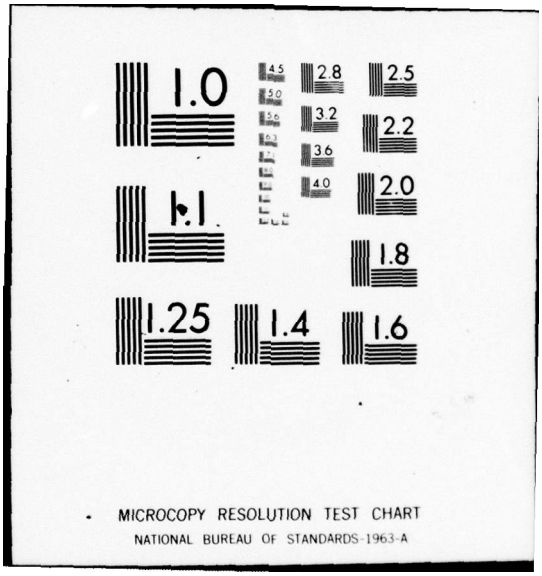
AD-A049 473 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SEMANOL(76) REFERENCE MANUAL. VOLUME II.(U)
NOV 77 F C BELZ
F30602-76-C-0238
UNCLASSIFIED RADC-TR-77-365-VOL-2 NL

1 of 2
ADA049473


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A049473

AD No.
DDC FILE COPY

RADC-TR-77-365, Vol II (of four)
Final Technical Report
November 1977

SEMANOL(76) REFERENCE MANUAL

F. C. Belz

TRW Defense and Space Systems Group

Approved for public release; distribution unlimited.

DDC
RECEIVED
FEB 3 1978
D

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-77-365, Vol II (of four) has been reviewed and approved for publication.

APPROVED: John M. Ives

JOHN M. IVES, Captain, USAF
Project Engineer

APPROVED: Alan R Barnum

ALAN R. BARNUM, Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

(A) TR-77-365-VOL-2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

(18) 1. REPORT NUMBER: RADC-TR-77-365, Vol II (of four)

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE *(and Subtitle)*: SEMANOL(76) REFERENCE MANUAL. Volume II.

(9) 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report.

6. PERFORMING ORG. REPORT NUMBER: N/A

(10) 7. AUTHOR(s): F. C. Belz

(15) 8. CONTRACT OR GRANT NUMBER(s): F30602- 76-C-0238

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
TRW Defense and Space Systems Group
One Space Park
Redondo Beach CA 90278

(16) 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
P.E. 63728F (17) 08
J.O. 55580840

(11) 11. CONTROLLING OFFICE NAME AND ADDRESS:
Rome Air Development Center (ISIS)
Griffiss AFB NY 13441

12. REPORT DATE: Nov[redacted]77 (12) 127.

13. NUMBER OF PAGES: 122

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same

15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Captain John M. Ives (ISIS)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

SEMANOL	standardization
SEMANOL(76)	language control
semantics	metalanguage
syntax	interpreter
language definition	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

SEMANOL(76) is a metaprogramming language designed specifically for use in writing formal, operational, specifications of the syntax and semantics of contemporary programming languages. The SEMANOL(76) Reference Manual provides a detailed description of the metaprogramming language. The context-free syntax is given in the SEMANOL(76) notation, while the context-sensitive constraints and semantics are given by prose text. This manual does not explain how SEMANOL(76) ought to be used when writing formal specifications.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

409 637

CONTENTS

I. Introduction. 1

II. General Overview 2

A. SEMANOL(76) program structure 2

B. Evaluation description. 3

C. SEMANOL(76) Data Types. 5

D. Global assignment sequence. 7

E. SEMANOL(76) Names 9

F. Sequences and parse trees 9

G. SEMANOL(76) Execution rules -- general. . . 10

H. SEMANOL(76) Arithmetic. 11
(1) Standard form numerals 11
(2) Sign-magnitude notation. 12
(3) General Notes on Arithmetic. 12

I. SEMANOL(76) Iterators 12

J. SEMANOL(76) Parameters. 14

III. Description of SEMANOL(76). 16

A. Format. 16

B. Conventions and notation. 17

C. Definitions 19
SEMANOL(76) Program Structure 19
Declarations. 21
Control Statements. 24
Syntactic Definitions 29
Semantic Definitions. 35

Appendix A. 99

ACCESSION for
NTIS White Section X
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

D D C
RECEIVED
FEB 3 1978
D

PRECEDING PAGE BLANK

SEMANOL(76) Reference Manual

I. Introduction

I. Introduction

This manual is a description of the semantic definition metalanguage known as SEMANOL(76). It is intended to be as complete and precise a description of SEMANOL(76) as can be achieved using English. It is not a tutorial; it presupposes considerable familiarity with the underlying ideas of SEMANOL(76).

The presentation is in a "top-down" sequence -- many structures are defined in terms of structures not yet defined; thus it is intended for use by experienced (or at least well versed) SEMANOL(76) programmers.

The manual itself consists of three chapters, including this Introduction. Chapter II consists of a basic description of the elements of SEMANOL(76) programs. Details regarding the Translator conventions will be listed here as well.

Chapter III consists of the full SEMANOL(76) language description. The grammar will be used as a guide to the development of the chapter. Thus it is again assumed that the reader has basic familiarity with the SEMANOL(76) notation for grammars.

II. General Overview

A. SEMANOL(76) program structure

A SEMANOL(76) program may contain four sections, each consisting of a sequence of basic constructs.

The Declarations section consists of a collection of declarations, by which Syntactic components and Global variables can be identified.

The Context free syntax section contains a collection of Syntactic definitions, which are used to specify an (almost) context free grammar for a language (i.e. a set of strings) over the ASCII character set. Actually, several possibly independent grammars can be included in this section. Each syntactic definition serves to identify a set of strings with a syntactic class name, which can then be used in several ways in SEMANOL(76). Any syntactic class name may be used as the start symbol for a grammar.

The Semantic definitions section contains Semantic definitions, which are operationally similar to function declarations in other programming languages. Semantic definitions consist of a Semantic definition name followed by an optional Dummy parameter list followed by a Semantic definition body. A semantic definition may be "functional" or "procedural"; i.e., it specifies the selection of a SEMANOL(76) expression which is to be evaluated (functional), or it specifies a sequence of Statements which are to be executed in order (procedural). Semantic definitions may be directly or indirectly recursive.

The Control section contains a list of statements exactly like those in procedural semantic definitions; the statements are to be executed in order. Statements may contain semantic definition references within them -- in most reasonable SEMANOL(76) programs at least one statement will have such an imbedded reference. At least one such statement must appear here to begin the interpretation. In practice, several are used to specify the major steps in the interpretation of a program in the object language.

II. General Overview

B. Evaluation description

The evaluation of a SEMANOL(76) program can be described on a formal or symbolic basis; i.e., the SEMANOL(76) operations are performed on certain constants for underlying objects, producing new constants standing for other objects.

In this manual, a less formal description method is used in which evaluation is described as the process of scanning the text of the SEMANOL(76) program and producing objects which serve as the values of expressions.

For example, the SEMANOL(76) expression "a #CS b" where a and b are SEMANOL(76) expressions would be evaluated as follows: evaluate a -- i.e. create the object, $\tilde{a}$, specified by a; evaluate b similarly; evaluate #CS by finding the sequence concatenation operator denoted by #CS and applying it to $\tilde{a}$ and $\tilde{b}$ to form a resultant object, say $\tilde{c}$.

If the expression x is a primitive constant then the value, $\tilde{x}$, of x is the object,
$\hat{x}$, denoted by the constant.

[NOTATION: The "hat", ^, symbol is used to designate the object or function or relation denoted by a constant, and the "tilde", ~ , is used to designate the value of an expression. However, we may also write of the function, $\hat{x}$, using the phrase: the "x" function.]

SEMANOL(76) programs are executed. The process of execution associates values with expressions in the SEMANOL(76) program.

SEMANOL(76) expressions are built up from object, function, and relation constants, and names, with precedence rules implied in the normal manner by the official SEMANOL(76) grammar.

SEMANOL(76) syntactic expressions are used to form context free grammars, and are discussed elsewhere.

SEMANOL(76) semantic expressions are evaluated in the manner described below:

The simplest expressions consist of object constants or names alone. The evaluation of a name depends upon whether the name is a syntactic definition name, a semantic definition name, a declared global variable name, a dummy parameter name, or a dummy variable name. Syntactic definition names can only appear in more complex semantic expressions (keyword

expressions, discussed below) and the semantics of their evaluation depends upon the keyword expression in which they appear.

Semantic definition names may appear with or without an actual argument expression list, and are termed "semantic def calls", "#PROC-DF calls", or "#DF calls"; evaluation of such calls is analogous to function calls in most algebraic languages and is discussed in Chapter III in the discussion of Semantic-definitions.

The evaluation of a declared global variable name, x, in a semantic expression is synonymous with the evaluation of "#LATEST-VALUE('x')"; i.e., the result value is the latest value assigned to "x" in the global assignment sequence (see Section D, "Global assignment sequence", below).

The evaluation of a dummy parameter name (of a semantic definition) produces as a result the value of the corresponding actual argument expression in the invoking semantic def call (see the discussion of Semantic-definitions in Chapter III).

Dummy variable names appear in the scope defined by a high-level-iterator, discussed below. The value of any such dummy name (upon evaluation of the expression in which it appears) is determined by the semantics of the specific high-level-iterator, as discussed in detail in Chapter III. Also see the last paragraph of this section.

In addition to names, a SEMANOL(76) expression may include function constants, which are function keywords such as "+", "-", "*", "#ABS", #TH-ELEMENT-IN", etc.; each denotes a specific function which has a certain number of arguments, and these arguments are represented in the SEMANOL(76) program by (sub)expressions called "operand expressions" of the function keyword or operator constant.

Each step in the evaluation of a complicated expression involves the application of one function constant to its operand expressions. We call the expression being evaluated at this step a keyword expression.

Normally, the rule for evaluating a keyword expression is the call-by-value rule discussed above. That is, the operand expression(s) are evaluated, left - to - right, creating argument(s) as value(s); the function denoted by the function keyword is applied to the argument(s), yielding a new object as the result value of the keyword expression.

Some keywords are evaluated in a (partial) call-by-name fashion -- these include the high-level-iterator keywords. In these cases, one of the operand expressions (usually the last) is passed to the function as an argument -- the operand expression is not evaluated first. Then the function itself may call for evaluation of the operand expression or some modified version of it.

C. SEMANOL(76) Data Types

The objects which can be values of SEMANOL(76) expressions may be partitioned into five classes called Primary data types:

-Boolean objects: <true> and <false>;

-Undefined objects: <undefined>;

-String objects;

-Sequence objects;

-Parse tree or Node objects;

Boolean, undefined and string objects are said to be unstructured or simple objects; sequence and parse tree objects are said to be structured objects. The elements of a sequence may be objects of any type; the parse tree objects are constructed of Nodes, and any parse tree is uniquely represented by its Root node. Furthermore, any node is the root node for some unique parse tree. Thus to say that an object is a parse tree is in some sense equivalent to saying that the object is a node. The SEMANOL(76) interpreter represents a parse tree by its root node.

In addition, there are two SEMANOL(76) functions, <stop> and <error> (denoted by #STOP and #ERROR), which when invoked during evaluation, cause evaluation to stop; in the case of <error>, an error message is written on a standard output file. We will sometimes speak of expressions denoting or producing the value <error> (or <stop>) which is the same as saying that the function <error> (or <stop>) is invoked in the evaluation process.

Since the evaluation of expressions causes the construction of objects from the above five data types, expressions are said to have the type of the object which is created upon their evaluation. Thus we may speak of "string expressions", etc.

II. General Overview

The classification of SEMANOL(76) objects into primary data types is based upon the SEMANOL(76) functions. Every function accepts 0 or more arguments, each an object of one of the data types. A function constant is said to accept its operand expressions iff its function accepts the arguments obtained by evaluating the operand expressions.

In many cases, if a function accepts an object of type a as its n-th argument, it will not accept an object of type b (b not equal to a) as that argument. This rule is relaxed in certain cases. Some functions will accept arguments of any type. Others will nominally accept only string arguments-- for these, however, node arguments are accepted by convention, and the operator denoted by the keyword "#STRING-OF-TERMINALS" is implicitly applied to convert the node to its string of terminals before evaluation of the function itself.

Certain functions will accept only a subset of the strings -- those having a certain (numeric) syntax. Thus a collection of secondary data types (or syntactic or numeric data types) is induced:

-Bit string;

-Integer numeral;

The allowable syntax for members of each data type is given in Chapter III. Certain function constants will accept only bit-string expressions (e.g. #BOR, #BXOR), others will accept only integer expressions (e.g., +, -, #SUBSTRING-OF-CHARACTERS m #TO n #OF s (for operand expressions m and n), etc.

<error>

Each function has 0 or more arguments and, given those arguments, produces an object of one of the data types listed above. Each operator accepts for each argument a SEMANOL(76) object from one of the data types listed above. If during the execution of a SEMANOL(76) program, an object is used as an argument and the object is not of the correct type, the operator does not accept the argument and produces the value <error>, which causes immediate termination of the execution of the SEMANOL(76) program.

The keyword #CONTEXT-FREE-PARSE-TREE will accept as its first operand any SEMANOL(76) expression denoting a string, and as its second operand a Syntactic class name which is the start

symbol for an unambiguous grammar. However, it is impossible to test the grammar for ambiguity a priori; therefore, a grammar can be established as ambiguous only when a parse is attempted for the given string; in this case the keyword expression evaluates to <error>. For a different given string the same grammar may provide for an unambiguous parse; in this case the operator constructs the parse tree which results from the parse. For another string, the same grammar may admit no parse at all; in this case the result would be <undefined>.

D. Global assignment sequence

An executing SEMANOL(76) program can manipulate a structure called the Global assignment sequence which can be thought of as a sequence of pairs (𝕊,𝕍), where 𝕊 is a string called the receiving string (or global SEMANOL(76) variable) and 𝕍 is a value of any legal SEMANOL(76) data type. The primary operators which manipulate the global assignment sequence are denoted by #ASSIGN-LATEST-VALUE(s,v), and #LATEST-VALUE(s).

The value of the expression "#ASSIGN-LATEST-VALUE(s,v)" is the empty string; as a side effect of the evaluation of this keyword, the pair (𝕊,𝕍) is added to the global assignment sequence.

The value of "#LATEST-VALUE(s)" is the 𝕍 in the first pair (𝕊,𝕍) found on a reverse order search through the global assignment sequence at the time of evaluation of the keyword. If the search fails to find such a pair, the value is <undefined>.

Note that the "receiving string" may be designated by any SEMANOL(76) string expression. Thus, the set of receiving strings employed in the execution of a SEMANOL(73) specification may not be fixed at the start of execution; during execution, new ones may be constructed whose form depend upon the object program and its input.

An abbreviation for the two basic assignment operators is available in SEMANOL(76), using declared global variables. A receiving string which has the syntax of a SEMANOL(76) name may be declared a global variable. Such a name must appear in a "#DECLARE-GLOBAL" declaration. Then it may appear in a (semantic) expression; it may also appear as the assigned-to variable immediately after the "#ASSIGN-VALUE!" of a SEMANOL(76) assignment statement.

If a declared global variable name (say x) appears in any

semantic expression it is an abbreviation for the semantic expression "#LATEST-VALUE('x')". The statement "#ASSIGN-VALUE! x = semantic-expression" is an abbreviation for the statement "#COMPUTE! #ASSIGN-LATEST-VALUE ('x', semantic-expression)".

The following points should be noted:

-Writing a declared global variable is then a matter of writing the receiving string itself, rather than a SEMANOL(76) expression whose value is the string.

-When the declared global variable mechanism is being used, two methods may be used interchangeably to establish the latest value: (1) writing the declared global variable alone, or (2) using the #LATEST-VALUE keyword with the receiving string enclosed in primes.

-Thus in the SEMANOL(76) expressions below:

" 'AB' #CW 'CD' " denotes the string "ABCD";

" 'AB' #CW CD " denotes the string "ABEF" if CD is a declared global variable and #LATEST-VALUE('CD') denotes the string "EF" (i.e., if " 'AB' #CW #LATEST-VALUE('CD') " denotes "ABEF").

-When the declared global variable mechanism is being used, setting a new latest-value for a given receiving string may be done either (1) via the "#ASSIGN-VALUE!..." statement (writing the declared global variable itself as the receiving string) or (2) via the "#ASSIGN-LATEST-VALUE" keyword expression with the receiving string in string quotes (or denoted by a more complicated string expression). Note that method (1) may only be used in procedural semantic definitions or in the control commands section, while method (2) may be used in any SEMANOL(76) expression. This limitation, as well as the need to know the receiving string a priori, tends to limit the use of declared global variables to retaining control related information which is independent of individual object programs.

-The use of declared global variables is subject to certain context-sensitive syntactic constraints in a SEMANOL(76) program: the variable names must not conflict with any other names (see SEMANOL(76) names). The purpose of the "#DECLARE-GLOBAL" declaration is to identify the variable name as a

declared global variable name, thus facilitating the process of distinguishing it from a parameterless semantic definition name.

The global assignment sequence is completely independent of all local variables (see Section E. "SEMANOL(76) names", below). Thus, no variables with a local name (e.g., dummy variables of high-level-iterators) can ever be explicitly assigned a value via the "#ASSIGN-VALUE!..." or "#ASSIGN-LATEST-VALUE..." operators.

E. SEMANOL(76) Names

A unique name must be used for each syntactic and semantic definition name and declared global variable name. Definition and global variable names are collectively called global names. Each global name can be given to at most one of the definitions and global variables.

Names used for semantic definition dummy parameter names and for dummy variable names in such things as "#FOR-ALL..." loops are called local names. The set of local names must be distinct from the global names.

Each dummy parameter and dummy variable has a scope. For example, the scope of a semantic definition dummy parameter begins with "#DF" and ends with the closing "#." of the definition. The scope of a "#FOR-ALL..." statement dummy variable is the <Compound-statement>.

The same local name cannot be used for any two dummy parameters or variables which have overlapping scopes. Otherwise, local names need not be distinct.

The global assignment sequence is completely independent of all local variables. Thus, no variables with a local name (e.g., dummy variables of high-level-iterators) can ever be explicitly assigned a value via the "#ASSIGN-VALUE!..." or "#ASSIGN-LATEST-VALUE..." operators.

F. Sequences and parse trees

The structured objects called sequences are of a very general nature: their elements can be objects of any allowable SEMANOL(76) data type, including sequences. However, SEMANOL(76) sequences may not be re-entrant (i.e., none of the sequence elements of a sequence may be the sequence itself, nor

may any elements of the (sequence) elements of the sequence be the sequence itself,... etc.).

Suppose two separate evaluations produce x̃ and ỹ, each of which is a sequence. If "#LENGTH(x) = #LENGTH(y)" and "i #TH-ELEMENT-IN(x) #EQ i #TH-ELEMENT-IN(y)" for all i such that "1 <= i" and "i <= #LENGTH(x)", then x̃ is identical to ỹ. That is, the identity of a sequence is completely determined by the order and identity of its components. Here the notion of identity is defined in terms of the SEMANOL(76) operators for equality (the node-, sequence-, string-, and numeric-equality relations).

Every SEMANOL(76) parse tree is uniquely represented by its root node; however, parse trees are not necessarily disjoint. A parse tree ã "includes" parse tree b̃ if every node contained in b̃ is contained in ã. Also ã "properly includes" b̃ if ã includes b̃ and there is some node contained in ã but not in b̃. A parse tree is "most-inclusive" if it is not properly included in any parse tree. All most-inclusive parse trees are disjoint (they contain no nodes in common); any two (sub)trees, ã and b̃, both included in a most-inclusive tree, have the property that either ã properly includes b̃ or b̃ properly includes ã or ã is identical to b̃.

Each result of the "#CONTEXT-FREE-PARSE-TREE" function is (the root-node of) a unique most-inclusive parse tree; given any node in a most-inclusive parse tree, any other node in the tree can be obtained by composing certain SEMANOL(76) functions. Any node is the root node of a parse tree (possibly a subtree of a most-inclusive tree), and corresponds to a particular string in a set defined in the context-free syntax section; that string can be obtained via the "#STRING-OF-TERMINALS-OF" function.

G. SEMANOL(76) Execution rules -- general

Execution of a SEMANOL(76) program begins with the first statement in the Control section. When execution of that statement is completed the next sequential statement is executed; execution proceeds in this sequence unless the instruction list is exhausted, in which case execution terminates, with a standard error message.

Each statement is executed according to the semantics specified in Chapter III of this manual. Normally this will involve the evaluation of a SEMANOL(76) expression; the precise rules of construction for SEMANOL(76) expressions and rules of

evaluation are given in Chapter III also. (A general discussion appears in this chapter, Section B, "Evaluation description", above.)

H. SEMANOL(76) Arithmetic

Certain arithmetic functions are performed on numeric (string) values to produce new numeric values. It is important to note that the argument and result objects are always strings, albeit strings with a special syntax (as described in Section "SEMANOL(76) Data Types" above). However in many cases they may be thought of most naturally as numbers (for example, "37 + 21" evaluates as follows: construct the argument strings "37" and "21" and apply the "+" operator, which produces the result value--also a string--"58").

Numerals must be strings which satisfy the syntax given in Chapter III for <Numeric-constant>. Integer numerals consist of an "integer portion" possibly followed by a base suffix. (Negative integer numerals have a leading minus sign.)

Bit string numerals consist of a (binary) "integer portion" followed by a "#BITS" suffix.

Each integer arithmetic operator produces, as a result, an integer numeral of base 10, 8, or 2. To define these operators uniquely, a standard form numeral must be defined for each of these classes, which is the form of all result values of the arithmetic operators.

(1) Standard form numerals

A non-zero numeral string is in standard form if the leading zeros are suppressed in the constant.

Standard form zeros are:

Integers

0

0#B8

0#B2

(2) Sign-magnitude notation

The arithmetic operations can be defined formally as string operations, but we shall use a less formal means in this manual, referring to the underlying numbers obviously denoted by the SEMANOL(76) numeral strings. Sign-magnitude notation (with explicit "-" sign) is used for SEMANOL(76) numerals.

At the risk of belaboring the obvious, the rules for defining the number, X, named by an numeral x, are given in the next two paragraphs.

The number associated with any integer "Xn:::X1X0" is given by the polynomial Xn*b!n+...+X1*(b) X0*(1) where + stands for addition, * stands for multiplication, ! stands for exponentation and b is the base of the numeral.

An unsigned integer numeral, i, denotes the number, I, associated with the integer portion of i. Thus "13", "15#B8" and "1101#B2" all denote the integer with decimal representation "13". If i has a leading minus sign, I is negated ("-1101#B2" denotes the integer with decimal representation "-13").

(3) General Notes on Arithmetic

Integer arithmetic operators accept base 2, 8, or 10 arguments interchangeably. If the operators are binary arithmetic operators such as +, -, * or /, the result is in the minimal base among the arguments if the bases of the arguments differ; otherwise the result is in the base of the arguments.

All integer arithmetic operators produce standard-form result numerals; this guarantees uniqueness.

All arithmetic is officially of unbounded precision. There is no maximum precision of integer operands and results, for example. (See Section J, "SEMANOL(76) Parameters".)

I. SEMANOL(76) Iterators

Each SEMANOL(76) iterator consists of three parts: initial keyword, iteration control clause, and iterated clause. The iterators are identified by their initial keyword: #FOR-ALL, #WHILE (iterator commands), #FOR-ALL, #THERE-EXISTS,

n #TH, #FIRST, #LAST, #SUBSEQUENCE-OF-ELEMENTS, #SEQUENCE-OF-NODES (iterator expressions). The iteration control clause identifies either (1) a control expression (in the #WHILE command), or (2) a dummy variable name and an iteration control sequence; the control sequence may be explicit ("dummy-var #IN s", where s is a sequence), or implicit ("dummy-var #IN n", where n is a parse tree node and the implied sequence is the preorder sequence of nodes in n; "dummy-var: lb < = dummy-var < = ub, where the implied sequence is the sequence of integers: lb, lb + 1,...,ub. If the upper bound is deleted, the implied sequence of integers becomes infinite). For iterator commands, the iterated clause has the form "#DO compound-stmt"; for iterator expressions, the form is either "#SUCH-THAT (bool-exp)" or "#IT-IS-TRUE-THAT (bool-exp)".

The #WHILE command is evaluated in the following manner: the expression in the control clause is evaluated. If the result is (1) <true>, then the iterated clause is evaluated, and then the entire #WHILE command is re-evaluated; (2) <false>, then the termination condition has been achieved, and control is given to the statement following the iterated clause; (3) any other value, then the result of the #WHILE command is <error>.

The other iterators are evaluated in the following manner: the expression(s) in the control clause are evaluated (in left-to-right order). If the implied sequence is empty, a default action is taken: iterator commands are considered complete and control is given to the statement following the iterated clause; iterator expressions return a default value which depends upon the particular iterator (e.g., #FOR-ALL returns <true>, #THERE-EXISTS returns <false>, #FIRST returns <undefined>, #SUBSEQUENCE-OF-ELEMENTS returns <nilseq>). Otherwise, the compound-statement or boolean-expression of the iterated clause is repeatedly evaluated until the termination condition for the particular iterator is achieved. In each evaluation of the iterated clause, any occurrences of the dummy variable name are taken to denote an element of the control sequence; in the first iteration, the first element of the control sequence is denoted by the dummy variable (except in the #LAST iterators, where the last element is denoted); in the nth iteration, the nth element is denoted (nth from the last, for #LAST).

For the #FOR-ALL command, and the #SUBSEQUENCE-OF-ELEMENTS and #SEQUENCE-OF-NODES expressions, the iteration termination condition is reached just after the last element of the control sequence has been denoted in an iteration. For the other

iterator expressions, (n#TH, #FIRST, #LAST, #FOR-ALL, #THERE-EXISTS) the termination condition is reached just after the evaluation of the iterated clause first (nth for n#TH) yields <true>. Thus it may be the case that not all elements of the control sequence are used as the denoted value. Since side effects may occur as the result of any evaluation of the iterated clause, this termination rule may be significant in a SEMANOL(76) specification.

It is possible for an iterator to result in an <error> in the following ways: (1) the expression evaluation(s) in the control clause yield values inappropriate for the iterator, (2) evaluation of the iterated clause yields <error>, and (3) evaluation of the iterated clause yields a non-logical value (i.e., neither <true> nor <false>).

J. SEMANOL(76) Parameters

In this reference manual, no provision is made for parameters which depend upon a particular SEMANOL(76) language processor. (Such parameters as bits-per-word and maximum-precision-of-multiply are common for other implemented languages). Most parameters which can be simply characterized are introduced into programming language semantics to allow for certain space and time optimizations. However, the theoretical and practical expository benefit of such parameters in a metalanguage is quite doubtful. For this reason, we have adopted certain principles.

First, every SEMANOL(76) program should be written as if there were no restrictions upon the resources available to represent any SEMANOL(76) object, function, or relation. The "official" SEMANOL(76) language is itself defined without such restrictions; any "official" SEMANOL(76) implementation will process the "official" SEMANOL(76) language. However, unofficial implementations may be useful, especially if the following rule is applied: all limitations will be expressed in terms of a (large) class of objects rather than in terms of particular objects. For example, it may be useful to limit the maximum total number of string characters in use at any point in the execution of a SEMANOL(76) program, but it is much less usefel to limit the length of individual strings (by, for example, imposing a limit on the size of numeral strings as arguments of numeric operations).

Note that there is no theoretical reason that such constraints cannot be stated more precisely (by, for example,

assigning a resource requirements measure to each SEMANOL(76) keyword expression, and defining the effect of each SEMANOL(76) operator conditionally upon the availability of its required resources); however, we feel this to be unnecessary at present. Also, even in the presence of limited SEMANOL(76) implementations, we believe that SEMANOL(76) programs should be written as if such constraints were not present at all.

III. Description of SEMANOL(76)

III. Description of SEMANOL(76)

This chapter consists of detailed definitions of the SEMANOL(76) program constructs. The organization follows a "top-down" grammar for the SEMANOL(76) language.

A. Format

The text of this chapter in sections, each consisting of a portion of the SEMANOL(76) grammar, followed by a series of (Keyword schema, explanation) pairs for the Keywords just introduced in the grammar. The sections of the text have the following form:

Portion of the grammar

Keyword schema

> Explanation; i.e., a description of the result produced by evaluating a keyword expression having the format of the keyword schema.

Next keyword schema

> Next result description.

. . . . etc.

(Sometimes general discussions of complex points are interspersed with the (Keyword schema, explanation) pairs.)

III. Description of SEMANOL(76)

B. Conventions and notation

The conventions used in this chapter are derived from those in Chapter II: if x is a meta variable standing for a SEMANOL(76) expression, then

$\tilde{x}$ stands for the value of the expression, and

if x is a constant,

$\hat{x}$ stands for the object denoted by x.

There are certain special meta constants used to denote special SEMANOL(76) objects:

<true> meaning the boolean value, true;

<false> meaning the boolean value, false;

<nil> meaning the empty string;

<nilseq> meaning the empty sequence.

We also use <error> and <stop> to indicate invocation of the error and stop functions respectively, which terminate the evaluation process; the error function also causes an error indication to be written on the standard error file.

The notation "B[x]" is used in the description of the high level iterator keywords to designate the iterated Boolean expression B with every occurrence of the dummy variable name replaced with x, a constant for an object $\tilde{x}$.

A SEMANOL(76) object is said to be <u>string convertible</u> if it is a
string or a parse tree node.

If x denotes a string convertible object, $\tilde{x}$, then the "string value of x" is $\tilde{x}$ if $\tilde{x}$ is a string, and the string of terminals corresponding to $\tilde{x}$, if $\tilde{x}$ is a parse tree node.

Some SEMANOL(76) constructs are defined in terms of others. In some result descriptions for keyword schemata certain conditionals will be written as SEMANOL(76) expressions for precision or clarity, and they are said to hold if, when evaluated, they would return <true>. For example, the

expression "a = b" is written instead of ā equals b̄ because it is more precise -- it means evaluate the expression "a = b" according to the SEMANOL(76) evaluation rules, and if the result is <true> then the condition ā equals b̄ is assumed to hold and the condition is satisfied.

III. Description of SEMANOL(76)

C. *Definitions*

```
"******************************

SEMANOL(76) Program Structure

 ******************************"
```

```
#DF SEMANOL-76-program

        => <gap> <Declaration-section>
        <Optional-context-free-syntax> <gap>
        <Control-section>
        <Optional-semantic-definition-section> <gap>

        => <gap> <Declaration-section>
        <Optional-context-free-syntax> <gap>
        <Semantic-definition-section> <gap>
        <Control-section> <gap> #.

#DF Optional-context-free-syntax

        => <#NILSET> #U <<gap> <Context-free-syntax>> #.

#DF Optional-semantic-definition-section

        => <#NILSET> #U <<gap>
        <Semantic-definition-section>> #.
```

A SEMANOL(76) program may contain four sections, each consisting of a sequence of basic constructs.

The "Declarations section" consists of a collection of declarations, by which "Syntactic components" and "Global variables" can be identified.

The Context free syntax section contains a collection of

Syntactic definitions, which are used to specify an (almost) context free grammar for a language (i.e. a set of strings) over the ASCII character set. Actually, several possibly independent grammars can be included in this section. Each syntactic definition serves to identify a set of strings with a Syntactic class name, which can then be used in several ways in SEMANOL(76). Any syntactic class name may be used as the start symbol for a grammar.

The Semantic definitions section contains Semantic definitions, which are operationally similar to function declarations in other programming languages. Semantic definitions consist of a Semantic definition name followed by an optional Dummy parameter list followed by a Semantic definition body. A semantic definition may be functional or procedural; i.e., it specifies the selection of a SEMANOL(76) expression which is to be evaluated (functional), or it specifies a sequence of Statements which are to be executed in order (procedural). Semantic definitions may be directly or indirectly recursive.

The Control section contains a list of statements exactly like those in procedural semantic definitions; the statements are to be executed in order. Statements may contain semantic definition references within them -- in most reasonable SEMANOL(76) programs at least one statement will have such an imbedded reference. At least one such statement must appear here to begin the interpretation. In practice, several are used to specify the major steps in the interpretation of a program in the object language.

The Context free syntax section and the Semantic Definitions section are optional. The Declarations section may be empty. Thus the only absolutely required section of a SEMANOL(76) program is the Control section.

III. Description of SEMANOL(76)

```
"**************************

  Declarations

 **************************"
```

```
------------------------

 #DF Declaration-section

          => <%<Declaration>> #.

 #DF Declaration

          => <Declare-syntactic-components>

          => <Declare-global-variables>  #.
------------------------
```

The "Declarations section" consists of a collection of declarations, by which "Syntactic components" and "Global variables" can be identified.

```
------------------------

 #DF Declare-syntactic-components

          => <'#DECLARE-SYNTACTIC-COMPONENT'> <gap> <':'>
          <gap> <Semantic-definition-name> <gap> <%<<Comma>
          <gap> <Semantic-definition name> <gap>>> <'#.'>
          #.
------------------------
```

Certain semantic definitions may be declared to be "syntactic components" by having their names listed in a declaration of this type. To appear in such a declaration, a semantic definition must have these properties: (1) it must take exactly one argument, (2) it must in no way depend upon any global variable or hidden input variable, or external function. This last qualification is a static one -- i.e., (1) no reference, whether or not it is actually executable, may

appear to a global variable, or to "#INPUT" or to "#EXTERNAL-CALL-OF"; nor (2) may any semantic definition reference appear to a semantic definition which violates this qualification (parts 1 & 2).

Any reference to a syntactic component definition must be evaluated to have a parse tree node as the actual argument; if not, the result of the reference is <error>.

These restrictions are sufficient to guarantee that the evaluation of a syntactic-component reference can only involve the composition of SEMANOL functions upon (1) SEMANOL(76) object constants and (2) the parse tree containing the actual argument node. This implies that the result of every syntactic component is a function of the parse tree of the actual argument node, and therefore is truly syntactic in nature.

Furthermore, the evaluation of all references to a particular syntactic-component with a particular node argument will return identical results.

Therefore, an optimization of the evaluation of SEMANOL(76) syntactic-component references is possible: for every (syntactic-component, actual node) pair the normal evaluation rules need be followed only once to produce a result value; thereafter a simple table lookup can be used to determine the result value. This concept is implemented by associating with each node on a parse tree a potential list of pairs, each of the form: (s, $\tilde{v}$), where s is the semantic definition name of a syntactic component and $\tilde{v}$ is the unique result value of evaluating any reference to the named syntactic-component with the node as argument. A pair (s, $\tilde{v}$) is added to the list as a side effect of the first such evaluation of a reference to syntactic component, s, with the node as argument; every subsequent reference to syntactic component, s, with the node as argument will be evaluated by searching the list of pairs for (s, $\tilde{v}$) and returning the value $\tilde{v}$.

```
------------------------

    #DF Declare-global-variables

=> <'#DECLARE-GLOBAL'> <gap> <':'> <gap> <Name> <gap>
<%<<Comma> <gap> <Name> <gap>>> <'#.'> #.
------------------------
```

An abbreviation for the two basic assignment operators is available in SEMANOL(76), using "declared global variables". A receiving string which has the syntax of a SEMANOL(76) "name" may be declared a global variable. Then it may appear in a (semantic) expression; it may also appear as the assigned-to variable immediately after the "#ASSIGN-VALUE!" of a SEMANOL(76) assignment statement.

If a declared global variable name (say x) appears in any (semantic) expression it is an abbreviation for the semantic expression "#LATEST-VALUE('x')". The statement "#ASSIGN-VALUE! x = Semantic-expression" is an abbreviation for the statement "#COMPUTE! #ASSIGN-LATEST-VALUE ('x', Semantic-expression)".

"**************************

Control Statements

**************************"

```
----------------------

#DF Control-section

            => <'#CONTROL-COMMANDS'> <gap> <':'> <gap>
            <%1<<Compound-statement> <gap>>> <'#.'> #.
----------------------
```

The "Control section" contains a list of statements exactly like those in procedural semantic definitions; the statements are to be executed in order. Statements may contain semantic definition references within them -- in most reasonable SEMANOL(76) programs at least one statement will have such an imbedded reference. At least one such statement must appear here. In practice, several are used to specify the major steps in the interpretation of a program in the object language.

Execution of a SEMANOL(76) program begins with the first statement in the Control section. When execution of that statement is completed the next sequential statement is executed; execution proceeds in this sequence unless the statement list is exhausted, in which case execution terminates with a standard error message.

Note that the "RETURN-WITH-VALUE!" statement is not syntactically disallowed in the Control section, but may not be executed in the Control section (any attempt to do so will produce an <error>).

```
----------------------

 #DF Compound-statement

                => <For-statement>

                => <'#WHILE'> <gap> <Boolean-expression> <gap>
                <'#DO'> <gap> <Compound-statement>

                => <'#IF'> <gap> <Boolean-expression> <gap>
                <'#THEN'> <gap> <Compound-statement>

                => <'#BEGIN'> <gap> <Compound-statement> <gap>
                <%<Compound-statement> <gap>>> <'#END'>

                => <Simple-statement>  #.

 #DF For-statement

                => <For-all-clause> <gap> <'#DO'> <gap>
                <Compound-statement> #.

 #DF For-all-clause

                => <'#FOR-ALL'> <gap> <Name> <gap> <'#IN'> <gap>
                <Sequence-expression>

                => <'#FOR-ALL'> <gap> <Name> <gap> <':'> <gap>
                <<Bounded-interval> #U <Unbounded-interval>> #.

 #DF Bounded-interval

                => <String-expression> <gap> <'<='> <gap> <Name>
                <gap> <'<='> <gap> <String-expression> #.

 #DF Unbounded-interval

                => <String-expression> <gap> <'<='> <gap> <Name>
                #.
----------------------


 #FOR-ALL x #IN s #DO c

                <error> if s is not a sequence.

                Otherwise: perform the following steps in order.
```

(1) Construct a new dummy variable named x, with scope extending throughout c. Construct a new dummy variable named sx, with no scope. Evaluate s. Assign s̃ to sx.

(2) If s̃ is <nilseq>, the empty sequence, go immediately to the statement following c. (Halt on <error> if there is no following statement.) Otherwise,

(3) Assign first element in s̃ to x; Assign tail of s̃x to sx. [Note: tail of s̃x is given by #TERMINAL-SUBSEQ-OF-LENGTH (#LENGTH(sx) - 1) #OF sx".]

(4) Execute Compound-statement c; go to 2.

#FOR-ALL x: a < = x < = b #DO c

#FOR-ALL x: a < = x #DO c

<error> if "a #IS-NOT #INTEGER" or (if b is given) "b #IS-NOT #INTEGER".

Otherwise: perform the following steps in order

(1) Construct a new dummy variable named x, with scope extending throughout c. Evaluate a and assign ã-1 to x. If b is given, evaluate b and save b̃ for later use.

(2) Assign x̃+1 to x. If b is given, and if x̃ > b̃, go immediately to the statement following c. (Halt on <error> if there is no following statement.) Otherwise,

(3) Execute compound statement, c; go to 2.

#WHILE b #DO c

Perform the following steps in order.

(1) Evaluate b

(2) If b̃ is <true>, then execute compound statement, c and then go to step 1.

(3) If b̃ is <false>, go immediately to the statement following c. (Halt on <error> if

there is no following statement).

(4) Otherwise, halt on <error>.

#IF b #THEN s

If "b #EQ #TRUE", execute Compound-statement s.

If "b #EQ #FALSE", proceed to the statement following s. (Halt on <error> if there is no following statement.)

Otherwise, halt on <error>.

#BEGIN s1 s2 ... sn #END

Compose the block of statements s1,s2,...,sn into a single Compound-statement.

Execution of the composed block proceeds by executing s1,s2,...,sn in order.

#DF Simple-statement

=> <'#ASSIGN-VALUE'> <gap> <'!'> <gap> <Name> <gap> <'='> <gap> <Expression>

=> <'#COMPUTE'> <gap> <'!'> <gap> <Expression>

=> <'#RETURN-WITH-VALUE'> <gap> <'!'> <gap> <Expression> #.

#ASSIGN-VALUE! a = e

Note: a must be a declared global variable, and e must be a semantic expression.

Evaluate semantic expression e, assign resulting object to declared global variable a.

This statement is an abbreviation for "#COMPUTE! #ASSIGN-LATEST-VALUE('a',e)".

#COMPUTE! e

Evaluate semantic expression e.

#RETURN-WITH-VALUE! e

<error> if this statement is executed in the control section.

Evaluate e. Terminate evaluation of the procedural semantic definition containing this statement, returning ẽ as the value of the semantic definition reference.

III. Description of SEMANOL(76)

```
"**************************

    Syntactic Definitions

**************************"
```

```
----------------------

#DF Context-free-syntax

        => <'#CONTEXT-FREE-SYNTAX'> <gap> <':'> <%<<gap>
        <Syntactic-definition>> #.
----------------------
```

The "Context free syntax section" contains a collection of "Syntactic definitions", which are used to specify an (almost) context free grammar for a language (*i.e. a set of strings*) over the ASCII character set. Actually, several possibly independent grammars can be included in this section.

```
----------------------

#DF Syntactic-definition

        => <'#DF'> <gap> <Syntactic-class-name> <gap>
        <%1<<'=>'> <gap> <Syntactic-expression> <gap>>
        <'#.'> #.

#DF Syntactic-class-name

        => <Name> #.
----------------------
```

Syntactic definitions serve as the productions of (one or more) grammars collected in the Context free syntax section. Each syntactic definition serves to declare the syntactic class name (or syntactic definition name) for use in the remainder of the SEMANOL(76) programs. For purposes of explication, we will speak of the grammars in this section as generative grammars:

i.e., the syntactic definitions represent rules for the construction of possibly infinite sets of strings (languages). In fact, the grammars are used in SEMANOL(76) as recognition grammars, or parsing grammars, and are employed in the semantics of the "CONTEXT-FREE-PARSE-TREE..." operator, and certain membership relations discussed below.

The terminals of SEMANOL(76) syntactic expressions are the strings specified by the syntactic set constants discussed below. SEMANOL(76) syntactic expressions need not be grounded; an expression is not grounded if by using the standard rewriting algorithm for grammars (see Appendix A), one cannot reduce the expression to a string of terminals. However, any such expression cannot be employed in the successful parse of a string using the "#CONTEXT-FREE-PARSE-TREE" operator. There is one ungrounded syntactic set constant: #EMPTYSET.

Each syntactic class name or syntactic definition name is then associated with a possibly infinite set of strings, called a syntactic class, by virtue of the syntactic definition which declares the syntactic class name. Each such syntactic definition has the form:

```
#DF c => b1
      => b2
      => b3
      .
      .
      .
      => bn  #.
```

where the c is a syntactic definition name, and the bi are syntactic expressions. The name c is then associated with the syntactic class formed by taking the union of the sets b1, b2, ..., bn, represented by the corresponding syntactic expressions.

It is assumed that any string x in the set, c̆, associated with c is in exactly one of the sets b̆i. If this is not so (i.e., x is in more than one b̆i), the definition of c is said to be ambiguous. Ambiguity is checked for only when parsing or recognizing a given string, as described above. Thus a grammar may be ambiguous, which is not considered legal in SEMANOL(76), but that fact may not be detected during the execution of the SEMANOL(76) program. A precise definition of ambiguity is

given in Appendix A.

```
#DF Syntactic-expression

              => <Syntactic-term> <%< <gap> <'#U'> <gap>
              <Syntactic-term>>>

              => <Syntactic-term> <gap> <'#S-'> <gap> < <'>
              <gap> <String> <gap> <%<<Comma> <gap> <String>
              <gap>>> <'>'> #.

#DF Syntactic-term

              => <Syntactic-primary> <%<<gap>
              <Syntactic-primary>>>

              => <'('> <gap> <Syntactic-expression> <gap> <')'>

              => <Syntactic-class-name>

              => <'%'> <gap> <'1'> <gap> <Syntactic-primary>

              => <'%'> <gap> <Syntactic-primary>

              => <Syntactic-set-constant>  #.
```

---

a #U b

The union of the sets ã and ƀ.

a #S- <'s1','s2',...,'sn'>

The set of strings obtained by removing the strings, s1,s2,...,sn from ã.

<b1><b2>...<bn>

The set concatenation of ñ sets of strings, ƀ1,ƀ2,...,ƀn; i.e., the set of strings {a: a = "a1a2...an", and "a1" is in ƀ1, and "a2" is in ƀ2, ..., and "an" is in ƀn}.

(e)

The set, ẽ, also designated by the SEMANOL(76) syntactic expression, e.

% <b>

The Kleene-star set, b̃*. That is, the set which might be designated by the infinite SEMANOL(76)-like syntactic expression, "#NILSET #U <b> #U <b><b> #U ... ".

%1 <b>

The subset of the Kleene-star set, b̃*, found by removing the empty string, <nil>.

```
#DF Syntactic-primary

        => <'<'> <gap> <Syntactic-expression> <gap> <'>'>

        => <'<'> <gap> <String> <gap> <%<<Comma> <gap>
        <String> <gap>>> <'>'> #.

#DF Syntactic-set-constant

        => <'#ASCII'>

        => <'#CAP'>

        => <'#DECNUM'>

        => <'#DIGIT'>

        => <'#EMPTYSET'>

        => <'#GAP'>

        => <'#LETTER'>

        => <'#LOWCASE'>

        => <'#NAT-NOS'>

        => <'#NILSET'>

        => <'#SPACESET'> #.
```

<e>

The set, $\bar{e}$, also designated by the SEMANOL(76) syntactic expression, e.

#ASCII

The set of all ASCII characters.

#CAP

The set of all ASCII capital letters, A through Z.

#DECNUM

The set of ASCII strings formed by following an optional plus or minus sign by one or more digits. That is, the set also given by the SEMANOL(76) syntactic expression "<#NILSET #U <'+', '-'>> <%1 <#DIGIT>>". (see the discussion of the "CONTEXT-FREE-PARSE-TREE..." operator, below -- especially Section 2. --, for a more detailed discussion of constraints on the use of this set constant.)

#DIGIT

The set of ASCII decimal digits, 0 through 9.

#EMPTYSET

The set containing no elements.

#GAP

The set consisting of strings of 0 or more ASCII blank characters; except that the empty string, <nil>, is excluded in alphanumeric left and right contexts. (Here we are speaking of syntactic set constants in terms of generative grammars; see Appendix A, Section 2, for a more detailed discussion of constraints on the use of this set constant.)

#LETTER

The set of all ASCII capital and lower case letters, A through Z, and a through z. That is, the set also designated by the SEMANOL(76) syntactic expression "<#CAP #U #LOWCASE>".

#LOWCASE

The set of ASCII lower case letters, a through z.

#NAT-NOS

The set consisting of strings of one or more ASCII digits. That is, the set also designated by the SEMANOL(76) syntactic expression "<%1 <#DIGIT>>". (see Appendix A, Section 2, for a more detailed discussion of constraints on the use of this set constant.)

#NILSET

The set consisting of the null string.

#SPACESET

The set consisting of the single ASCII blank character.

<s1,s2,...,sn>

The set , {s1,s2,...,sn}, of strings of ASCII characters

III. Description of SEMANOL(76)

"**********

Semantic Definitions

**********"

```
#DF Semantic-definition-section

        => <'#SEMANTIC-DEFINITIONS'> <gap> <':'> <%<<gap>
        <Semantic-definition>>> #.
```

The "Semantic definitions section" contains "Semantic definitions", which are operationally similar to function declarations in other programming languages. Semantic definitions consist of a "Semantic definition name" followed by an optional "Dummy parameter list" followed by a "Semantic definition body". A semantic definition may be "functional" or "procedural"; i.e., it specifies the selection of a SEMANOL(76) expression which is to be evaluated ("functional"), or it specifies a sequence of "Statements" which are to be executed in order ("procedural"). Semantic definitions may be directly or indirectly recursive.

```
#DF Semantic-definition

        => <Functional-definition>

        => <Procedural-definition>  #.

#DF Functional-definition

        => <Simple-definition>

        => <Definition-by-cases> #.
```

III. Description of SEMANOL(76)

Semantic definitions are invoked by evaluating "semantic definition references", of the form "c(a1,...,an)" or "($a1,...an$)c", where c is a semantic definition name, and the ai are SEMANOL(76) semantic expressions in an optional "actual argument expression list". The sequence of steps for evaluating a semantic definition reference is as follows:

(1) Evaluate the actual argument expressions in the semantic definition reference, from left to right (if there are any).

(2) Find the corresponding semantic definition (if there is none, halt on <error>). Construct nuw local (formal parameter) variables, given the formal parameter names (if there are any). If the number of values does not equal the number of parameter variables, halt on <error>. Assign the values of the argument expressions to the formal parameter variables (if there are any).

(3) (a) If the semantic definition name has not been declared a syntactic component name in the declarations section, then do (c); otherwise do (b).

(b) If the first actual argument value is a parse tree node, n, and if the syntactic component pair, (c,v) is attached to n, v becomes the result value, which is the value of the semantic definition reference. Otherwise, do (c).

(c) Evaluate the semantic definition body as described in (4) below, giving a result value which is the value of the semantic definition reference. If the semantic definition has been declared a syntactic component, and if the first actual argument value is a parse tree node (which has no syntactic component pair (c,v) attached), then attach to the node the pair (c,v1) where v1 is the result value.

(4) This step is described in detail in the discussions below for each type of semantic definition.

#DF Simple-definition

=> <'#DF'> <gap> <Definiendum> <gap>

```
                <Unconditional-definiens> <gap> <'#.'>

#DF Definiendum

                => <Semantic-definition-name> <gap> <'('> <gap>
                <Formal-param-name> <gap> <%<<Comma> <gap>
                <Formal-param-name> <gap>> <')'>

                => <Semantic-definition-name>  #.

#DF Semantic-definition-name

                => <Name>  #.

#DF Formal-param-name

                => <Name>  #.

#DF Unconditional-definiens

                => <'=>'> <gap> <Expression> #.
```

The form of a simple definition is

```
#DF c (p1,...,pn)  =>  expr #.
```

where the c is the semantic definition name, and p1 through pn are formal parameter names in an optional formal parameter list, and expr is a SEMANOL(76) semantic expression possibly containing formal parameter names, and also possibly containing semantic definition references (though normally not to c itself, as this would almost surely be a circular (ungrounded) recursion).

Upon evaluation of a semantic definition reference corresponding to definition c, the evaluation sequence described above would be invoked, using step (4) below:

*** (4) The result value of the simple definition body is simply obtained by evaluating the unconditional definiens. ***

Note that any appearance of a formal parameter name in the unconditional definiens has, as its value, the value assigned in step (2) of the evaluation sequence above.

```
----------------------

 #DF Definition-by-cases

              => <'#DF'> <gap> <Definiendum> <gap>
              <%<<Conditional-definiens> <gap> <';'><gap>>>
              <Conditional-definiens> <'#.'>

              => <'#DF'> <gap> <Definiendum> <gap> <%1
              <<Conditional-definiens> <gap> <';'> <gap>>>
              <Unconditional-definiens> <'#OTHERWISE'>  #.

 #DF Conditional-definiens

              => <'=>'> <gap> <Expression> <gap> <'#IF'> <gap>
              <Boolean-expression> #.

----------------------
```

The form of a definition-by-cases is

```
#DF c (p1,...,pn)

     => expr-1 #IF cond-1 ;

     => expr-2 #IF cond-2 ;

     .
     .
     .

     => expr-n #OTHERWISE #.
```

where n > 1, and where the parameter list and the last definiens are optional. (If the last definiens is missing, a definiens of the form:

```
"=> #ERROR #OTHERWISE"
```

is assumed.) The value expressions, expr-i, and the conditional expressions, cond-i, may contain formal parameter names and semantic definition references to any semantic definitions (including c).

Upon evaluation of a semantic definition reference corresponding to definition c, say, the evaluation sequence described above would be invoked, using step (4) below.

*** (4) The evaluation of a definition-by-cases body proceeds as follows:

Evaluate the conditional expressions, in order, until a non-<false> result is obtained, say, for cond-j. If the value of cond-j is not <true>, then halt on <error>. Otherwise, evaluate the corresponding value expression, expr-j, thus producing the result value (evaluation of the body thus completing).

If <false> is the value obtained for all conditional expressions, then the value expression expr-n in the #OTHERWISE-definiens is evaluated to produce the result value. (Here an implicit "=> #ERROR #OTHERWISE" definiens is assumed if there is no explicit #OTHERWISE-definiens, and in this case, an <error> halt would ensue.) ***

#DF Procedural-definition

=> <'#PROC-DF'> <gap> <Definiendum> <gap>
<Compound-statement> <gap>
<%<<Compound-statement> <gap>>> <'#.'> #.

The form of a procedural definition is

```
#PROC-DF c(p1,...,pn)

        s1

        .

        .

        sn #.   (n > 0)
```

where the parameter list is optional and the si are Compound-statements. The expressions in the si may contain formal parameter names, and semantic definition references to any semantic definitions (including c).

Upon evaluation of a semantic definition reference corresponding to definition c, say, the evaluation sequence described above is invoked, using step (4) below:

*** (4) The evaluation of the procedural definition body proceeds by executing the compound statements s1 through sn, in order, according to the rules described above in the discussion of the control-section. ***

If evaluation ever produces a result value, it does so by executing the "RETURN-WITH-VALUE!" statement (again described above in the discussion of the control section).

III. Description of SEMANOL(76)

"**********

Expressions

**********"

SEMANOL(76) expressions are composed from keyword expressions, and each of the keywords has a precedence for evaluation as given in the grammar used in the following pages.

#DF Expression

=> <Boolean-expression>

=> <'#STOP'>

=> <'#ERROR'> #.

#STOP

<stop>; when this keyword is evaluated, the execution of the SEMANOL(76) program terminates.

#ERROR

<error>; when this keyword is evaluated, the execution of the SEMANOL(76) program terminates with a standard error message.

```
"**************************

  Boolean Expressions

 **************************"


-----------------------

 #DF Boolean-expression

              =>  <Implication>  <%<  <gap>  <'#IFF'>  <gap>
              <Implication>>> #.

 #DF Implication

              => <Disjunction>  <%<  <gap>  <'#IMPLIES'>  <gap>
              <Disjunction>>> #.

 #DF Disjunction

              =>  <Conjunction>  <%<  <gap>  <'#OR'>  <gap>
              <Conjunction>>> #.

 #DF Conjunction

              => <Negation>  <%<  <gap>  <'&','#AND'>  <gap>
              <Negation>>> #.

 #DF Negation

              => <Boolean-primary>

              => <'#NOT'> <gap> <Boolean-primary>  #.

 #DF Boolean-primary

              => <Relational-expression>

              => <Sequence-expression>  #.
-----------------------
```

a #IFF b

<true> if (ă = <true> and ŏ = <true>) or (ă = <false> and ŏ = <false>)

<false> if (ă = <false> and ŏ = <true>) or (ă = <true> and ŏ = <false>)

<error> otherwise.

a #IMPLIES b

<true> if ă = <false> or ŏ = <true>;

<false> if ă = <true> and ŏ = <false>;

<error> otherwise.

a #OR b

<true> if ă = <true> or ŏ = <true>;

<false> if ă = <false> and ŏ = <false>;

<error> otherwise.

a #AND b

a & b

<true> if ă = <true> and ŏ = <true>;

<false> if ă = <false> or ŏ = <false>;

<error> otherwise.

#NOT a

<false> if ă = <true>;

<true> if ă = <false>;

<error> otherwise.

"**********

Relational Expressions

**********"

#DF Relational-expression

=> <Membership-relation>

=> <Case-identification>

=> <Subword-relation>

=> <Precedes-relation>

=> <Quantifier-relation>

=> <Equality-relation>

=> <Arithmetic-inequality> #.

The SEMANOL(76) relations are really characteristic functions which return the value <true> if the arguments are members of the relation, <false> if the arguments are not in the relation but are acceptable to the function, and <error> if the arguments are not acceptable to the function.

#DF Membership-relation

=> <Type-identification>

=> <Character-identification>

```
                => <Syntactic-class-membership>

                => <Sequence-membership> #.

#DF Membership-operator

                => <'#IS'>

                => <'#IS-IN'>

                => <'#IS-NOT'>

                => <'#IS-NOT-IN'> #.
-----------------------
```

The four types of membership relation are distinguished according to the second operand expression, b; b must be either a type designator, a character set constant, a union of syntactic class references or a sequence expression. If b is not one of these kinds of expressions, then "a #IS b" produces <error> when evaluated.

The membership relations all have one of the following forms:

```
"a #IS b"
"a #IS-NOT b"
"a #IS-IN b"
"a #IS-NOT-IN b"
```

Since the other three forms are completely definable in terms of the #IS-expressions, we shall do so here. The rest of the membership relations will be described only for the #IS-expression.

a #IS-IN b

Synonymous with "a #IS b"; that is,

<true> (<false>) if and only if "a #IS b" evaluates to <true> (<false>);

<error> otherwise.

a #IS-NOT b

Synonymous with "#NOT(a #IS b)"; that is

<true> (<false>) if and only if "a #IS b"
evaluates to <false>(<true>);

<error> otherwise.

a #IS-NOT-IN b

Synonymous with "a #IS-NOT b"; that is

<true> (<false>) if and only if "a #IS b"
evaluates to <false>(<true>);

<error> otherwise.

#DF Type-identification

=> <Sequence-expression> <gap>
<Membership-operator> <gap> <Type> #.

#DF Type

=> <'#UNDEFINED'>

=> <'#BOOLEAN'>

=> <'#SEQUENCE'>

=> <'#NODE'>

=> <'#STRING'>

=> <'#INTEGER'>

=> <'#BIT-STRING'> #.

[ALSO SEE: Chapter II, Section C. "SEMANOL(76) Data Types"]

a #IS #UNDEFINED

<true> if a is <undefined>;

<false> otherwise.

a #IS #BOOLEAN

<true> if ā is either <true> or <false>;

<false> otherwise.

a #IS #SEQUENCE

<true> if ā is a SEMANOL(76) sequence object;

<false> otherwise.

a #IS ↓NODE

<true> if ā is a SEMANOL(76) parse tree node;

<false> otherwise.

a #IS #STRING

<true> if ā is a SEMANOL(76) string object;

<false> otherwise.

a #IS #INTEGER

<true> if ā is a string which satisfies the syntax for a SEMANOL(76) integer constant (c.f., the grammar for <Integer-constant> below).

<false> otherwise.

a #IS #BIT-STRING

<true> if ā is a string which satisfies the syntax for a SEMANOL(76) bit string constant (c.f., the grammar for <Bit-string-constant> below).

<false> otherwise.

#DF Character-identification

=> <Sequence-expression> <gap>
<Membership-operator> <gap> <Set-constant> #.

```
#DF Set-constant
                => <'#CAP'>
                => <'#DIGIT'>
                => <'#LETTER'>
                => <'#LOWCASE'>
                => <'#SPACESET'>  #.
-----------------------
```

a #IS #CAP

<error> if ã is not string convertible;

<true> if the string value of ã is in the set of ASCII capital letters, A through Z.

<false> otherwise.

a #IS #DIGIT

<error> if ã is not string convertible;

<true> if the string value of ã is in the set of ASCII decimal digits, 0 through 9.

<false> otherwise.

a #IS #LETTER

<error> if ã is not string convertible;

<true> if the string value of ã is in the set of ASCII capital and lower case letters, A through Z, and a through z;

<false> otherwise.

a #IS #LOWCASE

<error> if ã is not string convertible;

<true> if the string value of ā is in the set of ASCII lower case letters, a through z;

<false> otherwise.

a #IS-IN #SPACESET

<error> if ā is not string convertable;

<true> if the string value of ā is the ASCII blank character.

<false> otherwise.

#DF Syntactic-class-membership

=> <Sequence-expression> <gap>
<Membership-operator> <gap>
<Syntactic-class-union> #.

#DF Syntactic-class-union

=> <Syntactic-class-reference> <gap> <% < <gap>
<'#U'> <gap> <Syntactic-class-reference>>> #.

#DF Syntactic-class-reference

=> <'<'> <gap> <Syntactic-class-name> <gap> <'>'>

---

a #IS <b1>

a #IS <b1> #U <b2>

...

a #IS <b1> #U <b2> #U ... #U <bn>

<error> if ā is not string convertible, or if the bi are not syntactic class names appearing in the definienda of syntactic definitions;

CASE 1: ā is a node.

<true> if the syntactic class name label on the node, ā (cf, the discussion of the "#CONTEXT-FREE-PARSE-TREE..." operator, below) matches one of the syntactic class names, bi;

<false> otherwise.

CASE 2: ā is a string.

<true> if there is an i <= n such that bi is the first (reading from the left) syntactic class name for a syntactic class containing the string value of ā, and if for all j (1 <= j <= i) the grammars which have bj as their start symbol are unambiguous with respect to ā. If the grammars for the bj's include one which is ambiguous, the value is <error>;

<false> otherwise.

NOTE: Here the SEMANOL(76) processor invokes the #CONTEXT-FREE-PARSE-TREE operator in order to recognize the string with respect to the grammars "started" by each of the syntactic class names, bi, in order of increasing i.

#DF Sequence-membership

=> <Sequence-expression> <gap> <Membership-operator> <gap> <Sequence-expression> #.

---

a #IS-IN b

<error> if b is a SEMANOL(76) semantic expression, and b̄ is not a sequence;

<true> if there is an integer i such that "i #TH-ELEMENT-IN b #EQ a " holds;

<false> otherwise.

#DF Case-identification

=> <Sequence-expression> <gap> <'#IS','#IS-NOT'>
<gap> <'#CASE'> <gap> <Sequence-expression> <gap>
<'#OF'> <gap> <Syntactic-class-reference> #.

---

a #IS #CASE n #OF <c>

<error> if "n #IS-NOT #INTEGER #OR a #IS-NOT #NODE" holds;

<true> if "a #IS c" holds and if ã is tagged with the case number, ñ (c.f., the discussion of the "#CONTEXT-FREE-PARSE-TREE..." operator, below); that is, node ã was constructed by virtue of the ñ-th definiens in the syntactic definition named c;

<false> otherwise.

a #IS-NOT #CASE n #OF <c>

<true> (<false>) if and only if "a #IS #CASE n #OF c" evaluates to <false> (<true>);

<error> otherwise.

#DF Subword-relation

=> <Sequence-expression> <gap> <'#IS','#IS-NOT'>
<gap> <'#SUBWORD'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#IS','#IS-NOT'>
<gap> <#NODE-IN> <gap> <Sequence-expression> #.

---

a #IS #SUBWORD b

<error> if ā (or b̄) is not string convertible;

<true> if ā is a substring of b̄ (that is, b̄ = "b1a1b2", where "b1", "a1", and "b2" are strings -- possibly <nil> -- , and ā = "a1");

<false> otherwise.

a #IS-NOT #SUBWORD b

<true> (<false>) if and only if "a #IS #SUBWORD b" evaluates to <false> (<true>);

<error> otherwise.

a #IS #NODE-IN b

<error> if "a #IS-NOT #NODE #OR b #IS-NOT #NODE" holds;

<true> if b̄ is the "root" node of a (sub) parse tree which contains ā as a node;

<false> otherwise.

This is equivalent to "a #IS-IN #SEQUENCE-OF-NODES-IN b".

a #IS-NOT #NODE-IN b

<true> (<false>) if and only if "a #IS #NODE-IN b" evaluates to <false> (<true>);

<error> otherwise.

This is equivalent to "a #IS-NOT-IN #SEQUENCE-OF-NODES-IN b".

#DF Precedes-relation

=> <Sequence-expression> <gap> <'#PRECEDES'> <gap> <Sequence-expression> <gap> <'#IN'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#DOES-NOT-PRECEDE'> <gap> <Sequence-expression>

<gap> <'#IN'> <gap> <Sequence-expression>

---

a #PRECEDES b #IN c

<error> if neither of the following conditions hold: (1) c̆ is a sequence, or (2) ă, b̆, and c̆ are all nodes;

CASE 1 (c̆ is a sequence)

<false> if ă (or b̆) is not an element of c̆;

<true> if the ordinal position of the first occurrence of ă in c̆ is less than the ordinal position of the first occurrence of b̆ in c̆ (i.e., if "#ORDPOSIT a #IN c < #ORDPOSIT b #IN c");

<false> otherwise;

CASE 2 (ă, b̆, and c̆ are all nodes)

<false> if ă (or b̆) is not a node in the subtree of which c̆ is the root node;

<true> if ă appears before b̆ in a pre-order scan of the nodes in c̆;

<false> otherwise;

NOTE: This case is synonymous with the expression "a #PRECEDES b #IN #SEQUENCE-OF-NODES-IN(c)" which is an instance of case 1 of this relation.

#DF Quantifier-relation

=> <'#THERE-EXISTS'> <gap> <Name> <gap> <':'> <gap> <Bounded-interval> <'#SUCH-THAT'> <gap> <'('> <gap> <Boolean-expression> <gap> <')'>

=> <'#THERE-EXISTS'> <gap> <Name> <gap> <'#IN'>

<gap> <Sequence-expression> <'#SUCH-THAT'> <gap>
<'('> <gap> <Boolean-expression> <gap> <')'>

=> <'#FOR-ALL'> <gap> <Name> <gap> <':'> <gap>
<Bounded-interval> <gap> <'#IT-IS-TRUE-THAT'>
<gap> <'('> <gap> <Boolean-expression> <gap>
<')'>

=> <'#FOR-ALL'> <gap> <Name> <gap> <'#IN'> <gap>
<Sequence-expression> <gap> <'('> <gap>
<Boolean-expression> <gap> <')'>

#THERE-EXISTS b : a <= b <= c #SUCH-THAT (B)

<error> if "a #IS-NOT #INTEGER #OR c #IS-NOT #INTEGER" holds;

<false> if "c < a" holds or if, for all integers b̆, ă <= b̆ <= c̆, the expression "B[b]" evaluates to <false> (see note);

<true> if there is an integer, b̆, ă <= b̆ <= c̆, such that "B[b]" evaluates to <true> and, for all integers d̆, ă <= d̆ < b̆, "B[b]" evaluates to <false> (see note);

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̆. Also see Chapter II, Section I.

#THERE-EXISTS b #IN c #SUCH-THAT (B)

<error> if c̆ is not a sequence;

<false> if every element, b̆, of c̆ is such that "B[b]" evaluates to <false> (see note);

<true> if there is an element, b̆, of c̆ such that "B[b]" evaluates to <true>, and each element, ă, preceding b̆ in c̆ is such that "B[a]" evaluates to <false> (see note);

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̃. Also see Chapter II, Section I.

#FOR-ALL b: a <= b <= c #IT-IS-TRUE-THAT (B)

<error> if "a #IS-NOT #INTEGER #OR b #IS-NOT #INTEGER" holds;

<false> if there is an integer, b̃, ã <= c̃, such that "B[b]" evaluates to <false> and, for all integers d̃, ã <= d̃ < b̃, "B[d]" evaluates to <true> (see note);

<true> if "c < a" holds or if, for all integers b̃, ã <= b̃ <= c̃, the expression "B[b]" evaluates to <true> (see note);

<error> otherwise.

More precisely, this keyword expression is equivalent to

"#NOT #THERE-EXISTS b: a <= b <= c #SUCH-THAT (#NOT (B))".

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̃. Also see Chapter II, Section I.

#FOR-ALL b #IN c #IT-IS-TRUE-THAT (B)

<error> if "c #IS-NOT #SEQUENCE" holds;

<false> if there is an element, b̃, of c̃ such that "B[b]" evaluates to <false>, and each element, ã, preceding b̃ in c̃ is such that "B[a]" evaluates to <true> (see note);

<true> if every element, b̃, of c̃ is such that "B[b]" evaluates to <true> (see note);

<error> otherwise.

More precisely, this keyword expression is equivalent to "#NOT #THERE-EXISTS b #IN c #SUCH-THAT (#NOT (B))".

III. Description of SEMANOL(76)

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x. Also see Chapter II, Section I.

#DF Equality-relation

=> <Sequence-expression> <gap> <'#EQ'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#NEQ'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#EQS'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#NEQS'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#EQN'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#NEQN'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#EQW'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#NEQW'> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'='> <gap> <Sequence-expression>

=> <Sequence-expression> <gap> <'#N'> <gap> <'='> <gap> <Sequence-expression>

---

a #EQ b

<true> if a and b are both the value <undefined>;

<true> if a and b are identical string values (i.e., if "x #IS #STRING & y #IS #STRING & x #EQW y");

<true> if ā and b̄ are both <true> or are both <false> (i.e., if "x #IS #BOOLEAN & y #IS #BOOLEAN & x #IFF y");

<true> if ā and b̄ are the same identical node (i.e., if " x #IS #NODE & y #IS #NODE & x #EQN y");

<true> if ā and b̄ are identical sequences (i.e., if "x #IS #SEQUENCE & y #IS #SEQUENCE & x #EQS y");

<false> otherwise;

a #NEQ b

<true> if "a #EQ b" evaluates to <false>;

<false> otherwise.

a #EQS b

<error> if ā (or b̄) is not a sequence;

<false> if the length of the two sequences is not the same (i.e., if "#LENGTH(a) #N= #LENGTH(b)");

<true> if ā and b̄ are both empty sequences (i.e., they both have length 0), or if, for 1 <= i <= length of ā, "i #TH-ELEMENT-IN a #EQ i #TH-ELEMENT-IN b" holds;

<false> otherwise.

a #NEQS b

<true> (<false>) if and only if "a #EQS b" evaluates to <false> (<true>);

<error> otherwise.

a #EQN b

<error> if ā (or b̄) is not a node;

<true> if ā and b̄ are the same identical node (c.f., Chapter II, Section F "Sequences and parse

trees");

<false> otherwise.

a #NEQN b

<true> (<false>) if and only if "a #EQN b" evaluates to <false> (<true>);

<error> otherwise.

a #EQW b

<error> if ā (or b̄) is not string convertible;

<true> if the string value of ā is identical to the string value of b̄;

<false> otherwise.

a #NEQW b

<true> (<false>) if and only if "a #EQW b" evaluates to <false> (<true>);

<error> otherwise.

a = b

<error> if "a #IS-NOT #INTEGER #OR b #IS-NOT #INTEGER" holds;

<true> if the numbers denoted by the numerals ā and b̄ are equal;

<false> otherwise.

NOTE: This operator tests for arithmetic equality; it could be defined strictly in terms of the numeral strings ā and b̄, but such a definition seems unnecessarily obscure for the purposes of this manual. This comment applies to all the arithmetic operators to be discussed below as well.

a #N= b

Synonymous with "#NOT(a = b)"; that is,

<true> (<false>) if and only if "a = b" evaluates to <false> (<true>);

<error> otherwise.

```
#DF Arithmetic-inequality

        => <String-expression> <gap> <Less-than-operator>
        <gap> <String-expression>

        => <String-expression> <gap>
        <Greater-than-operator> <gap> <String-expression>
        #.

#DF Less-than-operator

        => <'<'>

        => <'<='> #.

#DF Greater-than-operator

        => <'>'>

        => <'>='> #.
```

---

a < b

<error> if "a #IS-NOT #INTEGER #OR b #IS-NOT #INTEGER" holds;

<true> if the number denoted by the numeral ā is less than the number denoted by b̄;

<false> otherwise.

a <= b

Synonymous with "#NOT(a > b)"; that is,

<true> (<false>) if and only if "a > b" evaluates to <false> (<true>);

<error> otherwise.

a > b

<error> if "a #IS-NOT #INTEGER #OR b #IS-NOT #INTEGER" holds;

<true> if the number denoted by the numeral ā is greater than the number denoted by b̄;

<false> otherwise.

a >= b

Synonymous with "#NOT(a < b)"; that is,

<true> (<false>) if and only if "a < b" evaluates to <false> (<true>);

<error> otherwise.

III. Description of SEMANOL(76)

"**********

Sequence Expressions

**********"

#DF Sequence-expression

=> <Subsequence-expression>

=> <Sequence-concatenation>

Sequence expressions have sequences as their result values. A SEMANOL(76) sequence is a structured object, whose elements may be any object of any type discussed in Chapter II, Section C "SEMANOL(76) Data Types", including sequences. However, SEMANOL(76) sequences may not be "re-entrant" (i.e., none of the sequence elements of a sequence may be the sequence itself, nor may any elements of a (sequence) element of the sequence be the sequence itself, etc.). The "length" of a sequence is the number of elements it has; there exists an "empty" sequence whose length is 0 -- we denote the empty sequence in this manual by "<nilseq>", and in SEMANOL(76) by "#NILSEQ".

The elements of a sequence are strictly ordered by the "Precedes" relation; we assign ordinal positions to the elements of a sequence as follows: the first element of a sequence precedes all other elements, the second precedes all others except the first, etc.

Suppose two separate evaluations produce $\tilde{x}$ and $\tilde{y}$, each of which is a sequence. If "#LENGTH(x) = #LENGTH(y)" and "i #TH-ELEMENT-IN(x) #EQ i #TH-ELEMENT-IN(y)" for all i such that "1 <= i" and "i <= #LENGTH(x)", then $\tilde{x}$ is identical to $\tilde{y}$. That is, the identity of a sequence is completely determined by the order and the identity of its components. Here the notion of identity is defined in terms of the SEMANOL(76) operators for equality (the node-, sequence-, string-, and numeric-equality relations).

#DF Subsequence-expression

=> <'#SUBSEQUENCE-OF-ELEMENTS'> <gap> <Name>
<gap> <'#IN'> <gap> <Sequence-concatenation>
<gap> <'#SUCH-THAT'> <gap> <'('> <gap>
<Boolean-expression> <gap> <')'>

=> <'#SUBEQUENCE'> <gap> <String-expression>
<gap> <'#TO'> <gap> <String-expression> <gap>
<'#OF'> <gap> <Sequence-expression>

=> <'#INITIAL-SUBSEQ-OF-LENGTH'> <gap>
<String-expression> <gap> <'#OF'> <gap>
<Sequence-expression>

=> <'#TERMINAL-SUBSEQ-OF-LENGTH'> <gap>
<String-expression> <gap> <'#OF'> <gap>
<Sequence-expression>

#SUBSEQUENCE-OF-ELEMENTS b #IN a #SUCH-THAT (B)

<error> if "a #IS-NOT #SEQUENCE" holds, or if there is an element, b̆, in ă such that "B[b]" evaluates to neither <true> nor <false>;

<nilseq> if there is no element, b̆, in ă such that "B[b]" holds;

Otherwise: s̆, the sequence formed by taking the elements, b̆, of ă such that "B[b]" holds. The order of the elements in s̆ is their order in ă (i.e., #b1 #PRECEDES b2 #IN s #IMPLIES b1 #PRECEDES b2 #IN a" holds).

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x. Also see Chapter II, Section I.

#SUBSEQUENCE m #TO n #OF a

<error> if m̆ (or n̆) is not an integer or if ă is not a sequence;

<undefined> if "m > n" or "m < 1" or "n > #LENGTH(a)";

Otherwise: the subsequence of ã beginning with the m̃-th element in ã and ending with the ñ-th element in ã, inclusive; that is, the sequence b̃ such that for every integer i in the interval 1 <= i <= n - m + 1, "i #TH-ELEMENT-IN b #EQ i + m -1 #TH-ELEMENT-IN a" holds.

#INITIAL-SUBSEQ-OF-LENGTH m #OF a

<error> if m̃ is not an integer or if ã is not a sequence;

<undefined> if "m > #LENGTH(a)" or "m < 0";

<nilseq> if "m = 0";

Otherwise: The sequence formed by taking as elements, the first m̃ elements of ã in order; i.e., the sequence, b̃, such that "LENGTH(b) = m" and there exists a (possibly empty) sequence c̃ such that "b #CS c #EQ a" holds; that is for every integer i in the interval 1 <= i <= m, "i#TH-ELEMENT-IN b #EQ i #TH-ELEMENT-IN a" holds.

#TERMINAL-SUBSEQ-OF-LENGTH m #OF a

<error> if m̃ is not an integer or if ã is not a sequence;

<undefined> if "m > #LENGTH(b)" or if "m < 0";

<nilseq> if "m = 0";

Otherwise: The sequence formed by taking as elements, the last m̃ elements of ã in order; i.e., the sequence, b̃, such that "LENGTH(b) = m", and there exists a (possibly empty) sequence c̃ such that "c #CS b #EQ a" holds; that is, for every integer i in the interval 1 <= i <= m, "i #TH-ELEMENT-IN b #EQ (#LENGTH(a)-m+i)#TH-ELEMENT-IN a" holds.

#DF Sequence-concatenation

=> <Sequence-constructor> <%< <gap> <'#CS'> <gap> <Sequence-constructor>>> #.

---

a #CS b

<error> if ã (or b̃) is not a sequence;

Otherwise: the sequence s̃ formed by concatenating ã and b̃; that is, the sequence s̃ such that for every i in the interval 1 <= i <= the length of ã, "i #TH-ELEMENT-IN a #EQ i #TH-ELEMENT-IN s" holds, and for every i in the interval 1 <= i <= the length of b̃, "i #TH-ELEMENT-IN b #EQ (i + #LENGTH(a)) #TH-ELEMENT-IN s" holds.

a1 #CS a2 #CS ... #CS an

Synonymous with the expression "(...((a1 #CS a2) #CS a3)...#CS an)"; that is, the n-fold sequence concatenation of ã1 through ãn (<error> if any of the ãi is not a sequence).

#DF Sequence-constructor

=> <'#SEQUENCE-OF'> <gap> <Syntactic-class-union> <gap> <'#IN'> <gap> <String-expression>

=> <'#SEQUENCE-OF-NODES-IN'> <gap> <String-expression>

=> <'#SEQUENCE-OF-NODES'> <gap> <Name> <gap> <'#IN'> <gap> <String-expression> <gap> <'#SUCH-THAT'> <gap> <'('> <gap> <Boolean-expression> <gap> <')'>

=> <'#SEQUENCE-OF-ANCESTORS-OF'> <gap> <String-expression>

=> <String-expression> #.

#SEQUENCE-OF <a1> #IN b

#SEQUENCE-OF <a1> #U <a2> #U ...#U <an> #IN b

<error> if b is not a parse tree node;

Otherwise: the sequence s formed by taking as elements, the nodes, bi, (of the parse tree whose root node is b) which have a Syntactic-class-name tag identical to one of the Syntactic-class-names, ai (cf, the discussion of the "#CONTEXT-FREE-PARSE-TREE..." operator, below). The nodes of the parse tree b are to appear in the sequence s in preorder (i.e., if the ordinal position of bi is less than that of bj in s, then bi would be listed before bj in a preorder listing of the nodes of b. This SEMANOL(76) expression is equivalent to "#SUBSEQUENCE-OF-ELEMENTS x #IN (#SEQUENCE-OF-NODES-IN b) #SUCH-THAT (x #IS-IN <a1> #U <a2> #U ...#U <an>)".

NOTE: A preorder listing of the nodes of a parse tree is one given by the following procedure:

1. Write a name for the root node of the tree;

2. For each descendent node (in increasing seg-number --i.e., left-to-right -- order), write the preorder listing of the nodes in the subtree having that node as root.

#SEQUENCE-OF-NODES-IN b

<error> if b is not a parse tree node;

Otherwise: the sequence, s, formed by taking as elements, in preorder, the nodes of the parse tree whose root node is b. If bi, and bj are nodes in the parse tree b, then bi appears before bj in a preorder listing of the nodes in b if and only if the ordinal position of bi is less than that of bj in the sequence, s.

NOTE: A preorder listing of the nodes of a parse tree is one given by the following procedure:

1. Write a name for the root node of the tree;

2. For each descendent node (in increasing seg-number --i.e., left-to-right -- order), write the preorder listing of the nodes in the subtree having that node as root.

#SEQUENCE-OF-NODES b #IN a #SUCH-THAT (B)

<error> if ă is not a parse tree node, or if there is a node, b̆, in the parse tree whose root is ă, such that B[b] evaluates to neither <true> nor <false>;

<nilseq> if there is no node b̆, in the parse tree whose root is ă, such that "B[b]" holds;

Otherwise: s̆, the sequence of nodes, b̆, taken in preorder from the parse tree whose root is ă, such that "B[b]" holds. Thus "b1 #PRECEDES b2 #IN s #IMPLIES b1 #PRECEDES b2 #IN a" holds.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̆. Also see Chapter II, Section I.

This SEMANOL(76) expression is equivalent to "#SUBSEQUENCE-OF-ELEMENTS b #IN (#SEQUENCE-OF-NODES-IN a) #SUCH-THAT (B[b])".

NOTE: A preorder listing of the nodes of a parse tree is one given by the following procedure:

1. Write a name for the root node of the tree;

2. For each descendent node (in increasing seg-number --i.e., left-to-right -- order), write the preorder listing of the nodes in the subtree having that node as root.

#SEQUENCE-OF-ANCESTORS-OF n

<error> if n̆ is not a node;

Otherwise: the sequence , s̆, formed by taking as elements, in preorder, those nodes b̆ from the most inclusive parse tree containing n̆ such that "n #IS #NODE-IN b #AND n #NEQN b" holds.

```
        This is equivalent to "#SEQUENCE-OF-NODES b #IN
        #ROOT-NODE (n) #SUCH-THAT (n #IS #NODE-IN b #AND
        n #NEQN b)"


        "*************************

          String-Expressions

         *************************"



----------------------
#DF String-expression

 => <Substring-extractor>

        => <String-concatenation>

        => <Position-detector>

        => <Sequence-element-extractor>

        => <Seg-selector>

        => <Sum> #.

 #DF Substring-extractor

        => <'#LEFT'> <Sum> <gap> <'#CHARACTERS-OF'> <gap>
        <String-expression>

        => <'#RIGHT'> <Sum> <gap> <'#CHARACTERS-OF'>
        <gap> <String-expression>

        => <Sum> <gap> <'#TH-CHARACTER-IN'> <gap>
        <String-expression>

        => <'#FIRST-CHARACTER-IN'> <gap>
        <String-expression>

        => <'#LAST-CHARACTER-IN'> <gap>
        <String-expression>

        => <'#SUBSTRING-OF-CHARACTERS'> <gap> <Sum> <gap>
        <'#TO'> <gap> <Sum> <gap> <'#OF'> <gap>
```

<String-expression>

=> <'#PREFIX-OF-FIRST'> <gap> <Sum> <gap> <'#IN'> <gap> <String-expression>

=> <'#SUFFIX-OF-FIRST'> <gap> <Sum> <gap> <'#IN'> <gap> <String-expression>

#LEFT m #CHARACTERS-OF b

<error> if m̃ is not an integer or if b̃ is not string convertible;

<undefined> if "m > #LENGTH(b)" or "m < 0";

<nil> if "m = 0";

Otherwise: the string c̃, such that c̃ is the initial substring of b̃ of length m̃. That is, the string given by "#SUBSTRING-OF-CHARACTERS 1 #TO m #OF b".

[Examples:

"#LEFT 3 #CHARACTERS-OF ('HI-THERE!')" denotes "HI-".

"#LEFT q #CHARACTERS-OF ('OOPS')" denotes <undefined> if "q > 4" or if "q < 0", and denotes "OOP" if "q = 3".]

#RIGHT m #CHARACTERS-OF b

<error> if m̃ is not an integer or if b̃ is not string convertible;

<undefined> if "m > #LENGTH(b)" or "m < 0";

<nil> if "m = 0";

Otherwise: the string c̃, such that c̃ is the rightmost substring of b̃ of length m̃. That is, the string given by "#SUBSTRING-OF-CHARACTERS #LENGTH(b) - m + 1 #TO #LENGTH(b) #OF b".

[Examples:

"#RIGHT 3 #CHARACTERS-OF ('FEE-FI-FO-FUM')" denotes "FUM".

"#RIGHT(#LENGTH(ix) - 1) #CHARACTERS-OF ix" denotes <undefined> if "#LENGTH(ix) < 1", and <nil> if "LENGTH(ix) = 1"; otherwise, it denotes the string formed from ix by dropping its first character.]

m #TH-CHARACTER-IN b

<error> if m is not an integer or if b is not string convertible;

<undefined> if "m < 1" or "m > #LENGTH(b)";

Otherwise: c, where c is a string of length 1 consisting of the m-th character in b counting from the left and starting the count at 1.

[Example:

"2 #TH-CHARACTER-IN 'pig'" denotes "i".]

#FIRST-CHARACTER-IN b

<error> if b is not string convertible;

<undefined> if b is <nil>, the empty string;

Otherwise: the leftmost character of b

[This is equivalent to "1 #TH-CHARACTER-IN b"]

#LAST-CHARACTER-IN b

<error> if b is not string convertible;

<undefined> if b is <nil>, the empty string;

Otherwise: the rightmost character of b

[This is equivalent to

"(#LENGTH (b)) #TH-CHARACTER-IN b"

#SUBSTRING-OF-CHARACTERS m #TO n #OF b

<error> if m (or n) is not an integer or if b is not string convertible;

<undefined> if "m > n" or "m < 1" or "n > #LENGTH(b)";

Otherwise: c where c is the substring of b beginning with the m-th character of b and ending with the n-th character of b, inclusive (the count is from left to right, beginning at 1).

[Example:

"#SUBSTRING-OF-CHARACTERS 2 #TO 3 #OF 'abcd'" denotes "bc".]

#PREFIX-OF-FIRST b #IN a

<error> if b (or a) is not string convertible;

<undefined> if b is not a substring of a;

<nil> if b is an initial substring of a or if "b #EQW #NIL";

Otherwise: c where c is the initial substring of a ending at (but not including) the first character of the first occurrence of b (scanning from the left).

[Examples:

"#PREFIX-OF-FIRST 'X' #IN 'aaXXX'" denotes "aa";

"#PREFIX-OF-FIRST ',' #IN alist" denotes the initial substring of string alist ending just before the first comma (if there is one).]

#SUFFIX-OF-FIRST b #IN a

<error> if b (or a) is not string convertible;

<undefined> if b is not a substring of a;

<nil> if the first occurrence of b in a (scanning from the left) is a right-most substring of a;

a if "b #EQW #NIL";

Otherwise: c̃ where c̃ is the right-most substring of ã beginning at the character immediately after the last character of the first occurrence of b̃ in ã.

[Examples:

"#SUFFIX-OF-FIRST 'a' #IN 'aaXXX'" denotes "aXXX";

"#SUFFIX-OF-FIRST ',' #IN alist" denotes the right-most string of alist beginning just after the first comma (if there is one).]

```
#DF String-concatenation

        => <Sum> <gap> <%1<<'#CW'> <gap> <Sum>>>

        #.
```

---

a #CW b

<error> if ã (or b̃) is not string convertible;

Otherwise: c̃ where c̃ is the string obtained by concatenating ã and b̃.

[Example:

"'A' #CW 'WFUL'" denotes "AWFUL"]

[Note: If several concatenations are specified in an expression with no order-imposing parenthesization, concatenation takes place from left-to-right.]

[Note: Since all numerals are also strings, all the string functions may be applied to them, also.]

III. Description of SEMANOL(76)

#DF Position-detector

=> <'#SUBSTRING-POSIT-OF'> <gap> <Sum> <gap>
<'#IN'> <gap> <String-expression>

=> <'#ORDPOSIT'> <gap> <Sum> <gap> <'#IN'> <gap>
<String-expression> #.

#SUBSTRING-POSIT-OF a #IN b

<error> if ā (or b̄) is not string convertible;

<undefined> is ā is not a substring of b̄ or if "a #EQW #NIL";

Otherwise: c̄, where c̄ is the integer corresponding to the first character position at which ā matches (is identical to) a substring of b̄.

[Example:

"#SUBSTRING-POSIT-OF 'a' #IN 'cat'" denotes "2".]

#ORDPOSIT a #IN b

<error> if b̄ is not a sequence;

<undefined> if element ā is not to be found in sequence b̄ (i.e., if #FOR-ALL x #IN b #IT-IS-TRUE-THAT (x#NEQ a)" holds);

Otherwise: c̄, where c̄ is a decimal numeral denoting the order position of the first occurrence of element ā in sequence b̄ (counting from 1).

```
----------------------

 #DF Sequence-element-extractor

                => <Sum> <gap> <'#TH-ELEMENT-IN'> <gap>
                <String-expression>

                => <'#FIRST-ELEMENT-IN'> <gap>
                <String-expression>

                => <'#LAST-ELEMENT-IN'> <gap> <String-expression>

                => <<'#FIRST'> #U <'#LAST'> #U <<Sum> <gap>
                <'#TH'>>> <gap> <Name> <gap> <'#IN'> <gap> <Sum>
                <gap> <Such-that-clause>

                => <<'#FIRST'> #U <'#LAST'> #U <<Sum> <gap>
                <'#TH'>>> <gap> <Name> <gap> <':'> <gap>
                <Bounded-interval> <gap> <Such-that-clause> #.

 #DF Such-that-clause

                => <'#SUCH-THAT'> <gap> <'('> <gap>
                <Boolean-expression> <gap> <')'> #.
----------------------


 n #TH-ELEMENT-IN b

                <error> if b is not a sequence or if n is not in
                the integer range.

                <undefined> if "n < 1" or "n > #LENGTH(b)".

                Otherwise: the nth element in the sequence b.

 #FIRST-ELEMENT-IN b

                <error> if b is not a sequence.

                <undefined> if b has length 0;

                Otherwise:  the first element in the Sequence b.

                [This is equivalent to "1 #TH-ELEMENT-IN b".]

 #LAST-ELEMENT-IN b
```

<error> if 𝔟 is not a sequence;

<undefined> if 𝔟 has length 0;

Otherwise: the last element in the Sequence 𝔟.

[This is equivalent to "#LENGTH(b) #TH-ELEMENT-IN b".]

n #TH b #IN s #SUCH-THAT(B)

<error> if 𝔰 is not a sequence or 𝔫 is not an integer;

<undefined> if there are fewer than 𝔫 elements, 𝔟, of 𝔠 such that "B[b]" evaluates to <true>, and all the others are such that "B[b]" evaluates to <false>;

𝔟, the element in 𝔠 such-that "B[b]" evaluates to <true> and there are 𝔫 - 1 elements, 𝔞, preceding 𝔟 in 𝔠 such that "B[b]" evaluates to <true>, and each other element 𝔡, preceding 𝔟 in 𝔠 is such that "B[d]" evaluates to <false>, if there is such a 𝔟;

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting 𝔵. Also see Chapter II, Section I.

#FIRST b #IN s #SUCH-THAT(B)

<error> if 𝔰 is not a sequence.

<undefined> if every element, 𝔟, of 𝔠 is such that "B[b]" evaluate to <false>;

𝔟, the element of 𝔠 such that "B[b]" evaluates to <true> and each element, 𝔞, preceding 𝔟 in 𝔠, is such that "B[a]" evaluates to <false>, if there is such a 𝔟;

<error> otherwise.

More precisely, this keyword expression is equivalent to "1 #TH b #IN s #SUCH-THAT (B)".

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̃. Also see Chapter II, Section I.

#LAST b #IN s SUCH-THAT(B)

<error> if s̃ is not a sequence.

<undefined> if every element, b̃, of c̃ is such that "B[b]" evaluates to <false>;

b̃, the element of c̃ such that "B[b]" evaluates to <true> and each element, ã, following b̃ in c̃, is such that "B[a]" evaluates to <false>, if there is such a b̃;

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̃. Also see Chapter II, Section I.

n #TH b : a <= b <= c #SUCH-THAT (B)

<error> if "a #IS-NOT #INTEGER #OR c #IS-NOT #INTEGER" holds;

<undefined> if there are fewer than ñ integers, b̃, ã <= b̃ <= c̃, such that "B[b]" holds, and all the others are such that #B[b]" evaluates to <false>;

The minimal integer, b̃, ã <= b̃ <= c̃ such that there are ñ integers, d̃, ã <= d̃ <= b̃ such that "B[d]" holds, and every other ẽ, ã <= ẽ <= b̃ is such that "B[e]" evaluates to <false>, if there is such a b̃;

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̃. Also see Chapter II, Section I.

This keyword expression is equivalent to

"n #TH b #IN sequence-of-integers(a,c) #SUCH-THAT (B)",

given:

#DF sequence-of-integers (a,c)

=> #ERROR #IF a #IS-NOT #INTEGER #OR c #IS-NOT #INTEGER;

=> #NILSEQ #IF a > c;

=> \a\ #CS sequence-of-integers (a + 1, c) #OTHERWISE #.

#FIRST b : a <= b <= c #SUCH-THAT (B)

<error> if "a #IS-NOT #INTEGER #OR b #IS-NOT #INTEGER" holds;

<undefined> if every integer b̆, ă <= b̆ <= c̆ is such that "B[b]" evaluates to <false>;

The (minimal) integer, b̆, ă <= b̆ <= c̆ such that "B[b]" holds, and all other integers d̆, ă <= d̆ < b̆ (if any) are such that "B[d]" evaluates to <false>, if there is such a b̆;

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x̆. Also see Chapter II, Section I.

More precisely, this keyword is equivalent to

"1 #TH b : a < = b < = c #SUCH-THAT(B)".

#LAST b : a <= b <= c #SUCH-THAT (B[b])

<error> if "a #IS-NOT #INTEGER #OR b #IS-NOT #INTEGER" holds;

<undefined> if every integer b̆, ă < = b̆ < = c̆ is such that "B[b]" evaluates to <false>;

the (maximal) integer, b̆, ă < = b̆ < = c̆ such that "B[b]" holds, and all others d̆, b̆ < d̆ < = c̆ (if any) are such that "B[d]" evaluate to <false>, if there is such a b̆;

<error> otherwise.

Note: B[x] is B with every occurrence of the dummy variable name replaced with x, a constant denoting x. Also see Chapter II, Section I.

This keyword expression is equivalent to

"#LAST b #IN sequence-of-integers (a,c) #SUCH-THAT(B)",

where "sequence-of-integers (a,c)" is defined in the explanation for "n #TH b: a < = b < = c #SUCH-THAT (B)", above

---

#DF Seg-selector

=> <'#SEG'> <gap> <Sum> <gap> <'#OF'> <gap>
<String-expression> #.

#SEG n #OF p

<error> if p̃ is not a parse tree node, or ñ is not an integer;

<undefined> if node p̃ has m̃ direct descendents and "n > m" *or* if "n < 1";

Otherwise: The ñth descendent q̃ of node p̃ (counting from the left from 1). q̃ is thus a parse tree node, possibly a terminal one.

[Example: "Standard-name(#SEG 1 #OF(#SEG 1 #OF stmnt))" starts at the node stm̃nt, finds its leftmost descendent and then, in turn, finds its "first" descendent.]

[Note: Normally, the SEG's of a node correspond to the set-concatenands in one of the alternatives of the node's Syndef.]

```
#DF Sum

        => <'#NEG'> <gap> <Product>
        <%<<Additive-operator> <Product>>>

        => <Product> <%<<Additive-operator> <Product>>>
        #.

#DF Additive-operator

        => <gap> <'+','#BOR','#BXOR'> <gap>

        => <special-gap> <'-'> <special-gap> #.

#DF Special-gap

        => <%1<#SPACE>>

        => <#GAP> <Special-delimiters> <#GAP> #.
```

Note: Also see Chapter II, Section H, "SEMANOL(76) Arithmetic".

Integer arithmetic

#NEG a

<error> if ă is not an integer;

Otherwise: the standard form integer numeral which is the negation of ă, in the same base as ă. That is, if ă has a leading minus sign, it is erased; otherwise, it is added.

a + b

<error> if ă (or ŏ) is not an integer;

Otherwise: the standard form integer numeral, ĕ, for the sum of numbers denoted by ă and ŏ. If the bases of ă and ŏ differ, the base of ĕ is the minimum of the two bases; otherwise the base of ĕ

is that of ă.

a - b

<error> if ă (or ŏ) is not an integer;

Otherwise: the standard form integer numeral, č, for the difference of numbers denoted by ă and ŏ. If the bases of ă and ŏ differ, the base of č is the minimum of the two bases; otherwise the base of č is that of ă.

Bit-string-arithmetic

a #BOR b

<error> if ă (or ŏ) is not a bit-string;

Otherwise: the bit string numeral, č, obtained by

(1) padding with leading zeros the shorter of ă and ŏ;

(2) constructing a result numeral č using bit-wise inclusive-or logic, given in the table below.

(i-th bit in ă) + (i-th bit in ŏ)	0	1
0	0	1
1	1	1

[Example:

"110#BITS #BOR 10101#BITS" evaluates to

"10111#BITS"]

a #BXOR b

<error> if ă (or ŏ) is not a bit-string;

Otherwise: The bit string constant, č, obtained by

(1) padding with leading zeros the shorter of ă and ŏ;

(2) constructing a result numeral using bit-wise exclusive-or logic, as given in the table below:

(i-th bit in ă) → / (i-th bit in ŏ) ↓	0	1
0	0	1
1	1	0

[Example:

"110#BITS #BXOR 10101#BITS" evaluates to "10011#BITS"]

#DF Product

=> <Primary> <%< <gap> <Multiplicative-operator> <gap> <Primary>>> #.

#DF Multiplicative-operator

=> <'*'>

=> <'/'>

=> <'#BAND'> #.

Integer arithmetic

a * b

<error> if ă (or ŏ) is not an integer;

Otherwise: the standard form integer numeral, č, for the product of numbers denoted by ă and ŏ. If the bases of ă and ŏ differ, the base of č is the minimum of the two bases; otherwise the base of č is that of ă.

a / b

<error> if ă (or ŏ) is not an integer;

<undefined> if "#SIGN(b) =0";

Otherwise: the standard form integer numeral, č, for the "integer portion" of the quotient of *numbers denoted by ă and ŏ. If the bases of ă* and ŏ differ, the base of č is the minimum of the two bases; otherwise the base of č is that of ă.

Note: This is truncated division; thus "3/2" evaluates to "1" and "(-3)/2" evaluates to "-1".

Bit-string-arithmetic

a #BAND b

<error> if ă (or ŏ) is not a bit-string;

Otherwise: the bit string constant, č, obtained by

(1) padding with leading zeros the shorter of ă and ŏ:

(2) constructing a result numeral using bit-wise and logic as given in the table below:

(i-th bit in ă)		
*	0	1
0	0	0
1	0	1

(i-th bit in ŏ)

[Example:

"110#BITS #B AND 10101#BITS" evaluates to "00100#BITS"]

#DF Primary

=> <'('> <gap> <Expression> <gap> <')'>

```
                => <Semantic-def-reference>

                => <Special-function>

                => <Numeric-function>

                => <Sequence-function>

                => <String-constant>

                => <Numeric-constant>

                => <Boolean-constant>

                => <Sequence-constant>

                => <Undefined-constant>

                => <Name> #.


 #DF Semantic-def-reference

                => <Semantic-definition-name> <gap> <'('> <gap>
                <Expression> <gap> <%<<Comma> <gap> <Expression>
                <gap>>> <')'>

                => <'($'> <gap> <Expression> <gap> <%<<Comma>
                <gap> <Expression> <gap> >> <'$)'> <gap>
                <Semantic-definition-name> #.

-----------------------


 c

 c ( a1, ..., an )

 ($ a1, ..., an $) c

                <error> if there is no semantic definition in the
                SEMANOL(76) program with the semantic definition
                name "c".

                Otherwise: the result of evaluating the semantic
                definition reference as described in the
                discussion of <Semantic-definition> above.
```

```
----------------------

 #DF Special-function

              => <External-function>

              => <Assignment-function>

              => <File-manipulation>

              => <Tree-function> #.
----------------------
```

```
----------------------

 #DF External-function

              => <'#EXTERNAL-CALL-OF'> <gap>
              <String-expression>

              => <'#EXTERNAL-CALL-OF'> <gap>
              <String-expression> <gap> <'#WITH-ARGUMENT'>
              <gap> <'('> <gap> <Sequence-expression> < gap>
              <')'> #.
----------------------
```

```
 #EXTERNAL-CALL-OF a

 #EXTERNAL-CALL-OF a #WITH-ARGUMENT (\b1,...,b2\)

 #EXTERNAL-CALL-OF a #WITH-ARGUMENT ( b )

              <error> if a (or b1,...,or bn) is not
              string-convertible, or if b is not a (possibly
              empty) sequence of string-convertible-objects.

              <error> If any error condition results from a
              call to an (external) function having as its name
              the string value of a, and as its arguments the
              string values of b1 through bn.  An error
              condition  includes the following: (1) the
              external function returns a non-string result;
```

(2) the external function is not available in the environment of the executing SEMANOL(76) program when the call is made (i.e., the call cannot be completed);

Otherwise: the string result of calling an available external function having as its name the string value of a, and as its arguments the string values of b1 through bn.

#DF Assignment-function

=> <'#LATEST-VALUE'> <gap> <'('> <gap> <String-expression> <gap> <')'>

=> <'#ASSIGN-LATEST-VALUE'> <gap> <'('> <gap> <String-expression> <gap> <Comma> <gap> <Expression> <gap> <')'> #. #.

#LATEST-VALUE (a)

<error> if ā is not string-convertible;

Otherwise: v̄, where s is the string-value of ā and (s,v̄) is the most recently added pair (which has first element s) to the global assignment sequence.

#ASSIGN-LATEST-VALUE (a,b)

<error> if ā is not string-convertible;

Otherwise: the result value is <nil>; as a side effect of evaluation, the pair (s,b̄) is added to the global assignment sequence, where s is the string-value of ā.

#DF File-manipulation

=> <'#INPUT'>

=> <'#OUTPUT'> <gap> <'('> <gap>
<String-expression> <gap> <')'>

=> <'#GIVEN-PROGRAM'> #.

#INPUT	#INPUT denotes a special argumentless function which returns the value of a unique special input variable. This variable is assigned a string value, implicitly, prior to the execution of the SEMANOL(76) program; it cannot be modified during execution.
#GIVEN-PROGRAM	#GIVEN-PROGRAM denotes a special argumentless function which returns the value of a unique special given-program input variable. This variable is assigned a string value, implicitly, prior to the execution of the SEMANOL(76) program; it cannot be modified during execution.
#OUTPUT (s)	<error> if s is not string convertible; Otherwise: #OUTPUT denotes a special function which returns the value <nil>, and as a side effect, concatenates s to the rightmost end of the latest string value of a unique special output variable, and assigns the resulting string as the (new) latest value of the special output variable. This variable always has the value <nil> at the initiation of the execution of the SEMANOL(76) program; it can only be altered by the "#OUTPUT" function. [Note: The exact manner in which the implicit assignments are made to the two special input variables depends upon the particular SEMANOL(76) implementation, as does the exact mechanism by which the value of the special output variable may be inspected.]

III. Description of SEMANOL(76)

#DF Tree-function

=> <'#CONTEXT-FREE-PARSE-TREE'> <gap> <'('> <gap>
<String-expression> <gap> <Comma> <gap>
<Syntactic-class-reference> <gap> <')'>

=> <'#STRING-OF-TERMINALS-OF'> <gap> <'('> <gap>
<String-expression> <gap> <')'>

=> <'#PARENT-NODE'> <gap> <'('> <gap>
<String-expression> <gap> <')'>

=> <'#ROOT-NODE'> <gap> <'('> <gap>
<String-expression> <gap> <')'>

#CONTEXT-FREE-PARSE-TREE (a,<b>)

<error> if a is not string-convertible or if b is not a syntactic definition *name* for a syntactic definition in the context free syntax section;

<error> if the grammar G(b), with start symbol b, is ambiguous with respect to the string value of a (See Section 3 in Appendix A);

<undefined> if the string value of a cannot be derived from the start symbol b of the grammar G(b);

Otherwise: The ordered parse tree obtained by parsing the string value of a with respect to the grammar G(b) to be found in the context free syntax section. Actually, the result value of this operation is the root node for the parse tree, from which the rest of the tree is accessible via other SEMANOL(76) functions.

Complete definition of the construction of the parse tree is given in the discussion in Appendix A.

The SEMANOL(76) parse tree has a few extensions over the most common notions of a parse tree; each non-terminal node has certain special labels or attributes: (1) syntactic class name, (2) case

(or definiens) number, (3) seg count, and (4) syntactic component pairs.

Each non-terminal node corresponds to a (sub)string of the parsed string, which belongs to a syntactic class defined by a syntactic definition (whose name is recorded as a label on the node). Each of its immediate descendants ("seg"s) correspond to a syntactic set (with a terminal or non-terminal syntactic class name) concatenand in one of the definiens of the definition. The segs of a node are ordered (left-to-right), and are given numbers from 1 to the seg-count of the node. The definiens used to produce the descendents is labeled (by case number) in the parent node; the total number of segs is also labeled in the parent node.

#STRING-OF-TERMINALS-OF (a)

<error> if ā is not a parse tree node;

Otherwise: The string, x, represented by ā as described in Section 5 of the discussion in Appendix A.

#ROOT-NODE(b)

<error> if "b #IS-NOT #NODE" holds;

Otherwise: the unique node, in the (most-inclusive) parse tree containing b̄, which has no parent; i.e., that node ā such that all nodes c̄ in the parse tree containing b̄ have the property that "c #IS #NODE-IN a".

Note: A parse tree ā "includes" parse tree b̄ if every node contained in b̄ is contained in ā. Also, ā "properly includes" b̄ if ā includes b̄ and there is some node c̄ contained in ā which is not contained in b̄. A parse tree is "most-inclusive" if it is not properly included in any parse tree.

#PARENT-NODE(b)

<error> if "b #IS NOT #NODE" holds;

<error> if "b #EQN #ROOT-NODE(b)" holds;

Otherwise: the unique node, ă, in the (most-inclusive) parse tree containing ŏ, for which there is an integer ĭ such that "b #EQN #SEG i #OF a" holds.

This keyword expression is equivalent to

"#LAST-ELEMENT-IN (#SEQUENCE-OF-ANCESTORS-OF b)".

Note: A parse tree ă "includes" parse tree ŏ if every node contained in ŏ is contained in ă. Also, ă "properly includes" ŏ if ă includes ŏ and there is some node č contained in ă which is not contained in ŏ. A parse tree is "most-inclusive" if it is not properly included in any parse tree.

#DF Numeric-function

=> Arithmetic-function

=> Count-length-function

#DF Arithmetic-function

=> <'#ABS'> <gap> <'('> <gap> <String-expression> <gap> <')'>

=> <'#SIGN'> <gap> <'('> <gap> <String-expression> <gap> <')'>

=> <'#CONVERT'> <gap> <'2', '8', '10'> <gap> <'('> <gap> <String-expression> <gap> <')'> #.

#ABS (a)

<error> if ă is not an integer;

Otherwise: the standard form numeral obtained from ā by

(1) converting ā to standard form, and

(2) taking ā if ā has no leading minus sign; otherwise

(3) erasing the leading minus sign of ā.

#SIGN (a)

<error> if a is not an integer;

Otherwise: the (standard form decimal integer) numeral given below:

"0" if "a = x" where x̄ is a standard form zero of the same numeric type as ā;

"-1" if ā has a leading minus sign;

"1" otherwise.

#CONVERT b (m) , where b is 2, 8, or 10

<error> if m̄ is not an integer;

Otherwise: the base b standard form representation of m̄.

#DF Count-length-function

=> <'#SEG-COUNT'> <gap> <'('> <gap> <String-expression> <gap> <')'>

=> <'#LENGTH'> <gap> <'('> <gap> <Sequence-expression> <gap> <')'> #.

#SEG-COUNT (a)

<error> if ā is not a parse tree node;

0 if ā is a terminal node (i.e., if ā corresponds to one of the Syntactic set constants in the parse of a given (sub)string -- see the discussion of the "#CONTEXT-FREE-PARSE-TREE..." operator, above).

Otherwise: The standard form decimal integer numeral, n, such that ā has n immediate descendants in the parse tree of which ā is the root (see the discussion of the "#CONTEXT-FREE-PARSE-TREE..." operator, above).

#LENGTH (a)

<error> if neither of the following conditions holds:

(1) ā is a sequence; (2) ā is string convertible.

CASE 1 (ā is a sequence)

The standard form integer numeral n, where n̄ is the number of elements in ā. ("0" if ā is the empty sequence <nilseq>.)

CASE 2 (ā is string convertible)

The integer numeral n, where n̄ is the number of characters in the string value of a ("0" if the string value of a is <nil>).

#DF Sequence-function

=> <'#REVERSE-SEQUENCE'> <gap> <'('> <gap> <Sequence-expression> <gap> <')'> #.

#REVERSE-SEQUENCE (s̄)

<error> if s̄ is not a sequence;

Otherwise:

<nilseq> if s̃ is <nilseq> ; otherwise,

The sequence obtained by taking the elements of s̃ in reverse order.

#DF Undefined-constant

=> <'#UNDEFINED'> #.

#DF Boolean-constant

=> <'#TRUE','#FALSE'> #.

#DF Sequence-constant

=> <'#NILSEQ'>

=> <'\'> <gap> <Expression> <gap> <%<<Comma> <gap> <Expression> <gap>>> <'\'> #.

#UNDEFINED	<undefined>, a distinguished value.
#TRUE	<true>
#FALSE	<false>
#NILSEQ	The empty sequence , <nilseq>
\ a \	The sequence containing the single element, ã.
\ a1, ... ,an \	The sequence containing n elements, whose i-th element is ãi.

[This is equivalent to

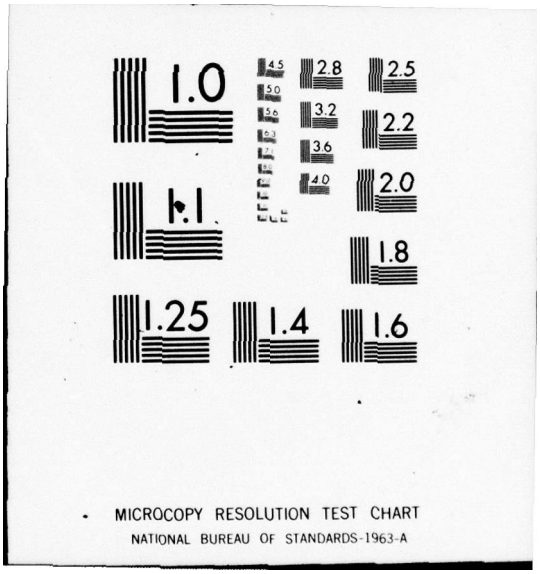
"\a1\ #CS ... #CS \an\" .]

AD-A049 473 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SEMANOL(76) REFERENCE MANUAL. VOLUME II.(U)
NOV 77 F C BELZ

UNCLASSIFIED F30602-76-C-0238
RADC-TR-77-365-VOL-2 NL

2 OF 2
ADA049473

END
DATE FILMED
3-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

III. Description of SEMANOL(76)

```
#DF String-constant
            => <String> #.
#DF String
            => <'['> <%<String-character>> <']'>
            => <'#SPACE'>
            => <'#NIL'> #.
#DF String-character
            => <Printing-ascii-character> #S- <'['] , '[[]'>
            => <'[[]'> <Escape-sequence> <']'> #.
#DF Escape-sequence
            => <'['], 'ENQ', 'FF', 'DC3', 'SUB',
                '[[]', 'ACK', 'CR', 'DC4', 'ESC',
                'NUL', 'BEL', 'SO', 'NAK', 'FS',
                'SOH', 'BS', 'SI' 'SYN', 'GS',
                'STX', 'HT', 'DLE', 'ETB', 'RS',
                'ETX', 'LF', 'DCI', 'CAN', 'US',
                'EOT', 'VT', 'DC2', 'EM', 'DEL'>
#DF Printing-ascii-character
            => <#LETTER>
            => <#DIGIT>
            => <#SPACE, '!', '"', '#', '$', '%',
                    '&', ['], '(', ')', '*', '+',
                    ',', '-', '.', '/', ':', ';',
                    '<', '=', '>', '?', '@', '[[]',
```

'\', ']', '^', '-', '`', '{',

'|', '}', '~'>

'x'

The string "x". The string constant, like every other SEMANOL(76) <Symbol>, is indivisible; that is, the evaluation of a string constant is direct and cannot be the composition of any other evaluative steps. In particular, if " 'a' " is a string constant appearing in an expression in a semantic definition which has a formal parameter named "a", the value of the string constant is the string "a", not the value of the actual argument associated with the formal parameter.

A special notation is used to represent non-printing ASCII characters, and imbedded single-quotes within a quoted string. The square bracket characters are used to enclose the standard ASCII name of the non-printing character. For example, '[CR][LF]' denotes the string of two characters -- ASCII carriage-return followed by ASCII line-feed. The same square brackets enclose the single-quote character. For example, '[']' denotes the string consisting of the ASCII single-quote character. Finally, it is necessary to use the special bracket notation to represent the "[" character, since that character normally serves as an "escape" character for this notation. Thus, '[[]' denotes the single "[" character; '[[]HI]' denotes the string "[HI]".

Note that the ASCII line-feed character itself (for example) cannot appear in a string, only the characters [LF], which stand for the line-feed. Thus the SEMANOL(76) end-of-line character cannot appear in a string; that is, string constants cannot be split across lines.

#SPACE The string consisting solely of ASCII space or blank character.

#NIL The empty string <nil>.

```
-----------------------

#DF Numeric-constant
                => <Integer-constant>
                => <Bit-string-constant>  #.
#DF Integer-constant
                => <Integer>
                => <Octal-integer> <'#B8'>
                => <Binary-integer> <'#B2'>
                => <'-'> <Integer>
                => <'-'> <Octal-integer> <'#B8'>
                => <'-'> <Binary-integer> <'#B2'>  #.
#DF Integer
                => <%1<#DIGIT>> #.
#DF Binary-integer
                => <%1<'0','1'>> #.
#DF Octal-integer
                => <%1<'0','1','2','3','4','5','6','7'>> #.
#DF Bit-string-constant
                => <Binary-integer> <'#BITS'> #.

-----------------------


XX:::X
```

XX:::X#B8

XX:::X#B2

Each integer numeral constant evaluates to itself as a SEMANOL(76) string object.

The integer portion, "XX:::X", may be any string of decimal digits, if there is no base suffix (or octal digits if the base suffix is "#B8"; or binary digits if the base suffix is "#B2"); no imbedded blanks are permitted.

XX:::X#BITS

Each bit-string constant evaluates to itself as a string object.

The integer portion, "XX:::X", may be any string of binary digits (with no imbedded blanks).

"**********

Basic Symbols

**********"

#DF Comma

=> <',>

=> <'#,'> <Name> #.

#,commentary-characters

,

The SEMANOL(76) comma may include some optional commentary characters for readability. WARNING: the use of "#," must always be immediately

followed by noise characters, with no gap and no imbedded blanks. There is some danger that the SEMANOL(76) programmer may forget the noise characters and/or their terminating blank, thus erroneously consuming the next name as noise. For example, the definition reference "blah(a#,b)" is erroneous.

```
#DF Comment

        => <'"'> <% <#ASCII #S- <'"'>>> <'"'> #.
```

" blah "

A SEMANOL(76) comment may appear in any gap in a SEMANOL(76) program. It is ignored during execution of the SEMANOL(76) program. The first occurrence of a double-quote after the opening double-quote terminates the comment.

```
#DF gap

        => <#GAP>

        => <#GAP> <Special-delimiters> <#GAP> #.

#DF Special-delimiters

        => <Special-delimiter>
        <%1<<#GAP><Special-delimiter>>> #.

#DF Special-delimiter

        => <Comment> #U <End-of line> #.

#DF End-of-line

        => <'[LF]'> #.
```

III. Description of SEMANOL(76)

The character '[LF]' partitions the SEMANOL(76) program into a sequence of "lines", which are substrings over the alphabet <#ASCII> #S-<'[LF]'>. No line may be greater than 72 characters in length.

```
#DF Name

        => <#LETTER> <%<<#LETTER> #U <#DIGIT> #U <'-'>>>
        #.
```

The possible meanings of names depend upon context. Some rules concerning names are given in Section E of Chapter II.

Appendix A

The following paragraphs are given to fill in the details of the SEMANOL(76) implementation of the notions of grammars, derivations, derivation- and parse-trees, syntactic components (but also see Chapter II), and the string of terminals of any node of a parse tree.

1. Derivations

The derivation of a string, s, from the "start-symbol", S, of a grammar, G(S), can best be explained by an example.

```
  Definition                                    (Definiens number)

#DF  S  => <T> <%1<<'+'> <T>>> #.                    (1)

#DF  T

   => <V>                                            (1)

   => <V> <'*'> <T> #.                               (2)

#DF  V

   => <#CAP> <%<#CAP #U #DIGIT>> #.                  (1)
```

Fig. A.1. Example grammar (with start symbol S).

Step (a): To begin, rewrite the grammar so that all syntactic class names are bracketed (G(S) is in this form). Then write the start-symbol, bracketed.

Step (b): Then we perform a direct-derivation, by (1) choosing a bracketed Syntactic class name in the current string; (2) choosing a definiens in the Syntactic definition which defines that Syntactic class name; and (3) replacing the bracketed Syntactic class name with the chosen definiens. If the definiens is a syntactic expression of the form "a #S- <

b1, ..., bn >", then bracket it before performing the replacement. At step (b), there are no choices, so we perform the only possible direct-derivation. We also record the (Syntactic class name; definiens number) pair used to make the derivation.

Step (c): After performing each direct-derivation, we check for spontaneous-derivations, which must be performed if there are any occurrences of the #U, %, or %1 operators. Corresponding to each occurrence of such an operator, O, is an "O-induced" spontaneous-derivation. (If any derivation contains more than one such operator occurrence, they may be nested by the parenthesizing brackets; in that case the induced spontaneous derivations are done "from the outside inward".) At step (c), a %1-induced spontaneous derivation is performed. Where the expression <%1<a>> appears, we choose a positive integer, n, and replace the expression with n occurrences of <a>; at step (c), n=1.

Spontaneous derivations must be performed in left to right order (unless the inducing operators are nested), and all inducing operators must be eliminated before proceeding to the next direct-derivation. When all such operators have been eliminated the direct-derivation and the subsequent chain of spontaneous derivations (if any) constitute a complete derivation step. Thus steps (b & c) constitute a complete derivation step.

Step (d): Perform a direct derivation, replacing the second occurrence of <T> with its second definiens, <V> <'*'> <T>. Since this derivation introduces none of the inducing operators, this is a complete derivation step.

Step (e): Replace the first occurrence of <T> with its first definiens, <V>. Complete derivation step.

Step (f): Replace the first occurrence of <V> with its only definiens. This introduces two (nested) inducing operators, % and #U. Thus we are forced to perform a %-induced spontaneous derivation, followed by a #U-induced spontaneous derivation:

```
(a)     <S>
          |
          |(S;1)
          v
(b)  <T><%1<<'+'><T>>>
          |
          |(%1)
          v
(c)  <T><'+'><T>
          |
          |(T;2)
          v
(d)  <T><'+'><V><'*'><T>
          |
          |(T;1)
          v
(e)  <V><'+'><V><'*><T>
          |
          |(V;1)
          v
(f)  <#CAP><%<#CAP #U #DIGIT>><'+'><V><'*'><T>
          |
          |(%)
          v
(g)  <#CAP><#CAP #U #DIGIT><#CAP #U #DIGIT><'+'><V><'*'><T>
```

Fig. A.2. Derivation of String S1 (part 1)

```
(g)  <#CAP><#CAP #U #DIGIT><#CAP #U #DIGIT><'+><V><'*'><T>
          |
          |(#U(twice))
          v
(h)  <#CAP><#DIGIT><#DIGIT><'+'><V><'*'><T>
          |
          |(V;1)
          v
(i)  <#CAP><#DIGIT><#DIGIT><'+'><#CAP><%<#CAP #U #DIGIT>>
                                                <'*'><T>
          |
          |(%)
          v
(j)  <#CAP><#DIGIT><#DIGIT><'+'><#CAP><'*'><T>
          |
          |(T;1)
          v
(k)  <#CAP><#DIGIT><#DIGIT><'+'><#CAP><'*'><V>
          |
          |(V;1)
          v
(l)  <#CAP><#DIGIT><#DIGIT><'+'><#CAP><'*'><#CAP>
          |                                <%<#CAP#U#DIGIT>>
          |(%)
          v
(m)  <#CAP> <#DIGIT> <#DIGIT> <'+'> <#CAP><'*'><#CAP>
        |       |       |      |      |    |     |
        v       v       v      v      v    v     v
(n)     A       1       2      +      B    *     C     = string S1
```

Fig. A.2. Derivation of String s1 (part 2)

Step (g): If the expression <%<a>> appears, we choose a non-negative integer n and replace the expression by n occurrences of <a>; here n = 2. This is a %-induced spontaneous derivation.

Step (h): If any of the expressions <a1 #U a2> ,..., <a1 #U...#U an> appear, choose a positive integer m (<=n) and replace the expression with <am>. Here m=2 is chosen twice,

and two #U-induced spontaneous derivations are performed.

There are no inducing operators left. Thus steps (f-h) constitute a complete derivation step.

Steps (i thru m): Proceed as discussed above, until there are no more Syntactic class names and no more inducing operators in the most recent string. All bracketed expressions will contain Syntactic set constants. This condition is known as derivation up to terminal nodes.

Step (n): The terminal derivation is performed by selecting a string from the set denoted by each Syntactic Set Constant and replacing the corresponding Set Constant with its string.

Our derivation is complete.

Note that what we have done here is to describe a method for producing members of the set denoted by the start-symbol.

The strings constructed in any derivation (e.g., that of Fig. A.2) are called sentential forms. Each lettered line of Fig. A.2 is a sentential form. Only the last sentential form in a derivation is in the language generated by the start-symbol. Sentential-forms which contain any of the inducing operators, %, %1, or #U, are called temporary forms. The rest are called permanent forms.

There are some special amendments to and constraints upon the terminal derivation:

(1) #S-

Suppose the grammar of the above example had the following syntactic definition for "V" rather than the one give above:

```
V => <#CAP> <% <#CAP #U #DIGIT>> #S- <'A12'>
```

(This says that no derivation may produce "A12" from the syntactic class V). If the sample derivation were repeated with the amended grammar, the sentential form at line (m) would be:

```
(m') < <#CAP><#DIGIT><#DIGIT> #S- <'A12'>>
     <'+'> <<#CAP> #S- <'A12'>>
     <'*'> <<#CAP> #S- <'A12'>>
```

There are five "bracketed components" of (m'):

(1) <<#CAP><#DIGIT><#DIGIT>#S-<'A12'>>

(2) <'+'>

(3) <<#CAP>#S-<'A12'>>

(4) <'*'>

(5) <<#CAP>#S-<'A12'>>

Bracketed components (1), (3), and (5) are said to be "#S-conditioned"; the selection of strings from the syntactic set constants in each bracketed component must be such that the resultant "component string" is not among those listed in the #S- condition (the set of strings following the "#S-").

For example, the string S1 of Fig. A.2 would no longer be derivable from S, since "A12" is specifically excluded from syntactic class "V". However the string "A13+B*C" is derivable from either the original or the modified grammar.

2. Some Special Cases

Define the relation "directly-derives" or -> as follows: A -> B if and only if A and B are permanent forms and there is a single complete derivation step (or a terminal derivation) which (given A) will produce B. This relation is understood to be relative to the particular grammar given in a particular Context Free Syntax Section; also if a denotes A and b denotes B and A -> B then we say a -> b. For example, in Fig. A.2, (a) -> (c); (c) -> (d); (d) -> (e); (e) -> (h); (h) -> (i); (j) -> (k); (k) -> (m); and (m) -> (n). No other pairs of line letters are in the relation.

Define the relation "derives" or *-> as the transitive closure of ->; that is A *-> B if and only if A -> B or there exist permanent forms a1, a2, a3.....an such that A -> a1 -> a2...-> an -> B. (Also, if a denotes A and b denotes B and A *-> B, then a *-> b).

Thus in Fig. A.2., <S> *-> "A12+B*C" or equivalently (a) *-> (n); also (a) *-> (h); (d) *-> (k); etc.

Let "right (c)" denote the right-most character of the string c (if c is not the empty word; and c otherwise). Let "left (c)" be defined symmetrically.

If S is the start-symbol of a grammar of the context free syntax section, and

(1) <S> *-> ABC *-> abc ; and

(2) A *->a; B *->b; C *->c; and

CASE 1

(3) B is "<#DECNUM>"

or

B is "<#NAT-NOS>"

then left(c) may not be an ASCII digit (from the set named #DIGIT).

CASE 2

(3) B is "<#GAP>"

then left(c) may not be an ASCII blank character; and b may be <nil> only if right(a) and left(c) are not both ASCII alphanumeric characters.

As an example of the #GAP case, recall the definition of n-place set concatenation, specified in SEMANOL(76) by <b1><b2>...<bn>, where the bi are syntactic expressions denoting sets of strings. The SEMANOL(76) constant <b1><b2>...<bn> denotes the set of strings A={ a : a="a1a2...an" and "a1" is in ƀ1; "a2" is in ƀ2;...; "an" is in ƀn}.

Normally, if the empty string <nil> is in ƀi (1<i<n) then A contains

a=a1a2...a(i-1)<nil>a(i+1)...an

=a1a2...a(i-1)a(i+1)...an.

However, suppose that bi is the syntactic expression #GAP. Then A contains a = a1a2...a(i-1)<nil>a(i+1)...an if and only if right (a1a2...a(i-1)) is not alphanumeric or left (a(i+1)...an) is not alphanumeric.

3. Ambiguity

A grammar in the context free syntax section is ambiguous if there exist two different derivations for any string in the language of the grammar. A grammar, G(S), is ambiguous with

respect to a particular string if there exist two different derivations for that string from the start symbol S of the grammar.

If the expression "#CONTEXT-FREE-PARSE-TREE(S,p)" is evaluated, where G(S) is ambiguous with respect to the string value of p, the result is <error>, as described above.

4. Derivation and Parse Trees

From the list of derivations in Fig. A.2 we can construct a tree representation of the derivation process. Fig. A.3 shows the first three steps in the construction of such a derivation tree.

Note that only the permanent forms (a), (c), (d), (e), (h), (j), (k), (m), and (n) are reflected in the construction of the derivation tree (although the "definiens number" of each direct derivation is also recorded). Each addition to the derivation tree (begun in Fig. A.3 (a)) corresponds to a complete derivation step in the derivation of Fig. A.2.

Step (a). The "parent" or "root" node is labeled with the start-symbol.

Steps (b-c). The "immediate descendent" nodes of any parent are labeled with the syntactic class names obtained from the parent by a complete derivation step. The "definiens number" used in the direct derivation (for that step) is appended to the parent node's label.

Step (d). The process is repeated for the next complete derivation step. Fig. A.4 shows the derivation tree "up to terminal nodes".

The Interpreter uses the Context-Free Syntax Section to recognize programs rather than to generate them. However, in the recognition process, a parse tree is formed whose basic structure is that of the derivation tree.

In terms of parsing a given string the following interpretations may be made regarding certain derivation rules given above:

1. A substring which is parsed by virtue of a definiens containing the operators #U, %, or %1, will produce a single level in the subtree corresponding to the definiens (i.e., Levels are added to the parse tree only for explicitly named syntactic classes; put another way, temporary forms are not

reflected in the parse tree).

2. The special case Syntactic set constants #DECNUM and #NAT-NOS which are used to parse a a substring ai...an will "consume" all the consecutive numeric digits of that portion of the overall string being parsed (i.e., character a(n+1) will never be an ASCII digit). Similary, if #GAP is used to parse "ai...an", character "a(n+1)" will never be an ASCII blank character. Furthermore "ai...an" can never be <nil> if both characters "a(i-1)" and "a(n+1)" are alphanumeric.

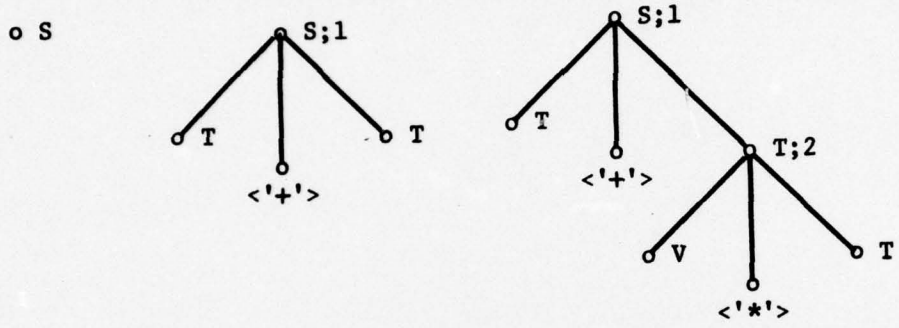


Fig. A.3 Construction of a Derivation Tree

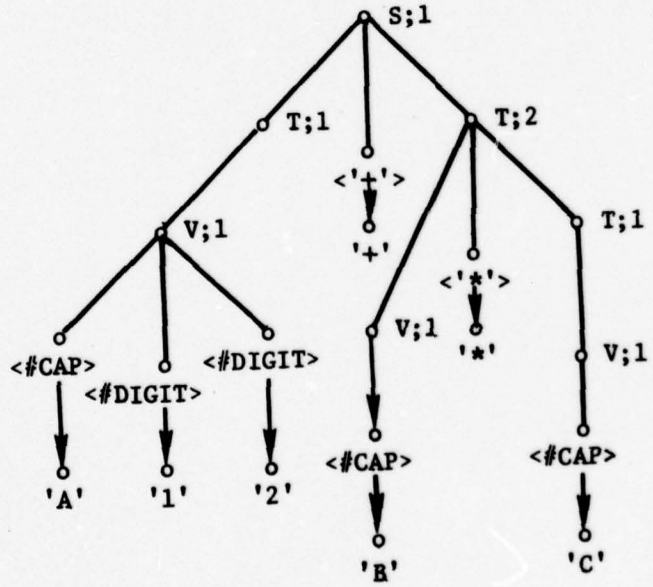


Fig. A.4 Derivation Tree "up to Terminal Nodes"

5. The string of terminals of a node

Now we can write the derived string by tracing the tree in "preorder". For each immediate descendent di of the parent node (<i=1,...> in left to right order), if di is a terminal node write down its associated string; otherwise trace the subtree of di, using di as the parent. The string thus written is the one represented by the derivation tree.

In a similar manner, each non-terminal node corresponds to a particular (sub)string occurrence of the derived string. See Fig. A.4: the first descendent of the parent node can represent the initial substring "A12" of S1; the second descendent of the parent represents the substring "+"; the third represents "B*C".

Note that the parse tree does not contain the strings of terminals themselves. Every node and the string of terminals it represents are separate and distinct; one can obtain the string from the node by using the "#STRING-OF-TERMINALS-OF..." operator as described above.

INDEX

! 27, 93

" 93, 97

93

#. 24

$ 93

$) 83

% 93

% 31

& 42, 93

(31, 37, 53, 54, 62, 64, 73, 82, 83, 84, 85, 86, 87, 89, 90, 91, 93

($ 83

) 31, 37, 53, 54, 62, 64, 73, 82, 83, 84, 85, 86, 87, 89, 90, 91, 93

* 81, 93

+ 79, 93

, 93, 96

- 79, 93, 94, 95

. 93

/ 81, 93

: 21, 23, 24, 29, 35, 53, 54, 73, 93

; 38, 93

< 32, 49, 93

<= 25

= 27, 56, 93

=> 29, 37, 38

> 32, 49, 93

? 93

@ 93

` 94

[31

{ 94

['] 93

[ACK] 93

[BEL] 93

[BS] 93

[CAN] 93

[CR] 93

[DC2] 93

[DC3] 93

[DC4] 93

[DCI] 93

[DEL] 93

[DLE] 93

[EM] 93

[ENQ] 93

[EOT] 93

[ESC] 93

[ETB] 93

[ETX] 93

[FF] 93

[FS] 93

[GS] 93
[HT] 93
[LF] 93
[NAK] 93
[NUL] 93
[RS] 93
[SI] 93
[SOH] 93
[SO] 93
[STX] 93
[SUB] 93
[SYN] 93
[US] 93
[VT] 93
[[] 93
\ 92, 94
¦ 94
] 31, 94
} 94
^ 94
_ 94
#, 96
#,Name 96
#. 21, 23, 29, 35, 39
#ABS 89

INDEX

#AND 42

#ASCII 32

#ASSIGN-LATEST-VALUE 85

#ASSIGN-VALUE 27

#B2 95

#B8 95

#BAND 81

#BEGIN 25

#BIT-STRING 46

#BITS 95

#BOOLEAN 46

#BOR 79

#BXOR 79

#CAP 32, 48

#CASE 51

#CHARACTERS-OF 67

#COMPUTE 27

#CONTEXT-FREE-PARSE-TREE 87

#CONTEXT-FREE-SYNTAX 29

#CONTROL-COMMANDS 24

#CONVERT 89, 90

#CS 63

#CW 71

#DECLARE-GLOBAL 23

#DECLARE-SYNTACTIC-COMPONENT 21

#DECNUM 32

#DF 29, 35, 38

#DIGIT 32, 48, 93

#DO 25

#DOES-NOT-PRECEDE 53

#EMPTYSET 32

#END 25

#EQ 56

#EQN 56

#EQS 56

#EQW 56

#ERROR 41

#EXTERNAL-CALL-OF 84

#FALSE 92

#FIRST 73

#FIRST-CHARACTER-IN 67

#FIRST-ELEMENT-IN 73

#FOR-ALL 54

#GAP 32

#GIVEN-PROGRAM 86

#IF 25, 38

#IFF 42

#IMPLIES 42

#IN 52, 53, 54, 62, 64, 68, 72, 73

#INITIAL-SUBSEQ-OF-LENGTH 62

#INPUT 85

#INTEGER 46

#IS 45, 51

#IS-IN 45

#IS-NOT 45, 51

#IS-NOT-IN 45

#IT-IS-TRUE-THAT 54

#LAST 73

#LAST-CHARACTER-IN 67

#LAST-ELEMENT-IN 73

#LATEST-VALUE 85

#LEFT 67

#LENGTH 90

#LETTER 32, 48, 93

#LOWCASE 32, 48

#N 56

#N= 56

#NAT-NOS 32

#NEG 79

#NEQ 56

#NEQN 56

#NEQS 56

#NEQW 56

#NIL 93

#NILSEQ 92

#NILSET 32

#NODE 46

#NODE-IN 51

#NOT 42

#OF 51, 62, 68, 78

#OR 42

#ORDPOSIT 72

#OTHERWISE 38

#OUTPUT 86

#PARENT-NODE 87

#PRECEDES 52

#PREFIX-OF-FIRST 68

#PROC-DF 39

#RETURN-WITH-VALUE 27

#REVERSE-SEQUENCE 91

#RIGHT 67

#ROOT-NODE 87

#S- 31

#SEG 78

#SEG-COUNT 90

#SEMANTIC-DEFINITIONS 35

#SEQUENCE 46

#SEQUENCE-OF 64

#SEQUENCE-OF-ANCESTORS-OF 64

#SEQUENCE-OF-NODES 64

#SEQUENCE-OF-NODES-IN 64

#SIGN 89

INDEX

#SPACE 93

#SPACESET 32, 48

#STOP 41

#STRING 46

#STRING-OF-TERMINALS-OF 87

#SUBSEQUENCE 62

#SUBSEQUENCE-OF-ELEMENTS 62

#SUBSTRING-OF-CHARACTERS 68

#SUBSTRING-POSIT-OF 72

#SUBWORD 51

#SUCH-THAT 53, 54, 62, 64, 73

#SUFFIX-OF-FIRST 68

#TERMINAL-SUBSEQ-OF-LENGTH 62

#TH 73

#TH-CHARACTER-IN 67

#TH-ELEMENT-IN 73

#THEN 25

#THERE-EXISTS 53, 54

#TO 62, 68

#TRUE 92

#U 31

#UNDEFINED 46, 92

#WHILE 25

#WITH-ARGUMENT 84

10 89

2 89

INDEX

8 89

<Additive-operator> 79

<Arithmetic-function> 89

<Arithmetic-inequality> 59

<Assignment-function> 85

<Binary-integer> 95

<Bit-string-constant> 95

<Boolean-constant> 92

<Boolean-expression> 42

<Boolean-primary> 42

<Bounded-interval> 25

<Case-identification> 51

<Character-identification> 47

<Comma> 96

<Comment> 97

<Compound-statement> 25

<Conditional-definiens> 38

<Conjunction> 42

<Context-free-syntax> 29

<Control-section> 24

<Count-length-function> 90

<Declaration-section> 21

<Declaration> 21

<Declare-global-variables> 23

<Declare-syntactic-components> 21

INDEX

<Definiendum> 37

<Definition-by-cases> 38

<Disjunction> 42

<End-of-line> 97

<Equality-relation> 56

<Escape-sequence> 93

<Expression> 41

<External-function> 84

<File-manipulation> 85

<For-all-clause> 25

<For-statement> 25

<Formal-param-name> 37

<Functional-definition> 35

<gap> 97

<Greater-than-operator> 59

<Implication> 42

<Integer-constant> 95

<Integer> 95

<Less-than-operator> 59

<Membership-operator> 45

<Membership-relation> 44

<Multiplicative-operator> 81

<Name> 98

<Negation> 42

<Numeric-constant> 95

<Numeric-function> 89

<Octal-integer> 95

<Optional-context-free-syntax> 19

<Optional-semantic-definition-se19

<Position-detector> 72

<Precedes-relation> 52

<Primary> 82

<Printing-ascii-character> 93

<Procedural-definition> 39

<Product> 81

<Quantifier-relation> 53

<Relational-expression> 44

<Seg-selector> 78

<SEMANOL-76-program> 19

<Semantic-def-reference> 83

<Semantic-definition-name> 37

<Semantic-definition-section> 35

<Semantic-definition> 35

<Sequence-concatenation> 63

<Sequence-constant> 92

<Sequence-constructor> 64

<Sequence-element-extractor> 73

<Sequence-expression> 61

<Sequence-function> 91

<Sequence-membership> 50

<Set-constant> 48

INDEX

<Simple-definition> 36

<Simple-statement> 27

<Special-delimiter> 97

<Special-delimiters> 97

<Special-function> 84

<Special-gap> 79

<String-character> 93

<String-concatenation> 71

<String-constant> 93

<String-expression> 67

<String> 93

<Subsequence-expression> 62

<Substring-extractor> 67

<Subword-relation> 51

<Such-that-clause> 73

<Sum> 79

<Syntactic-class-membership> 49

<Syntactic-class-name> 29

<Syntactic-class-reference> 49

<Syntactic-class-union> 49

<Syntactic-definition> 29

<Syntactic-expression> 31

<Syntactic-primary> 32

<Syntactic-set-constant> 32

<Syntactic-term> 31

<Tree-function> 87

<Type-identification> 46

<Type> 46

<Unbounded-interval> 25

<Unconditional-definiens> 37

<Undefined-constant> 92