A META COMPILER SYSTEM
FOR THE SDS 940

By
D. I. Andrews and J. F. Rulifson
Stanford Research Institute
Menlo Park, California

Note:  This work was supported jointly by:
1) National Aeronautics and Space Administration
   Langley Research Center
2) Rome Air Development Center
   Griffiss Air Force Base
3) Advanced Research Projects Agency
   Department of Defense
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 are drawn from algebraic and special-purpose languages, as well as the process of bootstrapping the comprehensive, self-defining, tree language from a simpler metalanguage. Thorough implementation documentation for the Scientific Data System 940 appears in the discussion of the support subroutines and in the appendices. A history of computer metalanguages, a tutorial guide to Tree Meta, and the practical usefulness and scope of the system are other topics of the report.

This is an interim project report and reflects the current status of a portion of a constantly evolving programming system.

Documentation level as of 29 December 1967 is TM1.3.
1 Introduction

2 Basic Syntax

3 Program Environment

4 Formal Description

5 Detailed Examples

6 Conclusions and Future Plans

7 Bibliography

8 Tree Meta in Tree Meta

9 Tree Meta Support Package

10 Metalib

11 Meta II in a Macro Language (not included)

12 Extended Meta in Meta II (not included)
Outline for a 30 Minute talk on Meta (not included)
Terms such as "metalanguage" and "metacompiler" have a variety of meanings. Their usage within this report, however, is well defined.

1a "Language," without the prefix "meta," means any formal computer language. These are generally languages like ALGOL or FORTRAN. Any metalanguage is also a language.

1b 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.

1c A language which can be used to describe other languages is a metalanguage. English is an informal, general metalanguage which can describe any formal language. Backus-Naur Form or BNF (NAUR1) is a formal metalanguage used to define ALGOL. BNF is weak, for it describes 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.

1d 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 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 standards and scope of Tree Meta.

2a Tree Meta is built to deal with a specific set of languages and an even more specific set of users. This project, therefore, adds to the ever-increasing problem of the proliferation of machines and languages, rather than attempting to reduce it. There is no attempt to design universal languages, or machine independent languages, or any of the other goals of many compiler-compiler systems.

2b 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.

2b1 Tree Meta is intended to parse context-free languages using limited backup. There is no intent or desire on the part of the users to deal with such problems as the FORTRAN "continue" statement, the PL/I "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 take
the easy way out rather than fight old problems.

2b2 Many of the features considered necessary for a compiler-compiler system are absent in Tree Meta. Such things as symbol-tables that handle ALGOL-style blocks and variable types are not included. Neither are there features for multidimensional subscripts or higher level macros. These features are not present because the users have not yet needed them. None, however, would be difficult to add.

2b3 Tree Meta translates directly from a high-level language to machine code. This is not for the faint of heart. There is a very small number of users (approximately 3); all are machine-language coders of about the same high level of proficiency. The nature of the special-purpose languages dealt with is such that general formal systems will not work. The data structures and operations are too diverse to produce appropriate code with current state-of-the-art formal compiling techniques.

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

3a 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 describe a method of generating all the possible strings of the language.
3b 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 compiled.

3c The metacompilers are a combination of both schemes. They are
neither purely productive nor purely reductive, but merge both
techniques into a powerful working system.

4 The metacompiler class of compiler-compiler systems may be
characterized by a common top-down parsing algorithm and a common
syntax. These compilers are expressible in their own language, whence
the prefix "meta."

4a 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 \(V\), nonterminal 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 \(S \Rightarrow u\)
\((S\) produces \(u)\).

4a1 Let
\[ V = [E, T, F, +, *, (, ), X] \]
\[ N = [E, T, F] \]
\[ P = [E := T / T + F, \]
\[ T := F / F * T, \]
\[ F := X / (E)] \]
\[ L = (V, N, P, E) \]

4a2 The following intentionally incomplete ALGOL procedures will perform a top-down analysis of strings in \( L \).

4a2a boolean procedure \( E; \)
\[
E := \begin{cases} 
\text{if } T \text{ then (if issymbol('*') then } E \text{ else true) else false; } & \text{comment issymbol(arg) is a Boolean procedure which compares the next symbol in the input string with its argument, arg. If there is a match the input stream is advanced;} \\
\end{cases}
\]

4a2b boolean procedure \( T; \)
\[
T := \begin{cases} 
\text{if } F \text{ then (if issymbol('*') then } T \text{ else true) else false; } \\
\end{cases}
\]

4a2c boolean procedure \( F; \)
\[
F := \begin{cases} 
\text{if issymbol('X') then true else if issymbol('(') then (if } E \text{ then (if issymbol(')') then true else false) else false) else false; } \\
\end{cases}
\]

4a3 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 ....

4a3a 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 E loops infinitely.

4a3b The solution to the problem used in Tree Meta is the arbitrary number operator. In Tree Meta the first production could be

E ::= T$( "*" T)

where the dollar sign and the parentheses indicate that the quantity can be repeated any number of times, including 0.

4a3c Tree Meta makes no check to ensure that the compiler it is producing lacks syntax rules containing left recursion. This problem is one of the more common mistakes made by inexperienced metalanguage programmers.

4b The input language 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 simply delimited basic entities. The
arbitrary-number operator and parenthesis construction 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.

The early history of metacompilers is closely tied to the history of
SIG/PLAN Working Group 1 on Syntax Driven Compilers. The group was
started in the Los Angeles area primarily through the effort of Howard
Metcalfe (SC/MIDT1).

In the fall of 1962, he designed two compiler-writing
interpreters (METCALFE1). One used a bottom-to-top analysis
technique based on a method described by Ledley and Wilson (LEDLEY1).
The other used a top-to-bottom approach based on a work by Gtennie
(GLENIE1) to generate random English sentences from a context-free grammar.

5b At the same time, Val Schorre described two "metamachines"—one generative and one analytic. The generative machine was implemented, and produced random algebraic expressions. Schorre implemented Meta I, the first metacompiler, on an IBM 1401 at UCLA in January 1963 (SCHORRE1). His original interpreters and metamachines were written directly in a pseudo-machine language. Meta I, however, was written in a higher-level syntax language able to describe its own compilation into the pseudo-machine language. Meta I is described in an unavailable paper given at the 1963 Colorado ACM conference.

5c Lee Schmidt at Bolt, Beranek, and Newman wrote a metacompiler in March 1963 that utilized a CRT display on the time-sharing PDP-1 (SCHMIDT2). This compiler produced actual machine code rather than interpretive code and was partially bootstrapped from Meta I.

6 Schorre bootstrapped Meta II from Meta I during the Spring of 1963 (SCHORRE2). The paper on the refined metacompiler system presented at the 1964 Philadelphia ACM conference is the first paper on a metacompiler available as a general reference. The syntax and implementation technique of Schorre's system laid the foundation for most of the systems that followed. Again the system was implemented on a small 1401, and was used to implement a small ALGOL-like language.
7 Many similar systems immediately followed.

7a Roger Rutman of A. C. Sparkplug developed and implemented LOGIK, a language for logical design simulation, on the IBM 7090 in January 1964 (RUTMANI). This compiler used an algorithm which produced efficient code for Boolean expressions.

7b Another paper in the 1964 ACM proceedings describes Meta III, developed by Schneider and Johnson at UCLA for the IBM 7090 (SCHNEIDERI). Meta III represents an attempt to produce efficient machine code for a large class of languages. It was implemented completely in assembly language. Two compilers were written in Meta III--CODOL, a compiler-writing demonstration compiler, and PUREGOL, a dialect of ALGOL 60. (It was pure gall to call it ALGOL). The rumored METAFORE, able to compile full ALGOL, has never been announced.

7c Late in 1964, Lee Schmidt bootstrapped a metacompiler from the PDP-1 to the Beckman 420 (SCHMIDT3). It was a logic equation generating language known as EQGEN.

8 Since 1964, System Development Corporation has supported a major effort in the development of metacompilers. This effort includes powerful metacompilers written in LISP which have extensive
9 An outgrowth of one of the Q-32 systems at SDC is Meta 5 (OPPENHEIM1) (SCHAFFER1). This system has been successfully released to a wide number of users and has had many string-manipulation applications other than compiling. The Meta 5 system incorporates backup of the input stream and enough other facilities to parse any context-sensitive language. It has many elaborate push-down stacks, attribute setting and testing facilities, and output mechanisms. The fact that Meta 5 successfully translates JOVIAL programs to PL/1 programs clearly demonstrates its power and flexibility.

10 The LOT system was developed during 1966 at Stanford Research Institute and was modeled very closely after Meta II (KIRKLEY1). It had new special-purpose constructs allowing it to generate a compiler which would in turn be able to compile a subset of PL/1. This system had extensive statistic-gathering facilities and was used to study the characteristics of top-down analysis. It also embedded system control, normally relegated to control cards, in the metalanguage.

11 The concept of the metamachine originally put forth by GLENNIE is so simple that three hardware versions have been designed and one actually implemented. The latter at Washington University in St. Louis. This machine was built from macromodular components and has for instructions the codes described by Schorre (SCHORRE2).
A metaprogram is a set of metalanguages rules. Each rule has the form of a BNF rule, with output instructions embedded in the syntactic description.

1a The Tree Meta compiler converts each of the rules to a set of instructions for the computer.

1b 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;

201
2a 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.

2b 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".

2b1 A string of characters between quotation marks (" ) represents a literal string. These literal strings are tested against the input stream as characters are read.

2b2 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.

2b3 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.

2b4 The dollar sign is the arbitrary number operator in the metalanguage. A dollar sign must be followed by a single element, and it indicates that this element may occur an arbitrary number of times (including zero). Parentheses may be used to group a set of elements into a single element as in R1A and R2A.

2b5 The final basic entities may be seen in rule R4A. These represent the basic recognizers of the metacompiler system. 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. In Tree Meta the three recognizers are "identifier" as .ID, "number" as .NUM, and "string" as .SR. There is another basic entity which is treated as a recognizer but does not look for anything. It is .EMPTY and it always returns a value of "true."

3 Suppose that the input stream contains the string X+Y when the rule EXP is invoked during a compilation.

3a 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.

3b 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 in TERM, however, means that the rule has succeeded because TERM has found a FACTOR followed by zero occurrences of "asterisk FACTOR" or "slash FACTOR." Thus TERM returns a "true" value to EXP. EXP now tests for a plus sign and finds it. The input stream advances another character.

3c 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.

3d 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.

4 Tree Meta is different from most metacompiler systems in that it
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.

4a A parse tree is a structural description of the input stream in terms of the given grammar.

4a1 Using the four rules above, the input stream

null
null
null
has the following parse tree

null
null
null
null
null
null
null
null
null
null
null
null
null
null
null
null
null
4a2 In this tree each node is either the name of a rule or one of the primary entities recognized by the basic recognizer routines.

4a3 In this tree there is a great deal of subcategorization. For example, Y is a PRIM which, is a FACTOR, which is the left member of a TERM. This degree of subcategorization is generally undesirable.

4b The tree produced by the metacompiler program is simpler than the one above, yet it contains sufficient information to complete the compilation.

4b1 The parse tree actually produced is

```
null
null
null
ADD
null
null
X
null
null
MULT
null
Y
null
Z
```

4b2 In this tree the names of the nodes are not the rule names of the syntactic definitions, but rather the names of rules which will be used to generate the code from the tree.

4b3 The rules which produce the above tree are the same as the
four previous rules with new syntax additions to perform the appropriate node generation. The complete rules are:

R1B
\[ \text{EXP} = \text{TERM} ("+" \text{EXP} : \text{ADD} / "-" \text{EXP} : \text{SUB}) [2] . \text{EMPTY} ); \]

R2B
\[ \text{TERM} = \text{FACTOR} S ("*" \text{FACTOR} : \text{MULT} / "/" \text{FACTOR} : \text{DIVD}) [2] ) ; \]

R3B
\[ \text{FACTOR} = "-" \text{FACTOR} : \text{MINUS}[1] / \text{PRIM} ; \]

R4B
\[ \text{PRIM} = \text{ID} / \text{NUM} / "(" \text{EXP} ")" ; \]

4c As these 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 elements remove them to make trees. We will step through these rules with the same sample input stream:

R1B
\[ \text{X} + \text{Y} \ast \text{Z} \]

4c1 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.

4c2 PRIM returns to FACTOR, which returns to TERM, which returns to EXP. The plus sign is recognized and EXP is again called.
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.

4c3 PRIM returns to FACTOR, which returns to TERM. The asterisk is recognized and the input is advanced another character.

4c4 The rule TERM now calls FACTOR, which calls PRIM, which recognizes the Z, advances the input stream, and puts the Z on the push-down stack.

4c5 The :MULT in 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.

4c5a The stack now contains
The parse tree is now

```
null
null
null
null
null
```

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` which names the next node 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.

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.
5a 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 metacompiler system is enough for most examples.

5a1 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 (upper-case L on a Teletype) as an element of an unparse rule.

5a2 To make the output more readable, there is a tab feature. To put a tab character into the output stream, one writes a comma as an element of an output rule.

5a3 A literal string can be inserted in the output stream by merely writing the literal string 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 line of code which has L as a label, ADD as an operation code, and SYS as an address, one would write the following string of elements in an unparse rule:

\texttt{\textbackslash{null} \texttt{"L"}, \texttt{"ADD"}, \texttt{"SYS"}}

5a4 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 \(X\).

5a4a Suppose that the expression \(X+Y*Z\) has been parsed and the program is in the ADD unpars 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 a line of code with the operation code ADA and the operand field \(X\), one would write:

`.null , "ADA", *1`

5a4b To generate the code for the left-hand node of the tree one merely mentions "*1" as an element of the unpars 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.

5a5 Generated labels are handled automatically. As each unpars rule is entered, a new set of labels is generated. A label is referred to by a number sign (upper-case 3 on a Teletype) followed by a number. Every time a label is mentioned during the execution of a rule, the label is appended to the output stream. If another rule is invoked in the middle of a rule, all the labels are saved.
and new ones generated. When a return is made the previous labels are restored.

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

6a An unpars rule is a rule name followed by a series of output rules. Each output rule begins with a test of nodes. The series of output rules make up a set of highest-level alternatives. When an unpars 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 unpars rule with the general flag set "true." If no test is successful, a return is made with the general flag "false."
6b 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

```
.null 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.

6b1 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

```
.null 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.

6b2 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.

6b3 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,-].

6b4 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.

6b5 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.
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 a variable and on the right is the tree for an expression. An unparse rule may begin as follows:

```
.null  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

```
.null
.null
.null
.null
.ID
.null
.null
.ID
.add

.null
.null

```

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. The code generated (for the SDS 940) would be the single instruction `MIN X`, which increments the cell \( X \) by one.

Each out-rule, or highest-level alternative, in an unparse rule is also made up of alternatives. These alternatives are separated 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 is 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 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.

**6c2a** Any nonterminal 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.

**6c2b** The other type of test is made by invoking another
umparse 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. 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:

```
.null
ADD[MINUS[-],[-]] ⇒ SUB[*2,*1:*1]
```

6c2b1 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, 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 MUL T were being evaluated, ↑1*1 would be the X.

6c2b2 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—a number sign followed by a number.

6c3 The calls on other unparsable rules may occur anywhere in an 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.

6c4 The rest of an out-expr is made up of output elements appended to the output stream, as discussed above.

6d 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.

6e Out-exprs may be nested, using parentheses, in the same way as the alternatives of the parse rules.

7 There are a few features of Tree Meta which are not essential but do make programming easier for the user.

7a If a literal string is only one character long, 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.

7b 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 a 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 write a question mark followed by a number followed by a rule name after any test except the first in the parse equations. For example:

```
.null
ST = .ID '=' question 2 E EXP question 3 E ';

.null
question 4 E :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, the arrow, and an error comment which contains the number 3, and transfers control to the parse rule E.

7c Comments may be inserted anywhere in a metalanguage program where blanks may occur. A comment begins and ends with a percent sign, and may contain any character except---of course, a percent sign.

7d In addition to the three basic recognizers .ID, .NUM and .SR, there are two others which are occasionally very useful.
7d1 The symbol .LET indicates a single letter. It could be thought of as a one-character identifier.

7d2 The symbol .CHR indicates any character. In the parse rules, +.CHR causes the next character on the input stream to be taken as input regardless of what it is. Leading blanks are not discarded as for .ID, .NUM, etc. The character is stored in a special way, and hence references to it are not exactly the same as for the other basic recognizers. In node testing, if one wishes to check for the occurrence of a particular character that was recognized by a .CHR, one uses the single quote-character construct. When outputting a node item which is a character recognized by a .CHR, one adds a :C to the node indicator. For example, *1:C.

7e Occasionally some parts of a compilation are very simple and it is cumbersome to build a parse tree and then output from it. For this reason the ability to output directly from parse rules has been added.

7e1 The syntax for outputting from parse rules is generally the same as for unparsable rules. The output expression is written within square brackets, however. 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 stack they are kept in.

7e2 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 stack. They are removed by writing an ampersand at the end of the parse rule (just before the semicolon). This causes all input items added to the stack by that rule to be removed. The input stack is thus the same as it was when the rule was called.
1 When a Tree Meta program is compiled by the metacompiler, a machine-language version of the 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 machine-language version of the Tree Meta program to make it complete.

1a The environment of the Tree Meta program, as it is running, is the library of routines plus the various data areas.

1b This section describes the environment in its three logical parts: input, stack organization, and output.

1b1 This is a description of the current working version, with some indications of planned improvements.

2 Input Machinery

2a 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 80 characters is broken into two
lines. This line orientation is necessary for the following:

2a1 Syntax-error reporting

2a2 A possible anchor mode (so the compiler can sense the end of a line)

2a3 An interlinear listing option.

2a4 In the future, characters for the input buffer will be obtained from another input buffer of arbitrary block size, but at present they are obtained from the system with a Character I/O command.

2b 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 (prefixed with a comment character—an asterisk for our assembler). It also checks for an End-of-File, and for a multiple blank character, which is a system feature built into our text files. There is a buffer pointer which indicates which character is to be read from the line buffer next, and RLINE resets that pointer to the first character of the line.

2c 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. A program-character counter (PCC) points to the next character to be read by the program. This may be moved back by the program to effect backup. A library-character counter (LCC) is never changed except by a library routine when a new character is stored in the input window. PCC is used to compute the third pointer, the input-window pointer (IWP). Actually, PCC and LCC are counters, and only IWP points into the array RING which is the character ring buffer. LCC 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 BACK, and is simply the difference between PCC and LCC. BACK is always negative or zero.

2d There are several routines which deal directly with the input window.

2d1 The routine PUTIN takes the next character from the input line buffer and stores it at the input-window position indicated by IWP. This involves incrementing the input-buffer pointer, or calling RLINE if the buffer is empty. PUTIN does not change IWP.

2d2 The routine INC is used to put a character into the input window. It increases IWP by one by calling a routine, IWP, which makes IWP wrap around the ring buffer correctly. If there is
backup (i.e., if BACK is less than 0), BACK is increased by one and INC returns, since the next character is in the window already. Otherwise, LCC is increased by one, and PUTIN is called to store the new character.

2d3 A routine called INCS is similar to INC except that it deletes 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 IWP. From then on, PUTIN is called to store succeeding characters in the input window in the same slot. As long as the current character (at IWP) is a blank, INCS calls PUTIN to replace it with the next character. The nonblank character is then compared with a comment character. INCS returns if the comparison fails, but otherwise skips to the next comment character. When the end of the comment is located, INCS returns to its blank-checking loop.

2d3a Note that comments do not get into the input window. For this reason, BACK should be zero when a comment is found in the loop described above, and this provides a good opportunity for an error check.

2d4 Before beginning any input operation, the IWP pointer must be reset, since the program may have set PCC back. The routine WPREP
computes the value of BACK from PCC-LCC. This value must be between 0 and the negative of the window size. IWP is then computed from PCC modulo the window size.

2d5 The program-library interface for inputting items from the input stream consists of the routines ID, NUM, SR, LET, and CHR. The first four are quite similar. ID is typical of them, and works as follows: First MFLAG is set false. WPREP is called to set up IWP, then INCS is called to get the first character. If the character at IWP 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 GOBL which updates PCC to the last character read in (which was not part of the identifier). That is, PCC is set to LCC+BACK-1.

2d6 The occurrence of a given literal string in the input stream is tested for by calling routine TST. The character count and the string follow the call instruction. 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 PCC to LCC+BACK, and returns. In addition to TST, there is a simple routine to test for a single character string (TCI). It inputs one character (deleting blanks), compares it to the given character and returns false, or adjusts PCC and returns true.

3 Stacks and Internal Organization

3a Three stacks are available to the program. A stack called MSTACK is used to hold return locations and generated labels for the program's recursive routines. Another stack, called KSTACK, 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, 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.

3a1 The operation of MSTACK is very simple. At the beginning of each routine, the current generated labels and the location that the routine was called from are put onto MSTACK. The routine is then free to use the generated labels or call other routines. The routine ends by restoring the generated labels from MSTACK and returning.

3a2 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.

3a2a A terminal node is a 24-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, or CHR) is to read the item from the input stream.
3a2b 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,

```
.null TREE NSTACK KSTACK
.null
.null ADD
.null X MULT
.null Y Z
.null
.null
```

```
null node ptr.
node ptr.
```

```
node.ptr.
SS item X
2
node ADD
SS item Z
SS item Y
2
node.MULT
```

3a2c KSTACK contains terminal nodes (input items) and
nonterminal node pointers which point to nodes already in NSTACK. NSTACK contains nonterminal nodes.

3b String Storage is another stack-like area. All the items read from the input stream by the basic recognizers (except CHR) are stored in the string-storage area (SS). This consists of a series of character strings prefixed by their character counts. An index into SS consists of the address of the character count for a string. Strings in SS are unique. A routine called STORE will search SS for a given string, and enter it if it is not already there, returning the SS index of that string.

3c 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 teletype and gives the line number. A routine called FINISH closes the files, writes the number of cells used for each of the four stack areas (KSTACK, MSTACK, NSTACK, and SS), and terminates the program.

3c1 At many points in the library routines, parameters are checked to see if they are within their bounds. The system error routine is called if there is something wrong. This routine writes a number indicating what the error is, and terminates the
program. In the current version, the numbers correspond to the following errors.

3cla (1) Class codes are illegal

3clb (2) Backup too far

3clc (64) Character with code greater than 63 in ring buffer

3cld (4) Test for string longer than ring size

3cle (5) Trying to output a string longer than maximum string length

3clf (6) String-storage overflow

3clg (7) Illegal character code

3cli (11) MSTACK overflow

3clj (12) NSTACK overflow

3clk (13) KSTACK overflow
There is a set of routines used by Tree Meta which are not actually part of the library, but are loaded with the library for Tree-Meta. They are not included in the library since they are not necessarily required for every Tree Meta program, but more likely only for Tree Meta. They are called "support routines". The routines perform short but frequently needed operations and serve to increase code density in the metacompiler. Examples of the operations are generating labels, saving and restoring labels and return addresses on MSTACK, comparing flags in NSTACK, generating nodes on NSTACK, etc.

4 Output Facilities

4a The output from a Tree Meta program consists of a string of characters. In the future it might be a string of bits constituting a binary program, but at any rate it can be thought of as a stream of data. The output facilities available to the program consist of a set of routines to append characters, strings, and numbers to the output stream.

4a1 A string in SS can be written on the output stream by calling the routine OUTS with the SS index for that string. OUTS checks the SS index and generates a system-error message if it is not reasonable.
4a2 A literal string of characters is written by calling the routine LIT. The literal string follows the call as for TST.

4a3 A number is written using routine OUTS. The binary representation is given, and is written as a signed decimal integer.

4a4 All of the above routines keep track of the number of characters written on the output stream (in CHNO). 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 8-character intervals. The routine CRLF will output a carriage return and a line feed and reset CHNO.

4a5 There are several routines that are convenient for debugging. One (WRSS) will print the contents of SS. Another (WRIW) will print the contents of the input window.
1 This chapter is a formal description of the complete Tree Meta language. It is designed as a reference guide.

1a For clarity, strings which would normally be delimited by quotation marks in the metalanguage are capitalized instead, in this chapter only.

1b Certain characters cannot be printed on the report-generating output media but are on the teletypes and in the metalanguage--their names, preceded by periods, are used instead. They are .exclamation, .question, .pound, .ampersand, .backslash, and .percent.

2 Programs and Rules

2a Syntax

2a1 program = .META .id (.LIST / .empty) size / .CONTINUE $rule .END;

2a2 size = 'siz $(', siz ')' / .empty;

2a3 siz = .chr '=' .num;
2a4  rule = .id ('=' expr (.ampersand / .empty) / ')' '=' gen1 / outrul) ';

2b  Semantics

2b1  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.

2b1a  The mandatory identifier following the string .META in a main file names the rule at which the parse will begin.

2b1b  The optional .LIST, if present, will cause the compiler currently being generated to list input when it is compiling a program.

2b1c  The size construct sets the allocation parameters for the three stacks and string storage used by the Tree Meta library. 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.
2b2 When a file begins with .CONTINUE, no initialization or storage-allocation code is produced.

2b3 There are three different kinds of rules in a Tree Meta program. All three begin with the identifier which names the rule.

2b3a 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.

2b3b Rules with the string / => following the identifier may only be composed of elements which produce output. There is no testing of flags within a rule of this type.

2b3c Unparse rules have a left bracket following the identifier. This signals the start of a series of node tests.

3 Expressions

3a Syntax

3a1 \( \text{exp} = \text{\texttt{\textasciitilde subback}} ('/ \text{exp} / \text{\texttt{\textperiodcentered empty}}) / \text{subexp} ('/ \text{exp} / \text{\texttt{\textperiodcentered empty}}); \)
3a2 suback = ntest (suback / .empty) / stest (suback / .empty);

3a3 subexp = (ntest / stest) (noback / .empty);

3a4 noback = (ntest / stest ('.question .num (.id / '.question ) / .empty) ) (noback / .empty);

3b Semantics

3b1 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 stream.

3b2 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.

3b3 The subexpressions are series of tests. Only subexpressions which begin with a leftarrow are allowed to back up the input stream and rescan it.
3b3a Without the arrow at the head of a subexpression, any test other than the first within the subexpression may be followed by an error code. If the error code is absent and the test fails during compilation, the system prints an error comment and stops. If the error code is present and the test fails, the system prints the number following the question in the error code, and if the optional identifier is given the system then transfers control to that rule; otherwise it stops.

3b3b 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. Normally, backup is over identifiers or words and the buffer is long enough.

4 Elements of Parse Rules

4a Syntax

4a1 ntest = (': .id / '[: .num '] / genp ') (',backslash / .empty) / '< genp ' (',backslash / .empty) / (.CHR / '*') / "=>" / comm;
4a2 genp = genp1 / .empty;

4a3 genp1 = genp2 (genp1 / .empty);

4a4 genp2 = '*(S .num / .empty) (L / C / N / .empty) / genu;

4a5 comm = .EMPTY / '.exclamation .sr;

4a6 stest = '. .id / .id / .sr / ' ( exp ' ) / ' .chr / (.num ' $ / ' $) (.num / .empty) stest / '- (.sr / ' .chr);

4b Semantics

4bl 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.

4bl a The : .id construct names the next node to be put into the tree. The identifier must be the name of another rule.

4bl b The [ .num ] constructs a node with the name used in the last : .id construct, and puts the number of nodes specified after the arrow on the new node in the tree.

4bl c The [ genp ] is used to write output into the normal
output stream during the parse phase of the compilation.

4b1d The < genp > is used to print output back on the user teletype instead of the normal output stream. This is generally used during long compilations to assure the user that the system is still up and running correctly.

4b1e The occurrence of a .chr causes one character to be read from the input stream into a special register which may be put into the tree just as the terminal symbols recognized by the other basic recognizers are.

4b1f An asterisk causes the rule currently in execution to perform a subroutine call to the rule named by the top of the tree.

4b1g The "=>" ntest construct causes the input stream to be moved from its current position past the first occurrence of the next stest. This may be used to skip over comments, or to move the input to a recognizable point such as a semicolon after a syntax error.

4b2 The comm elements are common to both parse and unparse rules.

4b2a The .EMPTY in any rule sets the general flag true.
4b2b The .exclamation-string construct is used to insert patches into the compiler currently being produced. The string following the .exclamation is immediately copied to the output stream as a new line. This allows the insertion of any special code at any point in a program.

4b3 Stests always test the input stream for a literal string or 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.

4b3a A .id construct provides a standard machine-language subroutine call to the identifier. Supplied with the Tree Meta library are subroutines for .id, .num, .sr, .chr, and .let which check for identifier, number, string, character, and letter respectively.

4b3b An identifier by itself produces a call to the rule with the name of the identifier.

4b3c 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.

4b3d The number-$-number construct is the arbitrary-number operation of Tree Meta. m$n preceding an 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.

4b3e The hyphen-string and hyphen-character constructs test in the same way as the literal string and apostrophe-character constructs. After the test, however, the flag is complemented and the input-stream pointer is never moved forward. This permits a test to be sure that something does not occur.

5 Unparse Rules

5a Syntax

5a1 outrul = '][ outr (outrul / .empty);

5a2 outr = items ']' "=" " outexp;

5a3 items = item (', items / .empty);

5a4 item = '- / .id '][ outest / nsimp1 / '. .id / .sr / '.chr /
5b Semantics

5b1 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.

5b2 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 test within the alternation, but not complete failure as with the parse rules.

5b3 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.

5b4 If no alternation is successful a false value is returned.

5b5 Each item of an unparse alternation test may be one of five different kinds of test.

5b5a A hyphen is merely a test to be sure that a node is there. This sets up appropriate flags and pointers so that the node may be referred to later in the unparse expression if the complete match is successful.

5b5b 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.

5b5c 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.

5b5d The .id, .num, .chr, .let, or .sr checks to see if the node is terminal and was put on the tree by a .id recognizer,
num recognizer, etc. during the parse phase. This test is very simple, for it merely checks a flag in the upper part a word.

5b5e If a node is a terminal node in the tree, and if it has been recognized by one of the basic recognizers in meta, 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 a .sr recognizer; it can be any string, even one put in with a .let.

5b5f If the node is terminal and was generated by the .chr recognizer it may be matched against another specific character by writing the apostrophe-character construct as an item.

5b5g Finally, the node may be tested to see if it is a generated label. The labels may be generated in the unparsable expressions and then passed down to other unparsable rules. The test is made writing a .pound-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 unparsable expression as the number following the .pound.
6a Syntax

6a1 outexp = subout ('/ outexp / .empty);

6a2 subout = outt (rest / .empty) / rest;

6a3 rest = outt (rest / .empty) / gen (rest / .empty);

6a4 outt = .id '[' arglst ']' / '(' outexp ')' / nsimp1 (':' (S / L / N / C) / empty);

6a5 arglst = argmnt (',', arglst / .empty) / .empty;

6a6 argmnt = nsimp / '.pound .num;

6a7 nsimp1 = '+' nsimp / nsimp;

6a8 nsimp = '* .num (':' nsimp / .empty);

6a9 gen1 = (out / comm) (gen1 / .empty);

6a10 gen = comm / genu / '< / >';

6b Semantics
6b1 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.

6b2 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.

6b3 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.

6b3a 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 .pound followed by a number. Only the numbers 1 and 2 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.
6b3b Any already constructed node also may be put into the tree in this new position. The old node is not removed--rather a copy is made. An asterisk-number construct refers to nodes in the same way as the highest-level alternation.

6b4 This process of making new structures from the already-existing tree is a very powerful way of optimizing the compiler and condensing the number of rules needed to handle compilation.

6b5 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.

6b6 Just as in the parse rules, brokets may be used to send immediate printout to the user Teletype.

6b7 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.

6b7a If the node specified is the name of another rule, then
control is transferred to that node by the standard subroutine linkage.

6b7b If the node is terminal, then the terminal string associated with the node is copied onto the output stream.

6b7c 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-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.

6b7d 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.

6b7d1 :s puts out the literal string
6b7d2 :l puts out the length of the string as a decimal number.

6b7d3 :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.

6b7d4 :c puts out the character if the node was constructed with a .chr recognizer.

7 Output

7a Syntax

7a1 genu = out / '. .id ' [ ( .id / .num ) / .empty ] ' ] / ' .pound .num ( ':' / .empty ) ;

7a2 out = ( '.backslash / ' , / .sr / ' .chr / "+w" / "-w" / ".w" / ".pound" ) ;

7b Semantics

7b1 The standard primitive output features include the following:
7b1a Write a carriage return with a backslash

7b1b Write a tab with a comma

7b1c Write a literal string by giving the literal string

7b1d Write a single character using the apostrophe-character construct

7b1e 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 and writes the current value of the counter onto the output stream. -W subtracts one from the counter and does not write anything. .W writes the current value without changing it. Finally, .pound W writes the maximum value that the counter ever reached during the compilation.

7b2 The .id [ (.num/.id) ] is used to generate a call (940 BRM instruction) with a single argument in the A register. It has been used mostly as a debugging tool during various bootstrap sessions with the system. For example, .CERR[5] generates a call to the subroutine CERR with a 5 in the A register.

7b3 .pound 2 means "define generated label 2 at this point in the
program being compiled." It writes the generated label in the output stream followed by an EQU *. This construct is added only to save space and writing.
1 This section of the report is merely the listings of compilers for two languages.

2 The first language, known as SAL for "small algebraic language," is a straightforward algebraic ALGOL-like language.

3 The second example resembles Schorre's META II. This is the original metacompiler that was used to bootstrap Tree Meta. It is a one-page compiler written in its own language (a subset of Tree Meta).
META PROGRAM

PROGRAM = "PROGRAM" DEC $ ( DEC *) : STARTN[0] ST * $("; ST *)
"FINISH" ?E : ENDN[0] * FINISH ;


E = RESET => "; $(ST *) "END" ?99E : ENDN[0] * FINISH ;

ST = IFST / WHILEST / FORST / GOST / IOST / BLOCK /


WHILEST = " WHILE" EXP " DO" ST : WHL[2] ;

FORST = " FOR" VAR " EXP " BY" EXP " TO" EXP " DO" ST : FOR[5] ;

GOST = " GO" " TO" . ID : GO[1] ;

 " CLOSE" . ID : CLSFIL[1] /

IDLIST = VAR ( IDLIST : DO[2] / . EMPTY ) ;


BLOCK = " BEGIN" ST $("; ST : DO[2] " END" ;


UNION = INTERSECTION ( '/' UNION : OR[2] / . EMPTY ) ;


NEG = " NOT " NEGN [1] / RELATION ;

NEGN = " NOT " NEG / RELATION : NOT[1] ;

RELATION = SUM( " <=" SUM : LE /
 " <" SUM : LT /
 " >=" SUM : GE /
 " >" SUM : GT /
 " =" SUM : EQ /
 " #" SUM : NE ] [2] / . EMPTY ) ;
SUM = TERM ( ('+ SUM :ADD/ ' - SUM :SUB)(2) / EMPTY));
TERM = FACTOR ( ('* TERM :MUT :/ TERM :DIVID/ ' TERM :REM)(2) / EMPTY));
FACTOR = '- FACTOR :MINUS[1] / '+' FACTOR / PRIMARY;
PRIMARY = VARIABLE / CONSTANT / '(' EXP');
VARIABLE = .ID :VAR[1];
CONSTANT = .NUM :CON[1];
SIFTE[-,-,] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU "\ *2 SIFTE[2,3];
SIFTE[1,2,3] => ','BRU",#2 \ #1,"EQU "\ *2 ,#2,"EQU "\ ;
SI[] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU "\ *2 ,#2,"EQU "\ ;
WHL[-,] => #1,"EQU "\ WHL[1,*1,#2] 2,"BRU",#1 \ #2,"EQU "\ ;
WHL[1,-#2] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU "\ ;
GOL[-] => ,"BRU","*1;"
FOR[-,-,-,-] => "DO NOT USE FOR STATEMENTS;"
LBL[-] => *1,"EQU "\ ;
AIF[-,-,-] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU "\ ACC[2] AIF[2,3];
LOPR[OR[-,-],#1,-] => LOPR[*1:*1,#1,#2] BRT[*1:*1,#1]
#2,"EQU "\ LOPR[*1:*2,#1,1];
[AND[-,-],[1] => LOPR[*1:*1,#1,#2] BRF[*1:*1,#1]
#2,"EQU "\ LOPR[*1:*2,#2,1];
[NOT[-,-],#1,-] => LOPR[*1,*1,#2,1]
[->,#1] => *EMPY;"
BRT[OR[-,-],#1] => BRT[*1:*2,#1]
[AND[-,-],#1] => BRT[*1:*2,#1]
[NOT[-,-],#1] => BRF[*1:*1,#1]
[LEE[-,-],#1] => BLEE[*1:*1,*1:*2,#1]
[LT[-,-],#1] => BLE[*1:*1,*1:*2,#1]
[EQ[-,-],#1] => BEE[*1:*1,*1:*2,#1]
[GE[-,-],#1] => BGE[*1:*1,*1:*2,#1]
[GT[-,-],#1] => BLE[*1:*2,*1:*1,#1]
[NLE[-,-],#1] => BNEE[*1:*1,*1:*2,#1]
[->,#1] => ACC[1],"SKE = 0" ,"BRU",#1;"
[LEC',=] #1] => BLE[*1:*2,*1:*1,#1]
[LT(-,=] #1] => BGE[*1:*1,*1:*2,#1]
[EG(-,=] #1] => BLE[*1:*1,*1:*2,#1]
[GE(-,=] #1] => BLE[*1:*1,*1:*2,#1]
[GT(-,=] #1] => BLE[*1:*1,*1:*2,#1]
[NE(-,=] #1] => BLE[*1:*1,*1:*2,#1]
[=,-,1] => ACC[*1],"SKA" = 1","BRU",#1;

BLT(-,-,1] => (TOKEN[*1] ACC[*2],"SKE"*1,"SKG"*1/ 
    WORK[*1] ACC[*2],"SKE","T","W-W")
   ,"BRU"*2","BRU","#1;

BLE(-,-,1] => (TOKEN[*2] ACC[*1],"SKG","*2"/
    TOKEN[*1] ACC[*2],"SKG","*1","BRU"*2"/
    WORK[*2] ACC[*1],"SKG","T","W-W")
   ,"BRU","#1;

BEG(-,-,1] => (TOKEN[*2] ACC[*1],"SKE","*2"/
    TOKEN[*1] ACC[*2],"SKE",*1/
    WORK[*2] ACC[*1],"SKE","T","W-W")
   ,"BRU","#1;

BGE(-,-,1] => (TOKEN[*1] ACC[*2],"SKE","*1","SKG","*1/
    WORK[*1] ACC[*2],"SKE","T","W-W")
   ,"BRU","#1;

BNE(-,-,1] => (TOKEN[*2] ACC[*1],"SKE","*2"/
    TOKEN[*1] ACC[*2],"SKE","*1/
    WORK[*2] ACC[*1],"SKE","T","W-W")
   ,"BRU","#1;

STORE(-,VAR[*13] => "IS IT'S ALREADY THERE"
    [-,ADDVAR[*1],CONT["1"]]) => "MIN","*1"
    [-,ADDVAR[*1],O]) => ACC[*2:"2","ADM","*1"
    [-,SUBVAR[*1],O]) => ACC[*2:"2","CNA","ADM","*1"
    [-,]) => BREG[*2],"STA","*1"
        ACC[*2],"STA","*1"

ADD[MINUS,-] = SUB[*2,"1:*1"1]
    [-,]) => TOKEN[*2] ACC[*1],"ADD",*2/
        WORK[*1] ACC[*2],"ADD","T","W-W"

SUB(-,]) => TOKEN[*2] ACC[*1],"SUB","*2"/
    TOKEN[*11] (BREG[*2],"CBA; CNA; ADD"*1 /
        ACC[*2],"CNA; ADD"*1/)
        WORK[*2] ACC[*1],"SUB","T","W-W"

MINUS(-,] => TOKEN[*1],"LDA","*1","CNA"
    BREG[*11],"CBA; CNA"
    ACC[*1],"CNA"

DIVID(-,]) => TOKEN[*2] (BREG[*1],"CBA"/
    ACC[*11],"RSH 23; DIV"*2/)
    WORK[*2] (BREG[*1],"CBA")/
    ACC[*11],"RSH 23; DIV"*2"/
BREG[MULT[-,-]] => TOKEN[*1;*2] ACC[*1;*1] "MUL"*,1;*2", RSH 1" / 
TOKEN[*1;*1] ACC[*1;*2] "MUL"*,1;*1", RSH 1" / 
WORK[*1;*1] ACC[*1;*2] "MUL", "T+", W-W", RSH 1" / 
[REM[-,-]] => TOKEN[*1;*2] (BREG[*1;*1], "CBA" / 
ACC[*1]) "RSH 23; DIV "*1;*2" / 
WORK[*1;*2] (BREG[*1;*1], "CBA" / 
ACC[*1;*1]) "RSH 23; DIV T+", W-W", RSH 1" / 

ACCL[] => TOKEN[*1] "LDA", *1", BREG[*1] "CBA" / 
*1; 

WORK[-] => BREG[*1] "STB", "T+" W / 
ACC[*1] "STA", "T+" W; 

TOKEN[VAR[*1;ID]] => *EMPTY 
[CON[*NUM]] => *EMPTY; 

MULT / => *EMPTY; 
REM / => *EMPTY; 
AND / => *EMPTY; 
OR / => *EMPTY; 
NOT / => *EMPTY; 
ENDN / => "T", "BSS", W, "END"; 

VAR[*ID] => *1; 
CON[*NUM] => '*1; 
LE / => *EMPTY; 
LT / => *EMPTY; 
EQ / => *EMPTY; 
GE / => *EMPTY; 
GT / => *EMPTY; 
NE / => *EMPTY; 

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

OPNINP[-,-] => "CLEAR; BRS 15; BRU "*2"; BRS 16; BRU "*2"; STA "*1; 

OPNOUT[-,-] => "CLEAR; BRS 16; BRU "*2"; LDX =3; BRS 19; BRU " 
*2"; STA "*1;
CLSFILE => "LDA "*1""; BRS 20\n;

BRS38[.ID] => "LDA "*1""; LDB =10; BRS 38; STA "*2"\n
XCI0[.ID] => "LDA "*1""; STA "*2"

OUTCAR[.ID] => "LDA "*2""; CIO "*1"

OUTNUM[.ID] => "LDA "*1""; LDA =10; BRS 38; "\n
STARTN / => "START","EQU",""\n
DECN[.ID] => *1,"BSS 1"

END
```
.PROGRM = "META" .ID ?1? <"META II 1.1">
  [" NOLIST EXT NUL; $START BRM INITL"]
  ["SKSTKSZ EQU 1; SMSTKSZ EQU 100; SMSTKSZ EQU 1; $SSSIZE EQU 550"]
  ("LIST" ["CLA; STA LISTFG"] / "EMPTY")
  ["BRM RLINE; BRM ";; BRM FINISH"]
  (" SIZ ", " SIZ") ?17E / "EMPTY"
  $ST "END" ?2E
  ["STAR BSS 1; SSTOP DATA SS+SSSIZE-5; SSS BSS SSSIZE"]
  ["SMSP DATA MSTK; $MSP DATA MSTK+MSTK-5; SMSTK BSS MSTKSZ"]
  ["NSNP DATA NSTK; $NSP DATA NSTK+NSTK-5; $NSTK BSS NSTKSZ"]
  ["SKSP DATA KSTK; $SKP DATA KSTK+KSTK-5; $KSTK BSS KSTKSZ"]
  ["END"] ;("DONE")

  ST = .ID '=' ?3E <"ST"> [*,"ZRO; LDA *-1; BRM CLI."]
  EXP ?4E '; ?5E ["BRU RTN"];

  EXP = SUBEXP $(' .LDA MFLAG; SKE =0; BRU "*1")
    SUBEXP ["EQU "]

  SUBEXP = (GEN / ELT ["LDA MFLAG; SKE =1; BRU "])
  $REST (*$1"EQU "]

  REST = GEN / ELT ["LDA MFLAG; SKE =0; BRU "]
  (? , NUM ?12E ["LDA =" ] BRM ERR]"
      (* $BRR">*/ ? [", BRS EXIT"] ?13E /

  ELT = .ID ?1E ["BRM","*"; STA STAR"] /
    .ID ["BRM","*"]
    .SR ["BRM TST; DATA "; ASC "]
    ( EXP ?7E ") ?8E /
    "CHR ["LDA =" N; BRM TCH"]

  GEN = [ SOUT ']' ?10E ["BRM CRLF"] /
    ["EMPTY"] ELT ?9E
    ("LDA MFLAG; SKE =0; BRU "] MIN MFLAG") /
    ["CHR"] ["BRM WPREP; BRM INC; LDA* IWP; STA STAR; MIN NCCP") /
     ("SJR ?12E '>' ?13E ["BRM LIT; DATA "; ASC "] MIN NCCP; BRU "])

  OUT = .SR ["BRM LIT; DATA "; ASC "] /
    ["BRM TAB"] /
    (* ( NUM ["LDA =" 47]; CIO FNUMO; MIN CHNO; LDA GN"
    "*"; BRM GENLAB; STA GN"*"; BRM OUTN") /
     'L ["LDA* STAR; BRM OUTF"] /
     'N ["LDA STAR; BRM OUTF"] /
     'C ["LDA STAR; CIO FNUMO; MIN CHNO"] /
     "EMPTY ["LDA STAR; BRM OUTS"] /

  E = "BRU RTN" $ST "END" ?11E ["END"] FINISH;

  SIZ = "K" = .NUM ["SKSTKSZ EQU "] /
    "N" = .NUM ["SMSTKSZ EQU "] /
    "M" = .NUM ["MNSTKSZ EQU "] /
    "S" = .NUM ["SSSIZE EQU "]; "END"
```
1 Since the work on Tree Meta is still in progress, there are few conclusions and plentiful future plans.

2 (TAKE THIS BRANCH OUT FOR THE HOME REPORT.) This report needs extension in two areas, as well as constant updating as the system evolves.

2a Section 5 should be completed. This was intended to be a detailed example of a small algebraic-language compiler written in Tree Meta. The language is essentially completed, but the accompanying explanations are not.

2b Somewhere within the report there should be a thorough discussion of the bootstrap technique of meta.

3 There are many research projects that could be undertaken to improve the Tree Meta system.

3a Something which has never been done, and which we feel is very important, is a complete study of the compiling characteristics of top-down analysis techniques. This would include an accurate study of where all the time goes during a compilation as well as a study of the flow of control during both parse and unparse phases for
different kinds of compilers and languages. At the same time it would be worthwhile to try to get similar statistics from other compilers. It may be possible to interest some people at Stanford in cooperating on this.

3b SDC has added an intermediate phase to their metacompiler system. They call it a bottom-up phase, and it has the effect of putting various attributes and features on the nodes of the tree. This allows one to write simpler and faster node-matching instructions in the unparse rules. We would like to investigate this scheme, for it appears to hold the potential for allowing the compiler writer to conceptualize more complex tree patterns and thus utilize the node-matching features to a fuller extent.

3c Yet another intermediate phase could be added to Tree Meta which would do transformations on the tree before the unparse rules produce the final code. In attempts to write compilers in Tree Meta to compile code for languages with complex data structures (such as algebraic languages with matrix operations or string-oriented languages with tree operations) and to make these compilers produce efficient code, we have found that tree transformations similar to those used for natural-language translation allow one to specify easily and simply the rules for tree manipulation which permit the unparse rules to produce efficient, dense code. Implementation of the tree-transformation phase into the Tree Meta system would be an
extensive research project, but could add a completely new dimension to the power of Tree Meta.

3d There are a series of additions, some very small and some major, which we intend to add to Tree Meta during the next year.

3d1 Other metacompiler systems have had a construct which allows nodes to have an arbitrary number of nodes emanating from them. This requires additions in parse rules to specify such a search, additions in the node-matching syntax, and additions in the output syntax to scan and output any number of branches.

3d2 We have always felt that it would be nice to have the basic recognizers such as "identifier" defined in the metalanguage. There have been systems with this feature, but the addition has always had very bad effects on the speed of compilation. We feel that this new freedom can be added to Tree Meta without having telling effects on the compilation speed.

3d3 The error scheme for unparsable rules is rather crude--the compiler just stops. We would like to find a reasonable way of accommodating such errors and putting the recovery-procedure control in the metalanguage.

3d4 Currently the unparsable rules expand into 6 times as many
machine-language instructions as the parse rules. This happens because we did not choose the most appropriate set of subroutines and common procedures for the unparses rules. Without changing the syntax of Tree Meta or the way the stacks work, we feel that we can reduce the size of the unparses rules by a factor of 4. This would free a considerably larger amount of core storage for stacks and enlarge the size of programs which Tree Meta could handle. It would also make it run faster in time-sharing mode since less would have to be swapped into core to run it.

3d5 In doing some small tests on the speed of Tree Meta we found that better than 80 percent of the compilation time is spent outputting strings of characters to the system. The code that Tree Meta now produces is the simplest form of assembly code. It would be a very simple task to make Tree Meta able to directly produce binary code for the loader rather than symbolic code for the assembler. A similar change could also be made to output absolute code directly into core so that Tree Meta could be used as the compiler for systems that do incremental compilation.

3e Finally, there is the following list of minor additions or changes to be made to the Tree Meta system.

3e1 Make the library output routines do block I/O rather than character I/O. This could cut compilation times by more than 70
3e2 Fix Tree Meta so that strings can be put into the tree and passed down to other unparse rules. This would allow the unparse rules to be more useful as subroutines and thus cut down the number of unparse rules needed in a compiler.

3e3 Finally, we would like to add the ability to associate a set of attributes with each terminal entity as it is recognized, to test these attributes later, and to add more or change them if necessary. To do this we would associate a single 24-bit word with the string when it is put into string storage and add syntax to the metalanguage to set, reset, and test the bits of the word.


11 (SCHMIDT2) PDP-1

12 (SCHMIDT3) EQGEN


14 (Schorre1) D. V. Schorre, "A Syntax-Directed SMALGOL for the 1401,"

META PROGRAM %TREE 1.3%

PROGRAM = ""META"" + ID ? 1? ""LIST"" :LIST[0] / + EMPTY :MT[0] SIZE
BEGIN[3] /
""CONTINUE"" :MT[0] ) <"TREE 1.3"> :SETUP[1] * $( RULE * )
""END"" ?2E :ENDN[0] * ""DONE"";

SIZE = ( SIZ $(' SIZ :DO[2] ') ? 50E / + EMPTY :MT[0];
SIZ = + CHR ' ' = $54E + NUM $55E :SIZS[2];

RULE = + ID
' ' "=" $3E GEN1 + SIMP[2] /

SUBEXP ('/ EXP $9E :ALTER[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
(NOBACK :DO[2] / + EMPTY);

'[ ( + NUM ' ' ] $14E :MKNODE[1] /
("+CHR" :GCHR /
"* =GO) [0] /
COMM;

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

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

GENP2 = ' ' ('S + NUM $51E :PAROUT[1] / + EMPTY :ZRO[0] :PAROUT[1])

COMM = " + EMPTY" :SET[0] /
'! + SR ?18E :IMEDI[1];

+ ID :CALL[1] /
+ SR :STST[1] /
'( EXP ?20E ') ?21E /
' ' + CHR :CTST[1] /

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

OUTEST = ( (') :MT :"-1" :ONE / :"--" :TWO / :"--.--" :THRE) [0] / 


ITEM = ' :MT[0] / 

.ID [' :33E OUTEST ?34E :RITEM[2]/ 

NSIMP1 :NITEM[1] / 

'S :TTST[1] / 

'CHR :CHTST[1] / 

'NUM ?37E :GNITEM[1]; 

OUTEXP = SUBOUT ('/ OUTEXP :ALTEN[2] / .EMPTY); 


.EMPTY :DOIT[1]); 


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

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

GENU = OUT / 


.EMPTY :CALL[1]) '] / 


OUT = (/: :OUTCR / : :OUTAB) [0] / 


'CHR :OUTCH[1] / 


"W" :DNWRK[0] / 

"W" :MT[0] :OUTWRK / 

'"W" :MAXWRK[0]; 

E = .EMPTY RESET => '; $( RULE * ) ":: END" ?99E FINISH;
%OUT RULES%

SETUP [-] => "NOLIST NUL, EXT: GEN OPD 101B5, 1, 1; BF OPD 102B5, 1, 1"
 "BT OPD 103B5, 1, 1; FSHN OPD 104B5, 1, 1; FSHK OPD 105B5, 1, 1"
 "MKND OPD 106B5, 1, 1; NTLBL OPD 107B5, 1, 1; GET OPD 110B5, 1, 1"
 "BPTR OPD 111B5, 1, 1; BNPTR OPD 112B5, 1, 1; RI 1 OPD 113B5, 1, 1"
 "RI 2 OPD 114B5, 2; FLGT OPD 115B5, 1, 1; BE OPD 116B5, 1, 1"
 "LAB OPD 117B5, 1, 1; CE OPD 120B5, 1, 1; LDKA OPD 121B5, 1, 1"
 "$KSTKSZ EQU 100; $STKSZ EQU 130; $STKSZ EQU 1300; $STKSZ EQU 1400"

BEGIN[-,-,-] => "$START BRM INIT; CLA; STA WRK; STA XWRK" *3 *2
 "BRM RLINE; BRM "*1"; BRM FINISH"

LIST / => " CLA; STA LISTFG;"

OUTPT[-,-] => *1:S "ZRO; LDA -*1; BRM CLO" *2: "BRU RTNO"

SIMP[-,-] => *1, "ZRO" *2, "BRR "*1;

BALTER[-] => "BRM SAV" *1, "BRM RSTR"
 [-,-] => "BRM SAV" *1, "BRM RSTR; BT "#1 *2, #1.D[];

D / => "EQU "*1;"

ALTER[-,SET[]] => *1 *2
 [CONCAT[-,-],-] => PMT[*1:*1,#1] *1:*2, "BRU "#2, #1.D[] *2 #2.D[]
 [-,-] => *1, "BT "#1 *2, #1.D[];

PMT[PRIM[-],#1] => "BRM "*1:*1:S"; BF "#1"; MRG "*1:*1:S"FLG; PSHK =0"
 [-,-] => *1, "BF "#1;"

ER[ALTER[-,SET[]]] => *1
 [-] => *1, "BE =-1"

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

CONCAT[-,-] => *1, "BF "#1 *2, #1.D[];

LOAD[NUM] => "LDA ="*1:S
 [*1D] => "LDA "*1:S;

CALL[-] => "BRM "*1:
 [-,-] => *2, "BRM "*1;

MT / => .EMPTY;

CLA / => "CLA;"

ZRO / => "0";
ERCD[-,-,-] => *1 *2 ;"BE "*3;\nNDLE[->] => "NDRL ="*1;\nMKNODE[-] => ,"MKND ="*1;\nARBZRO[-,MT[-,-] => #1 DC] *3 ;"BT "; MIN MFLAG"\n[NUM,MT[-,-] => ARB[1] #1 DC] *3
"SKR* MSP; BT "; SKN* MSP; BRU *+3; BT "; MIN MFLAG"\nARB[1,-] => ,"BRM SAVJ; LDA ="*1 S"*1; MIN MSP; STA* MSP";\nARB[2,-,NUM] => ,"BRU *+4; CLA; STA MFLAG; BRU *+4; LDA* MSP; SKG ="*2
"-"*1; MIN MFLAG" ARB3[J
[->] => ,"BRU *+3; CLA; STA MFLAG" ARB3[J;\nARB3 / => ,"LDA =-1; AID MSP; BRM RSTR";\nGCHR / => ,"BRM WPREP; BRM INC; LDA* IWP; MRG CHRFLG; MIN NCCP; PSKH =0"X
GO / => ,"BRM OUTREE; BT *+3; LDA =2; BRM ERR";\nSET / => ,"LDA =1; STA MFLAG";\nTTY[-] => TTY[J *1 FIL[J
[+] => ,"LDA =1; STA FNUMO; XCHCH[J]\nFIL[J] => ,"LDA XFNUMO; STA FNUMO";\nXCHCH/ => ,"LDA TCHNO; XMA CHNO; STA TCHNO";\nSTRING[-] => " DATA "*1;L; ASC "*1";\nOSET[-] => ,"BRM BEGN"*1;\nCNTCK[-] => *1 ,"CLB; SKE NCT; STB MFLAG";\nONE / => ,"LDA =1";\nTWO / => ,"LDA =2";\nTHRE / => ,"LDA =3";\nITMSTR [-,-] => *1 ,"MIN CNT; EAX -1,2"; *2;\nLITEM [-] => *1 ,"MIN CNT; LDA CNT";\nRITEM[-,-] => ,"RI1 ="*1; BRU ";*2 ,"RI2"; #1 DC];\nOER[-,-] => *1 ,"CE =1"; *2;
OUTCLL [-, -] => ,"LDA NSP; STA NSP; NDLBL ="*1"; CLA; STA CNT"
    ,"LDA HT; STA ME"*2
    ,"WEND CNT; PSHN NSP; LDX HT; BRM* 0, 2; BRM POPK"
    ,"LDA* NSP; STA NSP"
ARGLD [-] => ,"LDA ME"*1;
ARG [-] => *1 ,"PSHK =0; MIN CNT"
CHASE [-, -] => ,"GET ="*1"; BPTR ++3; LDA =3; BRM CERR"*2;
LCHASE [-] => ,"GET ="*1";
DOIT [-] => *1 ,"BNPTR "*1
    ,"CAE; PSHK =0; BRM* 0, 2; BRM POPK; BRU **2"
    ,"#1* D\] ,"BRM OUTS"
NOPT [-, -] => *1 ,"BNPTR ++3; LDA =4; BRM CERR"
    ,"*2;
SCAN [-] => #1* D\] *1 ,"BT ++3; MIN NCCP; BRU "*1\]
PRIM [-] => ,"BRM "*1"; BF ++3; MRG "*1"FLG; PSHK =0"
    ,"*1"
STST [-] => ,"BRM TST;" STRING(*1]
CTST [-] => ,"LDA ="*1:N"; BRM TCH"
    ,"*1;"
OS / => " BRM OUTS"
    ,"*1;"
ON / => " ETR =77777B; BRM OUTN"
    ,"*1;"
OL / => " CAE; LDA 0, 2; BRM OUTN"
    ,"*1;"
OC / => " ETR =377B; CIO FNUMO; MIN CHNO"
    ,"*1;"
GNLBL [-] => ,"GEN GNLB"*1;
DEF [-] => *1 ,"BRM LIT; DATA 6; ASC "'" EQU "'"'"
    ,"*1;"
OUTCR / => ,"BRM CRLF"
    ,"*1;"
OUTAB / => ,"BRM TAB"
    ,"*1;"
OUTSR [-] => ,"BRM LIT; " STRING(*1]
    ,"*1;"
OUTCH [-] => ,"LDA ="*1:N"; CIO FNUMO; MIN CHNO"
    ,"*1;"
ENDN / => " SSTOP DATA SS+SSTKSZ-5; $SS BSS SSTKSZ"
    ,"MSP DATA MSTK; $MSP DATA MSTK+MSTKSZ-5; $MSTK BSS MSTKSZ"
    ,"NSP DATA MSTK; $NSP DATA MSTK+MSTKSZ-5; $NSP BSS MSTKSZ"
    ,"KSP DATA KSTK; $KSPT DATA KSTK+KSTKSZ-5; $KSTK BSS KSTKSZ"
    ,"WRK BSS 1; XWRK BSS 1; END"
SAVG [-] => ,"BRM SAVGN"*1 ,"BRM RSTGN;"
IME[ ] => "*1:

NITEM[ ] => "STX INDX; LDA KT" *1
       "CLB; LDX INDX; SKE 0,2; STB MFLAG"

FITEM[ ] => "FLGT "*1:S"FLG"

TTST[-] => "BRM STEST;" STRING[*1];

CHTST[-] => "CLB; LDA ="*1:N"; MRG CHRFLG; SKE 0,2; STB MFLAG"

GNITEM[-] => "FLGT GENFLG; ETR =77777B; STA GNLB"*1:S

GENARG[-] => "LAB GNLB"*1:S; MRG GENFLG"

NTST[-] => "LDA NCCP; STA SNCCP"*1
       "LDA =1; SKR MFLAG; BRU ++2; STA MFLAG; LDA SNCCP; STA NCCP"

NCHR[-] => "LDA ="*1:N"; BRM TCH"

NSR[-] => "BRM TST;" STRING[*1];

UP["1",-] => "LDA* KSP" *2
                     [-,-] => "LDX KSP; LDA ="*1:S",*2" *2;

LKT[-] => "LDA KT" *1:

UPWRK / => "MIN WRK; LDA WRK; SKG XWRK; LDA XWRK; STA XWRK"
DOWNWRK / => "LDA =-1; ADM WRK"
OUTWRK[-] => "LDA WRK; BRM OUTN"
MAXWRK / => "LDA XWRK; BRM OUTN"
SIZS[CHR,-] => *1:C"STKSZ EQU "*2:S"

KPOPK[-] => "MIN MSP; LDA KT; STA* MSP; MIN MSP; LDA KSP; STA* MSP"*1
       "LDX MSP; LDA 0,2; STA KSP; LDA =-1,2; STA KT; LDA =-2; ADM MSP"

PAROUTZRO[]] => "LDA KT"
                ["0"] => "LDA KT"
                [-] => "LDA ="*1:

* END
*POPS, SUBROUTINES FOR TREE META*

**GEN**

POPD 10100000B,1,1  // GENERATE LABEL

LDA =47B
CLO FNKNO
MIN CHNO
LDA* 0
SKE =0
BRU ++4
MIN GN
LDA GN
STA* 0
BRM OUTN
BRR 0

**BF**

POPD 10200000B,1,1  // BRANCH FALSE

LDB =77777777B
SKB MFLAG
BRR 0
BRU* 0

**BT**

POPD 10300000B,1,1  // BRANCH TRUE

LDB =77777777B
SKB MFLAG
BRU* 0
BRR 0

**PSHN**

POPD 10400000B,1,1  // PUSH THE N STACK

LDB =77777777B
SKE* 0
LDA* 0
MIN NSP
STA* NSP
OVN LDA NSP
SK6 NSPT
BRR 0
LDA =12
BRM SERR

**PSHK**

POPD 10500000B,1,1  // PUSH THE K STACK

LDB =77777777B
SKE* 0
LDA* 0
MIN KSP
XMA KT
STA* KSP
OVK LDA KSP
SK6 KSPT
BRR 0
LDA =13
BRM SERR

**MKND**

POPD 10600000B,1,1  // MAKE A NODE

LDA* 0
STA  MKND1
BRU  MK1

MK2
BRM  POPK
MIN  NSP
STA*  NSP

MK1
SKR  MKND1
BRU  MK2
LDA  MARK
MRG  PTRFLG
MIN  KSP
XMA  KT
STA*  KSP
LDA*  0
MIN  MARK
XMA*  MARK
STA  MARK
BRU  OVN

MKND1
ESS  1

* EMLBL
POPD  10700000B,1,1 NODE LABEL
LDA*  0
MIN  NSP
STA*  NSP
LDA  NSP
XMA  MARK
MIN  NSP
STA*  NSP
BRU  OVN

* GET
POPD  11000000B,1,1 GET A NODE
CAX
ADD  1,2
SUB*  0
CAX
LDA  2,2
BRR  0

* BPTR
POPD  11100000B,1,1 BRANCH IF (A) A POINTER
LDE  FLGMSK
SKM  PTRFLG
BRR  0
BRU*  0

* BNPTR
POPD  11200000B,1,1 BRANCH IF NO POINTER
LDE  FLGMSK
SKM  PTRFLG
BRU*  0
BRR  0

* R11
POPD  11300000B,1,1 REC* ITEM 1
LDA  0,2
LDE  FLGMSK
SKM  PTRFLG
BRU  RIF2
```plaintext
STX  RINDX
LDX  0,2
LDA* 0
SKE 0,2
BRU  RIF1
LDA  CNT
MIN  MSP
STA* MSP
MIN  MSP
LDA  NCNT
STA* MSP
MIN  MSP
LDA  RINDX
STA* MSP
LDA  MSP
SKG  MSPT
BRU  *+3
LDA  =11
BRM  SERR
CXA
BRM  SETA
CLA
STA  CNT
MIN  0
BRR  0

RIF1 LDX  RINDX
RIF2 CLA
STA  MFLAG
BRR  0
RINDX BSS  1
RICNT BSS  1

* R12  POPD  11400000B,2  REC. ITEM 2
LDA  =-1
LDX* MSP
ADM  MSP
LDB* MSP
STB  NCNT
ADM  MSP
LDB* MSP
STB  CNT
ADM  MSP
BRR  0

* FLGT  POPD  115E5,1,1  FLAG TEST
LDA  0,2
LDB  FLGMSK
SKM* 0
BRU  FLGTF
BRR  0
FLGTF CLA
STA  MFLAG
ERR  0

*
BE POPD 116B5,1,1
LDB =77777777B
SKE MFLAG
ERR 0
LDA* 0
SKE =-1
BRU **+2
CLA
BRM ERR
LDA* 0
SKE 0
SKG =0
BRS EXIT
BRU* 0

* LAB POPD 117B5,1,1
LDA* 0
SKE =0
ERR 0
MIN GN
LDA GN
STA* 0
BRR 0

* CE POPD 120B5,1,1
LDB =77777777B
SKE MFLAG
ERR 0
LDA* 0
BRM CERR

* LDKA POPD 121B5,1,1
LDA KSP
SUB* 0
CAF X
LDA 1,2
ERR 0

* SUES
*
$ POPK ZRO
LDB* KSP
LDA =-1
ADM KSP
CBA
XMA KT
BRR POPK

* $ SETA ZRO 0 SET X TO TOP OF NODE GROUP, COUNT IN NCNT
CAF X
LDE 1,2
ADD 1,2
CAF X
STE NCNT
EAX 1,2
BRR SETA

* $CLLS ZRO
MIN MSP
STA* MSP
LDA MSP
SKG MSPI
BRR CLLS
LDA = 11
BRM SERR

* $RTNS NOP
LDA = -1
LDB* MSP
ADM MSP
STB *+2
BRR *+1
BSS 1

* $SAV ZRO
LDA NCCP
MIN MSP
STA* MSP
LDA NSP
MIN MSP
STA* MSP
LDA KSP
MIN MSP
STA* MSP
LDA KT
MIN MSP
STA* MSP
LDA MSP
SKG MSPI
BRR SAV
LDA = 11
BRM SERR

* $RSTR ZRO
BT RSTT
LDA = -1
LDB* MSP
ADM MSP
STB KT
LDB* MSP
STB KSP
ADM KSP
LDB* MSP
STB NSP
ADM KSP
LDB* MSP
STB NCCP
ADM  MSP
BRR  RSTR

RSTT LDA  =-4
ADM  MSP
BRR  RSTR

* $OUTREE ZRO
   LDA  KT
   ENPTR OUTERR
   LDX+ KT
   BRM  0,2
   BRM  POPK
   LDA  =NSTK
   STA  NSP
   BRR  OUTREE

OUTERR LDA  =2
BRM  CERR

* $RESET ZRO
   LDA  =MSTK
   STA  MSP
   LDA  =KSTK
   STA  KSP
   LDA  =NSTK
   STA  NSP
   CLA
   STA  KT
   BRR  RESET

* $SAVGN ZRO
   LDA  GNLB1
   MIN  MSP
   STA* MSP
   LDA  GNLB2
   MIN  MSP
   STA* MSP
   CLA
   STA  GNLB1
   STA  GNLB2
   LDA  MSP
   SKG  MSPT
   BRR  SAVGN
   LDA  =11
   BRM  SERR

* $RSTGN ZRO
   LDA  =-1
   LDB* MSP
   STB  GNLB2
   ADM  MSP
   LDB* MSP
   STB  GNLB1
   ADM  MSP
   BRR  RSTGN
* $SSTEST ZRO
  LDA* SSTEST
  STA SSTCINT
  BRM MOD3
  LDB SSTEST
  ADM SSTEST
  STA SSTWDS
  STP SSTPTR
  MIN SSTPTR
  LDA 0,2
  BPT1 SST1+1
  LDA* 0,2
  SKE SSTCINT
  BRU SST1+1
  STX INDX
  LDA 0,2
  ADD =1
  LDB SSTPTR
  LDX SSTWDS
  BRM SKSE
  BRU SST1
  LDA CNT
  LDX INDX
  BRR SSTTEST
  SST1
  LDX INDX
  CLA
  STA MFLAG
  BRR SSTTEST
  SSTPTR BSS 1
  SSTCINT BSS 1
  SSTWDS BSS 1
* $BEGN ZRO
  LDA =1
  STA MFLAG
  LDA KT
  BRM SETA
  CLA
  STA CNT
  BRR BEGN
* $CLOZRO
  BRM CLLS
  BRM SAVGN
  BRR CLLO
* $RTNO NOP
  BRM RSTCN
  BRU RTNS
  NOP
* CELLS
*  
$ME   BSS   1  
$INDX  BSS  1  
$CNT   BSS  1  
$NCNT  BSS  1  
$SNSP  BSS  1  
$KI    BSS  1  
$SRFLG DATA  10B5  
$CHRFLG DATA  12B5  
$IDFLG DATA  4B5  
$NUMFLG DATA  6B5  
$UPRF LG DATA  2B5  
$FLGMSK DATA  776B5  
$GENFLG DATA  16B5  
$MARK  BSS  1  
$AN   DATA  0  
$ENLBI DATA  0  
$ENLBI2 DATA  0  
$SAVKT BSS  1  
$SAVKP BSS  1  
$LFTFLG DATA  14B5  
END
* ARPA's library for 940 meta II and tree systems.
* Parameters for size of K, M, N stacks, and SS area.

GOEL ZERO
    LDA MCCP
    ADD BACK
    SUB = 1
    STA NCCP
    BRR GOEL

* STORE ZERO
    LDA = SS
    STA SSP
    LDA LEN
    SKE = 0
    BRU *+3
    LDA = 8
    BRM SERR
    LDA = STR
    LDB = TEST
    LDX LEN
    BRM PACK

S1
    LDA SSL
    SKG SSP
    BRU SPUT
    LDA SSP
    STA SX
    LDA* SSP
    BRM MOD3
    MIN SSP
    AIM SSP
    LDA* SX
    SKE LEN
    BRU S1
    BRM MOD3
    CAX
    LDB = TEST
    LDA SX
    ADD = 1
    BRM SKSE
    BRU S1

SST
    LDA SX
    BRR STORE

* SPUT
    LDA SSL
    STA SX
    LDA LEN
    STA* SSL
    MIN SSL
    LDA = STR
    LDB SSL
    LDX LEN
    BRM PACK
    LDA LFN
    BRM MOD3
ADM SSL
LDA SSL
SKG SSTO P
BRU SST
LDA =6
BRU SERR

* 
SSP DATA SS
$SSL DATA SS
SX BSS 1
$MXSTR EQU 80
STPTR BSS 1
STR BSS MXSTR
STEST BSS MXSTR
$LISTFG DATA 1
$RLine ZRO
MIN LINCNT
LDA EOFLG
SKE =0
BRU REOF
LDA =12B
SKN LISTFG
CIO FNUM0
LDX BUFNO
BRU R1+1
R1 BRX R3
CIO FNUMI
SKN LISTFG
BRU R4
R15 STA IBUF 2
SKE =155B
BRU R2
LDA =152B
SKN LISTFG
CIO FNUM0
BRU FILL2
R2 SKE =137B
BRU R1
LDA =1
STA EOFLG
FILL CLA
STA IBUF 2
FILL2 BRX R3
BRU FILL
R3 LDA BUFNO
STA IBP
BRR RLINE
REOF BRM CHLFT
BRM LITT
DATA 18
ASC 'END OF FILE INPUT'
BRM CHLFT
BR4 EXIT
R4 SKE =152B
CIO FNU0
BRU R15
EOFLG DATA 0
$INC ZRO
BMI UPI WP
SKN BACK
BRU **3
MIN BACK
BRR INC
MIN MCCP
BMI PUTIN
BRR INC

* PUTIN ZRO
BMI PCHK
LDX IBP
MIN IBP
LDA IBUF, 2
SKE = 155B
BRU P2
BMI RLINE

P1 CLA
P11 STA* IWP
BRR PUTIN
P2 SKE = 135B
BRU P3
BMI PCHK
MIN IBF
BRU P1
P3 SKG = 63
BRU P11
BRU PUTIN+1

* PCHK ZRO
LDA MXIB
SKG IBP
BMI RLINE
BRR PCHK

* CHER ZRO
LDX* IWP
LDA = 64
SKG* IWP
BRU SERR
LDX CLASS, 2
CXA
SKG = 5
SKG = 0
BRU **2
BRR CHER
LDA = 1
BRU SERR

$WPREP ZRO
CLA
STA LEN
LDA = STR
STA STPTR
LDA NCCP
SUB MCCP
STA BACK
SKG = 0
SKG MRSIZ
BRU WPER
LDA NCCP
ETR MODRSZ
ADD = RING
STA IWP
BRR WPREP
WPER LDA = 2
BRR SERR
*
$INCS ZRO
INCS2 BRM INC
LDA* IWP
SKE = 0
BRU *+2
BRU INCS3
SKE CMNT
BRR INCS
LDA = 9
SKN BACK
BRU *+2
BRR SERR
BRRM PUTIN
LDA* IWP
SKE CMNT
BRU *-3
INCS3 BRM PUTIN
BRR INCS2+1
*
$ID ZRO
CLA
STA MFLAG
BRM WPREP
BRM INC
BRR CHER
BRU IDT1, 2
ID1 BRM CIC
BRM INC
BRM CHER
BRU IDT2, 2
IDF LDA = 1
STA MFLAG
BRM GOBL
BRM STORE
BRR ID
IDT1 BRU STER
BRR ID
BRU    ID1
BRR    ID
BRN    ID
BRR    ID
IDT2  BRU    STER
BRU    IDF
BRU    ID1
BRU    ID1
BRU    IDF
BRU    IDF

*  CIC    ZRO
   LDA*    IWP
   STA*    STPTR
   MIN    STPTR
   MIN    LEN
   BRR    CIC

*  LEN    BSS    1

*  UPIWP    ZRO
   MIN    IWP
   LDA    IWP
   SKG    MXI
   BRU    UPIWP
   LDA    RING
   STA    IWP
   BRR    UPIWP

$OUTS    ZRO
   STA    OUTP
   LDA    =-2
   STA    OUTS
   LDA    TELNO
   STA    LITF
   BRU    OUTSA

$OUTS    ZRO
   STA    OUTP
   LDA    FNUMO
   STA    LITF

OUTSA  LDA*    OUTP
   SKG    RSIZ
   BRU    ++3
   LDA    =5
   BRU    SERR
   ADM    CHNO
   CAB
   MIN    OUTP
   LDA    OUTP
   ETR    =77777B
   LDX    LITF
   BRS    34
   BRR    OUTS

*  OUTP    BSS    1
*  
$OUTH ZRO  
SKG = -1  
BRU OUTNN  
OUTH STA OUTNB  
LDB = 10  
LDX FNUMO  
BRS 36  
LDA = 1  
SKG OUTNB  
BRU **+2  
BRN OUTN  
MIN CHNO  
MUL = 10  
RSH 1  
CBA  
BRU **-7  
OUTH STA OUTNB  
LDA = 15B  
CIO FNUMO  
LDA OUTNB  
BRU OUTNP+1  
OUTNB BSS 1  
*  
$WRSS NOP  
LDA = SZ  
STA WRSSPT  
WR S1 BRM CRLFT  
LDA* WRSSPT  
STA WRSS1  
BRM WOUT  
MIN WRSSPT  
LDA WRSS1  
BRM MOD3  
LDB WRSSPT  
ADM WRSSPT  
LDA WRSS1  
XAB  
LDX TELNO  
BRS 34  
LDA WRSSPT  
SKG SSL  
BRU WR S1  
BRM CRLFT  
BRS EXIT  
WRSS1 BSS 1  
WRSSPT BSS 1  
*  
$CRLFT ZRO  
LDA = 155B  
CIO TELNO  
LDA = 152B
CIO TELNO
BRR CRLF T

* $CRLF ZRO
  LDA =155B
  CIO FNUMO
  LDA =152B
  CIO FNUMO
  LDA =1
  STA CHNO
  BRR CRLF

* $LITT ZRO
  LDA *- 1
  STA LIT
  LDA TELNO
  STA LITF
  MIN LIT
  LDA* LIT
  BRU LITW+3

* $LIT ZRO
  LDA FNUMO
  STA LITF
  LITW MIN LIT
  LDA* LIT
  ADM CHNO
  STA LIT1
  CAB MIN LIT
  LDA LIT
  ETR =77777B
  LDX LITF
  BRS 34
  LDA LIT1
  BRM MOD3
  SUB =1
  ADM LIT
  BRR LIT

LITF BSS 1
LIT1 BSS 1
$TABT ZRO
  LDA *- 1
  STA TAB
  LDA TELNO
  STA LITF
  BRU TABA

$TAB ZRO
  LDA FNUMO
  STA LITF

TABA LDA CHNO
  ADD =10B
  ETR =7770B
  STA TAB3
TAB2 MIN CHNO
CLA
CIO LITF
LDA TAB3
SKE CHNO
BRU TAB2
BRR TAB
TAB3 BSS 1
$WRIW NOP
LDA =RING
STA WRI1
NLIN BRM CRLFT
LDA BUFNO
ADD =10
CAX
WRCK LDA WRI1
SUB MXIW
SKG =0
BRU ++2
BRX EXIT
LDA* WRI1
CIO TELNO
MIN WRI1
BRX NLIN
BRU WRCK
WRI1 BSS 1
$INITL ZRO
AGAIN BRM CRLFT
BRM LITT
DATA 7
ASC 'INPUT: '
CLEAR
BRX 15
BRU AGAIN
STA FNUMI
CBA
SKE =16B
BRU ++2
BRU AGAIN2
LDA FNUMI
BRX 20
BRU AGAIN
AGAIN2 BRM CRLFT
BRM LITT
DATA 8
ASC 'OUTPUT: '
CLEAR
LDA =03000000B
BRX 16
BRU AGAIN2
STA FNUMO
STA XFNUMO
CBA
SKE =16B
BRU ++2
BRU ++4
LDA FNUMO
BRS 20
BRU AGAIN2
BRR CLRFT
BRR INITL
$FNUMO BSS 1
$FNUMI BSS 1
$XNUMO BSS 1
$TELNO DATA 1
$CHNO DATA 1
$TECHNO DATA 1
* A=UNPACKED POINTER, B=PACKED, X=LENGTH
PACK ZRO
STA UPP
STB PP
STX PLEN
PK1
BRM SKOK
BRR PACK
LDA* UPP
MIN UPP
STA PX
BRM SKOK
BRU PKR1
LDB* UPP
MIN UPP
LH 16
LDA PX
LSH 8
STA PX
BRM SKOK
BRU PKR2
LDB* UPP
MIN UPP
LH 16
LDA PX
LH 8
STA* PP
MIN PP
BRU PK1

* SKOK ZRO
SKR PLEN
MIN SKOK
BRR SKOK

* PKR1 LDA PX
CLB
LSH 16
STA* PP
BRR PACK
PKR2 LDA PX
CLB
LSH 8
STA* PP
BRR PACK

UPACK ZRO
STA UPP
STB PP
STX PLEN
SKR PLEN
BRU **2
BRR UPACK

PK2 LDA* PP
RSH 16
BRM PST
RSH 8
BRM PST
MIN PP
BRU PK2

PST ZRO
ETR = 377B
STA* UPP
MIN UPP
LDA* PP
SKR PLEN
BRR PST
BRR UPACK

PX BSS 1
UPP BSS 1
PP BSS 1
PLEN BSS 1

SKSE ZRO
STA PP
STB UPP
STX PLEN

SKS1 SKR PLEN
BRU **2
BRU SKST
LDA* PP
SKE* UPP
BRR SKSE
MIN UPP
MIN PP
BRU SKS1

SKST MIN SKSE
BRR SKSE

$MOD3 ZRO
SUB = 1
RSH 23
CLA
```
DIV = 3
ADD = 1
BRR MOD3
IBUF BES 80
BUFNO DATA 37660B
$IWP DATA RING-1
IBP DATA 37660B
MXIB DATA 40000B
MXIW DATA RING+255
BACK BSS 1
$NCCP DATA 0
MCCP DATA 0
RING BSS 256
$EXIT EQU 10
CLASS DATA 1, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 3, 3, 3, 3, 3
DATA 5, 5, 5, 5, 5, 5, 5, 5, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
DATA 2, 2, 2, 2, 2, 2, 5, 5, 5, 5, 5, 5, 5, 5, 0, 0, 0
$ERR ZERO STA ERRNO
BRM CRLFT
BRM LITT
DATA 13
ASC 'SYNTAX ERROR'
LDA =-1
XMA ERRNO
BRM WOUT
BRM LITT
DATA 5
ASC 'LINE'
LDX TELNO
LDB =10
LDA LINCNT
BRS 36
BRM CRLFT
LDX BUFNO
BRU ERRN+1
ERRC MIN ERRNO
ERRY CIO TELNO
ERRN BRX ERRR
CXA
SKE IBP
BRU **+3
LDA ERRNO
STA ERRX
LDA IBUF,2
SKE =155B
BRU ERR1
BRU ERRF
ERR1 SKE =152B
BRU ERR2
BRU ERRN
ERR2 SKE =135B
BRU ERRC
CIO TELNO
```
BRX  ERRF
LDA  IBUF, 2
AIDM  ERRNO
BRU  ERRY
ERRF  BRM  CRLFT
CLA
BRU  **2
CIO  TELNO
SKR  ERRX
BRU  **2
LDA  ARROW
CIO  TELNO
BRM  CRLFT
BRR  ERR

*  ERRNO  BSS  1
ERRX  BSS  1
ARROW DATA  76B
$SERR  NOP
STA  SE1
LDA  =SEM
LDB  =13
BRU  SERR1
$CERR  NOP
STA  SE1
LDA  =CEM
LDB  =15
SERR1  LDX  TELNO
BRS  34
LDA  SE1
LDB  =10
LDX  TELNO
BRS  36
SE1  BSS  1
*  RSIZ DATA  256
MRSIZ DATA  =256
MODRSZ DATA  377B
$MFLAG BSS  1
CMNT DATA  5
$LINCNT DATA  0
*  $WOUT ZRO
LDB  =10
LDX  TELNO
BRS  36
LDA  =14B
CIO  TELNO
CLA
CIO  TELNO
BRR   WOUT
*
$TST  ZRO
   CLA
   STA  MFLAG
   MIN  TST
   BRM  WPREP
   BRM  INCS
   LDA*  TST
   SKG  RSIZ
   BRU  *=3
   LDA  =4
   BRU  SERR
   STA  TST2
   BRM  MOD3
   LDB  TST
   ADM  TST
   CAB
   ADD  =1
   CAB
   LDA  =STEST
   LDX  TST2
   STA  TST1
   BRM  UPACK
   SKR  TST2
   BRU  TSTS1
   BRR  TST
   TSTS  BRM  INC
   MIN  TST1
   TSTS1 LDA*  TST1
    SKE*  IWP
    BRR  TST
    SKR  TST2
    BRU  TSTS
    LDA  MCCP
    ADD  BACK
    STA  NCCP
    LDA  =1
    STA  MFLAG
    BRR  TST
   TST1 BSS  1
   TST2 BSS  1
*
*$SR  ZRO
   CLA
   STA  MFLAG
   BRM  WPREP
   BRM  INCS
   BRM  CHER
   BRU  STT1,2
   STR1  BRM  CIC —
   BRM  INC
   BRM  CHER
BRU STT2,2
STT2 BRM GOEL
MIN NCCP
LDA =1
STA MFLAG
BRM STORE
BRR SR
*
STT1 BRU STER
BRR SR
BRR SR
BRR SR
BRR SR
BRU STR1+1 DON'T COPY QUOTE
BRR SR
STT2 BRU STER
BRU STR1
BRU STR1
BRU STR1
BRU STR2
BRU STR1
*
STER LDA =7
BRU SERR
*
$NUM ZRO
CLA
STA MFLAG
BRM WPREP
BRM INCS
BRM CHER
BRU NT1,2
NM1 BRM CIC
BRM INC
BRM CHER
BRU NT2,2
NMF LDA =1
STA MFLAG
BRM GOEL
BRM STORE
BRR NUM
*
NT1 BRU STER
BRR NUM
BRR NUM
BRU NM1
BRR NUM
BRR NUM
NT2 BRU STER
BRU NMF
BRU NMF
BRU NM1
BRU NMF
BRU NMF
SLET ZRO
CLA
STA MFLAG
BRM WPREP
BRM INCS
·BRM CHER
BRU LET1,2
LET2 BRM CIC
LDA = 1
STA MFLAG
BRM GOBL
MIN NCCP
BRM STORE
BRR LET
LET1 BRU STER
BRR LET
BRU LET2
BRR LET
BRR LET
BRR LET
*
*
* $FINISH NOP
LDA = 137B
CIO FNUMO
CIO FNUMO
CIO FNUMO
CIO FNUMO
LDA FNUMO
BRS 20
BRU LIMITS
*
* $TCH ZRO
STA TCH1
CLA
STA MFLAG
BRM WPREP
BRM INCS
LDA* IWP
SKE TCH1
BRR TCH
MIN MFLAG
LDA MCCP
ADD BACK
STA NCCP
BRR TCH
TCH1 BSS 1
*
* TOP MACRO D
LDA D(1) * SPT
STA D(1) * SP
LDA = -1
ALM D(1) * SP
LDA* D(1)* SP
SKE = 0
BRU *= 2
BRU *= 5
LDA D(1)* SP
SUB = D(1)* STK
SKG = 0
CLA
BRM WOUT
ENDM

* $LIMITS BRM CRLFT
BRM LITT
DATA 5
ASC 'USED '
TOP K
TOP M
TOP N
LDA SSL
SUB = SS
BRM WOUT
BRM CRLFT
BRS EXIT
END