK=0¤\
        ,¤CALL RLINE¤\,¤CALL MCALL(S¤*1:N ¤,'¤ #1')\
        #1,¤CALL FINISH¤\  , ¤CALL LIMITS¤\ , ¤STOP ENDCMP¤\;


OUTPT[-,-] => ¤C ¤*1\  *1:N,¤CONTINUE¤\ *2 ,¤CALL MRTN¤\;

BALTER[-] => ,¤CALL SAV¤\ *1 ,¤CALL RSTR¤\
        [-,-] => ,¤CALL SAV¤\ *1 ,¤CALL RSTR¤\,¤IF(MFLAG),,¤ #1 \
                 *2 #1,¤CONTINUE¤\;

0., => , ¤CONTINUE¤\ ;

ALTER[-,SET[ ]] => *1 *2
        [CONCAT[-,-],-] =>PMT[*1:*1,#1] *1:*2 ,¤GO TO ¤#2\
                 #1,¤CONTINUE¤\ *2 #2,¤CONTINUE¤\
        [-,-] => *1 ,¤IF(MFLAG),,¤ #1 \
                   *2 #1,¤CONTINUE¤\ ;

PMT[PRIME[-],#1] => ,¤CALL ¤*1:*1:S\,¤IF(MFLAG),¤    #1',   \
```

```
                 ,¤ISTAR = ISTAR+¤*1:*1:S¤FLG¤\,¤CALL KPUSH(ISTAR)¤
          [-,#1] => *1 ,¤IF(MFLAG),¤    #1',\;

ERRALTER[-,SET[ ]]] => *1
          [-] => *1 ,¤IF(MFLAG.EQ.0)CALL BIGERR¤\;

DO[-,-] => *1 *2;

CONCAT[-,-] => *1 ,¤IF(MFLAG),¤ #1 ',  \
                  *2 #1, ¤CONTINUE¤ \';

LOAD[-] => '( *1:S ') ;

CALL[-] => ,¤CALL MCALL($¤*1:N¤,$¤#1')\#1,¤CONTINUE¤\
            [-,-] => , ¤CALL ¤ *1:S *2\ ;

MT / => .EMPTY;

ZR / => ¤0¤;

ERCODE[-,-,ZROL ]] => *1 ,¤IERR=¤*2\,¤IF(MFLAG.EQ.0)CALL ERR(0)¤\
           -[-,-,-] => *1,¤IERR=¤*2\,¤IF(MFLAG.EQ.0)CALL ERR(¤¤*3:N

NOLB[-] => ,¤CALL NOLB($¤*1:N')\;

MKNODE[-] => ,¤CALL MKND(¤*1')\;

ARR[ZROL ],IFIN[ ],-] => #1,¤CONTINUE¤\ *3,¤IF(MFLAG),,¤#1 \
             ,¤MFLAG=1¤\
    [-,-,-] =>,¤CALL SAV¤\
           ,¤CALL MPUSH(-1)¤\
#1,¤CONTINUE¤\   *3
           ,¤MSTACK(MSP)=MSTACK(MSP)+1¤\
           ,¤IF(MSTACK(MSP)-¤*2¤),,¤#2\
           ,¤IF(MFLAG),,¤#1\
           ,¤IF(MSTACK(MSP).GE.¤*1¤)MFLAG=1¤\
           ,¤GO TO ¤#3\
#2,¤MFLAG=0¤\
#3,¤ISTUPD=MPOP(ISTUPD)¤\
           ,¤CALL RSTR¤\ ;

IFTY / => ¤32767¤ ;

INC[-] => ,¤INCFLG=1¤ \ *1 ,¤INCFLG=0¤\;

GO / => ,¤CALL OUTREE¤\,¤IF(MFLAG.EQ.0) CALL CERR(2)¤\;

SET / => ,¤MFLAG=1¤\;

TTY[-] => ¤C YOU NEED A TELETYPE FOR THIS¤\
```

```
    [ ] => ¤C YOU NEED A TELETYPE FOR THIS¤\;

FIL[ ] => ¤C NO TTY¤\;

STRING[-] => *1:L',*1:L'H*1:S;

OSET[-] => ,¤CALL BEGN¤\ *1;

CNTCK[-] => *1 ,¤IF(NCNT.NE.ICNT) MFLAG=0¤\;

ONE / => ,¤ICNT=1¤\;

TWO / => ,¤ICNT=2¤\;

THRE / => ,¤ICNT=3¤\;

ITMSTR [-,-] => *1 ,¤ICNT=ICNT+1¤\,¤KTX=KTX-1¤\ *2;

LITEM [-] => *1 ,¤ICNT=ICNT+1¤\;

RITEM[-,-] => ,¤CALL RI1($¤*1:N¤,$¤*1')\ *2 ,¤CALL RI2¤\ #1,¤CONTINUE

OERR[-,-] => *1, ¤IF(MFLAG.EQ.0) CALL CERR(1)¤\ *2;

OUTCLL [-,-] => ,¤CALL OTCLL1($¤*1:N')\ *2 ,¤CALL OTCLL2($¤#2 ') \
          #2 , ¤CONTINUE¤ \ ;

ARGLD[-] => ,¤IME=N'¤\ *1;

ARG [-] =>  *1 ,¤CALL KPUSH(IME)¤\,¤ICNT=ICNT+1¤\;

CHASE [-,-] => ,¤IME=GET(¤*1:S')\
      ,¤IF(AND(IME,PTRFLG).NE.PTRFLG)CALL CERR(3)¤\ *2;

LCHASE [-] => ,¤IME=GET(¤*1:S')\;

DOIT[-] => *1 ,¤CALL DOIT($¤ #1')\ #1 ,¤CONTINUE¤\;

NOPT [-,-] => *1 ,¤ISTAR=AND(IME,ADRMSK)¤\
                       ,¤IF(AND(IME,PTRFLG).EQ.PTRFLG) CALL CERR(4)¤\ *2;

SCAN [-] => #1,¤CONTINUE¤\ *1 ,¤IF(MFLAG),,¤ #2\
       ,¤NCCP=NCCP+1¤\,¤GOTO ¤#1\ #2,¤CONTINUE¤\;

PRIME[-] => ,¤CALL ¤*1\ ,¤IF(MFLAG.NE.0)CALL KPUSH(ISTAR+¤*1:S¤FLG)¤\;

STST [-] => ,¤CALL TST(¤ STRING[*1] ')\;

CTST [-] => ,¤CALL  TCH(1H¤*1:C')\;
```

```
OS / => ,¤CALL OUTS¤\;

ON / => ,¤CALL OUTN(AND(IME,ADRMSK)+1000)¤\;

OL / => ,¤CALL OUTN(NSYMTB(IME))¤\;

OC / => , ¤FLD(0,6,ISTUPD)=FLD(30,6,IME)¤\
          , ¤CALL CIO(OR(ISTUPD,BLANK5)) ¤\ ;

GNLBL [-] => ,¤CALL GEN(GNLB¤*1')\;

DEF [-] => *1 ,¤CALL LIT(8,8HCONTINUE)¤\;

OUTCR / => ,¤CALL CRLF¤\;

OUTAB / => ,¤CALL TAB¤\;

OUTSR [-] => ,¤CALL LIT(¤ STRINGE*1] ')\;

OUTCH [-] => ,¤CALL  CIO(1H¤ *1:C ')\;

EN N / =>.EMPTY ;

SAVG [-] => ,¤CALL SAVGN¤\ *1 ,¤CALL RSTGN¤\;

NITEM[-] => ,¤IME=KT¤\*1,¤IF(NSTACK(KTX).EQ.IME) MFLAG=0¤\;


FISE [-] => ,¤IF(AND(NSTACK(KTX),¤ *1:S ¤FLG).NE.¤ *1:S
                 ¤FLG) MFLAG=0¤\;

TTST[-] => ,¤CALL SSTEST(¤ STRINGE*1]')\;

CHTST[-] => ,¤IF (FLD(0,6,1H¤*1:C¤).NE.FLD(30,6,NSTACK(KTX)))MFLAG=0

GNITEM[-] => ,¤IF(AND(NSTACK(KTX),GENFLG).NE.GENFLG)MFLAG=0¤\
                 ,¤IF(MFLAG.EQ.1)GNLB¤*1:S¤=AND(NSTACK(KTX),ADRMSK)¤\;

GENARG[-] => ,¤IME=IAB(GNLB¤*1:S¤)+GENFLG¤\;

MNTST[-] =>     ,¤CALL MPUSH(KT)¤\
               ,¤CALL MPUSH(KSP)¤\
             ,¤CALL MPUSH(NCCP)¤\ *1
       ,¤NCCP=MPOP(ISTUPD)¤     \
             ,¤KSP=MPOP(ISTUPD)¤\
             ,¤KT =MPOP(ISTUPD)¤\     ;

[illegible] =>     ,¤CALL MPUSH(KT)¤\
               ,¤CALL MPUSH(KSP)¤\
             ,¤CALL MPUSH(NCCP)¤\ *1
```

```
     ,¤MFLAG=MOD(MFLAG+1,2)¤\  ,¤ICCP=MPOP(ISTUPD)¤     \
          ,¤KSP=MPOP(ISTUPD)¤\
          ;¤KT =MPOP(ISTUPD)¤\         ;

UP[¤1¤,-] => ,¤IME=KSTACK(KSP)¤\*2
      [-,-] => ,¤ITE=KSTACK(KSP-¤*1:S¤+1)¤\*2;
LKT[-] => ,¤IME=KT¤\ *1;

UPWRK / => ,¤WRK=WRK+1¤\, ¤XWRK=MAX0(WRK,XWRK)¤\;

DWNWRK / => ,¤WRK=WRK-1¤\;

OUTWRK /  =>    , ¤CALL OUTN(WRK)¤\;

MAXWRK / => ,¤CALL OUTN(XWRK)¤\;

SIZE['S,-] => ,¤COMMON /NSYMTB/NSYMTB(¤*2:S¤),LXSS,LSS¤ \
              , ¤LXSS=¤ *2:S \
      [.CHR,-] => ,¤COMMON /¤*1:C¤STACK/¤*1:C¤STACK(¤*2:S¤),¤*1:C
          ¤XSP,¤ *1:C ¤SP¤  \
              , *1:C ¤XSP=¤ *2:S \    ;

SIZEF / => ,¤COMMON /KSTACK/KSTACK(500),KXSP,KSP¤ \
              , ¤ KXSP=500¤ \
       ,¤COMMON /MSTACK/MSTACK(500),MXSP,MSP¤\
              , ¤ MXSP=500¤ \
       ,¤COMMON /NSTACK/NSTACK( 700),NXSP,NSP¤\
              , ¤ NXSP=700¤ \
       ,¤COMMON /NSYMTB/NSYMTB(3500),LXSS,LSS¤\
              , ¤ LXSS=3500¤\;

USEKID[-] => ,¤CALL IDSET¤ \          KPOPK[*1]    ,¤CALL IDGET¤\ ;

KPOPK[-] => ,¤CALL MPUSH(KT)¤\
           ,¤CALL MPUSH(KSP)¤\ *1
              ,¤KSP=MPOP(ISTUPD)¤\
              ,¤KT=MPOP(ISTUPD)¤\;

PAROUT[ZRO[ ]] => ,¤IME=KT¤\
      [¤0¤] => ,¤ITE=KT¤\
      [-] => ,¤IME=KSTACK(KSP+1-¤*1')\;


IMF[.SR] =>,*1:S \
   [-]    =>       IF[*1?*],*1:*2,I[,#2,#3]      ;
IDO[-,-,#1,#2,#3] =>  (  B1[*1]#1/  B2[*1]#2/  B3[*1]#3/  IXDO[*1])
         (  DOTST[*2]   IDO[*2:*1,*2:*2,#1,#2,#3]  /.EMPTY     )     ;
IA[.SR]     =>   *1:S
   [I,BL ]     =>    ,
   [ICRE ]     =>    \
```

```
      [.CHR]    =>    *1:C
      [INC-1]      =>*1:*1:N     ;
ILB1 / =>       .EMPTY ;
ILB2 / =>       .EMPTY ;
ILB3 / =>       .EMPTY ;
IT.. / =>        .EMPTY ;
ICR / =>         .EMPTY ;
INP-1]=>        .EMPTY ;
OUTSTLOC[-,-]] =>       .EMPTY ;
B1[TLB1[]] =>    .EMPTY ;
B2[TLB2[]] =>    .ENPTY ;
B3[TLB3[]] =>    .EMPTY ;

 .END TRMETA
```

BIBLIOGRAPHY

1 (BOOK1) Erwin Book, "The LISP Version of the Meta Compiler," TECH MEMO TM-2710/330/00, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406, 2 November 1965.

2 (BOOK2) Erwin Book and D.V. Schorre, "A Simple Compiler Showing Features of Extended META," SP-2822, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406, 11 April 1967.

3 (GLENNIE1) A.E. Glennie, "On the Syntax Machine and the Construction of a Universal Computer," Technical Report Number 2, AD 240-512, Computation Center, Carnegie Institute of Technology, 1960.

4 (KIRKLEY1) Charles R. Kirkley and Johns F. Rulifson, "The LOT System of Syntax Directed Compiling," Stanford Research Institute Internal Report ISR 187531-139, 1966.

5 (LEDLEY1) Robert Ledley and J.B. Wilson, "Automatic Programming Language Translation through Syntactical Analysis," <u>Communications of the Association for Computing Machinery</u>, Vol. 5, No. 3 pp. 145-55, March, 1962.

6 (METCALFE1) Howard Metcalfe, "A Parameterized Compiler Based on Mechanical Linguistics," Planning Research Corporation R-311, March 1, 1963, also in <u>Annual Review in Automatic Programming</u>, Vol. 4, 125-65.

7 (NAUR1) Peter Naur et al., "Report on the Algorithmic Language ALGOL 60," <u>Communications of the Association for Computing Machinery</u>, Vol. 3, No. 5, pp. 299-384, May, 1960.

8 (OPPENHEIM1) D. Oppenheim and D. Haggerty, "META 5: A Tool to Manipulate Strings of Data," Proceedings of the 21st National Conference of the Association for Computing Machinery, 1966.

9 (RUTMAN1) Roger Rutman, "LOGIK, A Syntax Directed Compiler for Computer Bit-Time Simulation," Master Thesis, UCLA, August, 1964.

10 (SCHMIDT1) L.O. Schmidt, "The Status Bit," Special Interest Group on Programming Languages Working Group 1 News Letter, 1964.

11 (SCHMIDT1) PDP-1

12 (SCHMIDT3) EQGEN

13 (SCHNEIDER1) F.W. Schneider and G.D. Johnson, "A Syntax-Directed Compiler-Writing Compiler to Generate Efficient Code," Proceedings of the 19th National Conference of the Association for Computing Machinery, 1964.

14 (SCHORRE1) D.V. Schorre, "A Syntax-Directed SMALGOL for the 1401," Proceedings of the 18th National Conference of the Association for Computing Machinery, Denver Colorado, 1963.

15 (SCHORRE2) D.V. Schorre, "META II, A Syntax-Directed Compiler Writing Language," Proceedings of the 19th National Conference of the Association for Computing Machinery, 1964.

16 Rosen, Saul (ed.), Programming Systems and Languages, McGraw-Hill Book Company, 1967.

17 Feldman, J. and Gries, D., "Translator Writing Systems," Communications of the ACM, February, 1968.

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Computer Science University of Utah Salt Lake City, Utah 84112	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP	

3. REPORT TITLE
THE TREE-META COMPILER-COMPILER SYSTEM: A Meta Compiler System for the Univac 1108 and the General Electric 645

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
technical report

5. AUTHOR(S) *(First name, middle initial, last name)*
C. Stephen Carr, David A. Luther, Sherian Erdmann

6. REPORT DATE March 1969	7a. TOTAL NO. OF PAGES 51	7b. NO. OF REFS 15
8a. CONTRACT OR GRANT NO. AF 30(602)-4277	9a. ORIGINATOR'S REPORT NUMBER(S) TR 4-12	
b. PROJECT NO. ARPA Order No. 829		
c. Program Code Number: 6D30	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* RADC-TR-69-83	
d.		

10. DISTRIBUTION STATEMENT
~~This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of RADC, GAFB, New York.~~

11. SUPPLEMENTARY NOTES Monitored by Rome Air Development Center (EMIIO), Griffiss Air Force, New York 13440	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency, Washington, D.C. 20301

13. ABSTRACT

Tree Meta is a compiler-compiler system for context-free languages. Parsing statements of the metalanguage resemble Backus-Naur Form with embedded tree-building directives. Unparsing rules include extensive tree-scanning and code-generation constructs. Examples in this report are drawn from algebraic and special-purpose languages. The process of bootstrapping from a simpler metalanguage is explored in detail.

This report is based on an earlier one by D. I. Andrews and J. F. Rulifson of Stanford Research Institute which described the SDS 940 version of Tree Meta. The Tree Meta system described in this report was bootstrapped from the SDS 940 with a minimum of hand coding.

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Metalanguage						
compiler-compiler						
parsing						
unparsing						
code generation						
tree building						

Unclassified
Security Classification