# A TREE META FOR THE XDS 940

by

J. F. Rulifson

April 1968

Augmentation Research Center

Stanford Research Institute

Menlo Park, California

This material was contained as Appendix D in the final Report for Rome Air Development Center on Contract AF 30(602)-4103

# APPENDIX D -- TREE META: Introduction

1 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 metalangauge is also a language.

1B A compiler is a computer program that reads a formal-language program as input and translates that program into instructions that may be executed by a computer. The term "compiler" also means a listing of the instructions of the compiler.

1C A language that can be used to describe other languages is a metalanguage. English is an informal, general metalanguage that can describe any formal language. Backus-Naur Form or BNF (Nur1) 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 that 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 that ease the compiler-building process.

2B1 Tree Meta is intended to parse context-free laguages 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 that describe a method of generating all the possible strings of the language.

3B The reductive or analytic technique states a set of rules that 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=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 := if T then (if issymbol('+') then E else true) else false; comment issymbol (arg) is a Boolean procedure that compares the next symbol in the input string with its argument, arg. If there is a match the input stream is advanced;

4A2B boolean procedure T; T := if F then (if issymbol('*') then T else true) else false;

4A2C boolean procedure F; F := if issymbol('X') then true else if issymbol('(') then (if E then (if issymbol(')') then true else false) else false) else false;

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.

4C 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.

5 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 Angles area primarily through the effort of Howard Metcalfe (Schmidt1).

5A In the fall of 1962, he designed two compiler-writing interpreters (Metcalf1). 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 Glennie (Glennie1) 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 (Rutman1). This compiler used an algorithm that 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 (Schneider1). 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 tree-searching and backup capability (Book1) (Book2).

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).

12 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.

12A The Tree Meta compiler converts each of the rules to a set of instructions for the computer.

12B 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.

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

```
R1A     EXP = TERM ("+" EXP / "-" EXP / .EMPTY);

R2A     TERM = FACTOR $("*" FACTOR / "/" FACTOR);

R3A     FACTOR = "-" FACTOR / PRIM;

R4A     PRIM = .ID / .NUM / "(" EXP ")";
```

13A 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.

13B 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".

13B1 A string of characters between quotation marks (") represents a literal string. These literal strings are tested against the input stream as characters are read.

13B2 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.

13B3 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.

13B4 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

13B5 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 tha is treated as a recognizer but does not look for anything. It is .EMPTY and it always returns a value of "true."

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

14A EXP first calls rule TERM, that calls FACTOR, that tests for a minus sign. This test fails and FACTOR then tests for a plus sign and fails again. Finally FACTOR calls PRIM, that tests for an identifier. The character X is an identifier; it is recognized and the input stream advances one character.

14B 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.

14C 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.

14D 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.

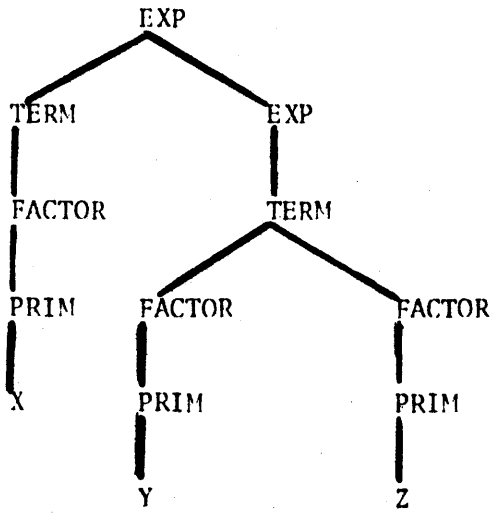
15 Tree Meta is different from most metacompiler systems in that it builds a parse tree of the input stream before producing any output. Before we describe the syntax of node generation, let us first discuss parse trees.

15A A parse tree is a structural description of the input stream in terms of the given grammar.

15A1 Using the four rules above, the input stream

X+Y*Z

has the following parse tree

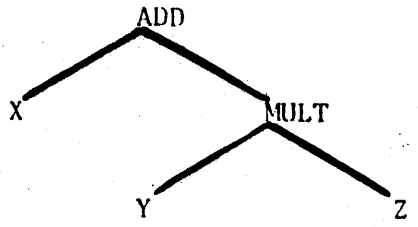



15A2 In this tree each node is either the name of a rule or one of the primary entities recognized by the basic recognizer routines.

15A3 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.

15B The tree produced by the metacompiler program is simpler than the one above, yet it contains sufficient information to complete the compilation. .

15B1 The parse tree actually produced is




15B2 In this tree the names of the nodes are not the rule names of the syntactic definitions, but rather the names of rules that will be used to generate the code from the tree.

15B3 The rules that 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     EXP = TERM ("+" EXP :ADD/ "-" EXP :SUB) [2] .EMPTY);

R2B     TERM = FACTOR $(("*" FACTOR :MULT/ "/" FACTOR :DIVD)
        [2]);

R3B     FACTOR = "-" FACTOR :MINUS[1] / PRIM;

R4B     PRIM = .ID / .NUM / "(" EXP ")";
```

15C 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:

```
X+Y*Z
```

15C1 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.

15C2 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.

15C3 PRIM returns to FACTOR, which returns to TERM. The asterisk is recognized and the input is advanced another character.

15C4 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.

15C5 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.

15C5A The stack now contains

MULT

X

15C5B The parse tree is now


15C5C Notice that the nodes are assembled in a left-to-right order, and that the original order of recognition is retained.

15C6 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 that 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.

16 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 alows the output to be based on the deep structure of the input, and hence better output may be produced.

16A 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.

16A1 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.

16A2 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.

16A3 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:

```
"L" , "ADD" , "SYS"
```

16A4 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.

16A4A Suppose that the expression X+Y*Z has been parsed and the program is in the ADD unparse rule processing the ADD node (later we will see how this state is reached). To put the identifier X into the output stream, one writes "*1" (meaning "the first node below") as an element. For example, to generate a line of code with the operation code ADA and the operand field X, one would write:

```
, "ADA", *1
```

16A4B To generate the code for the left-hand node of the tree one merely mentions "*1" as an element of the unparse rule. Caution must be taken to ensure that no attempt is made to append a nonterminal node to the output stream; each node must be tested to be sure that it is the right type before it can be evaluated or output.

16A5 Generated labels are handled automatically. As each unparse

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.

17 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.

17A An unparse 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 unparse rule is called, the test for the first output rule is made. If it is satisfied, the remainder of the alternative is executed; if it is false, the next alternative output rule test is made. This process continues until either a successful test is made or all the alternatives have been tried. If a test is successful, the alternative is executed and a return is made from the unparse rule with the general flag set "true." If no test is successful, a return is made with the general flag "false."

17B The simplest test that can be made is the test to ensure that the correct number of nodes emanate from the node being processed. The ADD rule may begin

```
ADD[-,-] =>
```

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

17B1 If one wishes to eliminate the test at the head of the out-rule, one may write a slash instead of the bracketed string of items. The slash, then, takes the place of the test and is always true. Thus, a rule which begins with a slash immediately after the rule name may have only one out-rule. The rule

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

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

17B2 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.

17B3 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,-].

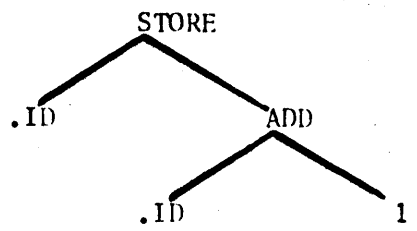
17B4 To check for a literal string on a node one may write a string as an item in an out-test. The construct [-,"1"] tests to be sure that there are two nodes and that the second node is a 1. The second node will have been recognized by the .NUM basic recognizer during the parse phase.

17B5 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.

17B6 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:

```
STORE[-,ADD[*1,"1"]] => , "MIN " *1
```

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




would satisfy the test, where the two identifiers must be the same or the test fails. An expression such as X ← X + 1 meets all the requirements. The code generated (for the SDS 940) would be the single instruction MIN X, which increments the cell X by one.

17C 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.

17C1 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.

17C2 The test in an out-expr resembles the test for the out-rule. There are two types of these tests.

17C2A 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 proceedes 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.

17C2B The other type of test is made by invoking another unparse rule by name and testing the flag on the completion of the rule. To call another unparse rule from an out-expr, one writes the name of the rule followed by an argument list enclosed in brackets. The argument list is a list of nodes in

the tree. These nodes are put on the node stack, and when the call is made the rule being called sees the argument list as its set of nodes to analyze. For example:

```
ADD[MINUS[-],-] => SUB[*2,*1:*1]
```

17C2B1 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 MULT were being evaluated,↑1*1 would be the X.

17C2B2 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.

17C3 The calls on other unparse 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.

17C4 The rest of an out-expr is made up of output elements appended to the output stream, as discussed above.

17D 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.

17E Out-exprs may be nested, using parentheses, in the same way as the alternatives of the parse rules.

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

18A 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.

18B 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:

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

     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.

18C 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.

18D In addition to the three basic recognizers .ID, .NUM, and .SR, there are two others which are occasionally very useful.

18D1 The symbol .LET indicates a single letter. It could be thought of as a one-character identifier.

18D2 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.

18E 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.

18E1 The syntax for outputting from parse rules is generally the same as for unparse rules. The output expression is written within square brackets, however. The items from the input stream that 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.

18E2 Normally the items are removed from the stack and put into the tree. However, if they are 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.

19 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.

19A The environment of the Tree Meta program, as it is running, is the library of routines plus the various data areas.

19B This section describes the environment in its three logical parts: input, stack organization, and output.

19B1 This is a description of the current working version, with some indications of planned improvements.

20 Input Machinery

20A 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:

20A1 Syntax-error reporting

20A2 A possible anchor mode (so the compiler can sense the end of a line)

20A3 An interlinear listing option.

20A4 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.

20B 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 that indicates which character is to be read from the line buffer next, and RLINE resets that pointer to the first character of the line.

20C 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.

20D There are several routines that deal directly with the input window.

20D1 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.

20D2 The routine INC is used to put a character into the input window. It increases IWP by one by calling a routine, UPIWP, 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.

20D3 A routine called INCS is similar to INC except that it deletes all blanks or comments that 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.

20D3A 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.

20D4 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.

20D5 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.

20D6 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 (TCH). It inputs one character (deleting blanks), compares it to the given character and returns false, or adjusts PCC and returns true.

## 21 Stacks and Internal Organization

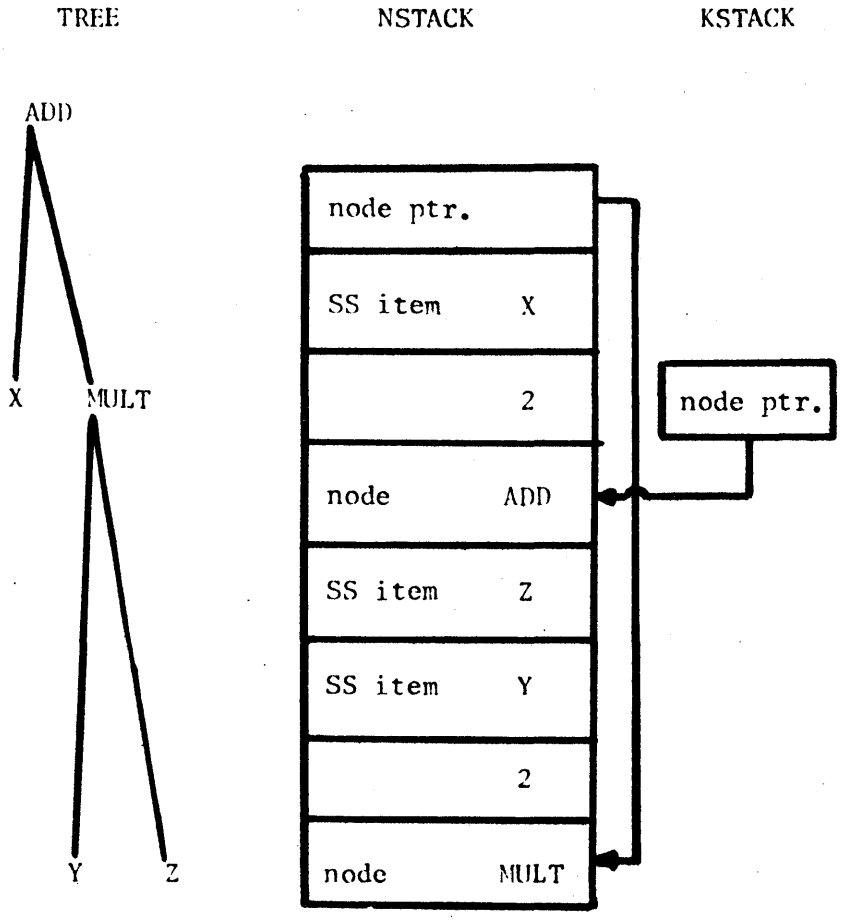
21A Three stacks are available to the program. A stack called MSTACK is used to hold return locations and generated labels for the program's recursive routines. Another stack, called KSTACK, contains references to input items. When a basic recognizer is executed, the reference to that input item is pushed 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.

21A1 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.

21A2 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.

21A2A 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.

21A2B 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,




21A2C KSTACK contains terminal nodes (input items) and nonterminal node pointers that point to nodes already in NSTACK. NSTACK contains nonterminal nodes.

21B 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.

21C 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.

21C1 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:

21C1A (1) Class codes are illegal

21C1B (2) Backup too far

21C1C (64) Character with code greater than 63 in ring buffer

21C1D (4) Test for string longer than ring size

21C1E (5) Trying to output a string longer than maximum string length

21C1F (6) String-storage overflow

21C1G (7) Illegal character code

21C1H (8) Trying to store SS element of length zero

21C1I (11) MSTACK overflow

21C1J (12) NSTACK overflow

21C1K (13) KSTACK overflow

21D There is a set of routines used by Tree Meta that 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.

22 Output Facilities

22A 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.

22A1 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.

22A2 A literal string of characters is written by calling the routine LIT The literal string follows the call as for TST.

22A3 A number is written using routine OUTS. The binary representation is given, and is written as a signed decimal integer.

22A4 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.

22A5 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.

23 This chapter is a formal description of the complete Tree Meta language. It is designed as a reference guide.

23A For clarity, strings that would normally be delimited by quotation marks in the metalanguage are capitalized instead, in this chapter only.

23B Certain characters cannot be printed on the report-generating output media but are on the teletypes and in the metalanguage--their names, preceeded by periods, are used instead. They are .exclamation, .question, .pound, .ampersand, .backslash, and .percent.

24 Programs and Rules

24A Syntax

24A1 program = .META .id (.LIST / .empty) size / .CONTINUE $rule .END;

24A2 size = '( siz $(', siz) ') / .empty;

24A3 siz = .chr '= .num;

24A4 rule = .id ('= exp (.ampersand / .empty) / '/ "=>" gen1 / outrul) '; ;

24B Semantics

24B1 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.

24B1A The mandatory identifier following the string .META in a main file names the rule at which the parse will begin.

24B1B The optional .LIST, if present, will cause the compiler currently being generated to list input when it is compiling a program.

24B1C 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 that must be determined by experience. The maximum number of cells used during each compilation is printed out at the end of the compilation.

24B2 When a file begins with .CONTINUE, no initialization or

storage-allocation code is produced.

24B3 There are three different kinds of rules in a Tree Meta program. All three begin with the identifier that names the rule.

24B3A Parse rules are distinguished by the = following the identifier. If all the elements that 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.

24B3B Rules with the string / => following the identifier may be composed only of elements that produce output. There is no testing of flags within a rule of this type.

24B3C Unparse rules have a left bracket following the identifier. This signals the start of a series of node tests.

25 Expressions

25A Syntax

25A1 exp = '←suback ('/ exp / .empty) / subexp ('/ exp / .empty);

25A2 suback = ntest (suback / .empty) / stest (suback / .empty);

25A3 subexp = (ntest / stest) (noback / .empty);

25A4 noback = (ntest / stest ('.question .num (.id / '.question ) / .empty) ) (noback / .empty);

25B Semantics

25B1 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 concatention of the test, are necessary to reduce the instructions executed when there is no backup of the input stream.

25B2 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 that invoked it.

25B3 The subexpressions are series of tests. Only subexpressions that begin with a leftarrow are allowed to back up the input stream and rescan it.

25B3A 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 stest fails during compilation, the system prints an error comment and stops. If the error code is present and the stest 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.

25B3B 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.

## 26 Elements of Parse Rules

### 26A Syntax

```
26A1  ntest = (':.id / '[ ( .num '] / genp '] ('.backslash /
.empty ) / '< genp '> ('.backslash / .empty) / (.CHR / '*) / "=>"
/ comm;

26A2  genp = genp1 / .empty;

26A3  genp1 = genp2 (genp1 / .empty);

26A4  genp2 = '* (S .num / .empty) (L / C / N / .empty) / genu;

26A5  comm = .EMPTY / '.exclamation .sr;

26A6  stest = '. .id / .id / .sr / '( exp ') / ''.chr / (.num '$ /
'$) (.num / .empty) stest / '- (.sr / ''.chr);
```

### 26B Semantics

26B1 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.

26B1A The : .id construct names the next node to be put into the tree. The identifier must be the name of another rule.

26B1B 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.

26B1C The [ genp ] is used to write output into the normal

output stream during the parse phase of the compilation.

26B1D 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.

26B1E 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.

26B1F An asterisk causes the rule currently in execution to perform a subroutine call to the rule named by the top of the tree.

26B1G 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.

26B2 The comm elements are common to both parse and unparse rules.

26B2A The .EMPTY in any rule sets the general flag true.

26B2B 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.

26B3 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.

26B3A 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.

26B3B An identifier by itself produces a call to the rule with the name of the identifier.

26B3C 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.

26B3D 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.

26B3E 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.

## 27 Unparse Rules

### 27A Syntax

27A1 outrul = '[ outr (outrul / .empty);

27A2 outr = items '] "=>" outexp;

27A3 items = item (', items / .empty);

27A4 item = '- / .id '[ outest / nsimpl / '. .id / .sr / ''.chr / '.pound;

### 27B Semantics

27B1 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.

27B2 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.

27B3 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.

27B4 If no alternation is successful a false value is returned.

27B5 Each item of an unparse alternation test may be one of five different kinds of test.

27B5A 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.

27B5B The name of the node may be tested by writing an identifier that is the name of a rule. The identifer must then be followed by a test on the subnodes.

27B5C 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.

27B5D 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.

27B5E 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.

27B5F 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.

27B5G Finally, the node may be tested to see if it is a generated label. The labels may be generated in the unparse expressions and then passed down to other unparse rules. The test is made writing a .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 unparse expression as the number following the .pound.

## 28 Unparse Expressions

### 28A Syntax

28A1 outexp = subout ('/ outexp / .empty);

28A2 subout = outt (rest / .empty) / rest;

28A3 rest = outt (rest / .empty) / gen (rest / .empty);

28A4 outt = .id '[ arglst '] / '( outexp ') / nsimp1 (': (S / L / N / C) / empty);

28A5 arglst = argmnt (', arglst / .empty) / .empty;

28A6 argmnt = nsimp / '.pound .num;

28A7 nsimp1 = '↑ nsimp / nsimp;

28A8 nsimp = '* .num ( ': nsimp / .empty);

28A9 genl = (out / comm) (genl / .empty);

28A10 gen = comm / genu / '< / '> ;

### 28B Semantics

28B1 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.

28B2 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.

28B3 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.

28B3A 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.

28B3B 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.

28B4 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.

28B5 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.

28B6 Just as in the parse rules, brackets may be used to send immediate printout to the user Teletype.

28B7 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 that exists in the tree.

28B7A If the node specified is the name of another rule, then control is transferred to that node by the standard subroutine linkage.

28B7B If the node is terminal, then the terminal string associated with the node is copied onto the output stream.

28B7C 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.

28B7D 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.

28B7D1 :s puts out the literal string

28B7D2 :l puts out the length of the string as a decimal number

28B7D3 :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.

28B7D4 :c puts out the character if the node was constructed with a .chr recognizer.

29 Output

29A Syntax

29A1 genu = out / '. .id '] ((.id / .num) / .empty) '] / '.pound .num (': / .empty);

29A2 out = ('.backslash / ', / .sr / ''.chr / "+w" / "-w" / ".w" / ".pound" ;

29B Semantics

29B1 The standard primitive output features include the following:

29B1A Write a carriage return with a backslash

29B1B Write a tab with a comma

29B1C Write a literal string by giving the literal string

29B1D Write a single character using the apostrophe-character construct

29B1E 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.

29B2 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.

29B3 .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.

30 Since the work on Tree Meta is still in progress, there are few conclusions and plentiful future plans.

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

31A Something that has never been done, and that 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 similiar statistics from other compilers. It may be possible to interest some people at Stanford in cooperating on this.

31B 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.

31C 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 that 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.

31D There are a series of additions, some very small and some major, that we intend to add to Tree Meta during the next year.

31D1 Other metacompiler systems have had a construct that 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.

31D2 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.

31D3 The error scheme for unparse 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.

31D4 Currently the unparse 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 unparse rules. Without changing the syntax of Tree Meta or the way the stacks work, we feel that we can reduce the size of the unparse rules by a factor of 4. This would free a considerably larger amount of core storage for stacks and enlarge the size of programs that 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.

31D5 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.

31E Finally, there is the following list of minor additions or changes to be made to the Tree Meta system.

31E1 Make the library output routines do block I/O rather than character I/O. This could cut compilation times by more that 70 percent.

31E2 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.

31E3 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.

APPENDIX D -- Tree Meta: Bibliography

1 (Book1) Erwin Book, "The LISP Version of the Meta Compiler," TECH MEMO TM-2710/330/00, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406, 2 November 1965.

2 (Book2) Erwin Book and D. V. Schorre, "A Simple Compiler Showing Features of Extended META," SP-2822, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406, 11 April 1967.

3 (Glennie1) A. E. Glennie, "On the Syntax Machine and the Construction of a Universal Computer," Technical Report Number 2, AD 240-512, Computation Center, Carnegie Institute of Technology, 1960.

4 (Kirkley1) Charles R. Kirkley and Johns F. Rulifson, "The LOT System of Syntax Directed Compiling," Stanford Research Institute Internal Report ISR 187531-139, 1966.

5 (Ledley1) Robert Ledley and J. B. Wilson, "Automatic Programming Language Translation Through Syntactical Analysis," Communications of the Association for Computing Machinery, Vol. 5, No. 3 pp. 145-155, March 1962.

6 (Metcalfe1) Howard Metcalfe, "A Parameterized Compiler Based on Mechanical Linguistics," Planning Research Corporation R-311, March 1, 1963, also in Annual Review in Automatic Programming, Vol. 4, 125-165.

7 (Naur1) Peter Naur et al., "Report on the Algorithmic Language ALGOL 60," Communications of the Association for Compting Machinery, Vol. 3, No. 5, pp.299-384, May 1960.

8 (Oppenheim1) D. Oppenheim and D. Haggerty, "META 5: A Tool to Manipulate Strings of Data," Proceedings of the 21st National Conference of the Association for Computing Machinery, 1966.

9 (Rutman1) Roger Rutman, "LOGIK. A Syntax Directed Compiler for Corputer Bit-Time Simulation," Master Thesis, UCLA, August 1964.

10 (Schmidt1) L. O. Schmidt, "The Status Bit," Special Interest Group on Programming Languages Working Group 1 News Letter, 1964.

11 (Schmidt2) PDP-1

12 (Schmidt3) EQGEN

13 (Schnieder1) F. W. Schneider and G. D. Johnson, "A Syntax-Directed Compiler-Writing Compiler to Generate Efficient Code," Proceedings of the 19th National Conference of the Association for Computing Machinery, 1964.

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

Proceedings of the 18th National Conference of the Association for Computing Machinery, Denver, Colorado, 1963.

15 (Schorre2) D. V. Schorre, "META II, A Syntax-Directed Compiler Writing Language," Proceedings of the 19th National Conference of the Association for Computing Machinery, 1964.

# APPENDIX D -- TREE META: Detailed Examples

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).

```
%TREE META SMALL ALGEBRAIC LANGUAGE - 29 SEPTEMBER 1967 %

.META PROGRAM .LIST

PROGRAM = ".PROGRAM" DEC * $( DEC *) :STARTN[0] ST * $('; ST *)
                ".FINISH" ?1E :ENDN[0] * FINISH ;

DEC = ".DECLARE" .ID $(', .ID :DO[2]) '; :DECN[1];


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

ST = IFST / WHILEST / FORST / GOST / IOST / BLOCK /
       .ID (': :LBL[1] ST :DO[2] / '← EXP :STORE[2]);

IFST = ".IF" EXP ".THEN" ST (".ELSE" ST :SIFTE[3] / .EMPTY :SIFT[2]);

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];

IOST = ".OPEN" ("INPUT" .ID '[ .ID '] :OPNINP[2] /
                "OUTPUT" .ID '[ .ID '] :OPNOUT[2]) /
       ".CLOSE" .ID :CLSFIL[1] /
       ".READ" .ID ': IDLIST :BRS38[2] /
       ".INPUT" .ID ': IDLIST :XCIO[2] /
       ".WRITE" .ID ': WLIST :OUTNUM[2] /
       ".OUTPUT" .ID ': WLIST :OUTCAR[2] ;
IDLIST = VAR (IDLIST :DO[2] / .EMPTY);

WLIST = (.ID / .NUM / .SR) (WLIST :DO[2] / .EMPTY);


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


EXP = ".IF" EXP ".THEN" EXP ".ELSE" EXP :AIF[3] / UNION;

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

INTERSECTION = NEG ('& INTERSECTION :AND[2] / .EMPTY);

NEG = "NOT " NEGNEG / RELATION;

NEGNEG = "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 :MULT/'/ 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 SIFTE1[#2,*3];

SIFTE1[#1,-] => ,"BRU",#2\ #1,"EQU *"\ *2 #2,"EQU *"\;

SIFT[-,-] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU *"\ *2 #2,"EQU *"\;

WHL[-,-] => #1,"EQU *"\ WHL1[*1,#2] *2 ,"BRU",#1\ #2,"EQU *"\;

WHL1[-,#2] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU *"\;

GO[-] => ,"BRU",*1\;

FOR[-,-,-,-,-] => <"DO NOT USE FOR STATEMENTS">;

LBL[-] => *1,"EQU *";

AIF[-,-,-] => LOPR[*1,#1,#2] BRF[*1,#2] #1,"EQU *"\ ACC[*2] AIF1[#2,*3];

AIF1[#1,-] => ,"BRU",#2\ #1,"EQU *"\ ACC[*2] #2,"EQU *"\;

LOPR[OR[-,-],#1,-] => LOPR[*1:*1,#1,#2] BRT[*1:*1,#1]
                                  #2,"EQU *"\ LOPR[*1:*2,#1,*3]
    [AND[-,-],-,#1] => LOPR[*1:*1,#2,#1] BRF[*1:*1,#1]
                                  #2,"EQU *"\ LOPR[*1:*2,*2,#1]
    [NOT[-],#1,#2]  => LOPR[*1:*1,#2,#1]
    [-,-,-] => .EMPTY;

BRT[OR[-,-],#1] => BRT[*1:*2,#1]
   [AND[-,-],#1] => BRT[*1:*2,#1]
   [NOT[-],#1]   => BRF[*1:*1,#1]
   [LE[-,-],#1]  => BLE[*1:*1,*1:*2,#1]
   [LT[-,-],#1]  => BLT[*1:*1,*1:*2,#1]
   [EQ[-,-],#1]  => BEQ[*1:*1,*1:*2,#1]
   [GE[-,-],#1]  => BGE[*1:*1,*1:*2,#1]
   [GT[-,-],#1]  => BLE[*1:*2,*1:*1,#1]
   [NE[-,-],#1]  => BNE[*1:*1,*1:*2,#1]
   [-,#1]        => ACC[*1] ,"SKE =0"\ ,"BRU",#1\;

BRF[OR[-,-],#1]  => BRF[*1:*2,#1]
   [AND[-,-],#1] => BRF[*1:*2,#1]
   [NOT[-],#1]   => BRT[*1:*1,#1]
```

```
   [LE[-,-],#1]   => BLE[*1:*2,*1:*1,#1]
   [LT[-,-],#1]   => BGE[*1:*1,*1:*2,#1]
   [EQ[-,-],#1]   => BNE[*1:*1,*1:*2,#1]
   [GE[-,-],#1]   => BLT[*1:*1,*1:*2,#1]
   [GT[-,-],#1]   => BLE[*1:*1,*1:*2,#1]
   [NE[-,-],#1]   => BEQ[*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\,"SKG","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\;

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

BGE[-,-,#1] => (TOKEN[*1] ACC[*2] ,"SKE",*1\,"SKG",*1\ /
                WORK[*1]  ACC[*2] ,"SKE","T+".W\,"SKG","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[*1]]          => "*ITS ALREADY THERE"\
     [-,ADD[VAR[*1],CON["1"]]] => ,"MIN",*1\
     [-,ADD[VAR[*1],-]]   => ACC[*2:*2] ,"ADM",*1\
     [-,SUB[VAR[*1],-]]   => ACC[*2:*2] ,"CNA; ADM "*1\
     [-,-]             => BREG[*2] ,"STB",*1\ /
                          ACC[*2] ,"STA",*1\;
ADD[MINUS[-],-] => SUB[*2,*1:*1]
   [-,-]        => TOKEN[*2] ACC[*1] ,"ADD",*2\ /
                   WORK[*1] ACC[*2] ,"ADD","T+".W-W\;

SUB[-,-] => TOKEN[*2] ACC[*1] ,"SUB",*2\ /
            TOKEN[*1] (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[*1] ,"CBA; CNA"\ /
            ACC[*1] ,"CNA"\;

DIVID[-,-] => TOKEN[*2] (BREG[*1] ,"CBA"\ /
                         ACC[*1]) ,"RSH 23; DIV "*2\ /
              WORK[*2] (BREG[*1] ,"CBA"\ /
                        ACC[*1]) ,"RSH 23; DIV T+".W-W\;
```

```
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"\;

ACC[-] => TOKEN[*1] ,"LDA",*1\ /
          BREG[*1] ,"CBA"\ /
          *1;

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

TOKEN[VAR[.ID]] => .EMPTY
     [CON[.NUM]] => .EMPTY;

MULT / => .EMPTY;

REM / => .EMPTY;

AND / => .EMPTY;

OR / => .EMPTY;

NOT / => .EMPTY;

ENDN / => "T","BSS",1W\ ,"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 18; BRU "*2"; LDX =3; BRS 19; BRU "
                                  *2"; STA "*1\;
```

```
CLSFIL[-] => ,"LDA "*1"; BRS 20"\;

BRS38[-,.ID] => ,"LDA "*1"; LDB =10; BRS 38; STA "*2\
     [-,-]   => BRS38[*1,*2:*1] BRS38[*1,*2:*2];

XCIO[-,.ID] => ,"CIO "*1"; STA "*2\
    [-,-]   => XCIO[*1,*2:*1] XCIO[*1,*2:*2];

OUTCAR[-,.ID] => ,"LDA "*2"; CIO "*1\
      [-,.NUM] => ,"LDA ="*2"; CIO "*1\
      [-,.SR]  => ,"LDA ="#1"; LDB ="*2:L"; LDX "*1"; BRS 36; BRU "*2\
                                  #1,"ASC "''*2''\
      [-,-]    => OUTCAR[*1,*2:*1] OUTCAR[*1,*2:*2];

OUTNUM[-,.ID] => ,"LDA "*1"; LDA =10; BRS 38;"\
      [-,.NUM] => ,"LDA ="*2"; CIO "*1\
      [-,.SR] => ,"LDA ="#1"; LDB ="*2:L"; LDX "*1"; BRS 36; BRU "*2\
                 #1,"ASC "''*2''\
      [-,-] => OUTNUM[*1,*2:*1] OUTNUM[*1,*2:*2];

STARTN / => "START","EQU","*"\;

DECN[.ID] => *1,"BSS 1"\
    [-]   => DECN[*1:*1] DECN[*1:*2] ;

.END
```

```
                .META PROGRM    %5%

PROGRM =        ".META" .ID ?1? <"META II 1.1">
        ["  NOLIST EXT,NUL;$START BRM INITL"]
        ["$KSTKSZ EQU 1;$MSTKSZ EQU 100;$NSTKSZ EQU 1;$SSSIZE EQU 550"]
        (".LIST" [,"CLA; STA LISTFG"] / .EMPTY)
        [,"BRM RLINE; BRM "*"; BRM FINISH"]
        ('( SIZ $(', SIZ) ') ?17E / .EMPTY)
                $ST ".END" ?2E
        ["STAR BSS 1;SSTOP DATA SS+SSSIZE-5;$SS BSS SSSIZE"]
        ["$MSP DATA MSTK;$MSPT DATA MSTK+MSTKSZ-5;$MSTK BSS MSTKSZ"]
        ["$NSP DATA NSTK;$NSPT DATA NSTK+NSTKSZ-5;$NSTK BSS NSTKSZ"]
        ["$KSP DATA KSTK;$KSPT DATA KSTK+KSTKSZ-5;$KSTK BSS KSTKSZ"]
        [,"END"]    <"DONE">;
ST =    .ID '= ?3E <"ST"> [*,"ZRO; LDA *-1; BRM CLL"]
        EXP ?4E '; ?5E  [,"BRU RTN"];
EXP =   SUBEXP $('/  [,"LDA MFLAG; SKE =0; BRU "*1]
                SUBEXP) [*1,"EQU *"];
SUBEXP =        (GEN / ELT [,"LDA MFLAG; SKE =1; BRU "*1])
                $REST [*1,"EQU *"];
REST =  GEN / ELT [,"LDA MFLAG; SKE =0; BRU *+4"]
                ('?  .NUM ?12E [,"LDA ="*"; BRM ERR"]
                                (.ID [,"BRM",*]/ '?  [,"BRS EXIT"])?13E/
                .EMPTY [,"CLA; BRM ERR; BRS EXIT"]);
ELT =  '.   .ID ?6E [,"BRM",*"; STA STAR"] /
        .ID [,"BRM",*]/
        .SR [,"BRM TST; DATA "*L"; ASC "''*''] /
        '( EXP ?7E  ') ?8E  /
        '' .CHR [,"LDA ="*N"; BRM TCH"];
GEN =  '[    $OUT '] ?10E  [,"BRM CRLF"] /
        '$  [*1,"EQU *"] ELT ?9E
                [,"LDA MFLAG; SKE =0; BRU "*1"; MIN MFLAG"] /
        ".EMPTY" [,"LDA =1; STA MFLAG"] /
        ".CHR" [,"BRM WPREP; BRM INC; LDA* IWP; STA STAR; MIN NCCP"] /
    '< .SR ?12E '> ?13E [,"BRM LITT; DATA "*L"; ASC "''*''"; BRM CRLFT"]/
        "=>" [*1,"EQU *"] ELT ?14E
                [,"LDA MFLAG; SKE =0; BRU *+3; MIN NCCP; BRU "*1]/
        '! .SR ?15E [,*];
OUT =   .SR [,"BRM LIT; DATA "*L"; ASC "''*''] /
        ',  [,"BRM TAB"] /
        '*  (.NUM [,"LDA =47B; CIO FNUMO; MIN CHNO; LDA GN"
                *"; BRM GENLAB; STA GN"*"; BRM OUTN"] /
                'L  [,"LDA* STAR; BRM OUTN"] /
                'N  [,"LDA STAR; BRM OUTN"] /
                'C [,"LDA STAR; CIO FNUMO; MIN CHNO"] /
                .EMPTY [,"LDA STAR; BRM OUTS"])/
        '' .CHR [,"LDA ="*N"; CIO FNUMO; MIN CHNO"]/
        ': [,"BRM CRLF"];
E = =>  ';  [,"BRU RTN"] $ST ".END" ?11E [,"END"] FINISH;
SIZ =   "K=" .NUM ["$KSTKSZ EQU "*] /
        "M=" .NUM ["$MSTKSZ EQU "*] /
        "N=" .NUM ["$NSTKSZ EQU "*] /
        "S=" .NUM ["$SSSIZE EQU "*];
                .END
```

```
                .META PROGRM  %TREE 1.3%

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

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

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

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

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

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

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

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

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

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

STEST = '. .ID ?19E :PRIM[1] /
      .ID :CALL[1]/
       .SR :STST[1] /
       '( EXP ?20E ') ?21E /
       '' .CHR :CTST[1]/
    (.NUM '$ ?23E /'$ :ZRO[0]) (.NUM /.EMPTY :MT[0]) STEST ?24E :ARB[3]/
       '- (.SR :NSR[1] / '' .CHR  :NCHR[1]) ?26E :NTST[1];
```

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

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

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

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

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

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

SUBOUT = OUTT (REST :CONCAT[2] / .EMPTY) / REST;

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

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

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

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

NSIMP1 = ← ': NSIMP :UP[2] / NSIMP :LKT[1];

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

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

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

GENU =  OUT /
        '. .ID ?42E '[ ?43E ((.ID / .NUM) :LOAD[1] :CALL[2] /
                                .EMPTY :CALL[1]) '] /
        '# .NUM :GNLBL[1] (':  :DEF[1] / .EMPTY) ;

OUT = ('\ :OUTCR / ', :OUTAB) [0] /
      .SR :OUTSR[1] /
      '' .CHR :OUTCH[1] /
      "+W" :UPWRK[0] :OUTWRK[1] /
      "-W" :DWNWRK[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;PSHN OPD 104B5,1,1;PSHK OPD 105B5,1,1"\
       "MKND OPD 106B5,1,1;NDLBL OPD 107B5,1,1;GET OPD 110B5,1,1"\
       "BPTR OPD 111B5,1,1;BNPTR OPD 112B5,1,1;RI1 OPD 113B5,1,1"\
       "RI2 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"\
    "$KSTKSZ EQU 100;$MSTKSZ EQU 130;$NSTKSZ EQU 1300;$SSTKSZ EQU 1400"\
       *1;

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

LIST / => " CLA; STA LISTFG;";


OUTPT[-,-] => *1:S ,"ZRO; LDA *-1; BRM CLLO"\ *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 *"\;

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\
       [.ID] => ,"LDA "*1:S\;

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


MT / => .EMPTY;

CLA / => "CLA";

ZRO / => "0";
```

```
FNCODE[-,-,-] => *1 *2 ,"BF "*3\;

NDLBL[-] => ,"NDLBL ="*1\;

MKNODE[-] => ,"MKND ="*1\;

ARB[ZRO[],MT[],-] => #1.D[] *3,"BT "#1"; MIN MFLAG"\
        [.NUM,MT[],-] => ARB1[*1] #1.D[]  *3
            ,"SKR* MSP; BT "#1"; SKN* MSP; BRU *+3; BT "#1"; MIN MFLAG"\
                .ARB3[]
        [-,.NUM,-] => ARB1[*2] #1.D[] *3
                ,"SKR* MSP; BT "#1"; SKN* MSP"\ ARB2[*1,*2];

ARB1[-] => ,"BRM SAV; LDA ="*1:S"+1; MIN MSP; STA* MSP"\;

ARB2[-,.NUM] => ,"BRU *+4; CLA; STA MFLAG; BRU *+4; LDA* MSP; SKG ="*2
                "-"*1"; MIN MFLAG"\ .ARB3[]
        [-] => ,"BRU *+3; CLA ; STA MFLAG"\ .ARB3[];

ARB3 / => ,"LDA =-1; ADM MSP; BRM RSTR"\;

GCHR /=> ,"BRM WPREP; BRM INC; LDA* IWP; MRG CHRFLG; MIN NCCP; PSHK =0"\

GO / => ,"BRM OUTREE; BT *+3; LDA =2; BRM CERR"\;

SET / => ,"LDA =1; STA MFLAG"\;

TTY[-] => TTY[] *1 FIL[]
   [] => ,"LDA =1; STA FNUMO"\ XCHCH[];

FIL[] => ,"LDA XFNUMO; STA FNUMO"\;

XCHCH/ => ,"LDA TCHNO; XMA CHNO; STA TCHNO"\;

STRING[-] => " DATA "*1:L"; ASC "''*1''"\;

OSET[-] => ,"BRM BEGN"\ *1;

CNTCK[-] => *1 ,"CLB; SKE NCNT; STB MFLAG"\;

ONE / => ,"LDA =1"\;

TWO / => ,"LDA =2"\;

THRE / => ,"LDA =3"\;

ITMSTR [-,-] => *1 ,"MIN CNT; EAX -1,2"\  *2;

LITEM [-] => *1 ,"MIN CNT; LDA CNT"\;

RITEM[-,-] => ,"RI1 ="*1"; BRU "#1\ *2 ,"RI2"\ #1.D[];

OER[-,-] => *1, "CE =1"\ *2;
```

```
OUTCLL [-,-] => ,"LDA NSP; STA SNSP; NDLBL ="*1"; CLA; STA CNT"\
                ,"LDA KT; STA ME"\ *2
       ,"MKND CNT; PSHN SNSP; LDX KT; 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
       "; CAX; 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"\;

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

CTST [-] => ,"LDA ="*1:N"; BRM TCH"\;

OS / => " BRM OUTS"\;

ON / => " ETR =77777B; BRM OUTN"\;

OL / => " CAX; LDA 0,2; BRM OUTN"\;

OC / => " ETR =377B; CIO FNUM0; MIN CHNO"\;

GNLBL [-] => ,"GEN GNLB"*1\;

DEF [-] => *1 ,"BRM LIT; DATA 6; ASC "'*'" EQU *"'*'\;

OUTCR / => ,"BRM CRLF"\;

OUTAB / => ,"BRM TAB"\;

OUTSR [-] => ,"BRM LIT; " STRING[*1];

OUTCH [-] => ,"LDA ="*1:N"; CIO FNUM0; MIN CHNO"\;

FNDN / => "SSTOP DATA SS+SSTKSZ-5;$SS BSS SSTKSZ"\
       "MSP DATA MSTK;$MSPT DATA MSTK+MSTKSZ-5;$MSTK BSS MSTKSZ"\
       "NSP DATA NSTK;$NSPT DATA NSTK+NSTKSZ-5;$NSTK BSS NSTKSZ"\
       "KSP DATA KSTK;$KSPT DATA KSTK+KSTKSZ-5;$KSTK BSS KSTKSZ"\
       "WRK BSS 1;XWRK BSS 1; END"\;

SAVG [-] => ,"BRM SAVGN"\ *1 ,"BRM RSTGN"\;
```

```
IMED [-] => ,*1\;

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

FITEM[-] => ,"FLGT "*1:S"FLG"\;

TTST[-] => ,"BRM SSTEST;" 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-"*1:S",2"\ *2;

LKT[-] => ,"LDA KT"\ *1;

UPWRK / => ,"MIN WRK; LDA WRK; SKG XWRK; LDA XWRK; STA XWRK"\;
DWNWRK / => ,"LDA =-1; ADM WRK"\;
OUTWRK[-] => *1, "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"\;

PAROUT[ZRO[]] => ,"LDA KT"\
       ["0"] => ,"LDA KT"\
       [-] => ,"LDKA ="*1\;

              .END
```