RULIFSON

STANFORD RESEARCH INSTITUTE

MENLO PARK, CALIFORNIA

TREE META

(WORKING DRAFT)

29 December 1967

A META COMPILER SYSTEM

FOR THE SDS 940

By

D. I. Andrews and J. F. Rulifson

Stanford Research Institute
Menlo Park, California

Note: This work was supported jointly by:

1) National Aeronautics and Space Administration
   Langley Research Center

2) Rome Air Development Center
   Griffiss Air Force Base

3) Advanced Research Projects Agency
   Department of Defense

Copy No. 1

0a Tree Meta is a compiler-compiler system for context-free languages. Parsing statements of the metalanguage resemble Backus-Naur Form with embedded tree-building directives. Unparsing rules include extensive tree-scanning and code-generation constructs. Examples are drawn from algebraic and special-purpose languages, as well as the process of bootstrapping the comprehensive, self-defining, tree language from a simpler metalanguage. Thorough implementation documentation for the Scientific Data System 940 appears in the discussion of the support subroutines and in the appendices. A history of computer metalanguages, a tutorial guide to Tree Meta, and the practical usefulness and scope of the system are other topics of the report.

0b This is an interim project report and reflects the current status of a portion of a constantly evolving programming system.

0c Documentation level as of 29 December 1967 is TM1.3. .newp

Tree Meta - CONTENTS - 29 DEC. 1967

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

1c A language which can be used to describe other languages is a metalanguage. English is an informal, general metalanguage which can describe any formal language. Backus-Naur Form or BNF (NAUR1) is a formal metalanguage used to define ALGOL. BNF is weak, for it describes only the syntax of ALGOL, and says nothing about the semantics or meaning. English, on the other hand, is powerful, yet its informality prohibits its translation into computer programs.

1d A metacompiler, in the most general sense of the term, is a program which reads a metalanguage program as input and translates that program into a set of instructions. If the input program is a complete description of a formal language, the translation is a compiler for the language.

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

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

2b Compiler-compiler systems may be rated on two almost independent features: the syntax they can handle and the features within the system which ease the compiler-building process.

2b1 Tree Meta is intended to parse context-free 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 which describe a method of generating all the possible strings of the language.

3b The reductive or analytic technique states a set of rules which describe a method of analyzing any string of characters and deciding whether that string is in the language. This approach simultaneously produces a structure for the input string so that code may be compiled.

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

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

4a The following is a formal discussion of top-down parsing algorithms. It relies heavily on definitions and formalisms which are standard in the literature and may be skipped by the lay reader. For a language L, with vocabulary V, nonterminal vocabulary N, productions P, and head S, the top-down parse of a string u in L starts with S and looks for a sequence of productions such that S=>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 <u>boolean</u> <u>procedure</u> E; E := <u>if</u> T <u>then</u> (<u>if</u> issymbol('+') <u>then</u> E <u>else</u> <u>true</u>) <u>else</u> <u>false</u>; comment issymbol (arg) is a Boolean procedure which compares the next symbol in the input string with its argument, arg. If there is a match the input stream is advanced;

4a2b <u>boolean</u> <u>procedure</u> T; T := <u>if</u> F <u>then</u> (<u>if</u> issymbol('*') <u>then</u> T <u>else</u> <u>true</u>) <u>else</u> <u>false</u>;

4a2c <u>boolean</u> <u>procedure</u> F; F := <u>if</u> issymbol('X') <u>then</u> <u>true</u> <u>else</u> <u>if</u> issymbol('(') <u>then</u> (<u>if</u> E <u>then</u> (<u>if</u> issymbol(')') <u>then</u> <u>true</u> <u>else</u> <u>false</u>) <u>else</u> <u>false</u>) <u>else</u> <u>false</u>;

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 := <u>if</u> T <u>then</u> <u>true</u> <u>else</u> <u>if</u> 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 (METCALFE1). One used a bottom-to-top analysis technique based on a method described by Ledley and Wilson (LEDLEY1). The other used a top-to-bottom approach based on a work by 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 which produced efficient code for Boolean expressions.

7b Another paper in the 1964 ACM proceedings describes Meta III, developed by Schneider and Johnson at UCLA for the IBM 7090 (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).

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

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

1b As the rules (acting as instructions) compile a program, they read an input stream of characters one character at a time. Each new character is subjected to a series of tests until an appropriate syntactic description is found for that character. The next character is then read and the rule testing moves forward through the input.

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

```
.null
R1A             EXP = TERM ("+" EXP / "-" EXP / .EMPTY);
.null
R2A             TERM = FACTOR S("*" FACTOR / "/" FACTOR);
.null
R3A             FACTOR = "-" FACTOR / PRIM;
.null
```

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

.null

2a The identifier to the left of the initial equal sign names the rule. This name is used to refer to the rule from other rules. The name of rule R1A is EXP.

2b The right part of the rule--everything between the initial equal sign and the trailing semicolon--is the part of the rule which effects the scanning of the input. Five basic types of entities may occur in a right part. Each of the entities represents some sort of a test which results in setting a general flag to either "true" or "false".

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

2b2 Rule names may also occur in a right part. If a rule is processing input and a name is reached, the named rule is invoked. R3A defines a FACTOR as being either a minus sign followed by a FACTOR, or just a PRIM.

2b3 The right part of the rule FACTOR has just been defined as "a string of elements," "or" "another string of elements." The "or's" are indicated by slash marks (/) and each individual string is called an alternative. Thus, in the above example, the minus sign and the rule name FACTOR are two elements in R3A. These two elements make up an alternative of the rule.

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

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

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

3a EXP first calls rule TERM, which calls FACTOR, which tests for a minus sign. This test fails and FACTOR then tests for a plus sign and fails again. Finally FACTOR calls PRIM, which tests for an identifier. The character X is an identifier; it is recognized and the input stream advances one character.

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

3c EXP now calls on itself. All necessary information is saved so that the return may be made to the right place. In calling on itself, it goes through the sequence just described until it recognizes the Y.

3d Thinking of the rules in this way is confusing and tedious. It is best to think of each rule separately. For example: one should think of R2A as defining a TERM to be a series of FACTORs separated by asterisks and slashes and not attempt to think of all the possible things a FACTOR could be.

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

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

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

4a1 Using the four rules above, the input stream

.null

.null X+Y*Z

.null

has the following parse tree

.null

.null

.null

.null

.null

.null

.null

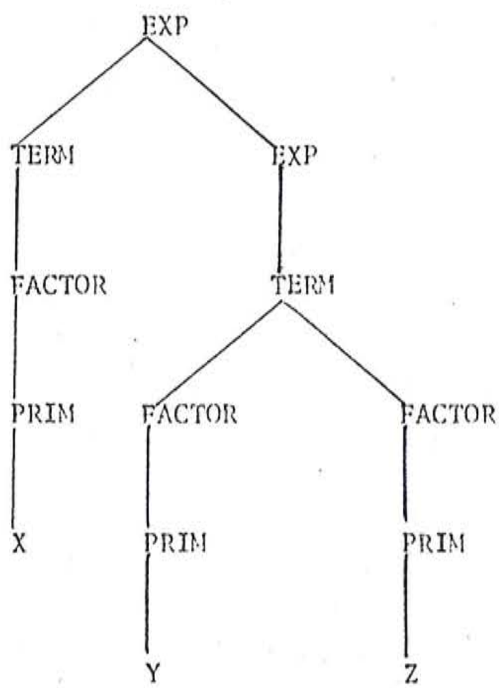
.null

.null

.null

.null

.null



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

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

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

4b1 The parse tree actually produced is

.null

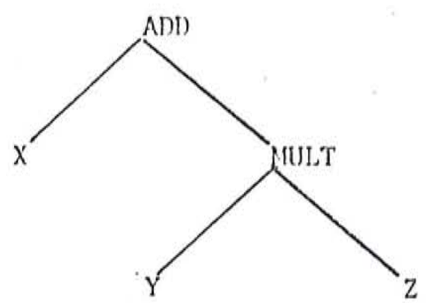
.null

.null

.null

.null

.null



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

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

.null

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

.null

R2B TERM = FACTOR $(("*" FACTOR :MULT/ "/" FACTOR :DIVD)

[2]);

.null

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

.null

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

4c As these rules scan an input stream, they perform just like the first set. As the entities are recognized, however, they are stored on a push-down stack until the node-generation elements remove them to make trees. We will step through these rules with the same sample input stream:

.null X+Y*Z

4c1 EXP calls TERM, which calls FACTOR, which calls PRIM, which recognizes the X. The input stream moves forward and the X is put on a stack.

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

Again EXP calls TERM, which calls FACTOR, which calls PRIM, which recognizes the Y. The input stream is advanced, and Y is put on the push-down stack. The stack now contains Y X, and the next character on the input stream is the asterisk.

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

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

4c5 The :MULT in now processed. This names the next node to be put in the tree. Later we will see that in a complete metacompiler program there will be a rule named MULT which will be processed when the time comes to produce code from the tree. Next, the [2] in the rule TERM is processed. This tells the system to construct a portion of a tree. The branch is to have two nodes, and they are to be the last two entities recognized (they are on the stack). The name of the branch is to be MULT, since that was the last name given. The branch is constructed and the top two items of the stack are replaced by the new node of the tree.

4c5a The stack now contains

.null MULT

.null X

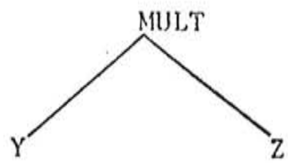
4c5b The parse tree is now

.null

.null

.null

.null



4c5c Notice that the nodes are assembled in a left-to-right order, and that the original order of recognition is retained.

4c6 Rule TERM now returns to EXP which names the next node by executing the :ADD, i.e., names the next node for the tree. The [2] in rule EXP is now executed. A branch of the tree is generated which contains the top two items of the stack and whose name is ADD. The top two items of the stack are removed, leaving it as it was initially, empty. The tree is now complete, as first shown, and all the input has been passed over.

5 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.

5a Before we discuss the node-testing features, let us first describe the various types of output that may be produced. The following list of output-generation features in the metacompiler system is enough for most examples.

5a1 The output is line-oriented, and the end of a line is determined by a carriage return. To instruct the system to produce a carriage return, one writes a backslash (upper-case L on a Teletype) as an element of an unparse rule.

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

5a3 A literal string can be inserted in the output stream by merely writing the literal string in the unparse rule. Notice that in the unparse rule a literal string becomes output, while in the parse rules it becomes an entity to be tested for in the input stream. To output a line of code which has L as a label, ADD as an operation code, and SYS as an address, one would write the following string of elements in an unparse rule:

```
.null            "L" , "ADD" , "SYS"
```

5a4 As can be seen in the last example of a tree, a node of the tree may be either the name of an unparse rule, such as ADD, or one of the basic entities recognized during the parse, such as the identifier X.

5a4a Suppose that the expression X+Y*Z has been parsed and the program is in the ADD 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:

```
.null          , "ADA", *1
```

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

5a5 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.

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

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

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

6a An 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."

6b The simplest test that can be made is the test to ensure that the correct number of nodes emanate from the node being processed. The ADD rule may begin

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

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

6b1 If one wishes to eliminate the test at the head of the out-rule, one may write a slash instead of the bracketed string of items. The slash, then, takes the place of the test and is always true. Thus, a rule which begins with a slash immediately after the rule name may have only one out-rule. The rule

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

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

6b2 The nodes emanating from the node being evaluated are referred to as *1, *2, etc., counting from left to right. To test for equality between nodes, one merely writes *i for some i as the desired item in an out-test. For example, to see if node 2 is the same as node 1, one could write either [-,*1] or [*2,-]. To see if the third node is the same as the first, one could write [-,*2,*1]. In this case, the *2 could be replaced by a hyphen.

6b3 One may test to see if a node is an element which was generated by one of the basic recognizers by mentioning the name of the recognizer. Thus to see if the node is an identifier one writes .ID; to test for a number one writes .NUM. To test whether the first node emanating from the ADD is an identifier and if the second node exists, one writes [.ID,-].

6b4 To check for a literal string on a node one may write a string as an item in an out-test. The construct [-,"1"] tests to be sure that there are two nodes and that the second node is a 1. The second node will have been recognized by the .NUM basic recognizer during the parse phase.

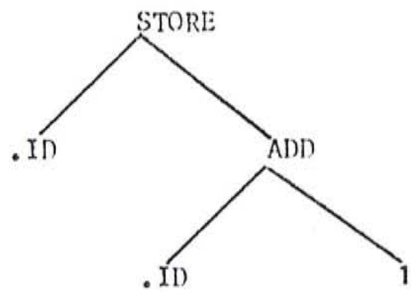
6b5 A generated label may be inserted into the tree by using it in a call to an unparse rule in another unparse rule. This process will be explained later. To see if a node is a previously generated label one writes a number sign followed by a number. If the node is not a generated label the test fails. If it is a generated label the test is successful and the label is associated with the number following the number sign. To refer to the label in the unparse rule, one writes the number sign followed by the number.

6b6 Finally, one may test to see if the name matches a specified name. Suppose that one had generated a node named STORE. The left node emanating from it is the name of a variable and on the right is the tree for an expression. An unparse rule may begin as follows:

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

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

.null
.null
.null
.null
.null
.null
.null



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

6c 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.

6c1 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.

6c2 The test in an out-expr resembles the test for the out-rule. There are two types of these tests.

6c2a Any nonterminal node in the tree may be transferred to by its position in the tree rather than its name. For example, *2 would invoke the second node from the right. This operation not only transfers control to the specific node, but it makes that node the one from which the next set of nodes tested emanate. After control is returned to the position immediately following the *2, the general flag is tested. If it is "true" the out-expr 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.

6c2b 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:

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

6c2b1 Only nodes and generated labels can be written as arguments. Nodes are written as *1, *2, etc. To reach other nodes of the tree one may write such things as *1:*2, which means "the second node emanating from the first node emanating from the node being evaluated." Referring to the tree for the expression X+Y*Z, if ADD is being evaluated, *2:*1 is Y. To go up the tree one may write an "uparrow" (↑) followed by a number before the asterisk-number-colon sequence. The uparrow means to go up that many levels before the search is made down the tree. If MULT were being evaluated, ↑1*1 would be the X.

6c2b2 If a generated label is written as an argument, it is generated at that time and passed to the called unparse rule so that that rule may use it or pass it on to other rules. The generated label is written just as it is in an output element--a number sign followed by a number.

6c3 The calls on other 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.

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

6d Somtimes it is necessary to set the general flag in an out-expr, just as it is sometimes necessary in the parse rules. .EMPTY may be used as an element in an out-expr at any place.

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

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

7a If a literal string is only one character long, one may write an apostrophe followed by the character rather than writing a quotation mark, the character, and another quotation mark. For example: 'S and

"S" are interchangeable in either a parse rule or an unparse rule.

7b As the parse rules proceed through the input stream they may come to a point where they are in the middle of a parse alternative and there is a failure. This may happen for two reasons: backup is necessary to parse the input, or there is a syntax error in the input. Backup will not be covered in this introductory chapter. If a syntax error occurs the system prints out the line in error with an arrow pointing to the character which cannot be parsed. The system then stops. To eliminate this, one may write a question mark followed by a number followed by a rule name after any test except the first in the parse equations. For example:

```
.null          ST = .ID '= question 2 E EXP question 3 E ';
.null                    question 4 E :STORE[2] ;
```

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

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

7d In addition to the three basic recognizers .ID, .NUM and .SR, there are two others which are occasionally very useful.

7d1 The symbol .LET indicates a single letter. It could be thought of as a one-character identifier.

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

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

7e1 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 which normally are put in the parse tree may be copied to the output stream by referencing them in the output expression. The most recent item recognized is referenced as * or *S0. Items recognized previous to that are *S1, *S2, etc., counting in reverse order--that is, counting down from the top of the stack they are kept in.

7e2 Normally the items are removed from the stack and put into the tree. However, if they are just copied directly to the output stream, they remain in the stack. They are removed by writing an ampersand at the end of the parse rule (just before the semicolon). This causes all input items added to the stack by that rule to be removed. The input stack is thus the same as it was when the rule was called.

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

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

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

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

2 Input Machinery

2a The input stream of text is broken into lines and put into an input buffer. Carriage returns in the text are used to determine the ends of lines. Any line longer than 80 characters is broken into two lines. This line orientation is necessary for the following:

2a1 Syntax-error reporting

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

2a3 An interlinear listing option.

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

2b It is the job of routine RLINE to fill the input line buffer. If the listing flag is on, RLINE copies the new line to the output file (prefixed with a comment character--an asterisk for our assembler). It also checks for an End-of-File, and for a multiple blank character, which is a system feature built into our text files. There is a buffer pointer which indicates which character is to be read from the line buffer next, and RLINE resets that pointer to the first character of the line.

2c Input characters for the Tree Meta program are not obtained from the input line buffer, but from an input window, which is actually a character ring buffer. Such a buffer is necessary for backup. There are three pointers into the input window. A program-character counter (PCC) points to the next character to be read by the program. This may be moved back by the program to effect backup. A library-character counter (LCC) is never changed except by a library routine when a new character is stored in the input window. PCC is used to compute the third pointer, the input-window pointer (IWP). Actually, PCC and LCC are counters, and only IWP points into the array RING which is the character ring buffer. LCC is never backed up and always indicates the next position in the window where a new character must be obtained from the input line buffer. Backup is registered in BACK, and is simply the difference between PCC and LCC. BACK is always negative or zero.

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

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

2d2 The routine INC is used to put a character into the input window. It increases IWP by one by calling a routine, 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.

2d3 A routine called INCS is similar to INC except that it deletes all blanks or comments which may be at the current point in the input stream. This routine implements the comment and blank deletion for .ID, .NUM, .SR, and other basic recognizers. INCS first calls INC to get the next character and increment IWP. From then on, PUTIN is called to store succeeding characters in the input window in the same slot. As long as the current character (at IWP) is a blank, INCS calls PUTIN to replace it with the next character. The nonblank character is then compared with a comment character. INCS returns if the comparison fails, but otherwise skips to the next comment character. When the end of the comment is located, INCS returns to its blank-checking loop.

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

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

2d5 The program-library interface for inputting items from the input stream consists of the routines ID, NUM, SR, LET, and CHR. The first four are quite similar. ID is typical of them, and works as follows: First MFLAG is set false. WPREP is called to set up IWP, then INCS is called to get the first character. If the character at IWP is not a letter, ID returns (MFLAG is still false); otherwise a loop to input over letter-digits is executed. When the letter-digit test fails the flag is set true, and the identifier is stored in the string storage area. The class of characters is determined by an array (indexed by the character itself) of integers indicating the class. Before returning, ID calls the routine GOBL which updates PCC to the last character read in (which was not part of the identifier). That is, PCC is set to LCC+BACK-1.

2d6 The occurrence of a given literal string in the input stream is tested for by calling routine TST. The character count and the string follow the call instruction. TST deletes leading blanks and inputs characters, comparing them one at a time with the characters of the literal string. If at any point the match fails, TST returns false. Upon reaching the end of the string, TST sets the flag true, sets PCC to LCC+BACK, and returns. In addition to TST, there is a simple routine to test for a single character string (TCH). It inputs one character (deleting blanks), compares it to the given character and returns false, or adjusts PCC and returns true.

3 Stacks and Internal Organization

3a Three stacks are available to the program. A stack called MSTACK is used to hold return locations and generated labels for the program's recursive routines. Another stack, called KSTACK, contains references to input items. When a basic recognizer is executed, the reference to that input item is pushed into KSTACK. The third stack is called NSTACK, and contains the actual tree. The three stacks are declared in the Tree Meta program rather than the library: the program determines the size of each.

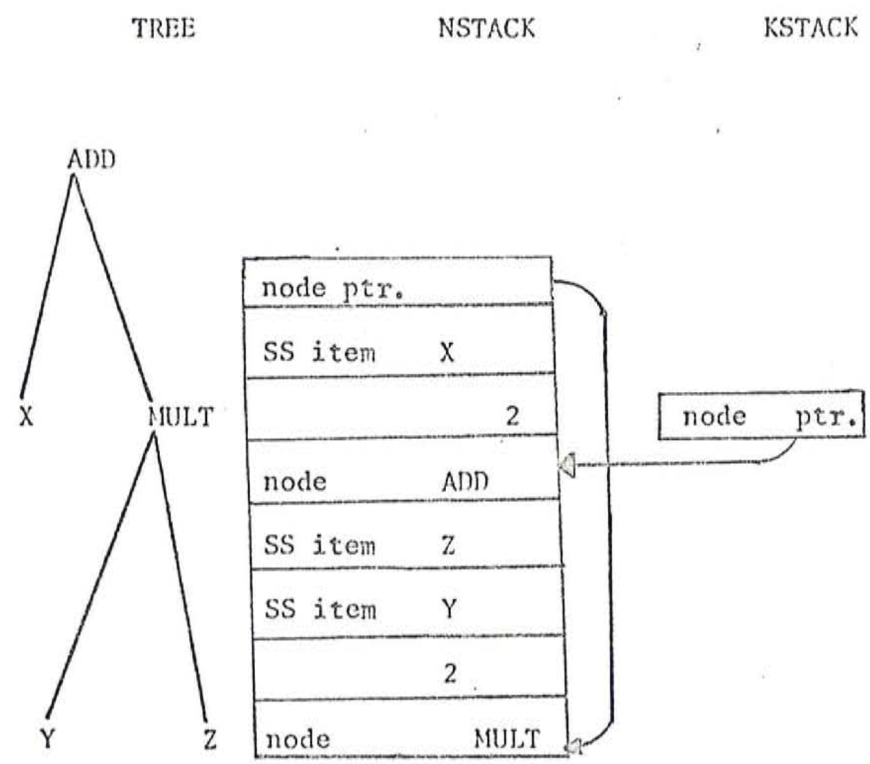
3a1 The operation of MSTACK is very simple. At the beginning of each routine, the current generated labels and the location that the routine was called from are put onto MSTACK. The routine is then free to use the generated labels or call other routines. The routine ends by restoring the generated labels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

3a2 KSTACK contains single-word entries. Each entry will eventually be placed in NSTACK as a node in the tree. The format of the node words is as follows: There are two kinds of nodes, terminal and nonterminal. Terminal nodes are references to input items. Nonterminal nodes are generated by the parse rules, and have names which are names of output rules.

3a2a A terminal node is a 24-bit word with either a string-storage index or a character in the address portion of the word, and a flag in the top part of the word. The flag indicates which of the basic recognizers (ID, NUM, SR, LET, or CHR) is to read the item from the input stream. .newp

3a2b A nonterminal node consists of a word with the address of an output rule in the address portion, and a flag in the top part which indicates that it is a nonterminal node. A node pointer is a word with an NSTACK index in the address and a pointer flag in the top part of the word. Each nonterminal node in NSTACK consists of a nonterminal node word followed by a word containing the number of subnodes on that node, followed by a terminal node word or node pointers for each subnode. For example,

.null
.null
.null
.null
.null
.null
.null
.null
.null
.null
.null
.null
.null



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

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

3c Other routines perform housekeeping functions like packing and unpacking strings, etc. There are three error-message writing routines to write the three types of error messages (syntax, system, and compiler). The syntax error routine copies the current input line to the teletype and gives the line number. A routine called FINISH closes the files, writes the number of cells used for each of the four stack areas (KSTACK, MSTACK, NSTACK, and SS), and terminates the program.

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

3c1a (1) Class codes are illegal

3c1b (2) Backup too far

3c1c (64) Character with code greater than 63 in ring buffer

3c1d (4) Test for string longer than ring size

3c1e (5) Trying to output a string longer than maximum string length

3c1f (6) String-storage overflow

3c1g (7) Illegal character code

3c1h (8) Trying to store SS element of length zero

3c1i (11) MSTACK overflow

3c1j (12) NSTACK overflow

3c1k (13) KSTACK overflow

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

4 Output Facilities

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

4a1 A string in SS can be written on the output stream by calling the routine OUTS with the SS index for that string. OUTS checks the SS index and generates a system-error message if it is not reasonable.

4a2 A literal string of characters is written by calling the routine LIT. The literal string follows the call as for TST.

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

4a4 All of the above routines keep track of the number of characters written on the output stream (in CHNO). Based on this count, a routine called TAB will output enough spaces to advance the current output line to the next tab stop. Tabs are set at 8-character intervals. The routine CRLF will output a carriage return and a line feed and reset CHNO.

4a5 There are several routines that are convenient for debugging. One (WRSS) will print the contents of SS. Another (WRIW) will print the contents of the input window.

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

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

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

2 Programs and Rules

2a Syntax

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

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

2a3 siz = .chr '= .num;

```
2a4   rule  =  .id  ('= exp (.ampersand / .empty) / '/ "=>" gen1 /
outrul) '; ;
```

2b Semantics

2b1 A file of symbolic Tree Meta code may be either an original main file or a continuation file. A compiler may be composed of any number of files but there may be only one main file.

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

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

2b1c The size construct sets the allocation parameters for the three stacks and string storage used by the Tree Meta library. The default sizes are those used by the Tree Meta compiler. M, K, N, and S are the only valid characters; the size is something which must be determined by experience. The maximum number of cells used during each compilation is printed out at the end of the compilation.

2b2 When a file begins with .CONTINUE, no initialization or storage-allocation code is produced.

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

2b3a Parse rules are distinguished by the = following the identifier. If all the elements which generate possible nodes during the execution of a parse rule are not built into the tree, they must be popped from the kstack by writing an ampersand immediately before the semicolon.

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

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

3 Expressions

3a Syntax

```
3a1  exp = '← suback ('/ exp / .empty) / subexp ('/ exp / .empty);
```

3a2 suback = ntest (suback / .empty) / stest (suback / .empty);

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

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

3b Semantics

3b1 The expressions in parse rules are composed entirely of ntest, stest, and error-recovery constructs. The four rules above, which define the allowable alternation and concatention of the test, are necessary to reduce the instructions executed when there is no backup of the input stream.

3b2 An expression is essentially a series of subexpressions separated by slashes. Each subexpression is an alternative of the expression. The alternatives are executed in a left-to-right order until a successful one is found. The rest of that alternative is then executed and the rule returns to the rule which invoked it.

3b3 The subexpressions are series of tests. Only subexpressions which begin with a leftarrow are allowed to back up the input stream and rescan it.

3b3a Without the arrow at the head of a subexpression, any test other than the first within the subexpression may be followed by an error code. If the error code is absent and the 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.

3b3b If the test fails, the input stream is restored to the position it had when the subexpression began to test the input stream and the next alternative is tried. The input stream may never be moved back more characters than are in the ring buffer. Normally, backup is over identifiers or words and the buffer is long enough.

4 Elements of Parse Rules

4a Syntax

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

4a2 genp = genp1 / .empty;

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

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

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

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

4b Semantics

4b1 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.

4b1a The : .id construct names the next node to be put into the tree. The identifier must be the name of another rule.

4b1b 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.

4b1c The [genp] is used to write output into the normal output stream during the parse phase of the compilation.

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

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

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

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

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

4b2a The .EMPTY in any rule sets the general flag true.

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

4b3 Stests always test the input stream for a literal string or basic entity. If the entity is found it is removed from the input stream and stored in string storage. Its position in string storage is saved on a push-down stack so that the entity may later be added as a terminal node to the tree.

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

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

4b3c A literal string merely tests the input stream for the string. If it is found it is discarded. The apostrophe-character construct functions like the literal string, except that the test is limited to one character.

4b3d The number-$-number construct is the arbitrary-number operation of Tree Meta. m$n preceding an element in a parse rule means that there must be between m and n occurrences of the next element coming up in the input. The default options for m and n are zero and infinity respectively.

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

5 Unparse Rules

5a Syntax

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

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

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

5a4 `item = '- / .id '[ outest / nsimp1  / '. .id / .sr / ''.chr /`

'.pound;

5b Semantics

5b1 The unparse rules are similar to the parse rules in that they test something and return a true or false value in the general flag. The difference is that the parse rules test the input stream, delete characters from the input stream, and build a tree, while the unparse rules test the tree, collapse sections of the tree, and write output.

5b2 There are two levels of alternation in the unparse rules. The highest level is not written in the normal style of Tree Meta as a series of expressions separated by slashes; rather, it is written in a way intended to reflect the matching of nodes and structure within the tree. Each unparse rule is a series of these highest-level alternations. The tree-matching parts of the alternations are tried in sequence until one is found that successfully matches the tree. The rest of the alternation is then executed. There may be further test within the alternation, but not complete failure as with the parse rules.

5b3 The syntax for a tree-matching pattern is a left bracket, a series of items separated by commas, and a right bracket. The items are matched against the branches emanating from the current top node. The matching is done in a left-to-right order. As soon as a match fails the next alternation is tried.

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

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

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

5b5b The name of the node may be tested by writing an identifer which is the name of a rule. The identifer must then be followed by a test on the subnodes.

5b5c A nonsimple construct, primarily an asterisk-number-colon sequence, may be used to test for node equivalence. Note that this does not test for complete substructure equivalence, but merely to see if the node being tested has the same name as the node specified by the construct.

5b5d The .id, .num, .chr, .let, or .sr checks to see if the node is terminal and was put on the tree by a .id recognizer,

.num recognizer, etc. during the parse phase. This test is very simple, for it merely checks a flag in the upper part a word.

5b5e If a node is a terminal node in the tree, and if it has been recognized by one of the basic recognizers in meta, it may be tested against a literal string. This is done by writing the string as an item. The literal string does not have to be put into the tree with a .sr recognizer; it can be any string, even one put in with a .let.

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

5b5g Finally, the node may be tested to see if it is a generated label. The labels may be generated in the 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.

6 Unparse Expressions

6a Syntax

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

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

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

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

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

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

6a7  nsimp1 = '↑ nsimp / nsimp;

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

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

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

6b Semantics

6b1 The rest of the unparse rules follow more closely the style of the parse rules. Each expression is a series of alternations separated by slash marks.

6b2 Each alternation is a test followed by a series of output instructions, calls of other unparse rules, and parenthesized expressions. Once an unparse expression has begun executing calls on other rules, elements may not fail; if they do a compiler error is indicated and the system stops.

6b3 The first element of the expression is the test. This element is a call on another rule, which returns a true or false value. The call is made by writing the name of the rule followed by a series of nodes. The nodes are put together to appear as part of the tree, and when the call is made the unparse rule called views the nodes specified as the current part of the tree, and thus the part to match against and process.

6b3a Two kinds of things may be put in as nodes for the calls. The simplest is a generated label. This is done by writing a .pound followed by a number. Only the numbers 1 and 2 may be used in the current system. If a label has not yet been generated one is made up. This label is then put into the tree.

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

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

6b5 The rest of the unparse expression is made up of output commands, and more calls on unparse rules. As noted above, if any except the first call of a expression fails a compiler error is indicated and the system stops.

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

6b7 The asterisk-number-colon construct is used frequently in the Tree Meta system. It appears in the node-matching syntax as well as in the form of an element in the unparse expressions. When it is in an expression it must specify a node which exists in the tree.

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

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

6b7c The simplest form of the construct is an asterisk followed by a number, in which case the node is found by counting the appropriate number of nodes from left to right. This may be followed by a colon-number construct which means to go down one level in the tree after performing the asterisk-number choice and count over the number of nodes specified by the number following the colon. This process may be repeated as often as desired, and one may therefore go as deep as one wishes. All of this specification may be preceded by an ↑-number construct which means to go up in the tree, through parent nodes, a specified number of times before starting down.

6b7d After the search for the node has been completed, a number of different types of output may be specified if the node is terminal. There is a compiler error if the node is not terminal.

6b7d1 :s puts out the literal string

6b7d2 :l puts out the length of the string as a decimal number

6b7d3 :n puts out the string-storage index pointer if the node is a string-storage element; otherwise it puts out the decimal code for the node if it is a .chr node.

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

7 Output

7a Syntax

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

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

7b Semantics

7b1 The standard primitive output features include the following:

7b1a Write a carriage return with a backslash

7b1b Write a tab with a comma

7b1c Write a literal string by giving the literal string

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

7b1e Write references to temporary storage by using a working counter. Three types of action may be performed with the counter. +W adds one to the counter and writes the current value of the counter onto the output stream. -W subtracts one from the counter and does not write anything. .W writes the current value without changing it. Finally, .pound W writes the maximum value that the counter ever reached during the compilation.

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

7b3 .pound 2 means "define generated label 2 at this point in the program being compiled." It writes the generated label in the output stream followed by an EQU *. This construct is added only to save space and writing.

1 This section of the report is merely the listings of compilers for two languages.

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

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

```
%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];


E = 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",↑W\ ,"END"\;

VAR[.ID] => *1;

CON[.NUM] => '= *1;

LE / => .EMPTY;

LT / => .EMPTY;

EQ / => .EMPTY;

GE / => .EMPTY;

GT / => .EMPTY;

NE / => .EMPTY;

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

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

OPNOUT[-,-] => ,"CLEAR; BRS 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"Z
       "=>" [*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
```

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

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

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

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

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

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

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

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

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

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

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

3d3 The error scheme for 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.

3d4 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 which Tree Meta could handle. It would also make it run faster in time-sharing mode since less would have to be swapped into core to run it.

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

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

3e1 Make the library output routines do block I/O rather than character I/O. This could cut compilation times by more that 70 percent.

3e2 Fix Tree Meta so that strings can be put into the tree and passed down to other unparse rules. This would allow the unparse rules to be more useful as subroutines and thus cut down the number of unparse rules needed in a compiler.

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

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., "ropert 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 Computer Bit-Time Simulation," Master Thesis, UCLA, August 1964.

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

11 (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.

```
.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
       :ERCODE[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";
```

```
ERCODE[-,-,-] => *1 *2 ,"BE "*3\;

NDLB[-] => ,"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 FNUMO; 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 FNUMO; MIN CHNO"\;

ENDN / => "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
```

```
*POPS, SUBROUTINES FOR TREE META.
*
GEN     POPD    10100000B,1,1   GENERATE LABEL
        LDA     =47B
        CIO     FNUMO
        MIN     CHNO
        LDA*    0
        SKE     =0
        BRU     *+4
        MIN     GN
        LDA     GN
        STA*    0
        BRM     OUTN
        BRR     0
*
BF      POPD    10200000B,1,1   BRANCH FALSE
        LDB     =77777777B
        SKB     MFLAG
        BRR     0
        BRU*    0
*
BT      POPD    10300000B,1,1   BRANCH TRUE
        LDB     =77777777B
        SKB     MFLAG
        BRU*    0
        BRR     0
*
PSHN    POPD    10400000B,1,1   PUSH THE N STACK
        LDB     =77777777B
        SKB*    0
        LDA*    0
        MIN     NSP
        STA*    NSP
OVN     LDA     NSP
        SKG     NSPT
        BRR     0
        LDA     =12
        BRM     SERR
*
PSHK    POPD    10500000B,1,1   PUSH THE K STACK
        LDB     =77777777B
        SKB*    0
        LDA*    0
        MIN     KSP
        XMA     KT
        STA*    KSP
OVK     LDA     KSP
        SKG     KSPT
        BRR     0
        LDA     =13
        BRM     SERR
*
MKND    POPD    10600000B,1,1   MAKE A NODE
        LDA*    0
```

```
        STA     MKND1
        BRU     MK1
MK2     BRM     POPK
        MIN     NSP
        STA*    NSP
MK1     SKR     MKND1
        BRU     MK2
        LDA     MARK
        MRG     PTRFLG
        MIN     KSP
        XMA     KT
        STA*    KSP
        LDA*    0
        MIN     MARK
        XMA*    MARK
        STA     MARK
        BRU     OVN
MKND1   BSS     1
*
NDLBL   POPD    10700000B,1,1   NODE LABEL
        LDA*    0
        MIN     NSP
        STA*    NSP
        LDA     NSP
        XMA     MARK
        MIN     NSP
        STA*    NSP
        BRU     OVN
*
GET     POPD    11000000B,1,1   GET A NODE
        CAX
        ADD     1,2
        SUB*    0
        CAX
        LDA     2,2
        BRR     0
*
BPTR    POPD    11100000B,1,1   BRANCH IF (A) A POINTER
        LDB     FLGMSK
        SKM     PTRFLG
        BRR     0
        BRU*    0
*
BNPTR   POPD    11200000B,1,1   BRANCH IF NO POINTER
        LDB     FLGMSK
        SKM     PTRFLG
        BRU*    0
        BRR     0
*
RI1     POPD    11300000B,1,1   REC. ITEM 1
        LDA     0,2
        LDB     FLGMSK
        SKM     PTRFLG
        BRU     RIF2
```

```
        STX     RINDX
        LDX     0,2
        LDA*    0
        SKE     0,2
        BRU     RIF1
        LDA     CNT
        MIN     MSP
        STA*    MSP
        MIN     MSP
        LDA     NCNT
        STA*    MSP
        MIN     MSP
        LDA     RINDX
        STA*    MSP
        LDA     MSP
        SKG     MSPT
        BRU     *+3
        LDA     =11
        BRM     SERR
        CXA
        BRM     SETA
        CLA
        STA     CNT
        MIN     0
        BRR     0                   SKIP IF ITEM MATCHES
RIF1    LDX     RINDX
RIF2    CLA
        STA     MFLAG
        BRR     0
RINDX   BSS     1
RICNT   BSS     1
*
RI2     POPD    11400000B,2      REC. ITEM 2
        LDA     =-1
        LDX*    MSP
        ADM     MSP
        LDB*    MSP
        STB     NCNT
        ADM     MSP
        LDB*    MSP
        STB     CNT
        ADM     MSP
        BRR     0
*
FLGT    POPD    115B5,1,1           FLAG TEST
        LDA     0,2
        LDB     FLGMSK
        SKM*    0
        BRU     FLGTF
        BRR     0
FLGTF   CLA
        STA     MFLAG
        BRR     0
*
```

```
BE      POPD    116B5,1,1
        LDB     =77777777B
        SKB     MFLAG
        BRR     0
        LDA*    0
        SKE     =-1
        BRU     *+2
        CLA
        BRM     ERR
        LDA*    0
        SKE     0
        SKG     =0
        BRS     EXIT
        BRU*    0
*
LAB     POPD    117B5,1,1
        LDA*    0
        SKE     =0
        BRR     0
        MIN     GN
        LDA     GN
        STA*    0
        BRR     0
*
CE      POPD    120B5,1,1
        LDB     =77777777B
        SKB     MFLAG
        BRR     0
        LDA*    0
        BRM     CERR
*
LDKA    POPD    121B5,1,1
        LDA     KSP
        SUB*    0
        CAX
        LDA     1,2
        BRR     0
*
*SUBS
*
$POPK    ZRO
        LDB*    KSP
        LDA     =-1
        ADM     KSP
        CBA
        XMA     KT
        BRR     POPK
*
$SETA    ZRO    0 SET X TO TOP OF NODE GROUP, COUNT IN NCNT
        CAX
        LDB     1,2
        ADD     1,2
        CAX
        STB     NCNT
```

```
         EAX      1,2
         BRR      SETA
*
$CLLS     ZRO
         MIN      MSP
         STA*     MSP
         LDA      MSP
         SKG      MSPT
         BRR      CLLS
         LDA      =11
         BRM      SERR
*
$RTNS     NOP
         LDA      =-1
         LDB*     MSP
         ADM      MSP
         STB      *+2
         BRR      *+1
         BSS      1
*
*
$SAV      ZRO
         LDA      NCCP
         MIN      MSP
         STA*     MSP
         LDA      NSP
         MIN      MSP
         STA*     MSP
         LDA      KSP
         MIN      MSP
         STA*     MSP
         LDA      KT
         MIN      MSP
         STA*     MSP
         LDA      MSP
         SKG      MSPT
         BRR      SAV
         LDA      =11
         BRM      SERR
*
$RSTR     ZRO
         BT       RSTT
         LDA      =-1
         LDB*     MSP
         ADM      MSP
         STB      KT
         LDB*     MSP
         STB      KSP
         ADM      MSP
         LDB*     MSP
         STB      NSP
         ADM      MSP
         LDB*     MSP
         STB      NCCP
```

```
         ADM      MSP
         BRR      RSTR
RSTT     LDA      =-4
         ADM      MSP
         BRR      RSTR
*
$OUTREE           ZRO
         LDA      KT
         BNPTR    OUTERR
         LDX*     KT
         BRM      0,2
         BRM      POPK
         LDA      =NSTK
         STA      NSP
         BRR      OUTREE
OUTERR   LDA      =2
         BRM      CERR
*
$RESET   ZRO
         LDA      =MSTK
         STA      MSP
         LDA      =KSTK
         STA      KSP
         LDA      =NSTK
         STA      NSP
         CLA
         STA      KT
         BRR      RESET
*
$SAVGN   ZRO
         LDA      GNLB1
         MIN      MSP
         STA*     MSP
         LDA      GNLB2
         MIN      MSP
         STA*     MSP
         CLA
         STA      GNLB1
         STA      GNLB2
         LDA      MSP
         SKG      MSPT
         BRR      SAVGN
         LDA      =11
         BRM      SERR
*
$RSTGN   ZRO
         LDA      =-1
         LDB*     MSP
         STB      GNLB2
         ADM      MSP
         LDB*     MSP
         STB      GNLB1
         ADM      MSP
         BRR      RSTGN
```

```
*
$SSTEST ZRO
        MIN     SSTEST
        LDA*    SSTEST
        STA     SSTCNT
        BRM     MOD3
        LDB     SSTEST
        ADM     SSTEST
        STA     SSTWDS
        STB     SSTPTR
        MIN     SSTPTR
        LDA     0,2
        BPTR    SSTT1+1
        LDA*    0,2
        SKE     SSTCNT
        BRU     SSTT1+1
        STX     INDX
        LDA     0,2
        ADD     =1
        LDB     SSTPTR
        LDX     SSTWDS
        BRM     SKSE
        BRU     SSTT1
        LDA     CNT
        LDX     INDX
        BRR     SSTEST
SSTT1   LDX     INDX
        CLA
        STA     MFLAG
        BRR     SSTEST
SSTPTR  BSS     1
SSTCNT  BSS     1
SSTWDS  BSS     1
*
$BEGN   ZRO
        LDA     =1
        STA     MFLAG
        LDA     KT
        BRM     SETA
        CLA
        STA     CNT
        BRR     BEGN
*
$CLLO   ZRO
        BRM     CLLS
        BRM     SAVGN
        BRR     CLLO
*
$RTNO   NOP
        BRM     RSTGN
        BRU     RTNS
        NOP
*
*CELLS
```

```
*
$ME     BSS     1
$INDX   BSS     1
$CNT    BSS     1
$NCNT   BSS     1
$SNSP   BSS     1
$KT     BSS     1
$SRFLG DATA     10B5
$CHRFLG DATA     12B5
$IDFLG DATA     4B5
$NUMFLG DATA    6B5
$PTRFLG DATA    2B5
$FLGMSK DATA    776B5
$GENFLG DATA    16B5
$MARK   BSS     1
$GN     DATA    0
$GNLB1  DATA    0
$GNLB2  DATA    0
$SAVKT  BSS     1
$SAVKP  BSS     1
$LETFLG DATA    14B5
        END
```

```
* ARPAS LIBRARY FOR 940 META II AND TREE SYSTEMS.
* PARAMETERS FOR SIZE OF K, M, N STACKS, AND SS AREA.
GOBL    ZRO
        LDA     MCCP
        ADD     BACK
        SUB     =1
        STA     NCCP
        BRR     GOBL
*
STORE   ZRO
        LDA     =SS
        STA     SSP
        LDA     LEN
        SKE     =0
        BRU     *+3
        LDA     =8
        BRM     SERR
        LDA     =STR
        LDB     =STEST
        LDX     LEN
        BRM     PACK
S1      LDA     SSL
        SKG     SSP
        BRU     SPUT
        LDA     SSP
        STA     SX
        LDA*    SSP
        BRM     MOD3
        MIN     SSP
        ADM     SSP
        LDA*    SX
        SKE     LEN
        BRU     S1
        BRM     MOD3
        CAX
        LDB     =STEST
        LDA     SX
        ADD     =1
        BRM     SKSE
        BRU     S1
SST     LDA     SX
        BRR     STORE
*
SPUT    LDA     SSL
        STA     SX
        LDA     LEN
        STA*    SSL
        MIN     SSL
        LDA     =STR
        LDB     SSL
        LDX     LEN
        BRM     PACK
        LDA     LEN
        BRM     MOD3
```

```
        ADM     SSL
        LDA     SSL
        SKG     SSTOP
        BRU     SST
        LDA     =6
        BRU     SERR
*
SSP     DATA    SS
$SSL    DATA    SS
SX      BSS     1
$MXSTR  EQU     80
STPTR   BSS     1
STR     BSS     MXSTR
STEST   BSS     MXSTR
$LISTFG         DATA    -1
$RLINE  ZRO
        MIN     LINCNT
        LDA     EOFLG
        SKE     =0
        BRU     REOF
        LDA     =12B
        SKN     LISTFG
        CIO     FNUMO
        LDX     BUFNO
        BRU     R1+1
R1      BRX     R3
        CIO     FNUMI
        SKN     LISTFG
        BRU     R4
R15     STA     IBUF,2
        SKE     =155B
        BRU     R2
        LDA     =152B
        SKN     LISTFG
        CIO     FNUMO
        BRU     FILL2
R2      SKE     =137B
        BRU     R1
        LDA     =1
        STA     EOFLG
FILL    CLA
        STA     IBUF,2
FILL2   BRX     R3
        BRU     FILL
R3      LDA     BUFNO
        STA     IBP
        BRR     RLINE
REOF    BRM     CRLFT
        BRM     LITT
        DATA    18
        ASC     'END OF FILE INPUT.'
        BRM     CRLFT
        BRS     EXIT
R4      SKE     =152B
```

```
        CIO     FNUMO
        BRU     R15
EOFLG   DATA    0
$INC    ZRO
        BRM     UPIWP
        SKN     BACK
        BRU     *+3
        MIN     BACK
        BRR     INC
        MIN     MCCP
        BRM     PUTIN
        BRR     INC
*
PUTIN ZRO
        BRM     PCHK
        LDX     IBP
        MIN     IBP
        LDA     IBUF,2
        SKE     =155B
        BRU     P2
        BRM     RLINE
P1      CLA
P11     STA*    IWP
        BRR     PUTIN
P2      SKE     =135B
        BRU     P3
        BRM     PCHK
        MIN     IBP
        BRU     P1
P3      SKG     =63
        BRU     P11
        BRU     PUTIN+1
*
PCHK    ZRO
        LDA     MXIB
        SKG     IBP
        BRM     RLINE
        BRR     PCHK
*
CHER   ZRO
        LDX*    IWP
        LDA     =64
        SKG*    IWP
        BRU     SERR
        LDX     CLASS,2
        CXA
        SKG     =5
        SKG     =0
        BRU     *+2
        BRR     CHER
        LDA     =1
        BRU     SERR
$WPREP ZRO
        CLA
```

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

```
        BRU     ID1
        BRR     ID
        BRR     ID
        BRR     ID
IDT2    BRU     STER
        BRU     IDF
        BRU     ID1
        BRU     ID1
        BRU     IDF
        BRU     IDF
*
CIC     ZRO
        LDA*    IWP
        STA*    STPTR
        MIN     STPTR
        MIN     LEN
        BRR     CIC
*
LEN     BSS     1
*
UPIWP   ZRO
        MIN     IWP
        LDA     IWP
        SKG     MXIW
        BRR     UPIWP
        LDA     =RING
        STA     IWP
        BRR     UPIWP
$TOUTS  ZRO
        STA     OUTP
        LDA     *-2
        STA     OUTS
        LDA     TELNO
        STA     LITF
        BRU     OUTSA
$OUTS   ZRO
        STA     OUTP
        LDA     FNUMO
        STA     LITF
OUTSA   LDA*    OUTP
        SKG     RSIZ
        BRU     *+3
        LDA     =5
        BRU     SERR
        ADM     CHNO
        CAB
        MIN     OUTP
        LDA     OUTP
        ETR     =77777B
        LDX     LITF
        BRS     34
        BRR     OUTS
*
OUTP    BSS     1
```

```
*
$OUTN   ZRO
        SKG     =-1
        BRU     OUTNN
OUTNP   STA     OUTNB
        LDB     =10
        LDX     FNUMO
        BRS     36
        LDA     =1
        SKG     OUTNB
        BRU     *+2
        BRR     OUTN
        MIN     CHNO
        MUL     =10
        RSH     1
        CBA
        BRU     *-7
OUTNN   MIN     CHNO
        CNA
        STA     OUTNB
        LDA     =15B
        CIO     FNUMO
        LDA     OUTNB
        BRU     OUTNP+1
OUTNB   BSS     1
*
$WRSS   NOP
        LDA     =SS
        STA     WRSPT
WRS1    BRM     CRLFT
        LDA*    WRSPT
        STA     WRSS1
        BRM     WOUT
        MIN     WRSPT
        LDA     WRSS1
        BRM     MOD3
        LDB     WRSPT
        ADM     WRSPT
        LDA     WRSS1
        XAB
        LDX     TELNO
        BRS     34
        LDA     WRSPT
        SKG     SSL
        BRU     WRS1
        BRM     CRLFT
        BRS     EXIT
WRSS1   BSS     1
WRSPT   BSS     1
*
$CRLFT  ZRO
        LDA     =155B
        CIO     TELNO
        LDA     =152B
```

```
        CIO     TELNO
        BRR     CRLFT
*
$CRLF   ZRO
        LDA     =155B
        CIO     FNUMO
        LDA     =152B
        CIO     FNUMO
        LDA     =1
        STA     CHNO
        BRR     CRLF
*
$LITT   ZRO
        LDA     *-1
        STA     LIT
        LDA     TELNO
        STA     LITF
        MIN     LIT
        LDA*    LIT
        BRU     LITW+3
*
$LIT    ZRO
        LDA     FNUMO
        STA     LITF
LITW    MIN     LIT
        LDA*    LIT
        ADM     CHNO
        STA     LIT1
        CAB
        MIN     LIT
        LDA     LIT
        ETR     =77777B
        LDX     LITF
        BRS     34
        LDA     LIT1
        BRM     MOD3
        SUB     =1
        ADM     LIT
        BRR     LIT
LITF    BSS     1
LIT1    BSS     1
$TABT   ZRO
        LDA     *-1
        STA     TAB
        LDA     TELNO
        STA     LITF
        BRU     TABA
$TAB    ZRO
        LDA     FNUMO
        STA     LITF
TABA    LDA     CHNO
        ADD     =10B
        ETR     =7770B
        STA     TAB3
```

```
TAB2    MIN     CHNO
        CLA
        CIO     LITF
        LDA     TAB3
        SKE     CHNO
        BRU     TAB2
        BRR     TAB
TAB3    BSS     1
$WRIW   NOP
        LDA     =RING
        STA     WRI1
NLIN    BRM     CRLFT
        LDA     BUFNO
        ADD     =10
        CAX
WRCK    LDA     WRI1
        SUB     MXIW
        SKG     =0
        BRU     *+2
        BRS     EXIT
        LDA*    WRI1
        CIO     TELNO
        MIN     WRI1
        BRX     NLIN
        BRU     WRCK
WRI1    BSS     1
$INITL  ZRO
AGAIN   BRM     CRLFT
        BRM     LITT
        DATA    7
        ASC     'INPUT: '
        CLEAR
        BRS     15
        BRU     AGAIN
        STA     FNUMI
        CBA
        SKE     =16B
        BRU     *+2
        BRU     AGAIN2
        LDA     FNUMI
        BRS     20
        BRU     AGAIN
AGAIN2  BRM     CRLFT
        BRM     LITT
        DATA    8
        ASC     'OUTPUT: '
        CLEAR
        LDA     =03000000B
        BRS     16
        BRU     AGAIN2
        STA     FNUMO
        STA     XFNUMO
        CBA
        SKE     =16B
```

```
        BRU     *+2
        BRU     *+4
        LDA     FNUMO
        BRS     20
        BRU     AGAIN2
        BRM     CRLFT
        BRR     INITL
$FNUMO BSS      1
$FNUMI BSS      1
$XFNUMO BSS     1
$TELNO DATA     1
$CHNO   DATA    1
$TCHNO DATA     1
* A=UNPACKED POINTER, B=PACKED, X=LENGTH
PACK  ZRO
        STA     UPP
        STB     PP
        STX     PLEN
PK1     BRM     SKOK
        BRR     PACK
        LDA*    UPP
        MIN     UPP
        STA     PX
        BRM     SKOK
        BRU     PKR1
        LDB*    UPP
        MIN     UPP
        LSH     16
        LDA     PX
        LSH     8
        STA     PX
        BRM     SKOK
        BRU     PKR2
        LDB*    UPP
        MIN     UPP
        LSH     16
        LDA     PX
        LSH     8
        STA*    PP
        MIN     PP
        BRU     PK1
*
SKOK    ZRO
        SKR     PLEN
        MIN     SKOK
        BRR     SKOK
*
PKR1    LDA     PX
        CLB
        LSH     16
        STA*    PP
        BRR     PACK
PKR2    LDA     PX
        CLB
```

```
        LSH     8
        STA*    PP
        BRR     PACK
*
UPACK  ZRO
        STA     UPP
        STB     PP
        STX     PLEN
        SKR     PLEN
        BRU     *+2
        BRR     UPACK
PK2     LDA*    PP
        RSH     16
        BRM     PST
        RSH     8
        BRM     PST
        BRM     PST
        MIN     PP
        BRU     PK2
*
PST     ZRO
        ETR     =377B
        STA*    UPP
        MIN     UPP
        LDA*    PP
        SKR     PLEN
        BRR     PST
        BRR     UPACK
*
PX      BSS     1
UPP     BSS     1
PP      BSS     1
PLEN    BSS     1
*
SKSE   ZRO
        STA     PP
        STB     UPP
        STX     PLEN
SKS1    SKR     PLEN
        BRU     *+2
        BRU     SKST
        LDA*    PP
        SKE*    UPP
        BRR     SKSE
        MIN     UPP
        MIN     PP
        BRU     SKS1
SKST    MIN     SKSE
        BRR     SKSE
*
$MOD3    ZRO
        SUB     =1
        RSH     23
        CLA
```

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

```
        BRX     ERRF
        LDA     IBUF,2
        ADM     ERRNO
        BRU     ERRY
ERRF    BRM     CRLFT
        CLA
        BRU     *+2
        CIO     TELNO
        SKR     ERRX
        BRU     *-2
        LDA     ARROW
        CIO     TELNO
        BRM     CRLFT
        BRR     ERR
*
ERRNO   BSS     1
ERRX    BSS     1
ARROW   DATA    76B
$SERR   NOP
        STA     SE1
        LDA     =SEM
        LDB     =13
        BRU     SERR1
$CERR   NOP
        STA     SE1
        LDA     =CEM
        LDB     =15
SERR1   LDX     TELNO
        BRS     34
        LDA     SE1
        LDB     =10
        LDX     TELNO
        BRS     36
        BRM     CRLFT
        BRS     EXIT
SEM     ASC     'SYSTEM ERROR '
CEM     ASC     'COMPILER ERROR '
SE1     BSS     1
*
RSIZ  DATA     256
MRSIZ DATA     -256
MODRSZ         DATA           377B
$MFLAG BSS      1
CMNT  DATA     5
$LINCNT         DATA          0
*
$WOUT   ZRO
        LDB     =10
        LDX     TELNO
        BRS     36
        LDA     =14B
        CIO     TELNO
        CLA
        CIO     TELNO
```

```
        BRR     WOUT
*
$TST    ZRO
        CLA
        STA     MFLAG
        MIN     TST
        BRM     WPREP
        BRM     INCS
        LDA*    TST
        SKG     RSIZ
        BRU     *+3
        LDA     =4
        BRU     SERR
        STA     TST2
        BRM     MOD3
        LDB     TST
        ADM     TST
        CBA
        ADD     =1
        CAB
        LDA     =STEST
        LDX     TST2
        STA     TST1
        BRM     UPACK
        SKR     TST2
        BRU     TSTS1
        BRR     TST
TSTS    BRM     INC
        MIN     TST1
TSTS1   LDA*    TST1
        SKE*    IWP
        BRR     TST
        SKR     TST2
        BRU     TSTS
        LDA     MCCP
        ADD     BACK
        STA     NCCP
        LDA     =1
        STA     MFLAG
        BRR     TST
TST1    BSS     1
TST2    BSS     1
*
*
$SR     ZRO
        CLA
        STA     MFLAG
        BRM     WPREP
        BRM     INCS
        BRM     CHER
        BRU     STT1,2
STR1    BRM     CIC
        BRM     INC
        BRM     CHER
```

```
        BRU     STT2,2
STR2    BRM     GOBL
        MIN     NCCP
        LDA     =1
        STA     MFLAG
        BRM     STORE
        BRR     SR
*
STT1    BRU     STER
        BRR     SR
        BRR     SR
        BRR     SR
        BRU     STR1+1              DON'T COPY QUOTE
        BRR     SR
STT2    BRU     STER
        BRU     STR1
        BRU     STR1
        BRU     STR1
        BRU     STR2
        BRU     STR1
*
STER    LDA     =7
        BRU     SERR
*
$NUM    ZRO
        CLA
        STA     MFLAG
        BRM     WPREP
        BRM     INCS
        BRM     CHER
        BRU     NT1,2
NM1     BRM     CIC
        BRM     INC
        BRM     CHER
        BRU     NT2,2
NMF     LDA     =1
        STA     MFLAG
        BRM     GOBL
        BRM     STORE
        BRR     NUM
*
NT1     BRU     STER
        BRR     NUM
        BRR     NUM
        BRU     NM1
        BRR     NUM
        BRR     NUM
NT2     BRU     STER
        BRU     NMF
        BRU     NMF
        BRU     NM1
        BRU     NMF
        BRU     NMF
$LET    ZRO
```

```
         CLA
         STA      MFLAG
         BRM      WPREP
         BRM      INCS
         BRM      CHER
         BRU      LET1,2
LET2     BRM      CIC
         LDA      =1
         STA      MFLAG
         BRM      GOBL
         MIN      NCCP
         BRM      STORE
         BRR      LET
LET1     BRU      STER
         BRR      LET
         BRU      LET2
         BRR      LET
         BRR      LET
         BRR      LET
*
*
*
$FINISH NOP
         LDA      =137B
         CIO      FNUMO
         CIO      FNUMO
         CIO      FNUMO
         CIO      FNUMO
         CIO      FNUMO
         LDA      FNUMO
         BRS      20
         BRU      LIMITS
*
$TCH     ZRO
         STA      TCH1
         CLA
         STA      MFLAG
         BRM      WPREP
         BRM      INCS
         LDA*     IWP
         SKE      TCH1
         BRR      TCH
         MIN      MFLAG
         LDA      MCCP
         ADD      BACK
         STA      NCCP
         BRR      TCH
TCH1     BSS      1
*
TOP      MACRO    D
         LDA      D(1).SPT
         STA      D(1).SP
         LDA      =-1
         ADM      D(1).SP
```

```
        LDA*    D(1).SP
        SKE     =0
        BRU     *+2
        BRU     *-5
        LDA     D(1).SP
        SUB     =.D(1).STK
        SKG     =0
        CLA
        BRM     WOUT
        ENDM
*
$LIMITS BRM     CRLFT
        BRM     LITT
        DATA    5
        ASC     'USED '
        TOP     K
        TOP     M
        TOP     N
        LDA     SSL
        SUB     =SS
        BRM     WOUT
        BRM     CRLFT
        BRS     EXIT
        END
```

Procedure return

```
%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];


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

ST = IFST / WHILEST / FORST / GOST / IOST / BLOCK / casest
       .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];
Casest = ".case"
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 /       ".ask" .SR
               "="  SUM :EQ /
               '#   SUM :NE ) [2] / .EMPTY);
```

```
SUM = TERM (('+ SUM :ADD/ '- SUM :SUB)[2]/ .EMPTY);
                                            .mod
TERM = FACTOR (('* TERM :MULT/'/ TERM :DIVID/'↑ TERM :REM)[2]/.EMPTY);

FACTOR = '- FACTOR :MINUS[1] / '+ FACTOR / PRIMARY;  '↑ factor: EXP[2]

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",↑W\ ,"END"\;

VAR[.ID] => *1;

CON[.NUM] => '= *1;

LE / => .EMPTY;

LT / => .EMPTY;

EQ / => .EMPTY;

GE / => .EMPTY;

GT / => .EMPTY;

NE / => .EMPTY;

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

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

OPNOUT[-,-] => ,"CLEAR; BRS 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
```

```
FOR [1, 2, 3, 4, 5]  =>   STORE [*1, *2]
                          WORK [*3]
                          WORK [*4]
                    #1 , "LDA", *1
                          LDB    =4B7
                          SKM    "T-14" .W
                          BRU    *+3
                          SKA    =-1
                          BRU    #2

                          #1 #5

                          LDA    "T+" .W
                          ADM    *1
                          BRU    #1

                    #2    EQU    *
```

ARPAS = '! .SR :ARP[1]

CASE = ".CASE" EXP ".OF" CASES :CS1[2] ;

CASES = ".BEGIN" ST $(';' ST :DQ[2]) ".END" / ST ;

```
CS1[-,-] =>    ACC[*1]
               CNA
               CAX
               CS2[*2,#2]
            #2 EQU *

CS2[DQ[-,-],#2] =>  ,BRX #1
                     *1:*1
                     BRU #2
                  #1 EQU *

   [-, - ] =>  *1
```

≤ 0

> max

CCALL
CRET

```
:CACMPL,  06/25/69  1254:27  JFR ; .SCR=1; .PLO=1; .MCH=75; .RTJ=0; .DSN=1;
.LSP=0; .MIN=70; .INS=3;
%.HED="CONTENT-ANALYZER SYNTAX DEFINITION";%
   %declarations% file camcl(orgcac)
      (+pttrnx) !" data 0; sbrm cac;1cacdx data cacode"  :x
      .content-analyzer-syntax
      .opcodes arb., bf., bt., bru., bfs., brc., brr., ccp., ldp., ln.,
      pdc., pps., sap., scv., pcp, lc, tst, tstf, fcp, rst, kset, sd.,
      cpf., ccv., scp,snc,bfr,ieq,ine,lda.,ldb.;
      .literals ccsp, cctab, ccld, ccl, ccd, cch, ccnp, ccpt, cccr, bggap,
      bglg, bgnlg, bgpts, bgnps, bgls, bgds, bglds, dirl, dirr, cpflf,
      cpfle, cpfsf, cpfse,  rellt, relle, releq, relge, relne, p0;
      pttrn
   %pattern definition%
      pttrn = pexp '; [brr cacdx] ;
      pexp =
         "if" posrl "then" [bf #1] pexp "else" [bru #2] :1 pexp :2 /
         .empty [pcp;kset] union [pps =2];
      posrl =
         pos [fcp] (".lt" pos [ccp =rellt] /
         ".le" pos [ccp =relle] /
         ".eq" pos [ccp =releq] /
         ".ge" pos [ccp =relge] /
         ".gt" pos [ccp =relne]);
   %logical expression%
      union = inter (
         "or" [bt #1; scp] union :1 /
         .empty);
      inter = neg (
         "and" [bf #1; scp] inter :1 /
         .empty);
      neg =
         "not" negneg /
         alter;
      negneg =
         "not" neg /
         alter [lc];
   %alternation & concatanation%
      alter = concat (
         '/ [bt #1; scp] alter :1 /
         .empty);
      concat = $(ele [bf #1] / nele) :1;
   %basic recognizers%
      ele = .sr [tst; *s] /
         cv /
         '( [pcp] pexp ') [pps =2] /
         arbno /
         '- ele [lc] /
         lmtskp /
         ".initials" (
            '= .id [lda =*i; ieq] /
            '# .id [lda =*i; ine]) /
         ".since" date [snc] /
```

```
        ".before" date [bfr] /
        unach;
    date = '( .num .call cacdt1 '/ .num .call cacdt2 '/ .num
        .call cacdt3 [lda =*n] .num .call cacdt1
        ': .num .call cacdt2 [ldb =*n] ');
    cv = scvs / ccvs;
    ccvs =
        "sp" [ccv =ccsp] /
        "tab" [ccv =cctab] /
        "ld" [ccv =ccld] /
        "l" [ccv =ccl] /
        "d" [ccv =ccd] /
        "ch" [ccv =cch] /
        "np" [ccv =ccnp] /
        "pt" [ccv =ccpt] /
        "cr" [ccv =cccr];
    scvs =
        "gap" [scv =bggap] /
        "lgap" [scv =bglg] /
        "nlgap" [scv =bgnlg] /
        "pts" [scv =bgpts] /
        "nps" [scv =bgnps] /
        "ls" [scv =bgls] /
        "ds" [scv =bgds] /
        "lds" [scv =bglds];
    nele =
        ".empty" [kset] /
        spceft;
    elles = ele / nele;
  %interative phrases%
    arbno =
        .num [ln =*n] '$ (
            .num [ln =*n] /
            .empty [ln =1000]) arbe /
        '$ (
            .num [ln =0; ln=*n] arbe /
            .empty :1 elles [bt #1; kset]);
    arbe = [ln =0] :1 elles [arb #1; pps =3];
    spceft =
        '< [sd =dirl] /
        '> [sd =dirr] /
        '| 'p .num .pchk [sap p0 *n *n] /
        '← 'p .num .pchk [pdc p0 *n *n] /
        pos [fcp];
    lmtskp = "within" .num "find" [ln =*n; pcp] :1 [rst] union skprt [brc
    #1; pps =3];
    skprt = "skip" [bt #1 1; kset] union :1;
    unach = '[ [pcp] :1 (
        .sr (
            '] [tstf; *s; pps =2] /
            [tst; *s; bf #2] union :2 [bfs #1; pps =2] '] )  /
```

```
        union [bfs #1; pps =2] '] );
  %position setting%
     pos = fcnpos / pntr;
     pntr = 'p .num .pchk [ldp p0 *n *n];
     fcnpos =
        "c(" pos ') /
        "sf(" pos ') [cpf =cpfsf] /
        "se(" pos ') [cpf =cpfse];
  .end : end of cac
```

```
%.HED="CONTENT-ANALYZER-COMPILER LIBRARY";%
   %initialization %
      (cac) procedure;
         %dummy declares%
         prefix for temporaries: 'clt';
         declare cacicv, %instruction counter value%
         cacgl1, %gen label 1%
         cacgl2, %gen label 2%
         cacnmv, %number value-last one recognized%
         cacnnl, %negative number of literals%
         cacltp, %literal pointer-into code array%
         cacnml, %number recognizer temporary%
         cstk, %the order of these is important%
         cstkd[200], cstkt, cstk1=cstk, carst1, carst2, carst3, %for
         cacrst%
         cactmp, %a good ole temporay%
         cacntp, %another%
         cacsrx=30, %max string length%
         cacsrl[11];
         cstk←cstk1;
         cstkt←$cstkt-1;
         cacltp ← $cacdnd-2;
         cacnnl ← -1;
         for cacicv from $cacode inc 1 to $cacdnd do [cacicv]←0;
         cacicv ← $cacode+1;
         cacgl1, cacgl2 ← 0;
         scndir←1;
         < ldp spskd; stp swork>;
         call fechc1(1,,$swork);
         < brm* cazc>;
         bump cacode;
         (caret0):
            call cacrst;
            < any gps>;
         (cart11): < abort 11>;
         (cart12): < abort 12>;
            null endp.
      (cacrst) procedure;
         frozen rsvstl, rsvst;
         for carst1 from 0 inc 1 to rsvstl do if rsvst[carst1] .ne 0 then
         begin
            call storsv(.lsh(carst1,0)8);
            carst2←.xr;
            for carst3 from 0 inc 1 to 253 do begin
               [carst2]←[carst2] .a 57777777b;
               carst2←+ 4 end end;
         return endp.
   %generated label generation & definition%
      (cacgn1) procedure;
         call cacgn(,,$cacgl1);
         return endp.
```

```
    (cacgn2) procedure;
       call cacgn(,,$cacgl2);
       return endp.
    (cacgn) procedure;
       if $0[.xr] .ncb 40000000b then begin
          .br ← cacicv;
          $0[.xr] ← .br end
       else .ar ← -.ar;
       [cacicv]←+.ar;
       return endp.
    (cacdf1) procedure;
       cacgl1 ← cacdf(,cacgl1);
       return endp.
    (cacdf2) procedure;
       cacgl2 ← cacdf(,cacgl2);
       return endp.
    (cacdf) procedure;
       if .br .ncb 40000000b then begin
          while .br .cb 37777b do begin
             .xr ← .br;
             .ar, .br ← $0[.xr];
             $0[.xr] ← (.ar .a 77740000b) .v cacicv end;
          return (-cacicv) end
       else call rerror endp.
 %recursive call & return%
    (caccll) procedure;
       call cpush(,$37777b[([caccll])]);
       call cpush(,cacgl1);
       call cpush(,cacgl2);
       cacgl1,cacgl2←0;
       return endp.
    (cacrtn) procedure;
       call cpop;
       cacgl2←.br;
       call cpop;
       cacgl1←.br;
       call cpop;
       .xr←.br;
       go to $1[.xr] endp.
 %stack operators%
    (cpush) procedure;
       if cstk .eq cstkt then goto cart12;
       bump cstk;
       [cstk]←.br;
       return endp.
    (cpop) procedure;
       if cstk .eq cstk1 then call rerror
       else begin
          .br←[cstk];
          cstk ←+ -1 end;
       return endp.
```

```
%basic recognizers%
   %main routines%
      (cacidr) procedure;
         call cacdeb;
         cacsrl← -1;
         < any pcp; ldx =swork; brm 2,2>;
         cactmp←.ar;
         if caclet(cactmp) then begin
            while caclet(cactmp) do begin
               call apchr(cactmp,,$cacsrx);
               .xr←$swork;
               < brm 2,2>;
               cactmp←.ar end;
            < any kset>;
            swork[1]←+ -1;
            call fechcl(1,,$swork) end
         else begin
            flag←0;
            < any scp> end;
         < pps =2>;
         return endp.
      (cacsrr) procedure;
         cacsrl ← -1;
         call cacdeb;
         < any pcp; any kset>;
         if nxtchr() .eq 7 then call apchr(nxtchr(),,$cacsrx)
         else if .ar .eq 2 then while nxtchr() .ne 2 do call
         apchr(.ar,,$cacsrx)
         else begin
            flag←0;
            < any scp> end;
         < pps =2>;
         return endp.
      (cacnmr) procedure;
         cacnmv ← 0;
         call cacdeb;
         < any pcp; ldx =swork; brm 2,2>;
         if .ar .gt 17b and .ar .le 31b then begin
            (cacnml):
            cacnml ← .ar;
            cacnmv ← cacnmv*10+cacnml-20b;
            < brm 2,2>;
            if .ar .gt 17b and .ar .le 31b then go to cacnml;
            < any kset>;
            swork[1]←+ -1;
            call fechcl(1,,$swork) end
         else begin
            flag←0;
            < any scp> end;
         < pps =2>;
         return endp.
```

```
    %uility routines%
       (caclet)procedure;
          return( if (.ar .gt 40b and .ar .le 72b) or (.ar .gt 100b and
          .ar .le 132b) then 1
             else 0) endp.
       (cacdeb) procedure;
          < any tst; data 0; asc ' '; bt *-3>;
          return endp.
       (capchk) procedure;
          if cacnmv .le 0 or cacnmv .gt 9 then go to cart12
          else return endp.
       (nxtchr) procedure;
          .xr←$swork;
          < brm 2,2>;
          return endp.
    %date and time make up%
       (cacdt1) procedure;
          cacntp ← .lsh(cacnmv,0)16;
          return endp.
       (cacdt2) procedure;
          cacnmv,cacntp← .lsh(cacnmv,0)8 .v cacntp;
          return endp.
       (cacdt3) procedure;
          cacnmv ← cacntp .v cacnmv;
          return endp.
 %code generators%
    (cacidc) procedure;
       return(cacsrl[1]) endp.
    (cacsrc) procedure;
       [cacicv] ← cacsrl;
       for cactmp from 0 inc 1 to cacsrl/3 do begin
          call cacicc;
          [cacicv] ← cacsrl[cactmp+1] end;
       return endp.
    (cacicc) procedure;
       bump cacicv;
       if cacicv .ge cacltp then call rerror;
       return endp.
    (cacltr) procedure;
       .xr ← cacnnl;
       < ske cacdnd,2; brx *-1>;
       if .ar .eq cacdnd[.xr] then .ar ← .xr + $cacdnd
       else begin
          [cacltp] ← .ar;
          .br ← cacltp;
          cacltp ←+ -1;
          cacnnl ←+ .ar;
          if cacltp .lt cacicv then go to cart11;
          .ar ← .br end;
       [cacicv]←+.ar;
       return endp.
```

```
%patch space%
    (cacf) procedure;
        return endp.
finish
```

```
:TXTEDT,  06/25/69  1257:47  JFR  ;  TEXT  EDITOR  .SCR=1; .PLO=1; .MCH=75;
.RTJ=0; .DSN=1; .LSP=0; .MIN=70; .INS=3;
% .HED="TXTEDT, SPL CODE"; .RES;
```

```
%
   file txtedt(orgtxt)
   %odds and ends%
      (+recred) display() goto [s]
      (+subs) call subst return
      (+subi) call subint return
   % used by parameter spec to build and break statements %
      (+gdmys) (a1)
         :c st a1 ← ", dummy ;" :
         return
      (+setrot) (a1)
         :p c(a1) > [ ";" ] |p2 :
         :c st a1 ← &fn*, ",", &stn2*, " ;", p2 se(p1): return
      (+sttxt) (d1,d2)
         :c st d2 ← sf(d1) se(d1) :
         return
      (+brkst) (a1,d2)
         :p c(a1) > ch $pt |p1←p1 $np |p2 :
         :c st d2 ← sp,p2 se(a1);
         st a1 ← sf(a1) p1 : return
      (+staptx) (d1,d2)
         :c st d2 ← sf(d2) se(d2),sp,sf(d1) se(d1):
         return
      (+stlit) (d1)
         :c st d1 ← &lit* :
         return
   % entity finding routines - known as delimiters %
      (+cdlim) (a1,a2,a3,a4,a5)
         :p c(a1) < |a2 ch |a4 c(a1) > |a3 ch |a5 :
         return
      (+idr) (a1,a2,a3,a4,a5)
         :p c(a1) < ch $np |a4 |a2←a2 c(a1) > ch $np |a3←a3 |a5 :
         return
      (+tdr) (a1,a2,a3,a4,a5,a6)
         :p if sf(a1) .ne sf(a2) then !" abort 6" else
            if a1 .lt a2 then
               c(a1) |a3 < ch |a5 c(a2) |a4 > ch |a6
               else c(a1) |a4 > ch |a6 c(a2) |a3 < ch |a5 :
         return
      (+ndr) (a1,a2,a3,a4,a5)
         :p c(a1) < $(d / ', / '.) ('$/.empty) ('-/.empty) |a2←a2 |a4
         > ch ('-/.empty) ('$/.empty)
         d (d (d/.empty) /.empty)
         (', d d d $(', d d d) / $d)
         ('. d $d / .empty) |a3←a3
         (sp |a5 /
         |a5 c(a4) < (sp/.empty) |a4) /
         !" abort 2" :
         return
      (+wdr) (a1,a2,a3,a4,a5)
         :p c(a1) >ch $ld |a3←a3
```

```
            (sp |a5 c(a1) <ch $ld |a2←a2 |a4/
            |a5 c(a1) <ch $ld (sp/.empty) |a2←a2 |a4) :
        return
    (+wdr2) (a1,a2,a3,a4,a5)
        :p c(a1)>ch$ld |a3←a3|a5 c(a1)<ch$ld |a2←a2|a4:
        return
    (+vdr) (a1,a2,a3,a4,a5)
        :p c(a1)>ch$pt |a3←a3
            (sp |a5 c(a1) <ch$pt |a2←a2 |a4 /
            |a5 c(a1) <ch $pt (sp/.empty) |a2←a2 |a4) :
        return
    (+vdr2) (a1,a2,a3,a4,a5)
        :p c(a1)>ch$pt |a3←a3|a5 c(a1)<ch$pt |a2←a2|a4:
        return
  % statement reconstruction routines %
    (+deltx) (d1)
        delptr(sf(d1) se(d1))
        :s deltx d1:
        return
    (+cpchtx) (a1,a2,a3,a4,a5)
        :c st a1 ← sf(a1) a2,$a4 a5,a3 se(a1):
        +recred;
    (+cshft)
        :c st b1 ← sf(b1) p3, +p1 p2, p4 se(b1):
        display() return
    (+del)
        :c st b1 ← sf(b1) p3,p4 se(b1):
        delptr(p1 p2) +recred;
    (+mvchtx) (a1,a2,a3,a4,a5,a6,a7,a8,a9,a10)
        :c if sf(a1) .eq sf(a2) then
            if a1 .lt a2 then
                st a1 ← sf(a1) a4,a7 a8,a6 a9,a10 se(a1)
                else st a1 ← sf(a1) a9,a10 a4,a7 a8,a6 se(a1)
            else begin
                st a1 ← sf(a1) a4,a7 a8,a6 se(a1);
                st a2 ← sf(a2) a9,a10 se(a2) end:
        +recred;
    (+mvwdvs) (a1,a2,a3,a4,a5,a6,a7,a8,a10)
        :c if sf(a1) .eq sf(a2) then
            if a1 .lt a2 then
                st a1 ← sf(a1) a4, sp,a7 a8,a6 a9,a10 se(a1)
                else st a1 ← sf(a1) a9,a10 a4, sp,a7 a8,a6 se(a1)
            else begin
                st a1 ← sf(a1) a4, sp,a7 a8,a6 se(a1);
                st a2 ← sf(a2) a9,a10 se(a2) end:
        +recred;
    (+rpl) (a1,a2,a3)
        :c st a1 ← sf(a1) a2, &lit*, a3 se(a1) :
        call rellit +recred;
    (+cpwdvs) (a1,a2,a3,a4,a5)
        :c st a1 ← sf(a1) a2, sp,$a4 a5, a3 se(a1):
```

```
        +recred;
  % control routines called from main control %
     % copy %
        (+qcc) +cdlim[b1,p1-4] +cdlim[b2,p5-8] +cpchtx[b1,p2,p4,p5,p6] ;
        (+qcw) +wdr2[b1,p1-4] +wdr2[b2,p5-8] +cpwdvs[b1,p2,p4,p5,p6] ;
        %(+qcn) +wdr2[b1,p1-4] +ndr[b2,p5-8] +cpwdvs[b1,p2,p4,p5,p6] ;%
        (+qci) +idr[b1,p1-4] +idr[b2,p5-8] +cpchtx[b1,p2,p4,p5,p6] ;
        (+qcv) +vdr2[b1,p1-4] +vdr2[b2,p5-8] +cpwdvs[b1,p2,p4,p5,p6] ;
        (+qct) +cdlim[b1,p1-4] +tdr[b2,b3,p5-8] +cpchtx[b1,p2,p4,p5,p6] ;
     % delete %
        (+qdc) +cdlim[b1,p1-4] +del;
        (+qdw) +wdr[b1,p1-4] +del;
        %(+qdn) +ndr[b1,p1-4] +del;%
        (+qdi) +idr[b1,p1-4] +del;
        (+qdv) +vdr[b1,p1-4] +del;
        (+qdt) +tdr[b1,b2,p1-4] +del;
     % insert %
        (+qic) +cdlim[b1,p1-4] +rpl[b1,p2,p4] ;
        (+qiw) +wdr2[b1,p1-4] +rpl[b1,p2,p4] ;
        %(+qin) +ndr[b1,p1-4] +rpl[b1,p2,p4] ;%
        (+qii) +idr[b1,p1-4] +rpl[b1,p2,p4] ;
        (+qiv) +vdr2[b1,p1-4] +rpl[b1,p2,p4] ;
        (+qit) +cdlim[b1,p1-4] +rpl[b1,p2,p4] ;
     % move %
        (+qmc) +cdlim[b1,p1-4] +cdlim[b2,p5-8] +mvchtx[b1,b2,p1-8] ;
        (+qmw) +wdr2[b1,p1-4] +wdr[b2,p5-8] +mvwdvs[b1,b2,p1-8] ;
        %(+qmn) +ndr[b1,p1-4] +ndr[b2,p5-8] +mvwdvs[b1,b2,p1-8] ;%
        (+qmi) +idr[b1,p1-4] +idr[b2,p5-8] +mvchtx[b1,b2,p1-8] ;
        (+qmv) +vdr2[b1,p1-4] +vdr[b2,p5-8] +mvwdvs[b1,b2,p1-8] ;
        (+qmt) +cdlim[b1,p1-4] +tdr[b2,b3,p5-8] +mvchtx[b1,b2,p1-8] ;
     % replace %
        (+qrc) +cdlim[b1,p1-4] +rpl[b1,p3,p4] ;
        (+qrw) +wdr2[b1,p1-4] +rpl[b1,p3,p4] ;
        %(+qrn) +ndr[b1,p1-4] +rpl[b1,p3,p4] ;%
        (+qri) +idr[b1,p1-4] +rpl[b1,p3,p4] ;
        (+qrv) +vdr2[b1,p1-4] +rpl[b1,p3,p4] ;
        (+qrt) +tdr[b1,b2,p1-4] +rpl[b1,p3,p4] ;
     % shift case %
        (+qsc) +cdlim[b1,p1-4] +cshft return
        (+qsw) +wdr2[b1,p1-4] +cshft return
        (+qsi) +idr[b1,p1-4] +cshft return
        (+qsv) +vdr2[b1,p1-4] +cshft return
        (+qst) +tdr[b1,b2,p1-4] +cshft return
        (+qss) :c st b1 ← +sf(b1) se(b1) : display() return
     %pointer specification%
        (+qpf) ptrfix(b1) call rellit +recred;
        (+qprs) getrf delptr(sf(b1) se(b1)) +recred;
        (+qprt) +tdr[b1,b2,p1-4] delptr(p1 p2) +recred;
        (+qprw) +wdr[b1,p1-4] delptr(p1 p2) +recred;
  end of txtedt
```

```
% .HED="TXTEDT, MOL CODE"; .RES;
```

```
% < NOLIST>;
   (txgd) procedure;
      prefix for generated labels: 'txl';
      prefix for temporaries: 'tet';
      call rerror endp.
   %.HED="txtedt, PONTER  FIXUP ROUTINES"; .RES;
```

```
%
    (fixptr) procedure(fxptr1,,fxptr2);
       frozen rplsid;
       frozen
          fxptr1, fxptr2, fxptr3, fxptr4,
          fxptr5, fxptr6, fxptr7, fxptr8;
       frozen ptrtb, ptrtbl;
       fxptr3 ← ncis(,,fxptr2)-$1[fxptr1]+1;
       fxptr4←fxptr1+1;
       fxptr5←fxptr1+3;
       fxptr6←0;
       while fxptr6 .lt ptrtbl do begin
          .xr ← fxptr6;
          fxptr6 ←+ 3;
          if
             [fxptr1] .eq ptrtb[.xr+1] .a 17777777b
             and [fxptr4] .le ptrtb[.xr+2]
             and [fxptr5] .ge ptrtb[.xr+2]
             and ptrtb[.xr+1] .a 20000000b .ne 0
             then ptrtb[.xr+1]←ptrtb[.xr+1] .v 40000000b end;
       fxptr7 ← 1;
       while fxptr7 do begin
          fxptr6, fxptr7 ← 0;
          while fxptr6 .lt ptrtbl and fxptr7 .eq 0 do begin
             if ptrtb[fxptr6+1] .lt 0 then begin
                fxptr7 ← 1;
                fxptr4 ← ptrtb[fxptr6];
                fxptr5 ← ptrtb[fxptr6+2]+fxptr3;
                for fxptr8 from fxptr6 inc 1 to ptrtbl do
                   ptrtb[fxptr8]←ptrtb[fxptr8+3];
                ptrtbl←+ -3;
                call insptr(rplsid,fxptr5,fxptr4) end;
             fxptr6 ←+ 3 end end;
       return(fxptr1,,fxptr2) endp.
    (delptr) procedure (delpt1);
       % pointer to table %
       frozen delpt1, delpt2, delpt3, delpt4;
       frozen ptrtb, ptrtbl;
       delpt2 ← delpt1+1;
       delpt3 ← 0;
       while
          delpt3 .lt ptrtbl
          and (
             [delpt1] .gt ptrtb[delpt3+1] .a 37777777b
             or (
                [delpt1] .eq ptrtb[delpt3+1] .a 37777777b
                and [delpt2] .gt ptrtb[delpt3+2]))
          do delpt3 ←+ 3;
       delpt4 ← delpt3;
       delpt2 ←+ 2;
       while
```

```
            delpt4 .lt ptrtbl
            and [delpt1] .eq ptrtb[delpt4+1] .a 37777777b
            and [delpt2] .ge ptrtb[delpt4+2]
            do delpt4 ←+ 3;
         if delpt3 .eq delpt4 then return;
         ptrtbl ← ptrtbl+delpt3-delpt4;
         for delpt1 from delpt3 inc 1 to ptrtbl do begin
            ptrtb[delpt1] ← ptrtb[delpt4];
            bump delpt4 end;
         return endp.
      (insptr) procedure(inspt1,inspt2,inspt3);
         frozen inspt1, inspt2, inspt3, inspt4, inspt5;
         frozen ptrtbl, ptrtxn, ptrtb;
         if ptrtbl .ge ptrtxn do-single
            < err 6>;
         for inspt4 from 0 inc 3 to ptrtbl do
            if ptrtb[inspt4] .eq inspt3 then return;
         bump [insptr];
         inspt4←0;
         while
            inspt4 .lt ptrtbl
            and (
               inspt1 .gt ptrtb[inspt4+1] .a 17777777b
               or (
                  inspt1 .eq ptrtb[inspt4+1] .a 17777777b
                  and inspt2 .gt ptrtb[inspt4+2]))
            do inspt4 ←+ 3;
         for inspt5 from ptrtbl dec 1 to inspt4
            do ptrtb[inspt5+3] ← ptrtb[inspt5];
         ptrtbl ←+ 3;
         .xr ← inspt4;
         ptrtb[.xr] ← inspt3;
         ptrtb[.xr+1] ← inspt1;
         ptrtb[.xr+2] ← inspt2;
         return endp.
   %.HED="txtedt, TEXT BUILDING"; .RES;
```

```
%
    (apachr) procedure;
       % character in b, work area address in x %
       null;
       < lsh 16 >;
       < lda* 1,2 >;
       < lsh 8 >;
       < sta* 1,2 >;
       < skr 2,2 >;
       < brr* apachr >;
       < lda =2 >;
       < sta 2,2 >;
       < lda 1,2 >;
       < min 1,2 >;
       < eor 1,2 >;
       < etr =77776000b >;
       < skg =0 >;
       < brr* apachr >;
       < sbrm gcol >;
       < brr* apachr > endp.
    (apastr) procedure (apasr1,,apasr4);
       % a-string in a, work area in x %
       frozen apasr1, apasr2, apasr3, apasr4;
       apasr3 ← $1[apasr1];
       for apasr2 from 0 inc 1 to apasr3 do
          call apachr(.ar,ldchr(apasr1,apasr2),apasr4);
       return endp.
    (aptstr) procedure (apts1,,apts2);
       % 1-ptr to 2 t-pointers, 2-ptr to cwork, 3-temp %
       % 6-case set, 7-option set, 8-option mask, 9-option set %
       frozen apts1, apts2, apts3, apts6, apts7, apts8, apts9, swork;
       .xr←.ar;
       if $0[.xr] .ne $2[.xr] then call rerror;
       swkflg ← -1;
       % turn on gcol %
       apts10 ← getsdb($0[.xr]);
       call lodsdb(.ar);
       apts10 ← sdbst[(.rsh(apts10)9)];
       call frzrfb(.ar,1);
       if apts7 then % mode set now in operation %
       begin
          swork[0]←[apts1];
          bump apts1;
          swork[1]←[apts1];
          apts1←+2;
          if swork[1] .gt 1 then begin
             swork[1]←+ -1;
             call fechc1(1,,$swork);
             < brm 2,2>;
             if .ar .ne endchr and .ar .cb 200b then goto aptsl end
          else call fechc1(1,,$swork);
```

```
            while swork[1] .le [apts1] do begin
               .xr←$swork;
               < brm 2,2>;
               apts3←.ar;
               if .ar .cb 200b then begin
                  (aptsl):
                  .ar←(.ar .a apts8) .v apts9;
                  if .ar .ne 200b then call apachr(,.ar,apts2);
                  .xr←$swork;
                  < brm 2,2>;
                  apts3←.ar end
               else  begin
                  .ar←apts9;
                  if .ar .ne 200b then call apachr(,.ar,apts2) end;
               case apts6 of begin
                  call apachr(
                     ,
                     if apts3 .gt 100b and .ar .le 132b then .ar-40b else
                     .ar, apts2
                     );
                  call apachr(
                     ,
                     if apts3 .gt 40b and .ar .le 72b then .ar+40b else
                     .ar,apts2
                     );
                 ll apachr(,apts3,apts2) end end;
            apts7←0 end
         else begin
            swork[0]←[apts1];
            bump apts1;
            swork[1]←[apts1];
            apts1←+2;
            if swork[1] .gt 1 then begin
               swork[1]←+ -1;
               call fechc1(1,,$swork);
               < brm 2,2>;
               if .ar .cb 200b then call apachr(,.ar,apts2) end
            else call fechc1(1,,$swork);
            while swork[1] .le [apts1] do begin
               .xr←$swork;
               < brm 2,2>;
               call apachr(,.ar,apts2) end end;
         swkflg ← 0;
         call frzrfb(apts10,-1);
         return endp.
   %.HED="TXTEDT, POPS"; .RES;
```

```
%
   (cchr) pop(13300000b,1,0) procedure;
      return(apachr(,[$0],$cwork)) endp.
   (anypop) pop(17700000b,1,0) proc;
      external bsc, esc, kps, kpr,dlp,cpw,cpx,pfx% %
      goto [$0];
      (bsc):
         rplsid←.ar;
         return(newsdb());
      (esc):
         call fresdb(rplsid);
         return(endsdb(rplsid));
      (cpp):
         call fixptr($sptr1,,$cwork);
         return(aptstr($sptr1,,$cwork));
      (kps):
         call apastr($0 .a 37777b,,$cwork);
         go to strrtn;
      (kpr):
         call regadr(smareg);
         return(apastr(.xr,,$cwork));
      (dlp):
         return(delptr($sptr1));
      (cpw):
         return(aptstr($sptr1,,$cwork));
      (cpx):
         call delptr($sptr1);
         call aptstr($sptr1,,$cwork);
         return;
      (pfx):
         call insptr(,,$2[litlc]);
         < abort 4>;
         return endp.
finish
```

NOTE: 02

This document represents the decisions reached by a combined Command Language / Novice-expert review groups meeting held the mornings of 29-May and 30-May. 03

With the NLS Utility will come a substantial increase in our NLS user community. These, in general, are users which we would like to please. Consequently, I herein propose modifications to the command language which I (chi) feel make it simpler, more consistent and somewhat more complete (although, I think it has a way to go before we can call it complete). I have discussed most of these issues with the "novice user" and "command language" groups. I think these changes should be in the running system BEFORE we begin training these new users (say by mid JUNE)! It is quite difficult to learn a command language which is changing while you are learning it! 04

The following changes should be made in the command language: 05

The command language should be made to consist of an editor and special purpose subsystems. The command language for each should consist of frequently used commands, which are recognized by their first letter (unless preceded by a space) and infrequently used, new, or experimental commands, which are preceded by a SPACE and which are recognized when the user types enough characters. There should be a consistent VERB-NOWN form to commands and verbs should be used in a consistent manner. 06

This allows commands to be reasonably named and added without worrying about first letter conflicts while "protecting" frequently used commands -- significant problems currently. 07

This should apply to operand types also, of course, but may not be widely used at first. 08

In "novice" mode, a system supplied SPACE will preceed each command the user gives. 09

Subsystem names should be recognized when the user has typed enough characters for uniqueness. All subsystems should terminate with the "Quit" command, as should NLS. All subsystems should have Execute, Goto, and help commands. 010

The concept of Address Expression should be generalized for DNLS, TNLS, and DEX such that wherever a statement name or number is currently used, an appropriate ADDRESS EXPRESSION should be allowed (see Appendix C). 011

For DNLS, a selection should be defined as 012

SEL = (BUG / OPTION DAE CA); 013

Note that the use of markers in DNLS by holding the rightmost mouse button down and typing the marker name should be eliminated, since one will be able to type OPTION and an arbitrary DAE followed by a CA. 014

In TNLS a selection should be defined as 015

SEL = DAE CA; 016

Note the syntactic conflict inherant in SEL / LIT alternatives in commands. This can be avoided by converting any such choice points in TNLS to LIT / OPTION SEL. 017

In DEX a selection is defined as 018

SEL = STAE CA; 019

Where DAE (Dynamic Address Expression) is defined in Appendix C, and STAE (STatic file Address Expression) is defined in the DEX-II design document. 020

A Dynamic Address Expression should be consistent with existing links, the same DAE should work in TNLS and in DNLS, and the elements of the expression should be reasonably mnemonic. A DAE should be available in NLS wherever a statement number, SID, or statement name is now used (as in links, jumps, etc.). 021

The replacement of the statement name/number field by a DAE provides a powerful extention to the link syntax and will be compatible with extant links. 022

Editing command changes 023

Terminating any editing command with INSERT or REPEAT (what used to be called CDOT, for historical reasons) is shorthand for Command Accept (CA) followed by the INSERT or REPEAT commands, respectively. 024

The INSERT command allows one to quickly insert a new statement after the CM. 025

The REPEAT command allows one to repeat the last used editing command, perhaps defaulting one of the operands to the Control Marker (CM) instead of asking the user to select it. 026

The notion of operand-type defaulting in DNLS should be eliminated. 027

Substitute changes 028

For the new form of the substitute command, see Appendix B. 029

An elipses capability should be made available in the substitute command. That is, text...text should be allowed for the specification of text to be replaced. This should result in instances of <text1 arbitrary-text text2> being replaced in the substitute. 030

This is subject to the constraint that the <text1...text2> be in one statement, and that an occurance of text1 can only be paired with the first occurance of text2 following text1. 031

If text1 is null then assume String Front (SF); if text2 is null then assume String End (SE). 032

If one wishes to actually substitute for a string of three periods (...), one must preceed each of the three periods by Literal Escape (LE). 033

Jump command changes 034

SP command in TNLS will be replaced by Jump to. 035

The Jump commands should be made to be like the rest -- no state of its own. 036

Jump to Successor, Jump to Predecessor should require one to type 'J 'S 'J 'P just as Insert Character, Insert Word requires one to type 'I 'C 'I 'W. 037

The sub commands of Jump to End should be deleted. 038

The Jump File Link command should be removed since Jump to Link is equivalent. 039

The order of operand selection in Move, Copy, Append, Assimilate, and Substitute should be changed. Please see the command syntax in Appendix B. Basically, the move/copy/assimilate should be of the form "Move/Copy/Assimilate This to There", rather than its current form "Move/Copy/Assimilate to There, This". Append should be "Append this to that", instead of "Append to that, this". 040

Note that this could be considered as being inconsistent with the insert command, although this inconsistency might prove minor and unimportant. 041

A more important problem is that the first operand selection in a command (given our current approach for switching commands) must allow us to differentiate between a keyword and a LIT. Consequently, we would not be able to allow move/copy/append to have LIT as a possible first operand. 042

A new control character "OPTION" should be available for specifying optional arguments to commands. 043

In addition, I propose that there be a FILTER command for content filters, a RESEQUENCE SID's command, and a COMPACT FILE command (doing away with Output File). Please refer to Appendix B for details. 044

Appendix A: Proposed Command Language (summary) 045

Commands which must be preceded by SP are preceded by SP in this list: 046

append 047

allow private modifications to file (old browse mode) 048

assemble program 049

assimilate 050

arm NLSDDT ↑H 051

break 052

copy 053

<copy> file 054

clear window 055

compact file 056

compile program 057

connect (display / tty) 058

create file 059

delete 060

<delete> markers 061

<delete> file 062

disarm NLSDDT ↑H 063

disconnect terminals 064

execute SUBSYSTEM-NAME command 065

edit 066

expert 067

expunge directory 068

goto SUBSYSTEM-NAME 069

help 070

insert 071

<insert> (sequential / assembler) file 072

<insert> Journal submission form 073

~~jump (DNLS and TNLS)~~ 074

k (unused) 075

~~load file FILENAME CA;~~ 076

Look {at file/link/session for [illegible]}

move 077

~~<move> boundary~~ 078

~~<move> file~~ 079

~~mark~~ 080

~~merge~~ {replaced by x-file editing} 081

novice 082

~~output~~ formatted output 083

~~p (unused in DNLS)~~ print (TNLS) 084

~~playback session~~ 085

~~quit [SUBSYSTEM-NAME / NLS]~~ 086

replace 087

~~receive connection from terminal~~ 088

~~record session~~ 089

~~relock file~~ 090

~~resequence SID's in file~~ 091

reset (~~tty-simulation window~~ /
viewspecs) 092

substitute 093

set (case /
character size for window /
filter /
link default for file /
? ~~name delimeters~~ /
~~tty-simulation window~~ /
viewspecs) 094

show (file status /
~~marker list~~ /
? name delimiters /
viewspecs [verbose] status) 095

~~simulate terminal type (display / TI-terminal / etc.)~~ 096 sort/merge ? 097

split window (vertically / horizontally) (DNLS) 098

transpose 099

terminate (recording / private modifications to file) 0100

update file 0101

unlock file 0102

verify file 0103

w (not used) 0104

x (unused) 0105

y (unused) 0106

z (unused) 0107

'. (TNLS) show CM 0108

'/ (TNLS) type context of CM 0109

'; Comment 0110

'\ (TNLS) print statement 0111

'↑ (TNLS) print back statement 0112

linefeed (TNLS) print next statement 0113

INSERT %Insert Statement after Control Marker% 0114

REPEAT last editing command 0115

TAB %to next occurrence of content or word% 0116

'? 0117

Appendix'B: Proposed Command Language (detail) 0118

Definitions 0119

CA = ↑D / CR (default); 0120

%SEL or command terminator% 0121

REPEAT = CDOT = ↑B / ESC/ALTMODE (default); 0122

%Terminate current editing command and begin REPEAT command, possibly defaulting a selection to CM% 0123

INSERT = ↑E / user-settable characters; 0124

%Terminate current editing command and begin INSERT command% 0125

CD = ↑X / DEL/RUBOUT (default); 0126

%abort current command specification% 0127

OPTION = ↑U / user-settable characters; 0128

%use optional parameter or use optional form of a command% 0129

Confirmation = CA / INSERT / REPEAT; 0130

%used to confirm editing commands% 0131

LEVADJ = $('u / 'd) (SP / CA); 0132

TextSpec1 = 0133

'c <... Character> / 'w <... Word> / 'v <... Visible> / 'i <... Invisible> / 'l <... Link> / 'n <... Number>; 0134

TextSpec2 = 0135

't <... Text>; 0136

StructureSpec1 = 0137

's <... Statement> / 'b <... Branch> / 'p <... Plex>; 0138

StructureSpec2 = 0139

'g <... Group>; 0140

Where <... WORD> denotes that WORD is appended to the command feedback. 0141

FILENAME = LIT / SEL; 0142

%where lit is of the form <dir>file.ext% 0143

LIT = 0144

literal text typed by the user, excluding control characters such as CA, CD, INSERT, OPTION, etc. unless preceded by the literal escape (LE) character (default LE is control-V). 0145

Command Language (Note: All commands and operand-types may be precedded by a SP and some MUST be. If a command or operand-type is not preceded by a SP, the system default command or operand-type for the first letter typed will be assumed and used. If it is preceded by a SP then recognition will take place when a sufficient number of characters have been typed to determine uniqueness.) 0146

append 0147

Syntax: 'a <Append> 0148

((TextSpec1 / StructureSpec1) <at> SEL / 0149

(TextSpec2 StructureSpec2) <from> SEL <(to)> SEL) 0150

<to> SEL LIT 0151

Confirmation; 0152

allow private modifications to file (old browse mode) 0153

assemble program 0154

Syntax: " asse" <Assemble Program at> SEL <Using> ASSEMBLER-NAME <to file> FILENAME CA; 0155

assimilate 0156

Syntax: " assi" <Assimilate> c 0157

(StructureSpec1 SEL / StructureSpec2 SEL SEL) 0158

<after Statement> SEL LEVADJ VIEWSPECS 0159

Confirmation; 0160

arm NLSDDT ↑H 0161

break 0162

Syntax: 'b <Break> 0163

((TextSpec1 / TextSpec2) <at> SEL / 0164

(StructureSpec1 / StructureSpec2) <at> SEL LEVADJ LIT) 0165

Confirmation; 0166

IT would be extremely nice if break Plex ad break group allowed us to convert a structure like 0167

0168

statement a 0169

statement b 0170

statement c 0171

statement d 0172

into a structure like 0173

0174

statement a 0175

statement b 0176

new statement 0177

statement c 0178

statement d 0179

by breaking at statement b. 0180

We could define break word, text etc. as just a quick insert of a SP. Break statement and branch would be equivalent (unless we wish to distinguish??). 0181

copy 0182

Syntax: 'c <Copy> 0183

((TextSpec1 <at> SEL / TextSpec2 <from> SEL <(to)> SEL) <to follow> SEL / 0184

(StructureSpec1 <at> SEL / StructureSpec2 <from> SEL <(to)> SEL) <to follow> SEL LEVADJ) 0185

Confirmation; 0186

<copy> file 0187

allows users to copy files from NLS. 0188

clear window 0189

compact file 0190

compile program 0191

Syntax: " com" <Compile Program at> SEL <Using> COMPILER-NAME <to file> FILENAME Confirmation; 0192

connect (display / tty) 0193

Syntax: 'c <Connect> 0194

('d <Display> / 't <TTY>) <to terminal> NUMBER ['i <Input and output> / 'o <Output only>] CA; 0195

"Input and output" type connection requires that the recipient issue a Receive connection command. 0196 create file 0197

Syntax: " cr" <Create File> FILENAME Confirmation; 0198

Creates a new (empty) file. 0199

delete 0200

Syntax: 'd <Delete> 0201

((TextSpec1 / StructureSpec1) <at> SEL / 0202

(TextSpec2 / StructureSpec2) <from> SEL <(to)> SEL) 0203

Confirmation; 0204

<delete> markers 0205

<delete> ('a <all markers> / 'm <marker named> LIT) CA; 0206

<delete> file 0207

allows users to delete files from NLS (will take care of Partial Copies). 0208

disarm NLSDDT ↑H 0209

disconnect terminals 0210

execute SUBSYSTEM-NAME command 0211

edit 0212

expert 0213

The Novice/Expert design group should specify this command. 0214

expunge directory 0215

goto SUBSYSTEM-NAME 0216

calculator subsystem 0217

identification subsystem 0218

journal subsystem 0219

includes submission, number assignment, secondary distribution, etc. 0220

measurement subsystem 0221

programs subsystem 0222

query subsystem 0223

user options subsystem 0224

The Novice/Expert design group should specify this subsystem. 0225

includes execute viewchange and show ... 0226

print parameters, feedback parameters, control characters 0227

<Show> 0228

('s <... Selections> / 'c <... Control Mark> / 'u <... Upper Case> / 'i <... Input Prompts> / 'l <... LEVADJ numbers>) 0229

Confirmation; 0230

help 0231

prints out instructions for new or confused users including definitions of terms used in syntax rules for commands. 0232

insert 0233

Syntax: 'i <Insert> 0234

((TextSpec1 / TextSpec2) <after> SEL / 0235

(StructureSpec2 / StructureSpec1) <after> SEL LEVADJ) 0236

LIT 0237

Confirmation; 0238

<insert> (sequential / assembler) file 0239

<insert> Journal submission form 0240

jump (DNLS and TNLS) 0241

Syntax for DNLS: 'j <Jump to> 0242

(SEL DAE CA VIEWSPECS) / 0243

(0244

(0245

'i <... Item> / 0246

's <... Successor> / 0247

'p <... Predecessor> / 0248

```
      'u <... Up> /                                   0249
      'd <... Down> /                                 0250
      'h <... Head> /                                 0251
      't <... Tail> /                                 0252
      'e <... End of Branch> /                        0253
      'b <... Back> /                                 0254
      'o <... Origin> /                               0255
      SP "ne" <... Next> /                            0256
      )                                               0257
   SEL VEIWSPECS                                      0258
   ) /                                                0259
(                                                     0260
   'l <... Link> ['l <... Locked>] ( SP LIT / SEL
   VEIWSPECS) /                                       0261
      %locked is a priviledged facility and is not
      available to the average user%                  0262
   'r <... Return> CA                                 0263
      %text from 'return' statement%                  0264
      $(NOT-CA %text from 'return' statement%) /      0265
   'a <... Ahead> CA                                  0266
      %text from 'ahead' statement%                   0267
      $(NOT-CA %text 'ahead' from statement%) /       0268
   'f <... File>                                      0269
      ((SP FILENAME / SEL) VIEWSPECS %load file%/     0270
      (('a <... Ahead> /                              0271
         'r <... Return>)                             0272
         %file name% $(NOT-CA %next file name%) /     0273
   'n <... Name>                                      0274
      ['f <... First> /                               0275
```

'n <... Next>] 0276

(SP LIT CA / SEL) VIEWSPECS / 0277

'c <... Content First> 0278

['f <... First> / 0279

'n <... Next>] 0280

(SP LIT CA / SEL SEL / OPTION %Accept old content%) VIEWSPECS / 0281

'w <... Word First> 0282

['f <... First> / 0283

'n <... Next>] 0284

(SP LIT CA / SEL / OPTION %Accept old word%) VIEWSPECS 0285

) 0286

CA; 0287

Syntax for TNLS: 'j <Jump to> SEL; 0288

k (unused) 0289

load file FILENAME CA; 0290

move 0291

Syntax: 'm <Move> 0292

((TextSpec1 <at> SEL / TextSpec2 <from> SEL <(to)> SEL) <to follow> SEL / 0293

(StructureSpec1 <at> SEL / StructureSpec2 <from> SEL <(to)> SEL) <to follow> SEL LEVADJ) 0294

Confirmation; 0295

<move> boundary 0296

<move> file 0297

allows users to move files from one directory to another from NLS. 0298

mark 0299

Syntax: "ma" <Mark> SEL <with marker name> LIT CA; 0300

merge 0301

Syntax: " mer" <Merge> 0302

((('b <... Branch> / 'p <... Plex>) <at> SEL <into> SEL) / 0303

('g <... Group> <from> SEL <(to)> SEL <into> SEL <(to)> SEL)) 0304

Confirmation; 0305

novice 0306

The Novice/Expert design group should specify the semantics of this command. 0307

output 0308

syntax: 'o <Output> 0309

(0310

('q <Quickprint> / 0311

'j <Journal Mail Quickprint> / 0312

'p <Printer> (COM, etc.)) 0313

FILENAME <Copies = 1?> [NUMBER]) / 0314

(0315

's <Sequential File> / 0316

'a <Assembler File>) 0317

FILENAME) 0318

CA; 0319

p (unused in DNLS) print (TNLS) 0320

Syntax for TNLS print: 0321

'p <Print> 0322

(StructureSpec1 <at> SEL / StructureSpec2 <from> SEL <(to)> SEL) VIEWSPECS [OPTION <using filter:> PATTERN] CA / CA; 0323

if no structure is specified, printing will continue until terminated by control o or until the end of the file is reached. 0324

playback session 0325

quit [SUBSYSTEM-NAME / NLS] 0326

Allows one to terminate NLS from within a subsystem. Also allows one to terminate several levels of subsystem with one command. 0327

replace 0328

Syntax: 'r <Replace> 0329

((TextSpec1 / StructureSpec1) <at> SEL <by> (LIT / rsel) / 0330

(TextSpec2 / StructureSpec2) <from> SEL <(to)> SEL <by> (LIT / r2sel)) 0331

Confirmation; 0332

where 0333

For DNLS: 0334

rsel = SEL; 0335

r2sel = SEL <(to)> SEL; 0336

For TNLS: 0337

rsel = OPTION SEL; 0338

r2sel = OPTION SEL <(to)> SEL; 0339

receive connection from terminal 0340

record session 0341

relock file 0342

resequence SID's in file 0343

reset (tty-simulation window / viewspecs) 0344

in TNLS, default (and reset) viewspecs will have statement numbers on (m). Should SID's (I) be on also??? 0345

substitute 0346

syntax: 's <Substitute> 0347

(TextSpec1 0348

<in> (StructureSpec1 SEL/ StructureSpec2 SEL SEL) 0349

<New> Collect1 <For Old> Collect1 / 0350

TextSpec2 0351

<in> (StructureSpec1 SEL/ StructureSpec2 SEL SEL) 0352

<New> Collect2 <For Old> Collect2) 0353

<Finished?> (NO %repeat at <New>% / YES / OPTION <Using filter:> PATTERN) 0354

Confirmation; 0355

Where: 0356

For TNLS: 0357

Collect1 = Collect2 = LIT; 0358

For DNLS: 0359

Collect1 = (LIT / SEL); 0360

Collect2 = (LIT / SEL SEL); 0361

%propagates the current awkwardness in specifying a NULL LIT% 0362

set (case /
character size for window /
filter /
link default for file /
name delimeters /
tty-simulation window /
viewspecs) 0363

Syntax for case: 0364

" ca" <Case> 0365

((TextSpec1 / StructureSpec1) <at> SEL / 0366

(TextSpec2 / StructureSpec2) <from> SEL <(to)> SEL) / 0367

'm <Mode>) 0368

[mtype] 0369

Confirmation; 0370

mtype = ('i <initial upper> / 'u <upper> / 'l <lower>); 0371

Note: allows temporary mode setting for a single instance of the command 0372

Syntax for Filter: 'f <filter> ('t <to> PATTERN / "on" <On> /

"of" <Off>) 0373

Syntax for name delimiters: 'n <Name Delimeters> 0374

(0375

StructureSpec1 <at> SEL / 0376

StructureSpec2 <from> SEL <(to)> SEL) 0377

<Left Delimiter> LIT <Right Delimiter> LIT 0378

) 0379

Confirmation; 0380

show (file status /
marker list /
name delimiters /
viewspecs [verbose] status) 0381

name delimiters for statement at > SEL Confirmation; 0382

file [lock / size / ownership / return ring] status 0383

simulate terminal type (display / TI-terminal / etc.) 0384

sort 0385

Syntax: "so" <Sort> 0386

(('b <... Branch> / 'p <... Plex>) <at> SEL / 0387

'g <... Group> <from> SEL <(to)> SEL) 0388

Confirmation; 0389

split window (vertically / horizontally) (DNLS) 0390

splits window into two equal halves. 0391

transpose 0392

Syntax: 't <Transpose> 0393

((TextSpec1 / StructureSpec1) <at> SEL <and> SEL / 0394

(TextSpec2 / StructureSpec2) <from> SEL <(to)> SEL <and
from> SEL <(to)> SEL) 0395

Confirmation; 0396

terminate (recording /
private modifications to file) 0397

update file 0398

Syntax: 'u <Update File> %default file name% (OPTION <Old version> / [FILENAME]) CA; 0399

unlock file 0400

verify file 0401

w (not used) 0402

x (unused) 0403

y (unused) 0404

z (unused) 0405

'. (TNLS) show CM 0406

'/ (TNLS) type context of CM 0407

'; Comment 0408

'\ (TNLS) print statement 0409

'↑ (TNLS) print back statement 0410

linefeed (TNLS) print next statement 0411

INSERT %Insert Statement after Control Marker% 0412

Syntax: INSERT 0413

LEVADJ LIT Confirmation; 0414

Insertion and LEVADJ is relative to CM. 0415

REPEAT last editing command 0416

TAB %to next occurrence of content or word% 0417

'? 0418

prints the names of all the commands available at the first level, with a comment about typing the first letter of any command followed by a '? to find out about that (set of) command(s). This should also include an explanation of A:, T:, etc. Also the user is advised to use the command Help to find out about definitions. 0419

Appendix C: Definition of Dynamic Address Expression 0420

Dynamic Address Expression elements 0421

location number 0422

A statement number is D $(L / D / '@). 0423 (no preceding period) 0424

name 0425

A statement name is as defined by the name delimiter routine
-- currently defined to be L S(L/ D/ ''/ '-). 0426

(no preceding period) 0427

System-supplied Statement IDentifiers (SID's) 0428

'O 1S D. 0429

(no preceding period) 0430

A sequence of digits and letters PRECEEDED IMMEDIATELY BY A PERIOD can contain the following letters, with associated "Jump" meaning. NOTE: default value for <number> is 1. 0431

[number]'s jump to successor <number> times 0432

[number]'p jump to predecessor <number> times 0433

[number]'u jump to up <number> times 0434

[number]'d jump to down <number> times 0435

[number]'a jump to ahead <number> times 0436

[number]'r jump to return <number> times 0437

[number]"fa" jump to file ahead <number> times 0438

[number]"fr" jump to file return <number> times 0439

[number]'o jump to origin 0440

[number]'e jump to end 0441

[number]'n jump to next <number> times 0442

[number]'b jump to back <number> times 0443

[number]'h jump to head 0444

[number]'t jump to tail 0445

[number]'l jump to the <number>th link 0446

[number]'w jump to next occurance of word <number> times 0447

[number]'c jump to next occurance of content <number> times 0448

a sequence of digits and letters PRECEEDED IMMEDIATELY BY A PLUS (SKIP FORWARD) OR MINUS (SKIP BACKWARD) can contain the following letters, with associated meaning. NOTE, default value of <number> is 1. 0449

[number]'c skip <number> characters 0450

[number]'w skip <number> word 0451

[number]'v skip <number> visible 0452

[number]'i skip <number> invisible 0453

[number]'n skip <number> number(s) 0454

[number]'l skip <number> link(s) 0455

'* name jumps to the next statement by that name 0456

'(text ') link 0457

text = directory, filename, DAE : Viewspecs 0458

'[text '] content search 0459

text excludes '] unless preceeded by the literal escape character 0460

allows elipses (...) notation 0461

'< text '> word search 0462

text excludes '> unless preceeded by the literal escape character 0463

allows elipses (...) notation 0464

'; text '; intra-statement content search 0465

text excludes '; unless preceeded by the literal escape character 0466

allows elipses (...) notation 0467

'i character character search 0468

← beginning of statement 0469

> end of statement 0470

'# text marker name, text = L $(L/D) 0471

'/ print context 0472

'\ print statement 0473

note that '/ and '\ are part of a DAE. In DNLS this is accomplished via the two line tty-simulation area above the Command Feedback Area. 0474

Appendix D. COMMAND SUMMARY

Section 1. EXECUTIVE COMMANDS

log SP USERNAME SP PASSWORD SP ACCOUNT NO. CR (p.1)

directory CR (p.4)

directory SP <OTHER DIRECTORY'S NAME> CR (p.4)

directory SP , CR (p.4)

@@size CR [CR]
@@ EMPTY CR (p.4)

@@everything CR [CR]
@@EMPTY CR (p.5)

@@deleted [files only] CR [CR]
@@EMPTY C (p.5)

connect [(to directory)] SP DIRECTORY CR (p.5)

delete SP FILENAME CR (p.6)

expunge CR (p.6)

undelete SP FILENAME CR (p.7)

rename [(existing file)] FILENAME [(to be)] FILENAME CR (p.7)

shut CR (p.8)

fullduplex CR (p.8)

halfduplex CR (p.8)

link [(to)] USERNAME CR
TERMINAL NO. (p.9)

break [(links)] CR (p.10)

sys CR (p.10)

Section 2. FILE COMMANDS

```
l[oad] f[ile] FILENAME CA                                  (p.11)

u[pdate file  CA
              o[(to old version)] CA                        (p.13)

o[utput]  f[ile] FILENAME  CA                              (p.14)

e[xecute] u[nlock]  CA  [really ?] CA                      (p.15)

o[utput] d[evice] t[eletype] CA                            (p.16)

e[xecute] f[ile verify]  CA                                (p.17)

e[xecute] r[eset] CA [really ?] CA                         (p.18)

e[xecute] a[ssimilate at] ADDR CA EMPTY CA [CR]
                                  $u
                                   d
[from file] FILENAME CA  [CR]
[structure] statement  [at]  ADDR        CA VIEWSPEC CA
            branch     [at]
            plex       [at]
            group      [at]  ADDR CA ADDR                  (p.19)
```

PRINT CURRENT CM LOCATION COMMAND (p. 22)

.

PRINT STATEMENT AT CM COMMAND (p. 22)

\

/

PRINT STATEMENT BACK FROM CM COMMAND (p. 23)

↑

PRINT STATEMENT NEXT TO CM COMMAND (p. 23)

LF

e[xecute[v[iewchange CR] (p.17)
t[ext area CR]
t[abs: A[aa] AA CA [CR]
i[ndenting=bb] BB
l[ines/page=cc] CC
r[ows/page=dd] DD
c[olumns=ee] EE

```
r[eplace] s[tatement] ADDR CA [by text?] y[es CR] LIT    CA
          b[ranch]                        n[o] ADDR      CDOT
          p[lex]
          g[roup] ADDR CA ADDR
          w[ord]
          c[haracter]
          v[isible]
          i[nvisible]
          n[umber]
          l[ink]
          t[ext] ADDR CA ADDR                       (p. 8)

t[ranspose] s[tatement at]
            b[ranch at] ADDR       CA [and] ADDR         CA
            p[lex at]                                    CDOT
            g[roup at] ADDR CA ADDR      ADDR CA ADDR
            w[ord at]
            c[haracter at]
            v[isible at]
            i[nvisible at]
            n[umber at]
            l[ink at]
            t[ext at]  ADDR CA ADDR      ADDR CA ADDR
                                              (p.10)

a[ppend to] ADDR CA [from] ADDR CA  EMPTY    CA
                                    LIT      CDOT (p.12)

b[reak statement at] ADDR CA  EMPTY    CA
                              $u       CDOT
                              d                   (p.14)

s[ubstitute] s[tatement at]              CA [CR]
             b[ranch at] ADDR
             p[lex at]
             g[roup at] ADDR CA ADDR

[text]  LIT CA  [for] LIT CA [Go?] y[es]
                                   CA
                                   n[o]...        (p.16)
```

<DORNBUSH>CML.NLS;1, 15-JUN-73 14:44 CFD ;

COMMAND META LANGUAGE -- CML 02

INTRODUCTION 03

The command meta-language (CML) is a vehicle for describing the syntax and semantics of the user interface to the NLS system. The syntax is described through the tree-meta alternation and succession concepts. The semantis are introduced via built-in functions and semantic conventions. 011

No attempt is made to describe the full semantics of any command via CML, but it is hoped that the front-end interface (parsing and feedback operations) may be explicitly accomodated with these facilities. It will still be necessary, and desirable, to use execution functions to perform the low-level semantics of the command. The CML describes how the command "looks" to the user, rather than what it does in the system. 012

ELEMENTS OF CML 09

RECOGNIZERS 05

Keyword Recognition 013

The process of keyword recognition is independent of the description of the keywords for CML. In the CML description, each keyword is represented by the full text of the keyword. The algorithm used to match a user's typed input against any list of alternative keywords is known as keyword recognition, and is a function of the command interpreter and is independent of the CML description of the command. 014

SELECTION SPECIFICATION 08

Three types of selections are built into CML. They are DSEL, SSEL, and LSEL (see -- the writeup on the command language for the explicit definition of the selections). Basically, they are recognizers which require some entity type as an argument and they return a pointer to a pair of text pointers. The entity type is obtained either by some previous invocation of the recognition function for some list of keyword entities, or use of the VALUEOF built in function. 089

The DSEL, SSEL, and LSEL functions perform all evaluation and feedback operations associated with the selection operations. 090

FEEDBACK CONTROL 06

The feedback control elements of CML are used to provide feedback in addition to the normal feedback generated by the recognizers. This is used to implement additional "noise words" and help feedback. 092

1) adding feedback to the command feedback link. 093

A string may be added to the current command feedback line by enclosing the quoted string in angle brackets. 094

extra feedback = '< .SR '> 095

2) replacing the last word in the feedback line. 096

It is possible to replace the last string in the command feedback line by using the string replace facility. This is similar to (1) above except the previous word in the feedback line is deleted before adding the new string. 097

replace extra feedback = "<..." .SR '> 098

FUNCTION EXECUTION 07

<DORNBUSH>CML.NLS;8, 3-OCT-73 16:05 CFD ;

COMMAND META LANGUAGE -- CML

INTRODUCTION

The command meta-language (CML) is a vehicle for describing the syntax and semantics of the user interface to the NLS system. The syntax is described through the tree-meta alternation and succession concepts. The semantics are introduced via built-in functions and semantic conventions.

No attempt is made to describe the full semantics of any command via CML, but it is hoped that the front-end interface (parsing and feedback operations) may be explicitly accomodated with these facilities. It will still be necessary, and desirable, to use execution functions to perform the low-level semantics of the command. The CML describes how the command "looks" to the user, rather than what it does in the system.

USE OF CML

The user interface for the NLS command language is defined in the CML specification language. This "program" is then compiled by the CML compiler (written using ARC's tree-meta compiler compiler system) to produce an interpretive text which drives a command parser. The command parser is cognizant of the device dependent feedback and addressing characteristics of the user's i/o device.

ELEMENTS OF CML

PROGRAM STRUCTURE

The basic compilation structure of a CML program is described by:

```
file                 = "FILE" .ID [system] (dcls / rule)
                       #subsys "FINISH";
system               = "SYSTEM" .ID %system name% '=
                       #<'/>.ID %names of subsystems % '; ;
subsys               = "SUBSYSTEM" .ID % subsystem name --  %
                        #(command / rule) "END.";
command              = ("COMMAND"/ "INITIALIZATION" /
                       "TERMINATION") rule ;
rule                 =  .ID '=  exp '; ;
```

The "file" construct brackets the definition of command language subsystems and may optionally include the system definition (which defines all subsystems contained in a particular system).

Parsing rules and declarations may appear at this global level.

The subsystem contruct brackets a set of rules or commands. Commands beginning with the keyword COMMAND are linked together to form a command language subsytem.

The subsystem may include a rule preceded by the keywords INITIALIZATION or TERMINATION. If specified, these rules will be executed once upon system initialization/termination respectively.

Each rule/command is named with an identifier. This name is a global symbol and should not conflict with any other variable names, rule names, or keywords.

DECLARATIONS

Declarations are used to associate attributes with identifier names which are used in cml programs. If not declared, identifiers are defined by their first occurrence according to the following rules.

1) Identifiers appearing on the left hand side of an assignment statement are defined as "VARIABLES".

2) Identifiers followed by a subscripted list are assumed to be of type "FUNCTION".

3) All other undefined identifiers are assumed to be names of parse rules or commands.

The syntax of the declare statement is given by:

```
dcls        = ("DCL" / "DECLARE") [dclattr] #<',> .ID;

dclattr     = ("VARIABLE" / "FUNCTION" / "PARSEFUNCTION");
```

If a declare attribute is not given, type VARIABLE is assumed. Identifiers which are implicitly defined as type variable are EXTERNAL symbols and will be linked by the loader to externally defined symbols with that name.

RECOGNIZERS

Keyword Recognition

The process of keyword recognition is independent of the description of the keywords for CML. In the CML description, each keyword is represented by the full text of the keyword. The algorithm used to match a user's typed input against any list of alternative keywords is known as keyword recognition, and is a function of the command interpreter and is independent of the CML description of the command.

Keywords are written in the meta language as upper-case identifiers enclosed in double quote marks optionally followed by a set of keyword qualifiers.

```
keyword = .SR [ '! #qualifier '! ]
```

The qualifiers serve to control the recognition process for the keywords and to override the system supplied internal identification for the keywords.

```
qualifier       = "NOTT"        % DNLS only keyword %

                 /"NOTD"        % TNLS only keyword %

                 /"L1"          % first level keyword %

                /.NUM           % explicit value for
keyword %
```

Selection Recognition

Three types of selections are built into CML. They are DSEL, SSEL, and LSEL (see -- <userguides,commands,1> for the explicit definition of the selections). Basically, they are recognizers which require some entity type as an argument and they return a pair of text pointers in the state record. The entity type is obtained either by some previous invocation of the recognition function for some list of keyword entities, or use of the VALUEOF built in function.

The DSEL, SSEL, and LSEL functions perform all evaluation and feedback operations associated with the selection operations.

```
selection       = ("SSEL"/ "DSEL"/ "LSEL") '( param ')
```

Other Recognizers

The processes of viewspec recognition, level adjust recognition and command confirmation recognition are represented in CML by built-in parameterless functions in the meta-language.

```
others          = "VIEWSPECS"   % viewspec collection %

                 /"LEVADJ"      % leveladj collection %

                 /"CONFIRM"     % command confirmation %
```

FUNCTION EXECUTION

Functions may be invoked at any point in the parse by writing a name of some routine and enclosing a parameter list in parentheses. All functions invoked by the interpreter must obey the groundrules set up for interpreter routines. The actual arguments are passed by address, rather than value, and two additional actual arguments are appended to the head of the argument list.

```
control     = .ID % routine name % '( $<',> param ')

param       = factor          % expression element %

              / "VALUEOF" '( .SR )    % keyword value %

              / '# .SR        % same as VALUEOF %

              / "TRUE"        % boolean TRUE value "

              / "FALSE"       % boolean FALSE value "

              / "NULL"        % null pointer value %
```

PARSING FUNCTIONS

Functions which are declared with the PARSEFUNCTION attribute are assumed to be parsing functions. They are called in "parsehelp" mode and when so called, are passed the address of a string as a third argument. The parsefunction routine then supplies a prompt string which tells what the parsing functon does. (see appendix 3 for example). Parse functions may appear as alternatives to non-failing recognizers and may themselves fail. Them must however, precede any non-failing recognizers in the list of alternatives.

FEEDBACK CONTROL

The feedback control elements of CML are used to provide feedback in addition to the normal feedback generated by the recognizers. This is used to implement additional "noise words" and help feedback.

1) adding feedback to the command feedback line.

A string may be added to the current command feedback line by enclosing the quoted string in angle brackets.

```
extra feedback = '< .SR '>
```

2) replacing the last word in the feedback line.

It is possible to replace the last string in the command feedback line by using the string replace facility. This is similar to (1) above except the previous word in the feedback line is deleted before adding the new string.

```
replace extra feedback = '<"..." .SR '>
```

A function is also provided to initialize the command feedback mechanisms and clear the command feedback line.

```
clear cfl  = "CLEAR"
```

EXPRESSION DEFINITION

CML is an expression languge. Commands are defined to be a single expression and expressions are composed of successive/alternative expression factors. Alternative paths are indicated by the character '/ in the expression.

The nesting of expressions may be explicitly defined with parenthesis and brackets are used to delimit optional expression elements. The dollar sign preceeding an optional construct is used to indicate that the optional element is repeated as long as the option character is typed in.

```
exp                 = #<'/>alternative;

alternative         = #factor;

factor              = term

                     / '( exp ')

                     / '[ exp ']     % optional element %

                     / '$ '[ exp '] % repeated opt elements
%

term                = subname        % id/ assign/ function %

                     / confirm       % command confirmation %

                     / feedback      % noise word feedback %

                     / recognition   % built-in recognizers %
```

COMPLETE FORMAL SYNTAX OF CML

```
file             = "FILE" .ID [system] $(rule/ dcls)

                 #subsys "FINISH";

system           = "SYSTEM" .ID %system name% '=

                 #<'/>.ID %names of subsystems % '; ;

subsys           = "SUBSYSTEM" .ID % subsystem name --  %

                  #(command / rule) "END.";

command          = ("COMMAND" / "INITIALIZATION" / "TERMINATION")
```

```
                    rule ;
rule             = .ID '=  exp '; ;
dcls             = ("DCL" / "DECLARE") [declattr] #<',> .ID;
dclattr          = ("VARIABLE" / "FUNCTION" / "PARSEFUNCTION");
exp              = #<'/>alternative;
alternative      = #factor;
factor           = term/ '( exp ')/ '[ exp ']/ '$ '[ exp '];
term             = subname/ confirm/ feedback/ recognition;
subname          = .ID  [ '← param/ '( $<',>param ')];
confirm          = "CONFIRM"; % call routine to terminate cmd %
recognition      = keyword/ builtinrec;
keyword          = .SR [ '! #qualifier '! ];
qualifier        = "NOTT"/ "NOTD"/ "L1"/ .NUM;
builtinrec       = (("SSEL"/ "DSEL"/ "LSEL") '( param '))
                   / "VIEWSPECS"/ "LEVADJ";
feedback         = "CLEAR"/ '< ["..."] .SR '>;
control          = .ID '( $<',>param ');
param            = factor/ ("VALUEOF" '( .SR ') / '# .SR)
                   /"TRUE"/ "FALSE"/ "NULL";
```

THE INTERPRETIVE TEXT

Each instruction of the interpretive text contains a structure word at least one function execution word. The structure word defines the alternation and successor paths of the grammar for the command language. The function execution words perform the actions of the interpreter.

The structure words

Each structure word consists of two pointers. The right half of the word defines the alternative node to the current node. The left half of the word points to the successor to the current node. Null paths are indicated by 0 valued pointers.

The executable function word formats

Format 1: [OP CTL MODIFIER ADDR]

This is the only interpreter instruction word format presently defined. OP is an operation code. CTL contains control bits used by the keyword regognition function. MODIFIER may contain an additional value. ADDR is the address or principal value for the function.

The functions of the interpreter.

RECOGNIZERS

KEYOP -- keyword recognition.

CTL = control bits for level 1 commands, DNLS commands, and TNLS commands.

ADDR = address of keyword literal string

The current input text is matched against the keyword string specified by the current node and all alternatives of the current node. This function performs keyword recognition on all of the alternative nodes of the current node simultaneously.

This function cannot fail. Control remains in the keyword recognition function until appropriate input is recognized or until the control is abnormally wrested via backup or command delete functions.

The value returned in the argument record is a single word containing the address of the string corresponding to the keyword actually recognized.

CONFIRM -- process command confirmation characters

This function interrogates the input text for one of the command confirmation characters. Control remains in this routine until a proper confirmation is recognized, and command termination state is appropriately set. This function always returns TRUE.

The value returned is a single word containing a command completion code which identifies the completion mode.

SSEL -- get a source selection

ADDR = not used

The sselect routine is invoked to process a source type selection. The return record contains two text pointers which delimit the selected entity.

DSEL -- get a destination selection

ADDR = not used

The dselect routine is invoked to process a destination type selection. The return record contains two text pointers which delimit the selected entity.

LSEL -- get a literal selection

ADDR = not used

The lselect routine is invoked to process a literal type selection. The selection type is passed as an actual argument. The return record contains two text pointers which delimit the selected entity.

VIEWSPECS -- process viewspecs information

The viewspec input routine is called to process the input stream for viewspec characters. The return record contains the two updated viewspec control words. This function always returns TRUE.

LEVADJ -- process level adjust information

The level adjust input routine is called to process the input stream for level adjust characters. The return record contains a single word which indicates the relative level adjust value (u = +1, d = -1, etc). This function always returns TRUE.

CONTROL FUNCTIONS

EXECUTE -- transfer of control to another point in the tree.

ADDR = address of root of tree for transfer of control

The current point in the tree is marked and control is transferred to the node pointed to by the address field. Control remains in the descendent node until it has been completely parsed, at which time control returns to the successor of the EXECUTE node.

CALL -- subroutine invocation

MODIFIER = number of actual parameters

ADDR = address of the subroutine

The appropriate number of actual arguments are popped off of the evaluation stack and passed to the routine whose address is contained in ADDR.

The resultptr from this routine is pushed onto the eval stack if it returns TRUE.

PFCALL -- parsing function invocation

MODIFIER = number of actual parameters

ADDR = address of the subroutine

The appropriate number of actual arguments are popped off of the evaluation stack and passed to the routine whose address is contained in ADDR.

The resultptr from this routine is pushed onto the eval stack if it returns TRUE.

This function is also called in "parsehelp" mode to find out what it does.

OPTION -- test for an optional construct.

If the next input character is the OPTION select character, then it is read and control is transferred to the node at address ADDR. If the next character is not the OPTION character, then control passes to the successor path of the current node.

ANYOF -- collect alternative optional keyword values

If the next input character is the OPTION select character, then it is read and control is transferred to the node at address ADDR. After the descendent nodes have been processed, control returns to the ANYOF node, permitting another optional selection to be made from among the set of alternatives. The result values from the succession of optional recognitions are logically OR'ed together to form the value for the ANYOF node. If the next character is not the OPTION character, then control passes to the successor path of the ANYOF node.

FEEDBACK ELEMENTS

FBCLEAR -- clear the contents of the feedback buffers.

The feedback state information and command feedback line are set to their initial or empty position.

ECHO -- appends a noise-word string to the command feedback link

ADDR = address of the text string to be appened

RECHO -- replaces the last noise-word string in the command feedback line

ADDR = address of the text string which is to replace the last item in the command feedback buffer

VALUE MANIPULATIONS

LOAD -- loads a pointer to an argument record into the top of the eval stck.

ADDR = address of the variable containing the pointer to the argument record.

The pointer value contained in the variable whose address is contained in ADDR is pushed onto the top of the eval stack.

STORE -- saves a pointer to an argument record in a variable

ADDR -- address of the variable

The address of an argument record is fetched from the top of the eval stack and is saved in the variable at address ADDR.

ENTER -- enters a constant value into the argument record pointed to by the top of the eval stack.

ADDR -- value to be entered (18 BITS only)

The value is taken from the ADDR field of the instruction and is entered into the argument record for the ENTER node in the path stack (whose address is at the top of the eval stack).

VALUEOF -- enters the system value for a keyword into the argument record .

ADDR -- address of the KEYWORD string.

The ADDR points to a string variable. The literal area is searched for a match with the argument string and the address of the literal string which matches the keyword string is entered into the argument record for VALUEOF, whose address is pushed onto the top of the eval stack.

FLOW OF CONTROL IN THE INTERPRETER

At any point in the process of parsing, the control pointer for the interpreter points to a structure word in the grammar. A path stack also exists which shows the nodes from which TRUE returns have been achieved. Some operations mark the path stack for halting the backup process. The parser has 4 distinct control states defined as follows:

1) parsing: recognition state where input text is compared with gramatical constructs to determine the parsing path in the parse tree.

2) backup: A FALSE return has been obtained from some

execution/recognition function. The path stack is backed up until a non-NULL alternative path is found, at which time the parse mode is set to parsing, and recognition of the alternative path is attempted. If no non-NULL alternative path is found, then the parse fails and the interpreter returns FALSE.

3) cleanup: A terminal parse has been achieved and control is passed to each execution routine to reset any state informations set by the routine.

4) repeat: The command is being repeated, and each execution function is given control to redo the operation it last performed (if its function is defaulted by the semantic action of the command).

The general flow of control is:

1) An initial path stack entry is constructed, and the parse mode is set to parsing. The execution function for the current node is evaluated. A pointer to the "function state record" is passed to the routine. The state record contains the return values for the function as well as a record of any state information saved by the function (for backup purposes).

2) If the function returns TRUE, then the successor to the current node becomes the current node. If this is NULL, then the ptrstk stack is backed up until a non-NULL successor path is found. If none is located before the bottom of the current parse state is reached, then the root of a parse tree has been reached, and a command has been successfully executed. In this case the command reset operation is performed and the interpreter is set to "parsing" mode once more.

3) If the function returns FALSE then the parser mode is set to "backup" and a non-NULL alternative path is sought.

After a command has been executed, the parsing path for the tree is re-evaluated in "reverse order" beginning with the terminal node of the path. Each execution function is re-invoked, in "cleanup" mode, and is passed the handle for the state information record which it generated on the forward pass through the grammar. Each execution routine has the responsibility of resetting any state information which it wishes to do at the termination of a command. Cleanup continues until a "starting point" is reached in the parse. This is generally the beginning of the command. At this point, the interpreter "shifts gears" and goes into forward or recognition mode and begins back down the grammar for the language.

The same backup mechanism is also used during command specification in order to back up the parse to allow the respecification of all or part of the command. The command delete function backs out of the parse tree until the beginning of the command is reached.

The same backup mechanism may be adapted to control the partial backup required for executing commands in "repeat mode" where at least one of the alternatives are defaulted to their current values.

The process of marking some nodes in the execution path as defaulted is as yet undefined. It seems that it should be possible to identify those execution functions which need not be re-evaluated in subsequent invocations of the command. The interpreter would then be smart enough to skip over defaulted parameters when in the forward or specification phase of the command and would not invoke backup for defaulted parammeters.

APPENDIX 1: USING THE CML SYSTEM

WRITING CML PROGRAMS

Source programs for the CML compiler are free form NLS files. Comments may be used wherever a blank is permitted and the structural nesting of the source file is ignored by the compiler.

COMPILING CML PROGRAMS

CML source programs are compiled into REL files with the Output Compiler command using CML as the compiler name. The current marker (top of display area) should point to the first statement of a CML program, not the top of an NLS file.

RUNNING CML PROGRAMS

After loading the user program for the parser (<rel-nls>parser) and your rel file, you must connect your grammar to the parser. This is done by using NDDT to change the address field of the instruction at PARSER+1 to point to your grammar (whose address is contained in the symbol table entry corresponding to your subsystem name).

Example:

IF your subsystem name is "expjournal" then you could connect the parser to your grammar with the following NDDT command:

S[how] L[ocation] PARSER+1← MOVEI A1,EXPJOURNAL<CR>

After connecting your test grammar to the parser, parsing is initiated by the NLS command :

G[o to] P[rograms] E[xecute program] PARSER CA

FUNCTION INTERFACE PROTOCAL

The syntax of the function call in the CML meta-language is similar to that of most programming languuages: the name of the function is followed by a list of expressions enclosed in parenthesis. In the CML system however, there are some strict rules which apply to all execution functions invoked by the interpreter. These rules are enumerated below:

1) Additional actual arguments

Preceeding any actual arguments which appear in a function reference in CML, the interpreter supplies two additional actual arguments. These are:

1) a pointer to the "function state record"

2) an integer which defines a parsing mode

= parsing: normal execution mode

= backup: backup after a FALSE path is taken

= cleanup: resetting of state after completion of command

These addtional arguments must be used by all execution functions to determine what they are to do. The pointer to the "function state record" is used to return values from the function and to save state information associated with a particular invocation of the function. The length of the function state record is presently 9 words and this record may be formatted in any manner appropriate to the function.

If 9 words is not sufficient space to record all of the state associated with a particular invocation of a function, then the function must use a storage allocator to allocate the additional storage and record the handles to the allocated storage in the function state record. Note that if this additional "local state" storage is required, then it is the responsibility of the execution function to de-allocaate the local state storage when called in backup or cleanup modes.

2) Returning parse failure

All execution functions are passed a pointer to their function state record. If the function processes normally, then it returns the same pointer as its only return value. If the function decides that the parse should fail at a given point, then it returns FALSE.

3) Passing arguments by address

All of the actual arguments in a function call on an execution function are passed by address rather than by value. The values actually passed are pointers to the function state records corresponding to the actual arguments. The format of the function state records are defined by the execution functions which manipulated them, and thus the location of parameter values in these records is determined by convention, the caller and callee having previously agreed to a particular layout for the function state record. The layout of the records for the built-in interpreter functions in given elsewhere in this appendix.

4) Order of control

An execution function will always be called in parsing mode before it is called in backup or cleanup modes.

A function routine which saves state information in the function state record must initialize its state record to some consistent state before it calls any subroutines which may cause SIGNALS or otherwise cause control to abnormally pass above the execution funtion.

Format of the function state records for the built-in CML recognizers.

Each of the functions of the CML parser utilitzes the function state records in a locally defined way summarized below.

REGOGNIZER	RECORD FORMAT	# WORDS USED
keyword	word 1: address of keyword str	1
viewspecs	word1: updated vs word 1	7
	word2: updated vs word 2	
	words 3-7: vs collection string	
levajd	word1: level adjust count	7
	(u = +1, d = -1, etc)	
	words 2-7: vs collection string	
ssel	words 1-2: txt ptr to start of entity	4
	words 3-4: txt ptr to end of entity	
dsel	same as ssel	
lsel	same as ssel	
confirm	word 1: confirmation code	1

APPENDIX 2: SAMPLE CML PROGRAM

% the following sample program should help illustrate the use of the CML language for describing NLS commands. %

% the grammar is taken from observation of a hypothetical first grade class in the process of receiving art instruction %

% for a more exhauative example, take a look at (dornbush,syntax,) %

FILE sampleprogram % CML to sample.rel %

SUBSYSTEM sample

```
objects =
     "GLUE"!L1!
   / "PASTE"!L1!
   / writingthings;
writingthings =
    "CRAYONS"!L1!
   / "PENS"
   / "PENCILS";
COMMAND zuse =
   "USE"!L1!  what ← writingthings
      <"to draw a pretty">
         ( whom ← "PICTURE"!L1! <"of Aunt Mary">
         / whom ← "SKETCH"!L1! <"of your dog"> )
      CONFIRM
      % call execution routine process the USE command
         *** commented out for now ***
         xuse( what, whom)
         ***    *** % ;


COMMAND ztake =
   "TAKE"!L1! what ← objects
   <"out of your">
      where ← ("EARS"!L1! / "NOSE"!L1! / "MOUTH"!L1!)
   <"PLEASE!!"> CONFIRM;
END.
FINISH
```

APPENDIX 3: SAMPLE INTERPRETER PARSEFUNCTION ROUTINE

Assume that in some command we want the typein of a number to appear as an alternative of some set of keywords. We can accomplish this by defining a parsefunction (call it looknum) which looks at the next input character and succeeds if the next character is a digit and fails otherwise. If we write this function as the first alternative in some command, then control will pass from the interpreter to the parsefunction before it passes to the keyword interpreter.

Suppose our command looks like:

```
COMMAND sample =

    "INSERT"!L1!

        ( looknum() <"number">  ent ← #"NUMBER"

        / ( ent ← ( "TEXT"!L1! / "LINK"!L1!) ))

        % entity now contains an entity type ( number, text, or
        LINK ).  We now use the LSEL function to get a selection of
        this type %

            source ← LSEL( entity)

        % get a command confirmation %

            CONFIRM

        % now invoke the insert execution function passing as
        arguments the entity type and the selection of that type %

            xinsert( entity, source);
```

Now take a look at the parsefunction looknum which is called by the interpreter both when prompting the user and also during the actual parse of the command .

```
% LOOK FOR A NUMBER %

    (looknum) PROC(

        % looknum looks at the next input character, if it is a
        digit, then a true return is taken else FALSE is returned %

        % FORMAL ARGUMENTS %

            resultptr,       % ptr to the function state record %

            parsemode,       % parsing mode for the interpreter %

            string);         % ptr to prompting string %

        REF resultptr, string;

        %------------%

        CASE parsemode OF
```

```
      = parsing:
         CASE lookc() OF
            IN ['0, '9]:
               NULL;
         ENDCASE RETURN (FALSE);
      = parsehelp:
         *string* ← "NUM:";
   ENDCASE;
   RETURN (&resultptr);
   END.
```

```
<NLS>SYNTAX.NLS;10, 29-OCT-73 13:03 CFD ;
FILE nlslanguage              % CML.SAV  TO <rel-nls>SYNTAX.REL %
   % COMMON RULES %
      % ENTITY DEFINITIONS %
         editentity = textent / structure;
      % TEXT ENTITY DEFINITIONS %
         textent = textl / "TEXT"!L1!;
         textl = "CHARACTER"!L1! / "WORD"!L1! / "VISIBLE"!L1! /
         "INVISIBLE"!L1! / "NUMBER"!L1! / "LINK"!L1!;
      % STRUCTURE ENTITY DEFINITIONS %
         structure = "STATEMENT"!L1! / notstatement;
         notstatement = "GROUP"!L1! / "BRANCH"!L1! / "PLEX"!L1! ;
      % SUBSYSTEM NAMES %
         nlssubs = "EDITOR"!L1! /  "CALCULATOR"!L1! /  "FORMS"!L1!
            /  "HELP"!L1! /  "IDENTIFICATION"!L1! /  "JOURNAL"!L1!
            /  "MEASUREMENT"!L1! /  "PROGRAM"!L1! /  "QUERY"!L1!
            /  "USEROPTIONS"!L1! ;
      % ANSWER %
         answer = ("YES"!L1 1!/"NO"!L1 0!/ "<CA>"!L1 1!);
      % SWITCH %
         switch = ("ON"!L1 1!/"OFF"!0!);
      % RECOGNITION MODES %
         rmode = ("EXPERT"!L1! / "ANTICIPATORY"!L1! / "DEMAND"!L1! /
         "FIXED"!L1! );
      % CASE SHIFT MODES %
         cshmode =
            ( "UPPER"!L1!
            / "LOWER"!L1!
            / "INITIAL"!L1! <"upper"> );
      % DIRECTORY OPTIONS %
         diropt =
            ( "ACCOUNT"!L1!/ "ARCHIVE"!L1!/ "CHRONOLOGICAL"!L1!/
            "CRAM"/ "DATES"!L1!/ "DELETE"/ "EVERYTHING"!L1!/
            "LENGTH"!L1!/ "PROTECT"!L1!/ "REVERSE"!L1!/ "SIZE"!L1!/
            "TIME"!L1!/ "VERBOSE"!L1!);
   % DECLARATIONS %
      DECLARE VARIABLE
         %Journal subsystem%
            authfield,
            clerfield,
            commfield,
            distfield,
            formfield,
            keywfield,
            linkfield,
            numbfield,
            numbtype,
            obsofield,
            subcfield,
            submdest,
            submtype,
            titlfield,
            updafield,
         aheadfilename,
         dent,
         dest,
```

```
      ent,
      filtre,
      filename,
      ff,
      fromwhom,
      level,
      literal,
      param,
      param2,
      param3,
      pb,
      port,
      retfilename,
      sent,
      sim,
      source,
      subsys,
      tip,
      vs;
   DECLARE PARSEFUNCTION
      sp,            % reads next char, TRUE if space %
      readca,        % reads next char if ca %
      lookca,        % TRUE if next char is CA %
      looknum,       % TRUE if next char is a number %
      notca;         % reads next char, TRUE iff not CA char %
SUBSYSTEM nlseditor
% NLS EDITOR COMMANDS %
   COMMAND %verify%
      zverify =
         "VERIFY"!L1! "FILE"!L1! CONFIRM xverify( ) ;
   COMMAND %update%
      zupdate =
         "UPDATE"!L1! "FILE"!L1!
            filename ← NULL
            ent ← #"NEW"
            [ent ←
               ( "OLD"!L1! <"version">
               / "COMPACT"!L1!
               / "RENAME"!L1! <"filename">
                 filename ← LSEL( #"FILE" ) )]
            CONFIRM
            xupdate( ent, filename ) ;
   COMMAND %undelete%
      zundelete =
         "UNDELETE"
            ent ←
               ( "FILE"!L1!
               / "ARCHIVE"!L1! <"file">
               / "MODIFICATIONS"!L1! <"to file"> )
            filename ← LSEL( #"FILE" )
            CONFIRM
            xundelete( ent, filename ) ;
   COMMAND %trim%
      ztrim =
         "TRIM" "DIRECTORY"!L1!
            <"no. versions to keep"> param ← LSEL(#"NUMBER")
```

```
            CONFIRM <"really?">
            CONFIRM
            xtrim( param ) ;
      COMMAND %transpose%
         ztranspose =
            "TRANSPOSE"!L1!
               sent ← editentity
               <"at"> source ← DSEL( sent )
               <"and"> dent ← sent
               dest ← DSEL( dent )
               vs ← NULL  filter ← FALSE
                  [ <"Filtered:"> filter ← TRUE  vs ← VIEWSPECS ]
               CONFIRM
               xtranspose( sent, source, dent, dest, filter, vs ) ;
      COMMAND %substitute%
         zsubstitute =
            "SUBSTITUTE"!L1!
               filter ← FALSE  vs ← NULL
               sent ← textent
               <"in"> dent ← structure
               <"at"> dest ← DSEL(dent)
               %** collect param ← subpairs( sent ) **%
                  param ← NULL
               copy2
               CONFIRM
               xsubstitute( sent, dent, dest, param, filter, vs );
      COMMAND %stop%
         zstop =
            "STOP" "RECORD"!L1! <"of Session"> CONFIRM
            xstop() ;
      COMMAND %split%
         zsplit =
            "SPLIT"!NOTT! "WINDOW"!L1!
               param ← ("HORIZONTALLY"!L1! / "VERTICALLY"!L1!)
               CONFIRM
               xsplit( param ) ;
      COMMAND %sort%
         zsort =
            "SORT"
               dent ← notstatement
               dest ← DSEL( dent )
               CONFIRM
               xsimulate( dent, dest ) ;
      COMMAND %simulate%
         zsimulate =
            "SIMULATE" "TERMINAL"!L1! <"type">
               ent ←
                  ( "DISPLAY"!L1!
                  / "TI"!L1! <"Terminal">
                  / "EXECUPORT"!L1!
                  / "33-TTY"!L1 33!
                  / "35-TTY"!L1 35!
                  / "37-TTY"!L1 37! )
               CONFIRM
               xsimulate( ent ) ;
      COMMAND %show%
```

```
      zshow =
         "SHOW"
         ent ←
            ( "FILE"!L1!
               param ←
                  ( "STATUS"!L1!
                  / "LINK"!L1! <"default directory">
                  / "MARKER" <"list">
                  / "MODIFICATIONS"!L1! <"status">
                  / "RETURN"!L1! <"ring">
                  / "SIZE"!L1! )
            / "ARCHIVE"!L1! <"directory">
               param ← (readca() / LSEL(#"NAME") )
                  %CAN WE DO THIS???%
            / "DIRECTORY"!L1!
               param ← (readca() / LSEL(#"NAME") $/diropt/ )
                  %CAN WE DO THIS???%
            / "DISK" <"space status">
            / "NAME"!L1! <"delimiters for statement at">
               param ← DSEL( #"STATEMENT" )
            / "VIEWSPECS"!L1! <"status">
               param ← NULL
               /param ← "VERBOSE"!L1!/ )
         CONFIRM
         xshow( ent, param ) ;
   COMMAND %set%
      zset =
         "SET"
         dest ← NULL param ← NULL  param2 ← NULL param3 ← NULL
         ent ←
            ( "CASE"!L1!
               ( param ← editentity
                  <"at"> dest ← DSEL( param )
                  param2 ← NULL  /param2 ← cshmode/
               / param ← "MODE"!L1!
                  param2 ← cshmode )
            / "CHARACTER"!NOTT!
               <"size for window to"> param ← DSEL( #"NUMBER" )
            / "FEEDBACK" <"mode">
               param ← ("VERBOSE"!L1! / "TERSE"!L1!)
            / "FILTER"!L1!
               param ←
                  ( "TO"!L1!
                     <"pattern"> param2 ← LSEL( #"CHARACTER" )
                  / switch )
            / "LINK"!L1!
               <"default for file to directory">
               param2 ← LSEL(#"NAME")
            / "NAME"!L1!
               <"delimiters in"> param ← structure
               <"at"> dest ← DSEL( param )
               CLEAR
               <"left delimiter"> param2 ← LSEL(#"CHARACTER")
               <"right delimiter"> param3 ← LSEL(#"CHARACTER")
            / "PROMPT"!L1!
               param ← ("OFF"!L1! / "PARTIAL"!L1! / "FULL"!L1!)
```

```
          / "RECOGNITION"!L1! <"mode">
             param ← rmode
          / "TEMPORARY"!L1! <"modifications for file">
          / "TTY"!NOTT! <"window to window">
             param ← LSEL(#"WINDOW")
          / "VIEWSPECS"!L1!
             param ← VIEWSPECS )
       CONFIRM
       xset( ent, param, param2, param3, dest ) ;
COMMAND %retrieve%
   zretrieve =
      "RETRIEVE" "FILE"!L1! <"from archive">
         filename ← LSEL(#"FILE")
         CONFIRM
         xretrieve( filename );
COMMAND %reset%
   zreset =
      "RESET"
         dest ← NULL dent ← NULL
         ent ←
            ( "ARCHIVE"!L1! <"request for file">
               dest ← LSEL(#"FILE")
            / "CASE"!L1! <"mode">
            / "CHARACTER"!L1 NOTT! <"size for window">
            / "FEEDBACK" <"mode">
            / "FILTER"!L1!
            / "LINK"!L1! <"default for file">
            / "NAME"!L1! <"delimiters in">
               dent ← structure
               <"at"> dest ← DSEL(dent)
            / "PROMPT"!L1!
            / "RECOGNITION"!L1! <"mode">
            / "TEMPORARY"!L1! <"modifications for file">
            / "TTY"!NOTT! <"window">
            / "VIEWSPECS"!L1! )
         CONFIRM
         xreset( ent, dent, dest );
COMMAND %replace%
   zreplace =
      "REPLACE"!L1!
         dent ← editentity
         <"at"> dest ← DSEL(dent)
         sent ← dent
         <"by"> source ← LSEL(sent)
         CONFIRM
         xreplace( dent, dest, sent, source );
COMMAND %renumber%
   zrenumber =
      "RENUMBER" "SIDS"!L1!
         <"in file"> CONFIRM
         xrenumber( );
COMMAND %release%
   zrelease =
      "RELEASE"!NOTT!
         ent ←
            ( "FROZEN"!L1! <"statement at">
```

```
                dest ← DSEL(#"STATEMENT")
             / "ALL"!L1! <"frozen statements"> dest ← NULL )
          CONFIRM
          xrelease( ent, dest );
   COMMAND %record%
      zrecord =
         "RECORD" "SESSION"!L1! <"on file"> filename ← LSEL(#"FILE")
            CONFIRM
            xrecord( filename );
   COMMAND %protect%
      zprotect =
         "PROTECT"!L1! "FILE"!L1! filename ← LSEL(#"FILE")
            <"from">
               fromwhom ← ("SELF"!L1! /"GROUP"!L1! / "OTHERS"!L1!)
            param ← #"ALL"
            [
               <"protection">  param ←
                  ( "READ"!L1 1!
                  / "WRITE"!L1 2!
                  / "EXECUTE"!L1 4!
                  / "APPEND"!L1 8!
                  / "LIST"!L1 16!
                  / "ALL"!L1 31! <"access"> )
                  ]
            CONFIRM
            xprotect( filename, fromwhom, param );
   COMMAND %print%
      zprint =
         "PRINT"!L1 NOTD!
            ( readca() ent ← #"REST"  dest ← NULL vs ← NULL
            / ent ← structure
               <"at"> dest ← DSEL( ent )
               vs ← VIEWSPECS )
            xprint( ent, dest, vs );
   COMMAND %playback%
      zplayback =
         "PLAYBACK" "SESSION"!L1!
            <"from file"> filename ← LSEL(#"FILE")
            CONFIRM
            xplayback( filename );
   COMMAND %output%
      zoutput =
         "OUTPUT"!L1!
            ( ( ent ←
               ("QUICKPRINT"!L1!
               / "JOURNAL"!L1!  <"Quickprint">
               / "PRINTER"!L1!
               / "COM"!L1! )
                  filename ← NULL
                  param ← TRUE % use default number of copies %
                  % construct default file name %
                     %** filename ← defilname( ent ) **%
                  [
                     "FILE"!L1! filename ← LSEL(#"FILE") /
                     "COPIES"!L1! param ← LSEL(#"NUMBER")
                     ]
```

```
                 CONFIRM
                 xoutl( ent, filename, param ))
            / ( ent ←
                ("SEQUENTIAL"!L1! /
                "ASSEMBLER"!L1! )
                   <"file"> filename ← LSEL(#"FILE")
                   param ← FALSE
                      ["FORCE"!L1! <"upper case"> param ← TRUE]
                   CONFIRM
                   xoutl( ent, filename, param ))
            / (
                ( ent ← "TERMINAL"!L1! tip ← NULL  port ← NULL
                / ent ← "REMOTE"!L1!
                   <"printer -- TIP"> tip ← LSEL(#"VISIBLE")
                   <"Port #"> port ← LSEL(#"NUMBER") )
                CLEAR
                <"Send Form Feeds?">
                   ff ←
                      ( "YES"!L1 1! sim ← FALSE
                      / "NO"!L1 0! <"Simulate?"> sim ← answer)
                CLEAR
                <"Wait at page break?"> pb ← answer
                CLEAR
                <"Go?">
                   ( readca()
                   / "YES"!L1!
                   / "NO"!L1!
                      <"Type CR when ready, CD to abort">
                      "
                      "!L1!)
                xout2( ent, tip, port, ff, sim, pb))
            );
   COMMAND %move%
      zmove =
         "MOVE"!L1!
            filtre ← FALSE
            vs ← NULL
            level ← NULL
            ( sent ← textl
                copyl
                dent ← sent
                dest ← DSEL(dent)
            / sent ← "TEXT"!L1!
                copyl
                dent ← #"CHARACTER"
                dest ← DSEL(dent)
            / sent ← structure
                copyl
                dent ← #"STATEMENT"
                dest ← DSEL(dent)
                level ← LEVADJ
                copy2
            / dent ← "FILE"!L1!
                sent ← dent
                <"from old filename"> source ← LSEL(sent)
                <"to new filename"> dest ← LSEL(dent)
```

```
            / dent ← "BOUNDARY"!NOTT!
               sent ← dent
               <"from"> source ← LSEL(#"WINDOW")
               <"to"> dest ← LSEL(#"WINDOW")
            )
            CONFIRM
            xmove(sent, source, dent, dest, level, filtre, vs);
COMMAND %merge%
   zmerge =
      "MERGE"
         sent ← notstatement
         <"at"> source ← SSEL(sent)
         dent ← sent
         <"into"> dest ← DSEL(dent)
         CONFIRM
         xmerge( sent, source, dent, dest);
COMMAND %mark%
   zmark =
      "MARK" "CHARACTER"!L1! <"at">
         dest ← DSEL( #"CHARACTER" )
         <"with marker named"> source ← LSEL(#"NAME")
         CONFIRM
         xmark( dest, source);
COMMAND %logout%
   zlogout =
      "LOGOUT"
         CONFIRM xlogout();
COMMAND %load%
   zload =
      "LOAD"!L1!
         ent ← ("FILE"!L1! /  "BUSY"!L1! <"file">)
         filename ← LSEL(#"FILE") CONFIRM
         xload(ent, filename);
%k (unused)%
COMMAND %tjump%
   ztjump =
      "JUMP"!NOTD L1! <"to">
         dest ← DSEL(#"CHARACTER")
         xjump(#"TITEM", dest, NULL);
COMMAND %djump%
   zdjump =
      "JUMP"!NOTT L1! <"to">
         (
            lookca() % look for a bug select %
            ent ← #"ITEM"
            dest ← DSEL( #"STATEMENT" )
            dest ← xjdae( dest ) %LSEL( #"ADDR" )%  vs ←
            VIEWSPECS
         /  ent ←
            (
               "ITEM"!L1!
               / "SUCCESSOR"!L1!
               / "PREDECESSOR"!L1!
               / "UP"!L1!
               / "DOWN"!L1!
               / "HEAD"!L1!
```

```
            / "TAIL"!L1!
            / "END"!L1! <"of Branch">
            / "BACK"!L1!
            / "ORIGIN"!L1!
            / "NEXT"
         )
         dest ← DSEL(#"STATEMENT") vs ← VIEWSPECS
      /
         (
         ent ← "LINK"!L1! (dest ← LSEL(#"LINK")
            vs ← VIEWSPECS)
         / ent ← "RETURN"!L1! dest ← NULL  vs ← NULL
            %** ca()
            retstat ← NULL
            $[
               retstat ← retext(retstat) **%
                  %displays textent from 'return' statement%
                  %returns next 'return' statement%
               %** notca() ] **%
         / ent ← "AHEAD"!L1! dest ← NULL  vs ← NULL
            %** ca()
            aheadstat ← NULL
            $[
               aheadstat ← aheadtext(aheadstat)
               notca() ] **%
                  %displays textent from 'ahead' statement%
                  %returns next 'ahead' statement%
         / ent ← "FILE"!L1! dest ← NULL  vs ← NULL
            (
             sp() dest ← LSEL(#"FILE")
            / lookca() dest ← DSEL(#"FILE") vs ← VIEWSPECS
            / "AHEAD"!L1! ent ← #"FILEAHEAD"
               aheadfilename ← NULL %**
               $[
                  aheadfilename ← aheadfile(aheadfilename)
                  notca()] **%
                     %displays name of 'ahead' file%
                     %returns next 'ahead' file%
            / "RETURN"!L1! ent ← #"FILERETURN"
               retfilename ← NULL %**
               $[
                  retfilename ← retfile(retfilename)
                  notca()]  **%
                     %displays name of 'return' file%
                     %returns next 'return' file%
            )
         / ent ← "NAME"!L1!
            ("FIRST"!L1! (ent ← #"FIRSTNAME") / "NEXT"!L1!
            (ent ← #"NEXTNAME") / "ONLY"!L1!)
            dest ← LSEL(#"NAME") vs ← VIEWSPECS
         / "CONTENT"!L1!
            ("FIRST"!L1! ent ← #"FIRSTCONTENT" / "NEXT"!L1!
            ent ← #"NEXTCONTENT")
            (dest ← LSEL(#"TEXT") %** / option() **% %Accept
            old content%) vs ← VIEWSPECS
               %textent may contain elipses notation (...)%
```

```
                 %Default IS "next"%
             / "WORD"!L1!
               ("FIRST"!L1! ent ← #"FIRSTWORD" / "NEXT"!L1! ent ←
               #"NEXTWORD")
               (dest ← LSEL(#"WORD") %** / option() **% %Accept
               old content%) vs ← VIEWSPECS
                 %textent may contain elipses notation (...)%
                 %Default is "next"%
             )
           )
           CONFIRM
           xjump(ent, dest, vs);
    COMMAND %insert%
      zinsert =
        "INSERT"!L1!
           level ← NULL
           ( ent ← textl
              <"to follow">
              dest ← DSEL(ent)
              param ← LSEL(ent)
           / ent ← "TEXT"!L1!
              <"to follow">
              dest ← DSEL(#"CHARACTER")
              param ← LSEL(ent)
           / ent ← structure
              <"to follow">
              dest ← DSEL(#"STATEMENT")
              level ← LEVADJ
              param ← LSEL(ent)
           / ent ← "JOURNAL"!L1! <"submission form">
              <"to follow">
              dest ← DSEL(#"STATEMENT")
              level ← LEVADJ
              param ← NULL
           )
           CONFIRM
           xinsert(ent, dest, level, param);
    % h (unused) %
    COMMAND %freeze%
      zfreeze =
        "FREEZE"!L1 NOTT! "STATEMENT"!L1! <"at">
           dest ← DSEL(#"STATEMENT")
           vs ← VIEWSPECS CONFIRM
           xfreeze(dest, vs);
    COMMAND %expunge%
      zexpunge =
        "EXPUNGE"
           ent ←
              ( "DIRECTORY"!L1!
              / "ARCHIVE"!L1! <"directory"> )
           CONFIRM
           xexpunge(ent);
    COMMAND %edit%
      zedit =
        "EDIT"!NOTD! "STATEMENT"!L1! <"at">
           dest ← DSEL(#"STATEMENT")
```

```
            xedit(dest);
   COMMAND %disconnect%
      zdisconnect =
         "DISCONNECT" "TERMINAL"!L1! CONFIRM
         xdisconnect();
   COMMAND %delete%
      zdelete =
         "DELETE"!L1!
            filtre ← FALSE dest ← NULL vs ← NULL
            (
               ( ent ← textent
                  <"at"> dest ← DSEL(ent)
               / ent ← structure
                  <"at"> dest ← DSEL(ent)
                  [filtre ← TRUE <"Filtered:"> vs ← VIEWSPECS] )
               / ( ent ← ("FILE"!L1! / "ARCHIVE" <"file">)
                  dest ← LSEL(#"FILE") )
               / ( ent ← "MARKER"
                  <"named"> dest ← LSEL(ent) )
               / ( ent ← "ALL"!L1! <"markers">)
               / (( ent ← "MODIFICATIONS"!L1!)
                  <"to file"> CONFIRM <"really?"> )
            )
            CONFIRM
            xdelete(ent, dest, filtre, vs);
   COMMAND %create%
      zcreate =
         "CREATE" "FILE"!L1!
            filename ← LSEL(#"FILE") CONFIRM
            xcreate(filename);
   COMMAND %copy%
      zcopy =
         "COPY"!L1!
            vs ← NULL
            level ← NULL
            filtre ← NULL
            param ← NULL
            ( sent ← text1
               copy1
               dent ← sent
               dest ← DSEL(dent)
            / sent ← "TEXT"!L1!
               copy1
               dent ← #"CHARACTER"
               dest ← DSEL(dent)
            / sent ← structure
               copy1
               dent ← #"STATEMENT"
               dest ← DSEL(dent)
               level ← LEVADJ
               copy2
            / sent ← "FILE"!L1!
               dent ← sent
               <"from"> source ← LSEL(sent)
               <"to"> dest ← LSEL(dent)
            / sent ← "DIRECTORY"!L1!
```

```
               copy3
            / sent ← "ARCHIVE"!L1! <"directory">
               copy3
            / sent ← "SEQUENTIAL"
               <"file from"> source ← LSEL(#"FILE")
               <" to follow"> dest ← DSEL(#"STATEMENT")
               level ← LEVADJ
               dent ← NULL
               [dent ← ("HEURISTIC"!L1!/ "JUSTIFIED"!L1!/
               "ASSEMBLER"!L1!)]
            )
            CONFIRM
            xcopy(sent, source, dent, dest, level, filtre, vs);
      copy1 =
         <"from"> source ← SSEL(sent)
         <"to follow">;
      copy2 =
         [filtre ← TRUE <"Filtered:"> vs ← VIEWSPECS];
      copy3 =
         <"from"> source ← LSEL( #"VISIBLE" )
         <"to follow"> dest ← DSEL(#"STATEMENT")
         level ← LEVADJ
         dent ← $[diropt];
   COMMAND %connect%
      zconnect =
         "CONNECT"
            <"to">
            (ent ← ("DISPLAY" / "TTY"!L1!)
               <"Number"> dest ← LSEL(#"NUMBER")
               param ← ("INPUT"!L1! <"and Output"> / "OUTPUT"!L1!
               <"Only">) /
            ent ← "DIRECTORY"!L1!  dest ← LSEL(#"NAME")
               param ← NULL
               [<"Password"> param ← LSEL(#"VISIBLE")]
               )
            CONFIRM
            xconnect(ent, dest, param);
   COMMAND %clear%
      zclear =
         "CLEAR"!NOTT! "WINDOW"!L1! <"at">
            dest ← DSEL(#"WINDOW") CONFIRM
            xclear(dest);
   COMMAND %break%
      zbreak =
         "BREAK"!L1!
            ent ← "STATEMENT"!L1!
               <"at"> dest ← DSEL(#"CHARACTER")
               level ← LEVADJ
            CONFIRM
            xbreak(ent, dest, level); %should also pass literal%
   COMMAND %archive%
      zarchive =
         "ARCHIVE" "FILE"!L1!
            filename ← LSEL(#"FILE") param ← $["DELETE"!L1! / "DO
            NOT DELETE" / "DEFERRED" / "IMMEDIATE"!L1! / "NOT
            ALLOWED"!L1!] CONFIRM
```

```
            xarchive(file, param);
    COMMAND %append%
       zappend =
          "APPEND"!L1!
          ( sent ← textl % text entities except "TEXT" %
             zappl
             dent ← sent
          / sent ← "TEXT"!L1!
             zappl
             dent ← #"CHARACTER"
          / sent ← structure
             zappl
             dent ← #"STATEMENT"
          )
          dest ← DSEL(dent)
          literal ← LSEL(#"TEXT")
          CONFIRM
          xappend(sent, source, dent, dest, literal);
       zappl =
          <"at">
          source ← SSEL(sent)
          <"to">;
    COMMAND %accept%
       zaccept =
          "ACCEPT"!NOTT! "CONNECT"!L1!
             <"from display #">  param ← LSEL(#"NUMBER")
             CONFIRM xaccept( param );
    COMMAND %comment%
       zcomment =
          ";"!L1!
             %** xcomment() **% ;
    COMMAND %period%
       zperiod =
          "."!L1 NOTD!
             xperiod() ;
    COMMAND %tab%
       ztab =
          "  "!L1 NOTD!
             xtab() ;
    COMMAND %slash%
       zslash =
          "/"!L1 NOTD!
             xslash() ;
    COMMAND %bslash%
       zbslash =
          "\"!L1 NOTD!
             xbslash() ;
    COMMAND %uparrow%
       zuparrow =
          "↑"!L1 NOTD!
             xuparrow() ;
    COMMAND %linefeed%
       zlinefeed =
          "<LF>"!L1 NOTD!
             xlinefeed() ;
    END.
```

```
% NLS SUPERVISOR COMMANDS %
   SUBSYSTEM subsupervisor
      COMMAND % display subsystem stack %
         zdspss =
         "<"!L1! xsublist();
      COMMAND % display current subsystem stack name %
         zdspcs =
         ">"!L1! xsubcurrent();
      COMMAND
         zquit =
            "QUIT"!L1!
               ( readca()  subsys ← NULL
               / subsys ← (nlssubs / "NLS"!L1!) CONFIRM )
               xquit(subsys);
      COMMAND
         zgoto =
            "GOTO"!L1! <"subsystem">
               subsys ← (nlssubs / "TENEX"!L1! )
               CONFIRM xgoto(subsys, FALSE);
      COMMAND
         zexecute =
            "EXECUTE"!L1! <"command in">
               subsys ← nlssubs
                xgoto(subsys, TRUE);
   END.
% USER PROGRAMMING SUBSYSTEM COMMANDS %
   SUBSYSTEM subprograms
      uprogtypes =
         ("CONTENT"!L1! <"analyzer program">/ "SORT"!L1! <"key
         extractor program">/ "SEGENERATOR"!L1! <"program"> );
      COMMAND
         zpset =
            "SET"
               ent ←
                  ( "BUFFER"!L1! <"size to">
                     source ← LSEL(#"NUMBER")
                  / "NDDT"!L1! <"control-h">
                     source ← NULL )
               CONFIRM xpset(ent, source );
      COMMAND
         zpshow =
            "SHOW"!L1! "STATUS"!L1! <"of programs buffer">
               CONFIRM xpshow( );
      COMMAND
         zpreset =
            "RESET"!L1!
               ent ←
                  ( "BUFFER"!L1! <"size">
                  / "NDDT"!L1! <"control-h"> )
               CONFIRM xpreset( ent );
      COMMAND
         zprun =
            "RUN"!L1! "PROGRAM"!L1!
               source ← LSEL(#"NAME")
               CONFIRM xprun( source );
      COMMAND
```

```
         zpload =
            "LOAD"!L1! "PROGRAM"!L1!
               source ← LSEL(#"FILE")
               CONFIRM xpload( source );
      COMMAND
         zpinstitute =
            "INSTITUTE"!L1! "PROGRAM"!L1!
               source ← LSEL(#"NAME") <"as">
               ent ←  uprogtypes
               CONFIRM xpinstitute( source  ent );
      COMMAND
         zpdeinstitute =
            "DEINSTITUTE"
               ent ←  uprogtypes
               CONFIRM xpdeinstitute( ent );
      COMMAND
         zpdelete =
            "DELETE"!L1!
               ent ←
                  ( "ALL"!L1! <"programs in buffer">
                  / "LAST"!L1! <"program in buffer"> )
                  CONFIRM xpdelete( ent );
      COMMAND
         zpcompile =
            "COMPILE"!L1!
               sent ←
                  ( ("FILE"!L1! / "ASSEMBLER"!L1! <"file"> )
                     <"at"> source ← DSEL(#"STATEMENT")
                     <"using"> compiler ← LSEL(#"FILE")
                     <"to file"> filename ← LSEL(#"FILE")
                  / "L10"!L1! <"user program at">
                     compiler ← NULL  filename ← NULL
                     source ← DSEL(#"STATEMENT") )
                  CONFIRM xpcompile( sent, source, compiler,
                  filename );
   END.
% JOURNAL SUBSYSTEM COMMANDS %
   SUBSYSTEM subjournal
      INITIALIZATION %possible reenter%
         zjinit =
            xjloaworfil() (
               xjsubinc() <"re-enter last submission?">
                  ((readca()/"YES"!L1!) xjgetworfil() / "NO"!L1!
                  jouinit ) /
               jouinit );
         jouinit =
            xjzapworfil()  %sets reenter flag%
            authfield ← NULL
            clerfield ← NULL
            commfield ← NULL
            distfield ← NULL
            formfield ← NULL
            keywfield ← NULL
            linkfield ← NULL
            numbfield ← NULL
            numbtype ← NULL
```

```
        obsofield ← NULL
        subcfield ← NULL
        submdest ← NULL
        submtype ← NULL
        titlfield ← NULL
        updafield ← NULL;
  TERMINATION %close workfile%
     zjterm =
        xjsavworfil() %saves state of variables in workfile%
        xjcloworfil(); %close workfile%
  COMMAND %assign%
     zjassign =
        "ASSIGN"!L1! (
           numbtype ←
              ("JOURNAL"!L1! / "RINS" / "XDOC"!L1! /
              "SPECIAL"!L1! / "NIC"!L1!)
              <"numbers -- how many?"> numbfield ←
              LSEL(#"NUMBER") /
           numbtype ← "RFC"!L1! <"number"> CONFIRM
              <"title"> titlfield ← LSEL(#"TEXT")
              <"author"> authfield ← LSEL(#"TEXT")
              <"distribute to"> distfield ← LSEL(#"TEXT")
              <"online document?"> submtype ← answer
              <"show status?"> ((readca()/"YES"!L1!)
              xjrfcshow()/
                 "NO"!L1! )
           )
        CONFIRM
        xjassign();
  COMMAND %author%
     zjauthor =
        "AUTHOR"!L1!
        authfield ← LSEL(#"TEXT")
        CONFIRM;
  COMMAND %clerk%
     zjclerk =
        "CLERK"
        clerfield ← LSEL(#"WORD")
        CONFIRM;
  COMMAND %comments%
     zjcomments =
        "COMMENTS"!L1!
        commfield ← LSEL(#"TEXT")
        CONFIRM;
  COMMAND %defer%
     zjdefnumber =
        "DEFER"!L1! "NUMBER"!L1! <"assignment">
        numbtype ← #"DEFER"
        CONFIRM;
  COMMAND %distribute%
     zjdistribute =
        "DISTRIBUTE"!L1! <"to"> distfield ← LSEL(#"TEXT")
        CONFIRM;
  COMMAND %finish%
     zjfinish =
        "FINISH"!L1! CONFIRM xjfinish( ); %resets re-enter flag%
```

```
            %performs secondary distribution and journal
            submission%
      COMMAND %interogate%
         zjinterrogate =
            "INTERROGATE"!L1! <"for submission"> CONFIRM
            <"select">  (
               submtype ← structure
                  <"at"> submdest ← DSEL(submtype) /
               submtype ← "FILE" (
                  ["NAMED" submdest ← LSEL(#"FILE")] /
                  submtype ← #"WINDOW"  submdest ← LSEL(#"WINDOW"))
                  /
               "MESSAGE" submtype ← #"STATEMENT"
                  submdest ← LSEL(#"TEXT") /
               submtype ← "HARDCOPY"!L1!
                  <"located at"> submdest ← LSEL(#"TEXT")
               )
            <"title"> titlfield ← LSEL(#"TEXT")
            <"distribute to"> distfield ← LSEL(#"TEXT")
            <"show status?"> (("YES"!L1!/CA) xjshow() / "NO"!L1!)
            <"finished?">
               (("YES"!L1!/CONFIRM) xjfinish() / "NO"!L1!);
      COMMAND %keywords%
         zjkeywords =
            "KEYWORDS"!L1!
            keywfield ← LSEL(#"TEXT")
            CONFIRM;
      COMMAND %lock%
         zjlock =
            "LOCK"!L1! "JOURNAL"!L1! <"password">
            keywfield ← LSEL(#"VISIBLE")
            CONFIRM xjlock(keywfield, TRUE);
      COMMAND %number%
         zjnumber =
            "NUMBER"!L1!
            numbtype ← #"NUMBER"
            numbfield ← LSEL(#"TEXT")
            CONFIRM;
      COMMAND %obsoletes%
         zjobsoletes =
            "OBSOLETES"!L1! <"documents">
            obsofield ← LSEL(#"TEXT")
            CONFIRM;
      COMMAND %place%
         zjplace =
            "PLACE" "LINK"!L1! <"at">
            linkfield ← LSEL(#"VISIBLE")
            CONFIRM;
      COMMAND %print%
         zjprint =
            "PRINT" "HARDCOPY"!L1! %wheel()%
            <"password"> updafield ← LSEL(#"VISIBLE") CONFIRM
            xjpriharcop(updafield);
      COMMAND %process%
         zjprocess =
            "PROCESS"!L1! "SUBMISSION"!L1! <"form at">
```

```
            formfield ← LSEL(#"STATEMENT")
            CONFIRM
            xjprocess(formfield);
      COMMAND %rfc number%
         zjrfcnumber =
            "RFC"!L1! <"number">
            numbtype ← #"RFC"
            numbfield ← LSEL(#"TEXT")
            CONFIRM;
      COMMAND %show%
         zjshow =
            "SHOW"!L1! "STATUS"!L1!
               CONFIRM xjshow( );
      COMMAND %subcollections%
         zjsubcollections =
            "SUBCOLLECTIONS"!L1!
            subcfield ← LSEL(#"TEXT")
            CONFIRM;
      COMMAND %submit%
         zjselect =
            "SELECT"!L1!
            (
               submtype ← structure
                  <"at"> submdest ← DSEL(submtype) /
               submtype ← "FILE" (
                  ["NAMED" submdest ← LSEL(#"FILE")] /
                  submtype ← #"WINDOW"  submdest ← LSEL(#"WINDOW"))
                  /
               "MESSAGE" submtype ← #"STATEMENT"
                  submdest ← LSEL(#"TEXT") /
               submtype ← "HARDCOPY"!L1!
                  <"located at"> submdest ← LSEL(#"TEXT") /
               submtype ← "JOURNAL"!L1! <"document">
                  submdest ← LSEL(#"NUMBER")
               )
            CONFIRM;
      COMMAND %title%
         zjtitle =
            "TITLE"!L1!
            titlfield ← LSEL(#"TEXT")
            CONFIRM;
      COMMAND %unlock%
         zjunlock =
            "UNLOCK"!L1! "JOURNAL"!L1! <"password">
            keywfield ← LSEL(#"VISIBLE")
            CONFIRM xjlock(keywfield, FALSE);
      COMMAND %updates%
         zjupdates =
            "UPDATES"!L1!  <"document(s)"> updafield ← LSEL(#"TEXT")
            CONFIRM;
   END.
SUBSYSTEM subidentification
   INITIALIZATION %not yet implemented%
      ziinit =
         xsubnotimp();
   END.
```

```
SUBSYSTEM subhelp
   INITIALIZATION %not yet implemented%
       zhelpinit =
         xsubnotimp();
   END.
SUBSYSTEM subcalculator
   INITIALIZATION %not yet implemented%
      zcalcinit =
         xsubnotimp();
   END.
FINISH OF NLSLANGUAGE
% IDENTIFICATION SUBSYSTEM COMMANDS %
   SUBSYSTEM subjournal
      INITIALIZATION %possible reenter%
         ideinit =
            xiloaidefil()  %sets reenter flag%
            capafield ← NULL %capabilities%
            commfield ← NULL %comments%
            cordfield ← NULL %cordinator%
            delifield ← NULL %delivery%
            expafield ← NULL %expand%
            funcfield ← NULL %function%
            identype ← NULL  %ident%
            membfield ← NULL %membership%
            hmaidest ← NULL  %hardcopy mail address%
            nmaifield ← NULL %NLS mail address%
            smaifield ← NULL %sequential mail address%
            namefield ← NULL %name%
            nhosfield ← NULL %NLS host name%
            orgafield ← NULL %organization%
            phonfield ← NULL  %phone%
            rtypfield ← NULL
            shosfield ← NULL
            sorgfield ← NULL
            subcfield ← NULL
            updafield ← NULL;
      TERMINATION %close workfile%
         zjterm =
            xjsavworfil() %saves state of variables in workfile%
            xjcloworfil(); %close workfile%
      COMMAND %assign%
         zjassign =
            "ASSIGN"!L1! (
               numbtype ←
                  ("JOURNAL"!L1! / "RINS" / "XDOC"!L1! /
                  "SPECIAL"!L1! / "NIC"!L1!)
                  <"numbers -- how many?"> numbfield ←
                  LSEL(#"NUMBER") /
               numbtype ← "RFC"!L1! <"number"> CONFIRM
                  <"title"> titlfield ← LSEL(#"TEXT")
                  <"author"> authfield ← LSEL(#"TEXT")
                  <"distribute to"> distfield ← LSEL(#"TEXT")
                  <"online document?"> submtype ← answer
                  <"show status?"> (("YES"!L1!/CA) xjrfcshow() /
                  "NO"!L1!)
               )
```

```
         CONFIRM
         xjassign();
   COMMAND %author%
      zjauthor =
         "AUTHOR"!L1!
         authfield ← LSEL(#"TEXT")
         CONFIRM;
   COMMAND %clerk%
      zjclerk =
         "CLERK"
         clerfield ← LSEL(#"WORD")
         CONFIRM;
   COMMAND %comments%
      zjcomments =
         "COMMENTS"!L1!
         commfield ← LSEL(#"TEXT")
         CONFIRM;
   COMMAND %defer%
      zjdefnumber =
         "DEFER"!L1! "NUMBER"!L1! <"assignment">
         numbtype ← #"DEFER"
         CONFIRM;
   COMMAND %distribute%
      zjdistribute =
         "DISTRIBUTE"!L1! <"to"> distfield ← LSEL(#"TEXT")
         CONFIRM;
   COMMAND %finish%
      zjfinish =
         "FINISH"!L1! CONFIRM xjfinish( ); %resets re-enter flag%
            %performs secondary distribution and journal
            submission%
   COMMAND %interogate%
      zjinterrogate =
         "INTERROGATE"!L1! <"for submission"> CONFIRM
         <"select">  (
            submtype ← structure
               <"at"> submdest ← DSEL(submtype) /
            submtype ← "FILE" (
               ["NAMED" submdest ← LSEL(#"FILE")] /
               submtype ← #"WINDOW"  submdest ← LSEL(#"WINDOW"))
               /
            "MESSAGE" submtype ← #"STATEMENT"
               submdest ← LSEL(#"TEXT") /
            submtype ← "HARDCOPY"!L1!
               <"located at"> submdest ← LSEL(#"TEXT")
            )
         <"title"> titlfield ← LSEL(#"TEXT")
         <"distribute to"> distfield ← LSEL(#"TEXT")
         <"show status?"> (("YES"!L1!/CA) xjshow() / "NO"!L1!)
         <"finished?">
            (("YES"!L1!/CONFIRM) xjfinish() / "NO"!L1!);
   COMMAND %keywords%
      zjkeywords =
         "KEYWORDS"!L1!
         keywfield ← LSEL(#"TEXT")
         CONFIRM;
```

```
COMMAND %lock%
   zjlock =
      "LOCK"!L1! "JOURNAL"!L1! <"password">
      keywfield ← LSEL(#"VISIBLE")
      CONFIRM xjlock(keywfield);
COMMAND %number%
   zjnumber =
      "NUMBER"!L1!
      numbtype ← #"NUMBER"
      numbfield ← LSEL(#"TEXT")
      CONFIRM;
COMMAND %obsoletes%
   zjobsoletes =
      "OBSOLETES"!L1! <"documents">
      obsofield ← LSEL(#"TEXT")
      CONFIRM;
COMMAND %place%
   zjplace =
      "PLACE" "LINK"!L1! <"at">
      linkfield ← LSEL(#"VISIBLE")
      CONFIRM;
COMMAND %print%
   zjprint =
      "PRINT" "HARDCOPY"!L1! %wheel()%
      <"password"> updafield ← LSEL(#"VISIBLE") CONFIRM
      xjpriharcop(updafield);
COMMAND %process%
   zjprocess =
      "PROCESS"!L1! "SUBMISSION"!L1! <"form at">
      formfield ← LSEL(#"STATEMENT")
      CONFIRM
      xjprocess(formfield);
COMMAND %rfc number%
   zjrfcnumber =
      "RFC"!L1! <"number">
      numbtype ← #"RFC"
      numbfield ← LSEL(#"TEXT")
      CONFIRM;
COMMAND %show%
   zjshow =
      "SHOW"!L1! "STATUS"!L1!
         CONFIRM xjshow( );
COMMAND %subcollections%
   zjsubcollections =
      "SUBCOLLECTIONS"!L1!
      subcfield ← LSEL(#"TEXT")
      CONFIRM;
COMMAND %submit%
   zjselect =
      "SELECT"!L1!
      (
         submtype ← structure
            <"at"> submdest ← DSEL(submtype) /
         submtype ← "FILE" (
            ["NAMED" submdest ← LSEL(#"FILE")] /
            submtype ← #"WINDOW"  submdest ← LSEL(#"WINDOW"))
```

```
                /
             "MESSAGE" submtype ← #"STATEMENT"
                submdest ← LSEL(#"TEXT") /
             submtype ← "HARDCOPY"!L1!
                <"located at"> submdest ← LSEL(#"TEXT") /
             submtype ← "JOURNAL"!L1! <"document">
                submdest ← LSEL(#"NUMBER")
             )
          CONFIRM;
    COMMAND %title%
       zjtitle =
          "TITLE"!L1!
          titlfield ← LSEL(#"TEXT")
          CONFIRM;
    COMMAND %unlock%
       zjunlock =
          "UNLOCK"!L1! "JOURNAL"!L1! <"password">
          keywfield ← LSEL(#"VISIBLE")
          CONFIRM xjunlock(keywfield);
    COMMAND %updates%
       zjupdates =
          "UPDATES"!L1!  <"document(s)"> updafield ← LSEL(#"TEXT")
          CONFIRM;
 END.
```

:DEL, 02/06/69 1010:58 JFR ; .DSN=1; .LSP=0; .SCR=1; .DPR=0; ['=] AND NOT SP ; ['?]; dual transmission?

Abstract.

The Decode-Encode Language (DEL) is a machine independent language tailored to two specific computer network tasks:

accepting input codes from interactive consoles, giving immediate feedback, and packing the resulting information into message packets for network transmission.

and accepting message packets from another computer, unpacking them, building trees of display information, and sending other information to the user at his interactive station.

This is a working document for the evolution of the DEL langauge. Comments should be made through Jeff Rulifson at SRI.

Foreword.

The initial ARPA network working group met at SRI on October 25-26, 1968.

It was generally agreed beforehand that the running of interactive programs across the network was the first problem that would be faced.

This group, already in aggrement about the underlaying notions of a DEL-like approach, set down some terminology, expectations for DEL programs, and lists of proposed semantic capability.

At the meeting were Andrews, Baray, Carr, Crocker, Rulifson, and Stoughton.

A second round of meetings was then held in a piecemeal way.

Crocker meet with Rulifson at SRI on November 18, 1968. This resulted in the incorporation of formal co-routines.

and Stoughton meet with Rulifson at SRI on December 12, 1968. It was deceided to meet again, as a group, probably at UTAH, in late Janurary, 1969.

The first public release of this paper was at the BBN NET meeting in Cambridge on February 13, 1969.

NET Standard Translators.

NST The NST library is the set of programs necessary to mesh efficiently with the code compiled at the user sites from the DEL programs it receives. The NST-DEL approach to NET interactive system communication is intended to operate over a broad spectrum.

The lowest level of NST-DEL useage is direct transmission to the server-host, information in the same format that user programs would receive at the user-host.

In this mode, the NST defaults to inaction. The DEL program does not receive universal hardware representation input but input in the normal fashion for for the user-host.

And the DEL program becomes merely a message builder and sender.

A more intermediate use of NST-DEL is to have echo tables for a TTY at the user-host.

In this mode, the DEL program would run a full duplex TTY fir the user.

It would echo characters, translate them to the character set of the server-host, pack the translated characters in messages, and on appropiate break characters send the messages.

When messages come from the server-host, the DEL progam would translate them to the user-host character set and print them on his TTY.

A more ambitous task for DEL is the operation of large, display-oriented systems from remote consoles over the NET.

Large interactive systems usually offer a lot of feedback to the user. The unusual nature of the feedback make it impossible to model with echo table, and thus a user program must be activated in a TSS each time a button state is changed.

This puts an unnecessarily large load on a TSS, and if the system is begin run through the NET it could easily load two systems.

To avoid this double overloading of TSS, a DEL program will run on the user-host. It will handel all the immediate feedback, much like a complicated echo table. At appropiate button pushes, message will be sent to the server-host and display updates received in return.

One of the more difficult, and often neglected, problems is the effective simulation of one non-standard console on another non-standard console.

We attempt to offer a means of solving this problem through the co-routine structure of DEL programs. For the complicated interactive systems, part of the DEL programs will be constructed by the server-host programmers. Interfaces between this program and the input stream may easily be inserted by programmers at the user-host site.

Universal Hardware Representation

To minimize the number of translators needed to map any facility's user codes to any other facility, there is a universal hardware representation.

This is simply a way of talking, in general terms, about all the hardware devices at all the interactive display stations in the initial newtork.

For example, a display is thought of as being a square, the mid-point has coordinates (0,0), the range is -1 to 1 on both axes. A point may now be specified to any accuracy, regardless of the particular number or density of rastor points on a display.

The representation is discussed in the semantic explanitations accompaning the formal description of DEL.

Introduction to the Network Standard Translatore (NST).

Suppose that a user at a remote site, say Utah, is entered in the AHI system and wants to run NLS.

The first step is to enter NLS in the normal way. At that time the Utah system will request a symbolic program from NLS.

REP This program is written in DEL. It is called the NLS Remote Encode Program (REP).

The program accepts input in the Universal Hardware Representation and translates it to a form usable by NLS.

It may pack characters in a buffer, also do some local feedback.

When the program is first received at Utah it is compiled and loaded to be run in conjunction with a standard library.

All input from the Utah console first goes to the NLS NEP. It is processed, parsed, blocked, translated, etc. When NEP receives a character appropriate to its state it may finally initiate transfers to the 940. The bits transferred are in a form acceptable to the 940, and maybe in a standard form so that the NLS need not differentiate between Utah and other NET users.

Advantages of NST

After each node has implemented the library part of the NST, it need only write one program for each subsystem, namely the symbolic file it sends to each user that maps the NET hardware represenataion into its own special bit formats.

This is the minimum programming that can be expected if each console is used to its fullest extent.

Since the NST which runs the encode translation is coded at the user site, it can take advantage of hardware at its consoles to the fullest extent. It can also add or remove hardware features without requiring new or different translation tables from the host.

Local users are also kept up to date on any changes in the system offered at the host site. As new features are added, the host programmers change the symbolic encode program. When this new program is compiled and used at the user site, the new features are automatically included.

The advantages of having the encode translation programs transferred symbolically should be obvious.

Each site can translate any way it sees fit. Thus machine code for each site can be produced to fit that site; faster run times and greater code density will be the result.

Moreover, extra symbolic programs, coded at the user site, may be easily interfaced between the user's monitor system and the DEL program from the host machine. This should ease the problem of console extension (e.g. accmodating unusual keys and buttons) without loss of the flexability needed for man-machine interaction.

It is expected that when there is matching hardware, the symbolic programs will take this into account and avoid any unnecessary computing. This is immediaely possible through the code translation constructs of DEL. It may someday be possible through program composition (when Crocker tells us how??).

AHI NLS - User Console Communication - An Example.

Block Diagram

The right side of the picture represents functions done at the user's main computer; the left side represents those done at the host computer.

Each label in the picture corresponds to a statement with the same name.

There are four trails associated with this picture. The first links (in a forward direction) the labels which are concerned only with network information. The second links the total information flow (again in a forward direction). The last two are equivalent to the first two but in a backward direction. They may be set with pointers t1 through t4 respectively. [">tif"] OR [">nif"]; ["<tif"] OR ["<nif"];

User-to-Host Transmission

keyboard is the set of input devices at the user's console. Input bits from stations, after drifting through levels of monitor and interrupt handlers, eventually come to the encode translator. [>nif(encode)]

encode maps the semi-raw input bits into an input stream in a form suited to the serving-host subsystem which will process the input. [>nif(hrt)<nif(keyboard)]

The Encode program was supplied by the server-host subsystem when the subsystem was first requested. It is sent to the user machine in symbolic form and is compiled at the user machine into code particularly suited to that machine.

It may pack to break characters, map multiple characters to single characters and vice versa, do character translation, and give immediate feedback to the user.

ldm Immediate feedback from the encode translator first goes to local display management, where it is mapped from the NET standard to the local display hardware.

A wide range of echo output may come from the encode translator. Simple character echoes would be a minimum, while command and machine-state feedback will be common.

It is reasonable to expect control and feedback functions not even done at the server-host user stations to be done in local display control. For example, people with high-speed displays may want to selectively clear curves on a Culler display, a function which is impossible on a storage tube.

Output from the encode translator for the server-host goes to the invisible IMP, is broken into appropriate sizes and labeled by the encode translator, and then ;oes to the NET-to-host translator.

Output from the user may be more than on-line input. It may be larger items such as computer-generated data, or files generated and used exclusively at the server-host site but stored at the user-host site.

Information of this kind may avoid translation, if it is already in server-host format, or it may undergo yet another kind of translation if it is a block of data.

hrp It finally gets to the host, and must then go through the host reception program. This maps and reorders the standard transmission-style packets of bits sent by the encode programs into messages acceptable to the host This program may well be part of the monitor of the host machine.[>tif(net mode)<nif(encode)]

Host-to-User Transmission

decode Output from the server-host initially goes through decode, a translation map similar to, and perhaps more complicated than, the encode map.[>nif(urt)>tif(imp ctrl)<tif(net mode)]

This map at least formats display output into a simplified logical-entity output stream, of which meaningful pieces may be dealt with in various ways at the user site.

The Decode program was sent to the host machine at the same time that the Encode program was sent to the user machine. The program is initially in symbolic form and is compiled for efficient running at the host machine.

Lines of characters should be logically identified so that

different line widths can be handled at the user site.
Some form of logical line identification must also me made. For example, if a straight line is to be drawn across the display this fact should be transmitted, rather than a series of 500 short vectors.
As things firm up, more and more complicated structural display information (in the manner of LEAP) should be sent and accomodated at user sites so that the responsibility for real-time display manipulation may shift closer to the user.

imp ctrl The server-host may also want to send control information to IMPs. Formatting of this information is done by the host decoder. [>tif(urt) <tif(decode)]
The other control information supplied by the host decoder is message break up and identification so that proper assembly and sorting can be done at the user site.

From the host decoder, information goes to the invisible IMP, and directly to the NET-to-user translator. The only operation done on the messages is that they may be shuffled.

urt The user reception translator accepts messages from the user-site IMP and fixes them up for user-site display. [>nif(d ctrl)>tif(prgm ctrl)<tif(imp ctrl)<nif(decode)]

The minimal action is a reordering of the message pieces.
dctrl For display output, however, more needs be done. The NET logical display information must be put in the format of the user site. Dispay control does this job. Since it coordinates between (encode) and (decode) it is able to offer features of display management local to the user site.[>nif(display)<nif(urt)]
prgmctrl Another action may be the selective translation and routing of information to particular user-site subsystems. [>tif(d ctrl)<tif(urt)]

For example, blocks of floating-point information may be converted to user-style words and sent, in block form, to a subsystem for processing or storage.
The styles and translation of this information may well be a compact binary format suitable for quick translation, rather than a print-image-oriented format.

(display) is the output to the user. [<nif(d ctrl)]

User-to-Host Indirect Transmission

(net mode) This is the mode where a remote user can link to a node indirectly through another node.[>tif(decode)<tif(hrt)]

DEL Syntax.

Notes for NLS Users.

All statements in this branch which are not part of the compiler must end with a period.
To compile the DEL compiler:

Set this pattern for the content aalyzer (↑P1 SE(P1) < -'.;).
The pointer "del" is on the first character of pattern.
Jump to the first statement of the compiler. The pointer "c" is on this statement.,
And output the compiler to file('/A-DEL'). The pointer "f" is on the name of the file for the compiler output .

Programs.

Syntax.

```
.meta file (k=100,m=300,n=20,s=900)
file = mesdecl $declaration $procedure "FINISH";
procedure =
   procname (
      (
         type "FUNCTION" /
         "PROCEDURE" ) .id (type .id / .empty)) /
      "CO-ROUTINE") '; /
   $declaration labeledst $(labeledst ';) "endp.";
labeledst = (←.id ': / .empty) statement;
type = "INTEGER" / "REAL" ;
procname = .id;
```

Functions are differentiated from procedures to aid compilers in better code production and run time checks.

Functions return values.

Procedures do not return values.

Co-routines do not have names or arguments. Their initial envocation points are given the pipe declaration.

It is not clear just how global declarations are to be??

Declarations.

Syntax.

```
declaration = numbertype / structuredtype / label / lcl2uhr /
uhr2rmt / pipetype;
numbertype = ("REAL" / "INTEGER") ("CONSTANT" conlist /
varlist);
conlist =
   .id '← constant
   $(', .id '← constant);
varlist =
   .id ('← constant / .empty)
   $(', .id ('← constant / .empty));
idlist = .id $(', .id);
structuredtype = ("tree" / "pointer" / "buffer" ) idlist;
label = "LABEL" idlist;
pipetype = "PIPE" pairedids $(', pairedids);
pairedids = .id .id;
procname = .id;
integerv = .id;
pipename = .id;
labelv = .id;
```

Variables which are declared to be constant, may be put in read-only memory at run time.

The label declaration is to declare cells which may contain the machine addresses of labels in the program as their values. This is not the B5500 label declaration.

In the pipe declaration the first .ID of each pair is the name of the pipe, the second is the initial starting point for the pipe.

Arithmetic.

Syntax.

```
exp = "IF" conjunct "THEN" exp "ELSE" exp;
sum = term (
   '+ sum /
```

```
        '- sum /
        .empty);
     term = factor (
        '* term /
        '/ term /
        '↑ term /
        .empty);
     factor = '- factor / bitop;
     bitop = compliment (
        ''/ bitop /
        '/' bitop /
        '& bitop /
        .empty);
     compliment = "--" primary / primary;
```

↑ means mod, and / means exclusive or.

Notice that the uniary minus is allowable, and parsed so you can write x*-y.

Since there is no standard convention with bitwise operators, they all have the same precedence, and parentheses must be used for grouping.

Compliment is the 1's compilment.

It is assumed that all arithmetic and bit operations take place in the mode and style of the machine running the code. Anyone who takes advantage of word lengths, two's compliment arithmetic, etc. will eventually have problems.

Primary.

Syntax.

```
     primary =
        constant /
        builtin /
        variable /
        block /
        '( exp ');
     variable = .id (
        '← exp /
        '( block ') /
        .empty);
     constant =  integer / real / string;
     builtin =
        mesinfo /
        cortnin /
        ("MIN" / "MAX") exp $(', exp) ' ;
```

parenthesised expressions may be a series of expressions. The value of a series is the value of the last one executed at run time.

Subroutines may have one call by name arguement.

Expressions may be mixed. Strings are a big problem?? Rulifson also wants to get rid of real numbers↑↑

Conjunctive Expression.

Syntax.

```
     conjunct = disjunct ("AND" conjunct / .empty);
     disjunct = negation ("OR" negation / .empty);
     negation = "NOT" relation / relation;
```

```
relation =
   '( conjunct ') /
   sum (
      "<=" sum /
      ">=" sum /
      '< sum /
      '> sum /
      '= sum /
      '# sum /
      .empty);
```

The conjunct construct is rigged in such a way that a conjunct which is not a sum need not have a value, and may be evaluated using jumps in the code. Reference to the conjunct is made only in places where a logical decision is called for (e.g. if and while statements).

We hope that most compilers will be smart enough to skip unnecessary evaluations at run time. I.e. a conjunct in which the left part is false or a disjunct with the left part true need not have the corresponding right part evaluated.

Arithmetic Expression.

Syntax.

```
statement = conditional / unconditional;
unconditional = loopst / casest / controlst / iost / treest /
block / null / exp;
conditional = "IF" conjunct "THEN" unconditional (
   "ELSE" conditional /
   .empty);
block = "begin" exp $('; exp) "end";
```

An expressions may be a statement. In conditional statements the else part is optional while in expressions is is manditory. This is a side effect of the way the left part of the syntax rules are ordered.

Semi--Tree Manipulation and Testing.

Syntax.

```
treest = setpntr / insertpntr / deletepntr;
setpntr = "set" "pointer" pntrname "to" pntrexp;
pntrexp = direction pntrexp / pntrname;
insertpntr = "insert" pntrexp "as"
   (("left" / "right") "brother") /
   (("first" / "last" ) "daughter") "of" pntrexp;
direction =
   "up" /
   "down" /
   "forward" /
   "backward" /
   "head" /
   "tail";
planttree = "plant" tree "in" treename;
replacepntr = "replace" pntrname "with" pntrexp;
deletepntr = "delete" pntrname;
tree = '( treel ') ;
treel = nodename $nodename ;
nodename = terminal / '( treel ') ;
```

```
      terminal = treename / buffername / pointername;
      treename = .id;
      treedecl = "pointer" .id / "tree" .id;
```
   Extra parentheses in tree building results in linear subcategorization, just as in LISP.

Flow and Control.

```
   controlst = gost / subst / loopst / casest;
```
   Go To Statements.

```
      gost = "GO" "TO" (labelv / .id);
         assignlabel = "ASSIGN" .id "TO" labelv;
```
   Subroutines.

```
      subst = callst / returnst / cortnout;
         callst = "CALL" procname (exp / .empty);
         returnst = "RETURN" (exp / .empty);
         cortnout = "STUFF" exp "IN" pipename;
      cortnin = "FETCH" pipename;
```
      FETCH is a builtin function whose value is computed by envoking the named co-routine.

   Loop Statements.

      Syntax.

```
         loopst = whilest / untilst / forst;
         whilest = "WHILE" conjunct "DO" statement;
         untilst = "UNTIL" conjunct "DO" statement;
         forst = "FOR" integerv '← exp ("BY" exp / .empty) "TO" exp
         "DO" statement;
```
      The value of while and until statements is defined to be false and true (or 0 and non-zero) respectively.

      For statements evaluate their initial exp, by part, and to part once, at initialization time. The running index of for statements is not available for change within the loop, it may only be read. If some compilers can take advantage of this (say put it in a register) all the better. The increment and the to bound will both be rounded to integers during the initialization.

   Case statements.

      Syntax.

```
         casest = ithcasest / condcasest;
         ithcasest = "ITHCASE" exp "OF" "BEGIN" statement $(';
         statement) "END";
         condcasest = "CASE" exp "OF" "BEGIN" condcs $('; condcs)
         "OTHERWISE" statement "END";
         condcs = conjunct ': statement;
```
      The value of a case statement is the value of the last case executed.

Extra statements.

```
   null = "NULL";
```
I/O Statements.

```
   iost = messagest / dspyst   ;
```
   Messages.

      Syntax.

```
         messagest =  buildmes / demand;
            buildmes = startmes / appendmes / sendmes;
               startmes = "start" "message";
```

```
          appendmes = "append" "message" "byte" exp ;
          sendmes = "send" "message";
        demandmes = "demand" "message";
      mesinfo =
        "get" "message" "byte"
        "message" "length" /
        "message" "empty" '? ;
      mesdecl = "message" "bytes" "are" .num "bits" "long" '; ;
```

Display Buffers.

Syntax.

```
dspyst = startbuffer / bufappend / estab;
startbuffer = "start" "buffer";
bufappend = "append" bufstuff $('& bufstuff);
bufstuff =
   "parameters" dspyparm $(', dspyparm) /
   "character" exp /
   "string" string /
   "vector" ("from" exp ': exp / .empty) "to" exp ', exp /
   "position" (onoff / .empty) "beam" "to" exp ': exp/
   "curve" ;
dspyparm =
   "intensity" "to" exp /
   "character" "width" "to" exp /
   "blink" onoff /
   "italics" onoff;
onoff = "on" / "off";
estab = "establish" buffername;
```

Logical Screen.

The screen is taken to be a square. The coordinates are normalized from -1 to +1 on both axes.

Associated with the screen is a position register, called PREG. The register is a triple <x,y,r>, where x and y specify a point on the screen and r is a rotation in radians, counter clockwise, from the x-axis.

The intensity, called INTENSITY, is a real number in the range from 0 to 1. 0 is black, 1 is as light as your display can go, and numbers in between specify the relative log of the intensity difference.

Character frame size.

Blink bit.

Buffer Building.

The terminal nodes of semi-trees are either semi-tree names or display buffers. A display buffers is a series of logical entities, called bufstuff.

When the buffer is initilized, it is empty. If no parameters are initially appended, those in effect at the end of the display of the last node in the semi-tree will be in effect for the display of this node.

As the buffer is built, the logical entites are added to it. When it is established as a buffername, the buffer is closed, and furthur appends are prohibited. It is only a buffername has been established that it may be used in a tree building statement.

Logical Input Devices.
Wand.
Joy Stick.
Keyboard.
Buttons.
Light Pens.
Mice.
Audio Output Devices.
.end

Sample Programs

Program to run display and keyboard as tty.
to run NLS.
input part
display part

```
DEMAND MESSAGE;
While LENGTH # 0 DO
    ITHCASE GETBYTE OF Begin
        IHTCASE GETBYTE OF %file area update% BEGIN
            %literal area%
            %message area%
            %name area%
            %bug%
            %sequence specs%
            %filter specs%
            %format specs%
            %command feedback line%
            %file area%
            %date time%
            %echo register%
        BEGIN %DEL control%
```

Distribution List

Steve Carr
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
Phone 801-322-7211 X8224

Steve Crocker
Boelter Hall
University of California
Los Angeles, California 90024
Phone 213-825-4864

Jeff Rulifson
Stanford Research Institute
333 Ravenswood
Menlo Park, California 94305
Phone 415-326-6200 X4116

Ron Stoughton
Computer Research Laboratory
University of California
Santa Barbara, California 93106
Phone 805-961-3221

Mehmet Baray
Corey Hall

University of California
Berekely, California 94720
Phone 415-843-2621