Y 3. AT 7:22

ANL-7022

ANL-7022

Yale University
New Haven, Connecticut

Argonne National Laboratory

COGENT
Programming Manual

by

John C. Reynolds

KLINE SCIENCE LIBRARY
AEC DEPOSITORY COLLECTION

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

Printed in USA. Price $4.00. Available from the Clearinghouse for Federal Scientific and Technical Information, National Bureau of Standards, U. S. Department of Commerce, Springfield, Virginia

ANL-7022
Mathematics and Computers
(TID-4500, 42nd Ed.)
AEC Research and Development Report

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60440

COGENT

Programming Manual

by

John C. Reynolds

Applied Mathematics Division

March 1965

Operated by The University of Chicago
under
Contract W-31-109-eng-38
with the
U. S. Atomic Energy Commission

This report replaces the following documents, which are now obsolete:

1) J. C. Reynolds, A Proposal for a Generalized Compiler, Technical Memorandum No. 33, Applied Mathematics Division, Argonne National Laboratory, August 16, 1962 (unpublished).

2) J. C. Reynolds, COGENT, A Compiler and Generalized Translator, informally distributed lecture notes, December 3, 1962.

3) J. C. Reynolds, COGENT Programming Manual, informally distributed report, September 9, 1964.

TABLE OF CONTENTS

		Page
CHAPTER I.	BASIC CONCEPTS AND MAJOR FEATURES OF COGENT	7
	A. Introduction	7
	B. Productions	8
	C. Construction Trees	11
	D. List Structures	12
	E. The Syntax Analyzer	15
	F. Generator Definitions: Constants	18
	G. Expressions and Assignment Statements	19
	H. Control Statements and Failure	23
	I. Declarations	25
	J. Generator Elements	26
	K. Primitive Generators: Arithmetic	28
	L. Identifiers	29
	M. Output	30
CHAPTER II.	SPECIFICATION OF THE COGENT LANGUAGE	32
	A. List Structures	32
	1. Types of List Elements	32
	2. Representation and Storage Allocation	33
	B. Basic Symbols of the COGENT Language	35
	C. Overall Structure of a COGENT Program	36
	D. The Character Description: Character Definitions	37
	1. Input and Skip Codes	37
	2. Output Codes	38
	3. Standard Character Definitions	38
	E. The Syntax Descriptions: Productions	39
	1. Special Labels	40
	2. Multiple Labels	44
	3. Forcing Markers	44
	4. Preliminary Processing of Productions	45
	5. Formal Definition of Syntactic Concepts	45
	6. Restrictions on Production Sets	48
	F. The Syntax Analyzer	50
	1. Parsing	50
	2. Treatment of Ambiguities	51
	3. Forcing Markers	52
	4. Goal Specifiers	53
	G. Constants	54
	H. The Generator Description: Nesting of Generator Definitions	56
	I. Declarations and Input Variable Sequences	59
	1. Local Declarations	60
	2. Input Variable Sequences	61

TABLE OF CONTENTS

		Page
	3. Own Declarations	61
	4. Pseudoconstant Declarations	62
	5. Generator Declarations	62
	6. Identifier Declarations	64
	J. Expressions and Assignment Statements	64
	1. Expressions	65
	2. Direct Assignment Statements	66
	3. Synthetic Assignment Statements	66
	4. Analytic Assignment Statements	67
	K. Statement Labels, Control Statements, and the Flow of Control	68
CHAPTER III.	PRIMITIVE GENERATORS AND INTERNAL VARIABLES	71
	A. Internal Variables	71
	1. Properties of the Internal Variables	72
	2. Standard Primitives for the Internal Variables	77
	B. Testing Primitives	79
	C. List-handling Primitives	80
	D. Dummy Primitives	81
	E. Arithmetic Primitives	81
	F. Identifier-handling Primitives	82
	G. Character-scanning Primitives	84
	1. Basic Character-scanning Primitives	85
	2. Higher-level Character-scanning Primitives	88
	3. Examples of Character Scanning	91
	H. Identifier-creating Primitives	93
	I. Number-creating Primitives	94
	J. Output Primitives	96
	1. P-medium Output Primitives	98
	2. C-medium Output Primitives	99
	3. Simultaneous P- and C-medium Output Primitives	101
	4. B-medium Output Primitives	101
	K. Dump Primitives	102
	L. Tape-control Primitives	103
	M. Exit Primitives	103

TABLE OF CONTENTS

		Page
CHAPTER IV.	AN ILLUSTRATIVE COGENT PROGRAM: SYMBOLIC DIFFERENTIATION	104
	A. Syntax Description	104
	B. Precedence	106
	C. Algebraic Operations	108
	D. The Main Generator DIFF	112
	E. Additional Generators	115
REFERENCES		116

LIST OF FIGURES

No.	Title	Page
1.	Construction Tree for the Polynomial "-(A+B)*DC"	12
2.	List Structure Representing the Polynomial "-(A+B)*DC"	14
3.	List Structures Communicated between the Syntax Analyzer and Generators	17
4.	List Structure Denoted by the Parametric Constant (TERM/(FACTOR/1)*(FACTOR/4)*(FACTOR/3))	19
5.	List Structures Used by Synthetic and Analytic Assignment Statements	22
6.	List Structure for "-(A+B)*DC" Using the Special Label "$IDENT,1/"	41
7.	List Structure Given to a Generator When the Special Label $NOTRAN/ Is Used	42
8.	Compressed List Structure for "-(A+B)*DC" Using the Special Labels $NOP/ and $IDENT,1/	44
9.	Numbering of Hole Positions for the B (Binary Card Output) Medium	97

COGENT

Programming Manual

by

John C. Reynolds

CHAPTER I

BASIC CONCEPTS AND MAJOR FEATURES OF COGENT

A. Introduction

The COGENT (COmpiler and GENeralized Translator) programming system is a compiler whose input language is designed for the description of symbolic or linguistic manipulation algorithms. Although the system is intended primarily for use as a compiler compiler, i.e., a compiler that compiles other compilers, it is also applicable to such problem areas as algebraic manipulation, mechanical theorem-proving, and heuristic programming.

In designing the system, the major objective was to achieve both the programming conciseness of a syntax-directed compiler system and the full generality of a recursive list-processing program. A second objective was to avoid any interpretive operations in the programs compiled by COGENT, in order to maximize the running speed of these programs. In general, the design of the system has drawn heavily upon the earlier work of several authors, particularly the syntax-directed compiler methods of E. T. Irons,(1) the Compiler Compiler of Brooker and Morris,(2) and the LISP list-processing system of J. McCarthy.(3)

An initial version of the COGENT system has been written for the Control Data 3600 computer. This manual is intended primarily to describe the input language for this system; operating procedures and details of the specific machine implementation are not discussed.

The fact that COGENT may be used to compile other compilers leads to some confusing terminology which must be clarified. The input language of the system is called the COGENT language, and a program written in this language is called a COGENT program. This program will in turn describe the manipulation of one or more object languages; when the program is a compiler, the object languages will be the input and output languages of the compiler. In general, the object languages are arbitrary languages whose characteristics are specified by part of the COGENT program, so that the COGENT language itself is a metalanguage which describes the object languages.

The COGENT system is a compiler which runs on the Control Data 3600 and translates COGENT language into 3600 assembly language. When the system is used as a compiler compiler, its output is a compiler which must run on the 3600 but which may translate an arbitrary input language into an arbitrary output language, e.g., numeric or symbolic code for any machine.

Fundamentally, a program compiled by COGENT is a list-processing program in which the list structures represent phrases of object language. The correspondence between the strings of object language which appear externally on input-output media and the list structures which represent these strings within the computer is determined by the syntax of the object languages. Thus the COGENT language itself contains two types of structures: productions, which specify the syntax of the object languages; and generator definitions, which specify list-processing procedures called generators. The format of the generator definitions is designed to let the programmer think directly in terms of the phrases of language that are being manipulated, rather than the list structures that actually represent these phrases.

The main routine of a COGENT program is always a syntax analyzer which is compiled from the productions describing the input object language. This analyzer reads character strings from the input medium and converts them into list structures. At certain points in this process, the analyzer will call a generator to operate on the currently recognized sublist; the result of this generator then replaces the current sublist in the list being constructed. In addition to being called by the syntax analyzer, generators may call each other and may call themselves recursively. Ultimately, primitive (built-in) output generators are called to decompose the list structures back into character strings and write these strings on the output medium.

B. Productions

Productions are the structures in the COGENT language that specify the syntax or grammar of the object language and thereby determine the correspondence between strings of object language and list structures. More specifically, productions serve three purposes:

1. To control the compilation of the syntax analyzer.

2. To control the compilation of tables that are used by various primitive generators to decompose list structures back into character strings. This process is called character scanning.

3. To specify the conversion of quoted phrases of object language, which appear as constants in generator definitions, into the list structures these phrases represent.

Productions specify syntax in the following manner: The phrases of an object language may be grouped into phrase classes, each of which is a set of phrases with the same syntactic behavior. Each phrase class is denoted by a phrase class name, which is a parenthesized alphanumeric name such as (LETTER) or (FACTOR). Then a production is a formula which defines one way (out of perhaps several alternatives) in which a phrase of a particular class may be constructed by juxtaposing characters and smaller phrases.

In format, the production consists of the name of the phrase class being defined (called the resultant of the production), an equal sign, a string of zero or more characters and/or phrase class names showing the construction of the phrase (called the construction string of the production), and a period. (Productions are thus written in a variant of the Backus notation[4] which avoids the use of special metalinguistic characters.)

As an example, consider a language consisting of algebraic polynomials, without numerical coefficients but with parentheses allowed. We will use strings of the letters A, B, C, D, and E to name the variables. To describe the syntax of the language, we first define the phrase class (LETTER), which includes the characters A, B, C, D, and E, by the productions

(LETTER) = A. (LETTER) = B. (LETTER) = C. (1-3)

(LETTER) = D. (LETTER) = E. (4-5)

Now the phrase class (VARIABLE) will consist of letter strings of arbitrary length. This may be specified by the productions

(STRING) = (LETTER). (6)

(STRING) = (STRING)(LETTER). (7)

(VARIABLE) = (STRING). (8)

Notice that the second of these productions is recursive; that is, it specifies that a string may consist of a string followed by a letter. This feature allows us to define strings of arbitrary length by using a finite number of productions; production (6) defines strings of one letter, while production (7) defines strings of two letters in terms of strings of one, strings of three in terms of strings of two, etc. Also notice the relation of strings and variables; all variables are strings but not all strings are variables. (In particular, strings that are subphrases of longer strings are not variables.)

Several productions used to define the same phrase class may be combined into a compound production, which has several construction strings

on the right, separated by commas. Such a compound production is simply an abbreviation for a set of simple productions which link the same resultant with each construction string. Thus we may write the productions we have given more compactly as

(LETTER) = A,B,C,D,E. (1-5)

(STRING) = (LETTER),(STRING)(LETTER). (6-7)

(VARIABLE) = (STRING). (8)

In discussing the meaning of COGENT programs, we will always assume that compound productions have been replaced by the equivalent simple productions.

We now come to the nontrivial phrase classes such as (FACTOR), (TERM), etc. A factor may simply be a variable, so that we have

(FACTOR) = (VARIABLE). (9)

On the other hand, a factor may be composed of a polynomial surrounded by parentheses. However, when we try to express this rule as a production, a problem arises. We would expect to write

(FACTOR) = ((POLYNOMIAL)).

But this is ambiguous, since the outer parentheses are meant to represent characters of the object language, while the inner parentheses are meant to indicate a phrase class name.

To overcome this ambiguity, we extend our specification of the format of productions, and specify that ambiguous characters will be enclosed in parentheses when they represent object characters. Thus we will write "(()" to represent "(" and "())" to represent ")", so that the production becomes

(FACTOR) = (()(POLYNOMIAL)()). (10)

In addition to "(" and ")", the object characters "," and "." must be parenthesized to avoid ambiguities in productions.

The remaining productions are straightforward. A term may be either a single factor, or a simpler term followed by an asterisk (multiplication sign) and a factor:

(TERM) = (FACTOR). (11)

(TERM) = (TERM)*(FACTOR). (12)

These two productions use recursion to define arbitrarily long terms in the same manner as it was used for strings. Polynomials are defined in a similar manner, except that the terms may be separated by either plus or minus signs, and a plus or minus sign may optionally precede the first term:

(POLYNOMIAL) = (TERM), +(TERM), -(TERM). (13-15)

(POLYNOMIAL) = (POLYNOMIAL)+(TERM). (16)

(POLYNOMIAL) = (POLYNOMIAL)-(TERM). (17)

This definition of polynomials is indirectly recursive, since a polynomial may be part of a factor, which may be part of a term, which may be part of a longer polynomial. This use of recursion allows parentheses to be nested to an arbitrary depth. (We will continue to use these illustrative productions describing polynomials in examples throughout this manual.)

C. <u>Construction Trees</u>

The set of productions that describes the syntax of an object language is essentially a set of rules for constructing all legal phrases of the language. For example, to construct a polynomial according to the productions we have given, we first use any one of the productions whose resultant is the phrase class (POLYNOMIAL). This production gives a sequence of phrase classes and characters which forms a legitimate polynomial. We then replace each phrase class name in this sequence by a subsequence which is determined by any one of the productions for that class. This process is repeated until only characters remain; the final sequence is then a phrase of the class (POLYNOMIAL).

A construction of this sort is conveniently represented by a tree structure in which terminal nodes represent characters and nonterminal nodes represent phrases. The nonterminal node for a particular phrase is connected to the subnodes for the characters and subphrases which replace the phrase; thus the relationship between a nonterminal node and its subnodes always corresponds to a particular production. Such a construction tree is shown in Figure 1 for the polynomial "-(A+B)*DC", according to the productions given above.

Given a set of productions describing an object language, a string of characters is said to be a phrase of a particular class if a construction tree exists which is headed by the name of the phrase class and terminates in the characters of the string. If more than one such construction tree exists, the phrase is said to be <u>ambiguous.</u>

Several further syntactical concepts may be defined in terms of the construction tree. Given a phrase and its construction tree, the production

that relates the main node of the tree to its subnodes is said to partition the phrase. Given any nonterminal node, other than the main node, the sequence of characters that terminate the subtree headed by this node is called a subphrase of the given phrase. If the subtree is headed by an immediate subnode of the main node, the corresponding subphrase is an immediate subphrase. Thus the polynomial "-(A+B)*DC" is partitioned by the production (POLYNOMIAL) = -(TERM), and its only immediate subphrase is the term "(A+B)*DC". (A more rigorous definition of these concepts is given in Chapter II, pp. 45-48.)

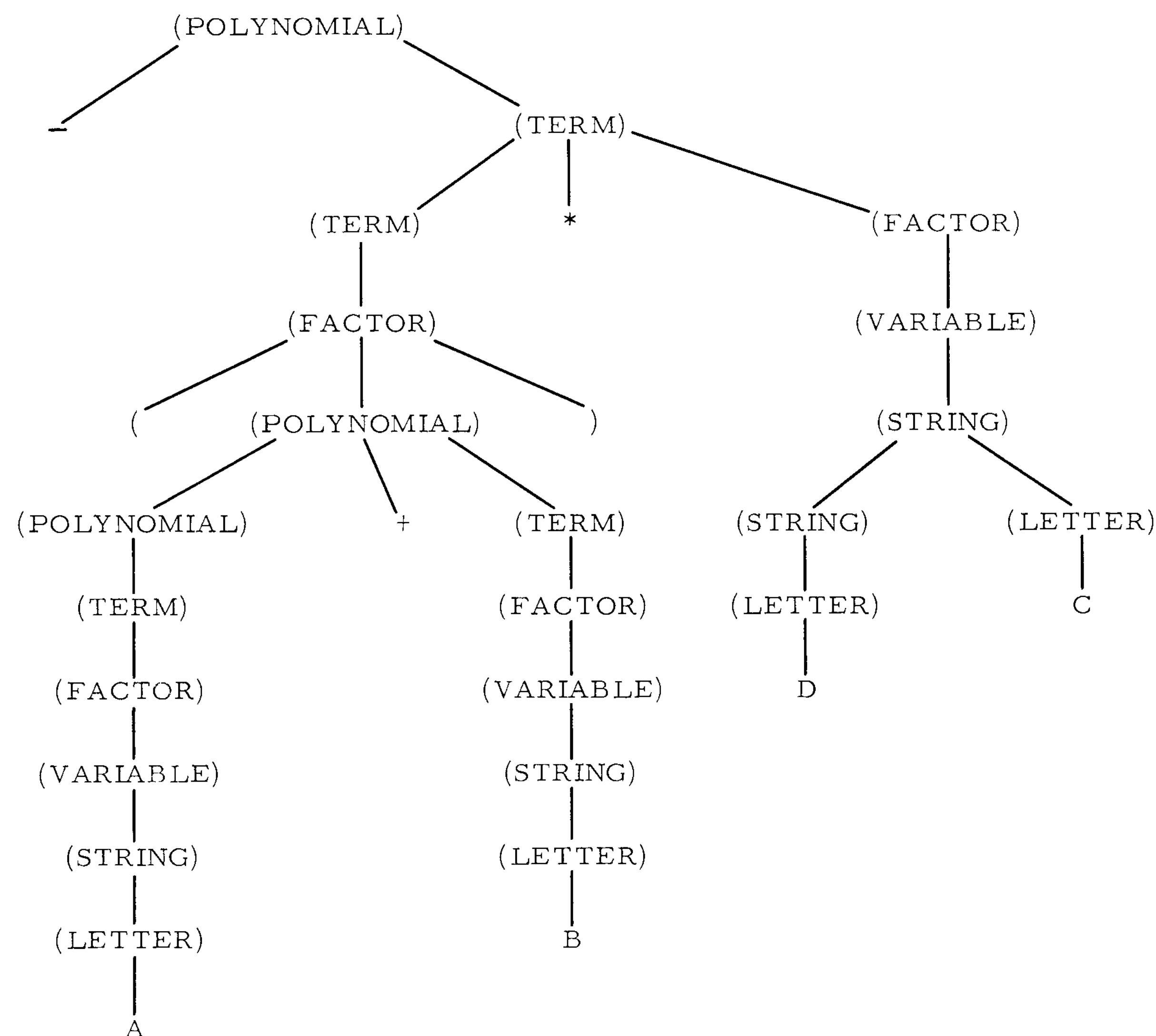


Fig. 1. Construction Tree for the Polynomial "-(A+B)*DC"

D. List Structures

Essentially, construction trees are the mechanism by which phrases of object language are mapped into list structures; i.e., the list structure representing a phrase is a machine representation of its construction tree. Before specifying this mapping in detail, we must first define the general concepts of list structures as used in COGENT.

List structures are composed of elements, which are ordered sets of components. Each element is denoted by a list name. (Within the computer, an element is normally a block of storage divided into subfields for the components, and the name of the element is the address of this block.) In general, a component may be either an item of data, such as a number or a BCD string, or else the name of an element. We will define a variety of types of elements; the type of an element specifies the nature of its components.

The list structure denoted by a name n is defined to consist of:

1. The element denoted by n. This element is called the head element of the structure.

2. The list structures denoted by each name that is a component of the head element. These are the sublists of the structure.

We will frequently speak of a constant or variable having a list structure or element as its value. More precisely, the value is actually the name of the structure or element.

It is important to distinguish between similar and identical list structures. Two list structures are similar if either:

1. They have the same name, or

2. The head elements of the structures have the same type and number of components, and corresponding components are either identical, if they are data items, or denote similar sublists, if they are names.

The two structures are identical if and only if case 1 holds.

To represent list structures pictorially, we will use a collection of boxes representing elements, which will be subdivided into boxes for the components. When a component is a name, an arrow will be drawn from the component box to the box for the named element; the component is then said to point to the element. The type of the element will sometimes be indicated by a flag in the upper left-hand corner of the element box.

We may now describe the relation between list structures and phrases of object language. The list structure that represents an object phrase is completely equivalent to the construction tree for the phrase, but is considerably more compact. A construction tree is highly redundant, since it contains information that is also given by the productions themselves. Actually it is only necessary to indicate for each nonterminal node of the tree the particular production that relates this node to its subnodes. The production

itself then specifies both the phrase class name of the node and the phrase class names and object characters of the subnodes.

To enable list structures to refer to productions, each (noncompound) production, when read by the COGENT system, is assigned a unique production code number. (The assignment of these code numbers is an internal process in the system and is normally of no consequence to the programmer.) When code numbers are given for the productions, the construction tree of a phrase may be converted into a list structure by replacing each nonterminal node of the tree by a list element whose first component is the code number of the production that relates the node to its subnodes, and whose remaining components are the names of the list elements that replace the nonterminal subnodes. Terminal nodes are simply discarded, since the corresponding characters are determined by the productions themselves.

The list structure shown in Figure 2 is derived from the construction tree in Figure 1 and represents the polynomial "-(A+B)*DC". The code

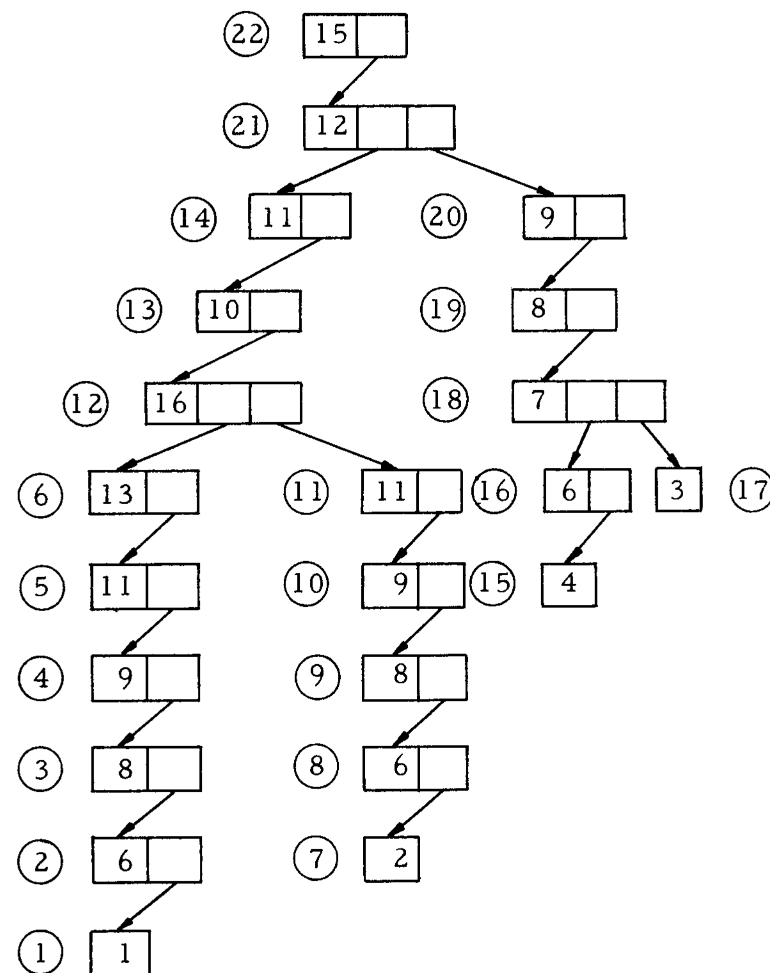


Fig. 2. List Structure Representing the Polynomial "-(A+B)*DC"

numbers used in Figure 2 correspond to the numbers given in the text to the right of each production.

The relation between a list structure and the phrase it represents may be described directly, without the intermediary construction tree, as follows: The first component of the head element of the list structure is the code number of the production that partitions the phrase, i.e., whose resultant gives the phrase class name of the phrase and whose construction string gives the decomposition of the phrase into characters and immediate subphrases. The remaining components of the head element correspond, in order, to the phrase class names in the construction string of this production, and name sublists which represent the corresponding immediate subphrases of the original phrase.

The list elements we have introduced always consist of a production code number followed by zero or more names. This type of element is called normal to distinguish it from other element types which will be introduced later.

E. The Syntax Analyzer

The main routine of a COGENT program is always a syntax analyzer which is compiled from the productions describing the input object language. This analyzer reads a character string from the input medium and produces the corresponding list structure. The list structure is always produced in a standard order, in which the sublists of any element are constructed in order from left to right, and the element itself is constructed immediately after the last sublist. In effect, the analyzer moves up each branch of the list structure, up to but not including the first element with further branches to the right. When an element with further branches is encountered, the analyzer jumps to the bottom of the next branch on the right. (In terms of the input character string, the analyzer recognizes phrases on the same level in order from left to right, and recognizes a phrase immediately after its last subphrase.) The circled numbers in Figure 2 indicate the order in which the list structure would be produced by the analyzer.

At certain points in its operation, the analyzer interrupts the construction of the list structure and calls in a generator to process the sublists of the current list element. When the generator returns control to the analyzer, the result of the generator replaces the current list element, and the construction of list structure continues.

This calling of generators is programmed by labeling certain productions with generator names. A label consists of a generator name, which is an alphanumeric string, followed by a slash. Thus for example, two of the productions describing polynomials might be labeled:

MULTCOMP/ (TERM) = (TERM)*(FACTOR). (12')

ADDCOMP/ (POLYNOMIAL) = (POLYNOMIAL) + (TERM). (16')

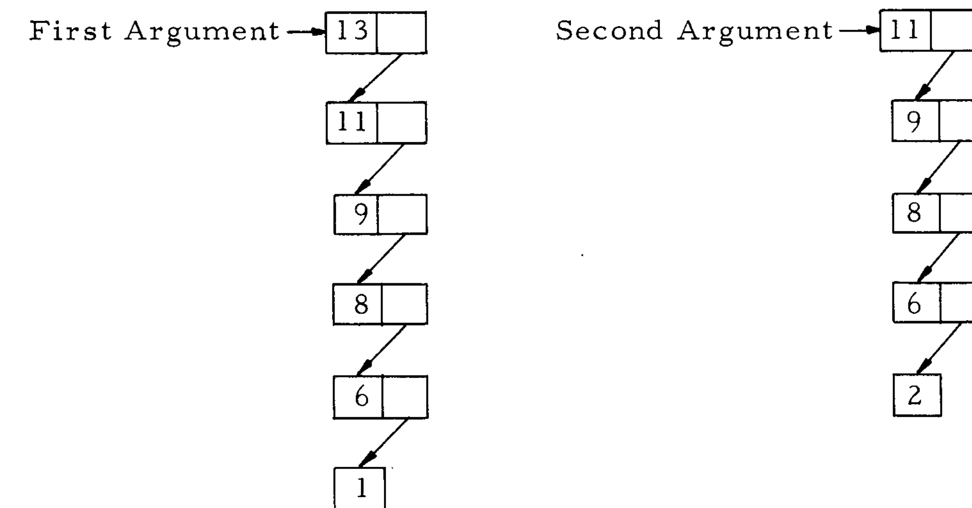
If compound productions are labeled, the label applies to all of the corresponding simple productions.

When the analyzer is about to construct a list element which represents a production with a label, it calls the generator indicated by the label and gives this generator as input arguments the sublists of the list element that would otherwise be constructed. When the generator returns control, the result of the generator replaces the list element that would have been constructed, and the analysis continues. Thus at a higher stage of the analysis the generator result may be passed on to other generators for further processing.

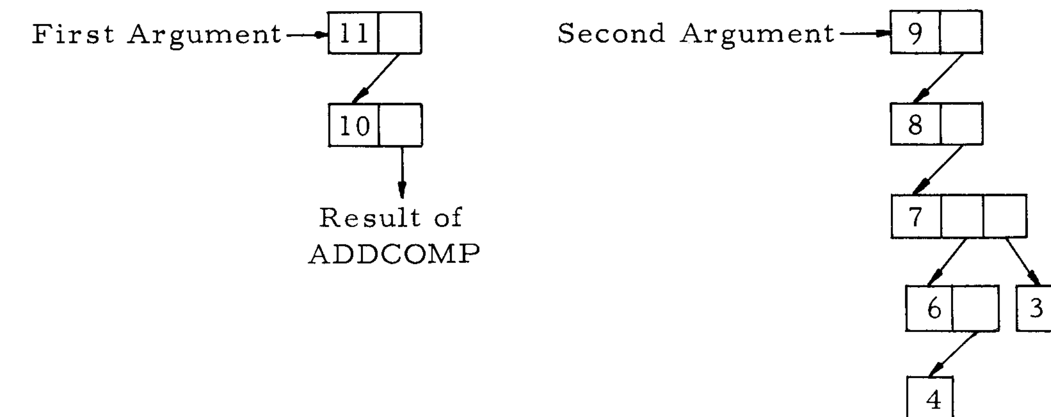
For example, suppose that the input string "-(A+B)*DC" is read by a syntax analyzer which was compiled from the productions we have given, including the labeled versions (12') and (16'). When the analyzer is about to recognize the list element beginning with code number 16 (see Figure 2), it will call the generator ADDCOMP and give it the input arguments shown in Figure 3a, which represent the (POLYNOMIAL) "A" and the (TERM) "B". When ADDCOMP returns, its result replaces the list element beginning with 16, and the analysis continues. Later, when the analyzer reaches the element beginning with 12, the generator MULTCOMP is called and given the arguments shown in Figure 3b, which represent the (TERM) "(...)" and the (FACTOR) "DC", where the dots indicate the substituted result of ADDCOMP. When MULTCOMP returns, its result replaces the element beginning with 12, and the analysis again continues. The final result is shown in Figure 3c.

This general approach, of performing translation by means of generators which are associated with productions and which are called in at each syntactic level where the production is used, was developed by E. T. Irons,[1] and called by him "Syntax-directed Compilation." It is especially useful for programming simple translation processes in which the translation of each phrase may be described in terms of the translation of its subphrases; in such situations, the generators may be simple substitution mechanisms. However, in more complex processes, in which the translation of a phrase may depend upon contextual material outside the phrase itself, the syntax-directed approach is less helpful. In these situations, the analyzer must convert large segments of input into list structure before the translation can begin, and the generator that accepts this structure must use facilities such as recursion and conditional list analysis to perform the translation.

(a) Input Arguments of ADDCOMP:



(b) Input Arguments of MULTCOMP:



(c) Final Result of the Analysis:

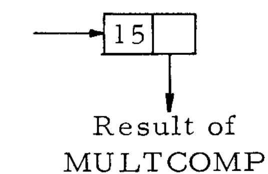


Fig. 3. List Structures Communicated between the Syntax Analyzer and Generators

F. Generator Definitions: Constants

We have seen that a generator is a subroutine for manipulating list structures, which is specified in the COGENT language by a generator definition. The generator definition contains a sequence of statements which compute and assign values to variables, perform tests, and transfer control, much as in a FORTRAN subroutine or an ALGOL procedure. More precisely, since a generator may have any number of input arguments but only a single result, it may be characterized as a function subroutine.

The essential peculiarity of generators is the nature of the values that their variables take on. These values are usually not numbers, but rather the names of list structures representing phrases of object language. Thus the most distinctive feature of generator definitions is the format of constants, which must denote such list structures.

Since COGENT is designed to let the programmer think directly in terms of the phrases of object language which are to be manipulated, the ideal approach would be to allow a constant to consist merely of the desired object phrase enclosed in some type of quotation marks, and to have the COGENT compiler convert this quoted phrase into the corresponding list structure according to the appropriate productions. But to perform this conversion, the compiler must know not only the characters of the object phrase, but also the class name of the phrase. Thus a constant must give both a phrase class name and a string of object characters. The exact format is: a left parenthesis, the phrase class name, a slash, the object string being quoted, and a closing right parenthesis. Just as in productions, the object characters "(", ")", ",", and "." must be enclosed in parentheses when appearing in the quoted string. Thus, for example, to denote the object phrase "-(A+B)*DC" one would write the constant

(POLYNOMIAL/-(()A+B())*DC)

The actual list structure denoted by this constant is the structure shown in Figure 2.

Now the basic operations used in linguistic manipulation are the analysis of phrases into subphrases, and the synthesis of phrases from subphrases. To facilitate these operations, it is extremely useful to extend the set of constants used in the COGENT language (and therefore the set of values that variables may assume) to include parametric phrases, i.e., phrases in which one or more subphrases, called parameters, are left unspecified. We will see that these parametric phrases play a central role in COGENT programs as templates which specify synthesis and analysis operations.

Parametric phrases are denoted by constants in which one or more parameters appear in the quoted object string. These parameters are indicated by the appropriate phrase class name, followed by a slash and a number called the parameter index, and enclosed in parentheses. For example, the constant

(TERM/(FACTOR/1)*(FACTOR/4)*(FACTOR/3))

denotes a term containing three parametric factors, with indices 1, 4, and 3. It is possible to omit the index and the preceding slash within a parameter; in this case, an implied index is assumed which is the order of appearance of the parameter in the constant (from left to right). Thus,

(TERM/(FACTOR)*(FACTOR/4)*(FACTOR))

is equivalent to the constant given above.

Parametric constants are converted into list structures in a manner similar to that used for conventional constants. A parameter containing the name of a particular phrase class behaves syntactically as a phrase of that class, but it is converted into a special type of list element called a parameter element. The parameter element contains a single component giving the index of the parameter. (We will indicate a parameter element in diagrams of list structures by the flag "P".) The list structure corresponding to either of the constants given above is shown in Figure 4.

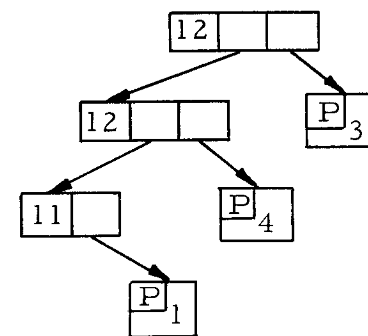


Fig. 4

List Structure Denoted by the Parametric Constant (TERM/(FACTOR/1)*(FACTOR/4)*(FACTOR/3))

G. Expressions and Assignment Statements

In generator definitions, variables are represented by alphanumeric names, as in most programming languages. Beginning with constants and variables, expressions may be built up by using a functional notation, in which the function names indicate the calling of other generators. Thus, an expression may be a constant, a variable, or a compound expression of the form

generator-name (expression , ... , expression)

Such a compound expression is evaluated by first evaluating the subexpressions, then calling the named generator with the subexpression values as input arguments, and finally taking the result of the generator as the value of the expression. For example,

ADDCOMP(MULTCOMP(X, (FACTOR/DC)), Y)

would be evaluated by calling the generator MULTCOMP with the values of X and the constant (FACTOR/DC) as input arguments, then calling the generator ADDCOMP with the result of MULTCOMP and the value of Y as input arguments, and finally taking the result of ADDCOMP as the value of the entire expression.

The main part of a generator definition is a sequence of statements, marked off by periods. These statements are divided into two categories: assignment statements, which alter the values of variables, and control statements, which perform tests and determine the order in which statements are executed.

The simplest type of assignment statement is the direct assignment statement, which consists of a variable name, equal sign, expression, and period. For example,

X = (FACTOR/DC).

X = ADDCOMP(Y, (TERM/AB*DC)).

A direct assignment statement causes the right-hand expression to be evaluated and its value to be assigned to the variable named on the left. A variant of the direct assignment statement consists simply of a compound expression followed by a period. For example,

OUTINST(X).

This variant causes the compound expression to be evaluated, but the resulting value is ignored. It is normally used to call a generator which has a meaningless result but performs some useful action as a side effect.

The basic operations of synthesizing and analyzing list structures are performed by the synthetic and analytic assignment statements. While these statements are actually general-purpose list-processing operations, they are designed to display the linguistic manipulations that these operations represent. As noted earlier, this is accomplished by using parametric constants as templates to control the construction and decomposition of the list structures.

The synthetic assignment statement is used to build up a list structure from one or more sublists. It consists of a variable name, the symbol "/=", an expression called the template expression (which is usually a parametric constant), a sequence of additional expressions each preceded by a comma, and a period; i.e.,

name /= $\text{expression}_{\text{template}}$, expression_1 , ... , expression_n.

This statement first evaluates all the expressions. It then creates a copy of the value of the template expression (i.e., a new list structure similar to the structure which is the template value) and assigns the copied list to the variable named on the left. However, as the copy is produced, each parameter element in the list structure is replaced by the value of expression_i, where i is the index of the parameter element.

For example, suppose that X has the value (FACTOR/ABE), and Y has the value (FACTOR/BED). Then the synthetic assignment statement

Z /= (TERM/(FACTOR)*(FACTOR)), X, Y.

will assign to Z a copy of (TERM/(FACTOR)*(FACTOR)) in which the first parameter is replaced by the value of X and the second parameter is replaced by the value of Y. Thus Z will be given the value (TERM/ABE*BED). (The explicit list structures for this example are shown in Figure 5.) Similarly, the statement

Z /= (TERM/(FACTOR/2)*(FACTOR/1)), X, Y.

would give Z the value (TERM/BED*ABE), while

Z /= (TERM/(FACTOR/1)*(FACTOR/1)), X.

would give Z the value (TERM/ABE*ABE).

The analytic assignment statement is used to decompose a list structure into sublists, and to compare two list structures. It consists of an expression called the test expression, the symbol "=/" , a template expression, a sequence of variable names each preceded by a comma, and a period; i.e.,

$\text{expression}_{\text{test}}$ =/ $\text{expression}_{\text{template}}$, name_1 , ... , name_n.

This statement first evaluates both expressions. It then compares, element-by-element, the list structures that are the values of these expressions. During the comparison, whenever a parameter element with index i is encountered in the template structure, the corresponding sublist in the test structure is made the value of the variable indicated by name_i.

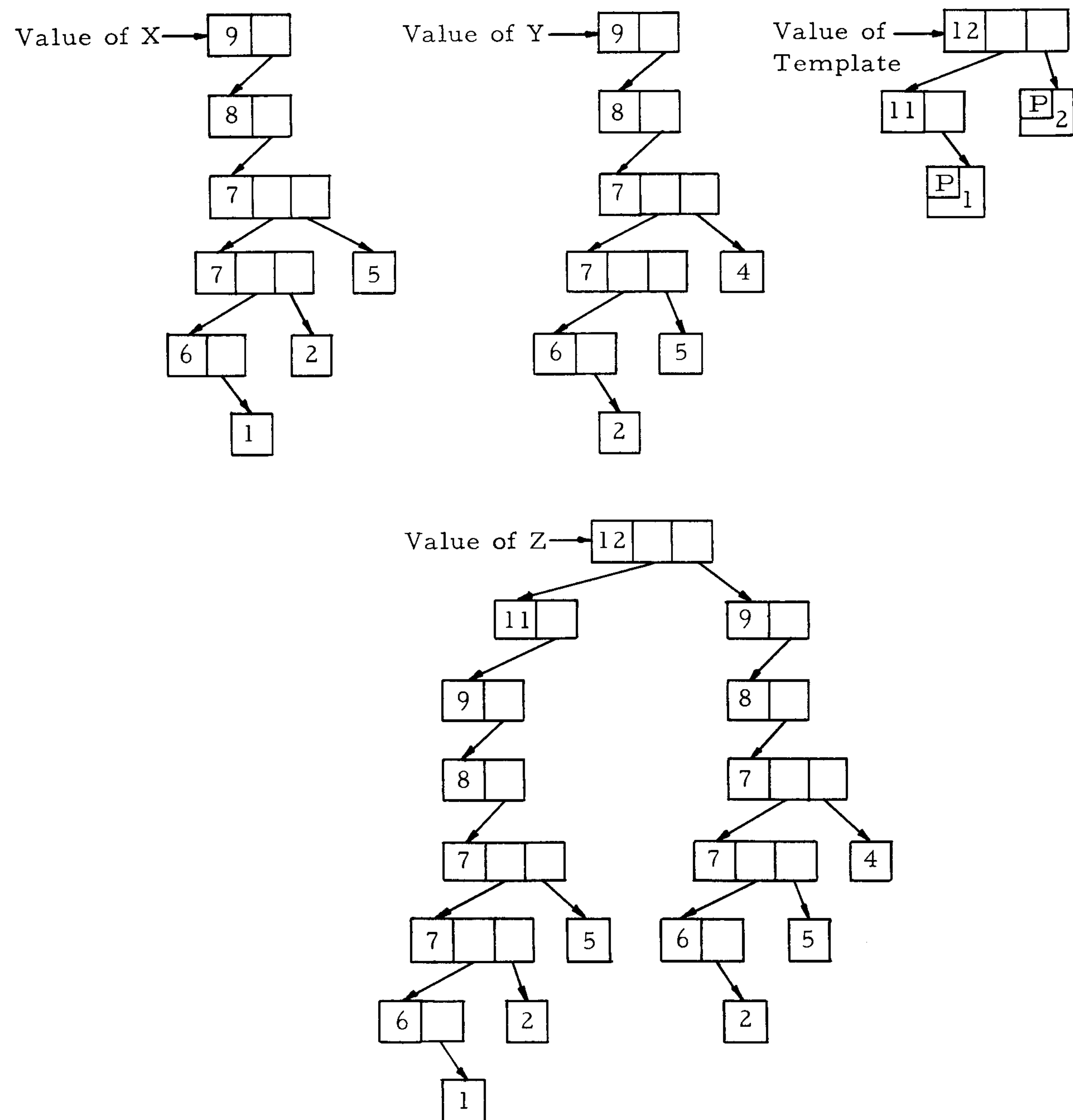


Fig. 5. List Structures Used by Synthetic and Analytic Assignment Statements

For example, suppose Z has the value (TERM/ABE*BED). Then the analytic assignment statement

Z =/ (TERM/(FACTOR)*(FACTOR)), X, Y.

will give X the value (FACTOR/ABE) and Y the value (FACTOR/BED). (The explicit list structures for this example are again given by Figure 5.)

An additional property of the analytic assignment statement arises from the possibility that the comparison of the test and template list structures may show that these structures are dissimilar (beyond the occurrence of parameter elements in the template in place of sublists in the test structure). If the list structures do not match, then the analytic statement fails, without changing the value of any variables. (Failure is a conditional control mechanism explained below.)

Thus, if Z has the value (TERM/ABE*BED*CAB), which is not a term composed of two factors, then the statement given above will fail, leaving the values of X and Y unchanged.

The synthetic and analytic assignment statements are generalizations of the "Parameter Operations" used by Brooker and Morris in their Compiler Compiler.(2)

H. Control Statements and Failure

Control statements are used to perform tests and to alter the normal statement-by-statement sequence of control. The control statements refer to other statements in the same generator definition by means of statement numbers, which are unsigned nonzero integers. Any statement may be labeled with a statement number by prefixing the number, followed by a slash, to the statement. For example,

10/ Z /= (TERM/(FACTOR)*(FACTOR)), X, Y.

The unconditional jump statement consists of a plus sign, a statement number, and a period. It causes control to jump to the statement which has been labeled with the statement number. Thus,

+10.

sends control to the statement labeled 10.

The conditional jump statement consists of a plus sign, a statement number, the word IF or UNLESS, and an unlabeled assignment statement; i.e.,

$$+ \text{ statement-number } \left\{ \begin{matrix} \text{IF} \\ \text{UNLESS} \end{matrix} \right\} \text{ assignment-statement}$$

This statement first performs the assignment statement on the right, and then conditionally executes a jump to the indicated statement number, depending on whether the assignment statement has failed. When IF is used, the jump is executed if failure has not occurred; when UNLESS is used, the

jump is executed if failure has occurred. If the jump is not executed, control passes to the next statement.

For example, consider the statement

+ 10 IF Z =/ (TERM/(FACTOR)*(FACTOR)), X, Y.

If the value of Z is a term composed of two factors, then this statement will assign these factors to the variables X and Y and transfer control to statement 10. On the other hand, if the value of Z is not a term containing two factors, control will pass to the next statement without altering X or Y.

Failure is the basic control mechanism used in COGENT. In general, a failure may occur in three ways:

1. Certain primitive generators used to perform tests may fail.

2. An analytic assignment statement will fail if the list structures being compared do not match.

3. A control statement written "$FAILURE." is provided which simply fails without taking any other action.

When a failure occurs, it is propagated upwards through the chain of generator calling sequences until a conditional jump statement or the syntax analyzer is reached. Thus, when an assignment statement calls a generator that fails, the statement fails without calling further generators or assigning values to its variables. When a statement in a generator fails, unless it is a substatement of a conditional jump statement, the generator fails. When a statement within a conditional jump statement fails, the jump statement does not fail, but branches appropriately. Finally, if a failure propagates all the way to the syntax analyzer without encountering a conditional jump statement, an error stop occurs.

The ability of a failure to propagate up a long chain of calling sequences is a useful mechanism for describing complex processes which may either run to completion and return a result, or else fail at an arbitrary point in their operation. Such a process, perhaps involving numerous recursions and many generators, may be coded to terminate upon a failure at any point in its operation.

A final type of control statement is the return statement, with the format:

$RETURN (expression) .

This statement evaluates the expression and then causes an exit from the generator containing the statement, with the value of the expression as the result of the generator.

I. Declarations

In format, a complete generator definition consists of the following:

1. The characters "$GENERATOR".

2. The name of the generator being defined, followed by two left parentheses.

3. A sequence of input variable names, separated by commas.

4. A right parenthesis.

5. A sequence of declarations.

6. A sequence of statements.

7. A right parenthesis and a period.

(Actually it is possible to insert, between items 5 and 6, one or more generator definitions which define subgenerators, but a discussion of subgenerators will be postponed until the next chapter, pp. 56-59.)

The purpose of the input variable sequence and the declarations is to specify the meaning of the alphanumeric names which appear within the statements of the generator definition. The input variable sequence and the declarations are said to control the appearances of names in the statements of the same generator definition. Two appearances of the same name (e.g., in different generator definitions) may have different meanings if the appearances are controlled by different declarations.

A local declaration normally has the format:

$LOCAL name , ... , name .

It specifies the names that it contains to be local variables of the generator containing the declaration. When this generator is called, new storage is allocated for its local variables, and when the generator exits, this storage is released. Thus the values of local variables are lost when the generator exits.

The input variable sequence specifies the names that it contains to be input variables of the generator containing the sequence. Input variables

behave in the same manner as local variables, except that when storage is allocated for the variables upon entrance to the generator, the variables are initialized to the input arguments of the generator. Thus, the input variable sequence determines the number and ordering of input arguments for a generator, so that all calls of the generator, both from the syntax analyzer and from other generators, must match the input variable sequence of the generator in number and ordering.

An own declaration, with the format:

$OWN name , ... , name..

specifies the names that it contains to be own variables of the generator containing the declaration. Storage for own variables is permanently allocated, although the variables can only be referred to by the generator to which they belong (or its subgenerators). Thus, when a generator exits, the values of its own variables are not lost but are available to later calls of the same generator.

The distinction between local variables (or input variables) and own variables appears most clearly in the case of recursive generators, which are generators that call themselves (either directly, or indirectly through one or more other generators). When a recursive generator calls itself, the current values of its local variables are "hidden" in a pushdown stack, and fresh storage is allocated for these variables. When the called generator returns to itself on a higher level, this storage is released, and the local variables reassume their hidden values. On the other hand, own variables are not hidden, so that values which have been set by the calling generator may be accessed and changed by the (same) called generator. In effect, local variables are independent on each level of the recursion, while own variables are the same on all levels.

A third type of declaration is the pseudoconstant declaration, which defines names to be abbreviations for long or frequently used constants. It has the form

$PCON name = constant , ... , name = constant .

and defines each name to represent the following constant. Any appearance of a name controlled by a pseudoconstant declaration is completely equivalent to the corresponding constant. A name controlled by a pseudoconstant declaration must never appear in a statement that assigns a value to this name, since it is meaningless to assign a new value to a constant.

J. Generator Elements

In certain types of COGENT programs, it is convenient to use variables that take on generators themselves as their values. This capability

is provided by defining a special type of list element called a generator element, which has a single component giving the entry address of a generator. A generator element may appear as the value of any variable or may even be imbedded within a larger list structure.

The introduction of generator elements allows us to give a more general interpretation to the concepts of generator names and compound expressions. A generator name is actually a special kind of pseudoconstant which denotes a generator element giving the appropriate entry address. On the other hand, the first item in a compound expression, which is normally a generator name, may actually be any expression whose value is a generator element.

For example, suppose that X has been declared as some type of variable, while PLUSCOMP is a generator name. Then the execution of the statement

X = PLUSCOMP.

will set X to a generator element giving the entry address of PLUSCOMP. Then at a later step in the computation, the statement

Z = X(Y).

will call PLUSCOMP with the value of Y as an input argument.

Similarly, if SILLY and PLUSCOMP are generator names, then the statement

Z = SILLY(Z, PLUSCOMP).

will call SILLY and provide the generator element for PLUSCOMP as the last argument of SILLY. The generator definition for SILLY might have the form:

$GENERATOR SILLY ((X, G) ...

... Y = G(Y). ...).

in which the expression G(Y) would cause SILLY to call the generator indicated by its last argument (in this case PLUSCOMP).

A more esoteric example is a compound expression such as

RIDICULOUS (X) (Y)

which is allowable if the result of the generator RIDICULOUS is a generator element. In the evaluation of this expression, RIDICULOUS is called with the value of X as its argument, and then the generator whose element is the result of RIDICULOUS is called with the value of Y as its argument.

The generalizations introduced here lead to the problem of determining when a name is a generator name, as opposed to the name of a variable whose value is a generator element. The general rules will be given in the next chapter (pp. 62-64); for the present, we state that (when subgenerators or main declarations are not involved) a name is a generator name if it is not controlled by any declaration or input variable sequence.

K. Primitive Generators: Arithmetic

A primitive or built-in generator is a generator that may be used in a COGENT program without being defined. Many primitive generators are provided in COGENT for a variety of purposes, including arithmetic, identifier-handling, and output operations.

Numbers in COGENT are represented by a special type of list element called a number element, which contains two components: a mode (integer or floating-point) and a value. The value components of floating-point numbers have fixed lengths, but the value components of integers have variable lengths depending upon the magnitudes of the integers. Thus there is no overflow in integer arithmetic; if the result of an operation has a length that exceeds the memory space used by its operands, additional space is automatically provided. This facility for arbitrarily long integers is particularly valuable in algebraic manipulation programs, in which exact fractional coefficients may be used throughout the computation.

Three primitive generators, named DECCON, OCTCON, and FLOATCON, are provided to convert list structures into number elements. Each of these generators accepts a single argument which is a list structure representing a string of object characters. This list structure is converted into a string of object characters by character scanning, and then the string is converted into a positive number element, as a decimal integer, octal integer, or floating-point (decimal) number.

Several primitives are also provided for performing arithmetic operations upon number elements. For example, ADD(X, Y) will produce a number element representing the sum of the elements X and Y. These generators accept either integer or floating-point arguments. If all the arguments are integers, the result will be an integer, but if any argument is floating-point, then any integer arguments will be converted into floating-point, the designated operation will be performed, and a floating-point result will be returned.

A special type of constant may be used in generator definitions to denote positive integer (but not floating-point) number elements. Such a constant is written simply as a string of digits followed by an optional "B". The letter B indicates that the digit string is to be interpreted as an octal number.

L. Identifiers

The examples we have given suggest that list structures frequently require an inordinate amount of storage to represent phrases of object language. In particular, an object language will usually contain certain phrases whose division into subphrases is of no interest, and which therefore should be carried along through a computation as packed character strings rather than as list structures. Such phrases (for example, variables in the polynomial language we have defined) are called identifiers and may be represented in COGENT programs by a special type of list element called an identifier element, which contains a component giving the appropriate packed character string.

Identifier elements are grouped into one or more identifier tables, which are distinguished by positive integers called table numbers. Each table has the property that no two elements within the table can contain the same character string. Thus if two list names refer to identifier elements in the same table and with the same character string, the names must refer to the same element; i.e., the names themselves must be equal.

Specifically, an identifier element contains four components:

1. The table number.

2. The association list name. This component may be set by the programmer to point to an arbitrary list structure, and is usually used to associate declarative or descriptive information with an identifier.

3. The table link name. This component is used to link identifier elements together for table searching. It is not directly accessible to the programmer.

4. The character string, stored in a packed BCD format.

Identifier elements are created by the primitive generator IDENT(X, N), which accepts a list structure X and a positive integer number element N denoting a table number. The list structure X is scanned to obtain a string of object characters, and then identifier table N is searched for an element with the same character string. If such an element is found, it is returned as the result of IDENT; otherwise a new element with the appropriate string is added to table N and returned as the result.

Two primitive generators are provided for setting and obtaining association lists. SETA(I, X) accepts an identifier element I and an arbitrary list structure X and replaces the association list name of I by X. It does not return a meaningful result. ALIST(I) accepts an identifier element I and returns the association list name from this element.

M. Output

Output operations in COGENT usually consist of two phases: character scanning, which reduces a list structure to a string of object characters, and character output, which assembles these characters into records and sends these records to the appropriate output device.

Character scanning may be performed by using the primitive generator STANDSCN(X, CR), which accepts a list structure X to be scanned, and a second argument CR, which must be a generator element denoting a generator called the character receiver. The character receiver is a one-argument generator which will be called repeatedly by STANDSCN and given on each call an argument representing a single object character, i.e., a positive integer number element giving the BCD code for the character.

STANDSCN will reduce a list structure containing normal elements according to the appropriate productions. If an identifier element is encountered in the scan, it will be replaced by its character string. If a positive integer number element is encountered, it will be replaced by a sequence of digits giving its decimal representation. If any other type of nonnormal element is encountered, STANDSCN will fail.

The programmer may alter the response of STANDSCN to number elements by methods described in Chapter III (pp. 84-93). He may also define his own character-scanning generator in terms of more basic primitive scanning generators.

The character-output phase is performed by the character receiver. For printed output, the primitive PUTP(C) may be used as a character receiver. On each call, PUTP accepts an integer C giving the BCD code for a single character, and repeated calls of PUTP place these characters in successive positions of a print line. When the character position reaches a margin limit, the current print line is written on the printed output tape, a new line is initialized, and the next character is placed at the left of the new line. A second primitive output generator OUTP() outputs the current line, even if the margin limit has not been reached, and then initializes a new line.

The primitive generators PUTC(C) and OUTC() are analogous to PUTP and OUTP, but produce BCD card images on the punched output tape.

Thus, for example, to print the object character string represented by the list structure X and then skip to a new line, one would use the statements:

STANDSCN(X, PUTP).   OUTP().

Similarly, to punch the character string on cards and then skip to a new card, one would use:

STANDSCN(X, PUTC).   OUTC().

Normally, PUTP and PUTC output character strings in a free-field format running from one print line or card to the next. However, the programmer may introduce more sophisticated output formats by altering the response of PUTP or PUTC to special character codes or to margin limits (as described in Chapter III, pp. 96-102). He may also define his own character-receiving generator in terms of more basic output primitives.

A separate set of output primitives is available for outputting numbers onto binary cards.

CHAPTER II

SPECIFICATION OF THE COGENT LANGUAGE

This chapter gives a complete specification of the COGENT language, including a number of features not described in Chapter I. The material is organized for convenient reference and assumes a familiarity with the basic concepts discussed in Chapter I.

A. List Structures

1. Types of List Elements

List structures in COGENT are composed of elements of the following types:

a. Normal Elements. A normal element indicates the partitioning of a phrase of object language into its immediate subphrases and/or characters, according to a particular production. It contains the following components:

1) A production code number p, such that $1 \le p \le 1023$.
2) Zero or more names of sublists. These names must correspond in number and order to the phrase class names appearing in the construction string of the production denoted by the production code number.

b. Identifier Elements. An identifier element represents a packed string of object characters. Each element is a member of the identifier table designated by its table number, and no other element in the same table may represent the same character string. An identifier element always has the following four components:

1) The table number n, such that $0 \le n \le 511$.
2) The association list name, which may be set by the programmer to point to an arbitrary list structure.
3) The table link name. This component is used to link identifier elements together for table searching. It is not directly accessible to the programmer.
4) The character string, represented by a packed sequence of output codes for the corresponding characters. The string may contain zero or more characters, subject to the limitation that the total number of characters, plus the number of characters with output codes larger or equal to 75_8, must not exceed 1016_{10}.

Identifier elements with a table number of zero are treated in a special manner. They are not placed in any table, and no attempt is made to coalesce two such elements with the same character string.

c. Number Elements. A number element represents an integer or floating-point number, and always contains two components: a mode component indicating integer or floating-point, and a value. The representable range of integers is limited only by the storage available for list structures. The range and precision of floating-point numbers is determined by the 3600 double-precision representation.

d. Parameter Elements. A parameter element represents a parameter in a parametric object phrase. Each parameter element contains a single component, the parameter index i, such that $1 \le i \le 50$.

e. Generator Elements. A generator element represents a generator and contains a single component, the entry address of this generator. Distinct generator elements always refer to distinct generators.

f. The Dummy Element. This is a special no-component element with a unique name. The dummy element is used as an initial value for variables and for the association list names of identifiers (when not otherwise specified by the programmer), and as the result of generators that do not produce a meaningful result.

In general, any component of a list element that is a name (except an identifier table link) may point to any type of list element. Note that the various limits given on the size of components apply specifically to the 3600 COGENT system.

2. Representation and Storage Allocation

Within the computer, during the running of a compiled COGENT program, a pushdown stack of consecutive words provides dynamically assigned storage for the variables of the generators, and thus contains the names of the list structures being used in the computation. The list elements themselves are represented by small arrays in a separate storage area called list storage; the name of an element is simply the address of the corresponding array. (Since elements may vary in their number of components, the list structures of COGENT are essentially plexes, in the sense defined by D. T. Ross.)(5) List storage is divided into active and free areas. When the free area is exhausted, a storage recovery routine is automatically called which marks all active elements and then retrieves the remaining elements to form a new free-storage area.

Thus the internal mechanism for handling list structures in COGENT follows the basic approach used in the LISP programming system(3) but differs in two important respects. First, the storage space

required to represent an element varies with the type and number of components of the element. This use of variable-sized elements requires that the free-list storage area must be a coalesced block, rather than a linked list of free elements, so that new elements of arbitrary size may always be created. To produce such a coalesced block, the storage-recovery algorithm must relocate the active elements after marking them. This algorithm is based on a scheme proposed by D. Edwards.[6]

Secondly, certain short but frequently used types of elements are represented by <u>literal names</u>, i.e., names that are not addresses of arrays in list storage, but are rather addressed-sized quantities giving a direct encoding of the components of the named elements. Literal names are used for normal elements with no sublists, integer number elements with magnitudes smaller than 1024, parameter elements, and the dummy element. This use of literal names provided significant savings in both storage space and execution speed.

The nature of the storage-recovery algorithm places one restriction on the form of list structures. In certain programs, either the syntax analyzer or the generators may create long chains of normal list elements, in which one component of each element points to the next element. When such chains are very long, i.e., more than a few hundred elements, they should be linked by the first component of each element (beyond the production code number component). Otherwise, the storage recovery routine may exhaust pushdown storage while attempting to mark a long chain of active elements.

The linkage of such chains is determined by the form of the productions associated with the elements. To obtain first-component linking, any production which recursively defines an arbitrarily long sequence of phrases (providing very long sequences of these phrases will actually occur in the object language) should recur on the first phrase class name in its construction string. For example, suppose that the phrase class (STATEMENT SEQUENCE) is to be defined as an arbitrarily long sequence of (STATEMENT)'s, and that very long sequences of (STATEMENT)'s will actually occur in the object language. Then the recursive production

(STATEMENT SEQUENCE) = (STATEMENT SEQUENCE)
(STATEMENT).

should be used instead of

(STATEMENT SEQUENCE) = (STATEMENT)
(STATEMENT SEQUENCE).

since the first production will cause a (STATEMENT SEQUENCE) to be represented by a chain of list elements linked on their first components.

B. Basic Symbols of the COGENT Language

To describe the syntax of the COGENT language itself, we will use the Backus notation.[4] COGENT productions could be used to define the syntax of the COGENT language, but for expository purposes it is better to use a distinct language, which is well-known in the literature on compilers, than to describe COGENT in terms of itself.

In the Backus notation, the names of phrase classes of the COGENT language are written as character strings enclosed by the brackets "<" and ">". The productions relating these phrase classes consist of the name of the phrase class being defined, the symbol "::=", and one or more strings of characters and phrase class names showing alternative constructions of the phrase class. In compound productions, the alternative construction strings are separated by the symbol "|". For clarity, the special phrase-class <empty> is used to denote an empty construction string.

The Backus notation is simply a transliteration of the notation for productions used within COGENT, in which the characters "=", "(", ")", ",", and "." are replaced by "::=", "<", ">", "|", and an end-of-line, respectively, and in which the object characters "(", ")", ",", and "." no longer need to be parenthesized to prevent ambiguity.

COGENT programs are prepared on punched cards, using columns 1 to 72 of each card. The standard 48-character FORTRAN set is used, excluding the redundant (4-8) minus sign. However, blanks and card boundaries are ignored, so that the program may be spaced and indented freely for readability. The characters are classified as:

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|
Y|Z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<normal character> ::= <letter>|<digit>|=|+|-|*|/|$

<special character> ::= .|,|(|)

In terms of these characters, the following basic symbols are defined:

<name string> ::= <letter>|<name string> <letter>|
<name string><digit>

<digit string> ::= <digit>|<digit string><digit>

<open phrase class name> ::= <name string>

<phrase class name> ::= (<open phrase class name>)

<name> ::= <name string>

<positive decimal integer> ::= <digit string>

<positive octal integer> ::= <digit string>

<positive integer> ::= <positive decimal integer>|
<positive octal integer>B

<object character representative> ::= <normal character>|
(<special character>)|($<name string>)

A name may represent a variable, pseudoconstant, or generator name, depending upon its controlling declaration. An object character representative is a symbol used in productions, constants, and character definitions to represent a character of object language. The format ($<name string>) is provided for representing object characters that are not characters of the COGENT language itself.

C. Overall Structure of a COGENT Program

A complete COGENT program has the overall structure:

<COGENT program> ::= <character description>
<primary syntax description> <secondary syntax description>
<generator description>

The character description allows the programmer to define arbitarily the meaning of object character representatives by using character definitions, which specify the representatives in terms of numerical codes for the object characters. The entire description may be omitted, in which case certain standard definitions are assumed for the object character representatives.

The primary syntax description gives a sequence of primary productions describing the input object language, which control the compilation of the syntax analyzer. This description also gives a sequence of goal specifiers, which determine the overall phrase classes for which the analyzer is to search.

The secondary syntax description gives a sequence of secondary productions describing the output object language plus any intermediate object languages which may be used in a computation. The basic distinction

between primary and secondary productions is that only primary productions affect the compilation of the syntax analyzer, while the combined set of primary and secondary productions is used to translate constants into list structures and to compile tables for the primitive character-scanning generators.

The generator description gives the generator definitions and associated declarations which control the compilation of generators.

D. The Character Description: Character Definitions

<character description> ::= <empty>|
$CHARDEF<character definition sequence>

<character definition sequence> ::= <empty>|
<character definition sequence><character definition>

<character definition> ::= <object character representative>=
<input code sequence> <skip code sequence>.<output code>.

<input code sequence> ::= (<open code sequence>)

<skip code sequence> ::= <empty>|(<open code sequence>)

<open code sequence> ::= <positive octal integer>|
<open code sequence>,<positive octal integer>

<output code> ::= <positive octal integer>

Each character definition defines the object character representative on its left by a set of octal integers consisting of one or more input codes, zero or more skip codes, and a single output code. (Note that these octal codes are not suffixed with the letter B.)

1. Input and Skip Codes

The input and skip codes for an object character representative determine how the corresponding object characters will be recognized in the input character string. The syntax analyzer reads the input medium by calling a machine-language routine called the input editor, which furnishes a sequence of character codes (in general, these codes may be integers between 0 and 377_8, inclusive). At a given instant, the analyzer inspects only a single member of this sequence, called the current character code, which always corresponds to the next-to-be-recognized object character.

Before recognizing an object character, the analyzer tests whether the current character code matches one of the input codes given by the corresponding character definition. If a match is found, the object character is recognized, and the current character code is advanced to the next code that does not match any of the skip codes for the just-recognized object character. Thus the input codes in a character definition specify the codes that may represent the defined character on the input medium, and the skip codes specify the codes to be ignored when they follow one of the input codes.

A standard input editor is provided with the COGENT system, but the programmer may write his own input editor in machine language if the standard editor is not suitable. The standard editor reads columns 1 to 72 of input card images and furnishes the corresponding 3600 internal BCD codes, deleting blanks and end-of-card indicators. Upon encountering an end-of-file, the standard input editor produces the single input code 101_8; this code is also produced when a card containing asterisks in all columns from 1 to 72 is encountered, so that such a card acts as a pseudo-end-of-file.

Note that meaningless character codes may be deleted either by the input editor or by the use of skip codes; the former method is faster in execution, but the latter is more general and easier to program.

2. Output Codes

The output code for an object character representative determines how the corresponding object character will be represented on the output media. More generally, whenever a list structure is subjected to a character scan to reduce it to a string of characters, each object character is replaced by its output code. Thus output codes are used to represent object characters, not only in output operations, but also in the packed strings within identifier elements and in the strings that are converted into number elements. All output codes must be numbers between 0 and 373_8 inclusive.

3. Standard Character Definitions

Before reading the character description, the COGENT system assumes a set of standard character definitions which define all object character representatives, except those with the ($<name string>) format, in terms of their 3600 internal BCD codes. If the character description is empty, these standard definitions are used in compiling a COGENT program. However, a nonempty character description may be used, either to define new representatives of the $-type, or to redefine the standard representatives, since a character definition in the character description will take precedence over a standard definition of the same representative.

The standard built-in character definitions are:

A = (21)21.	B = (22)22.	C = (23)23.	D = (24)24.
E = (25)25.	F = (26)26.	G = (27)27.	H = (30)30.
I = (31)31.	J = (41)41.	K = (42)42.	L = (43)43.
M = (44)44.	N = (45)45.	O = (46)46.	P = (47)47.
Q = (50)50.	R = (51)51.	S = (62)62.	T = (63)63.
U = (64)64.	V = (65)65.	W = (66)66.	X = (67)67.
Y = (70)70.	Z = (71)71.	0 = (00)00.	1 = (01)01.
2 = (02)02.	3 = (03)03.	4 = (04)04.	5 = (05)05.
6 = (06)06.	7 = (07)07.	8 = (10)10.	9 = (11)11.
= = (13)13.	+ = (20)20.	- = (40)40.	* = (54)54.
/ = (61)61.	$ = (53)53.	(() = (74)74.	()) = (34)34.
(,) = (73)73.	(.) = (33)33.		

E. The Syntax Descriptions: Productions

<primary syntax description> ::= $PRIMSYN(<goal specifier sequence>)<primary production sequence>

<primary production sequence> ::= <empty>| <primary production sequence><primary production>

<primary production> ::= <production>

<secondary syntax description> ::= <empty>| $SECSYN<secondary production sequence>

<secondary production sequence> ::= <empty>| <secondary production sequence><secondary production>

<secondary production> ::= <production>

<production> ::= <label sequence><special label><forcing marker> <resultant>=<construction string sequence>.

<construction string sequence> ::= <construction string>| <construction string sequence>,<construction string>

<construction string> ::= <empty>|
<construction string><object character representative>|
<construction string><phrase class name>

<resultant> ::= <phrase class name>

<forcing marker> ::= <empty>|*

<special label> ::= <empty>| $IDENT,<positive integer>/|
$OCT/| $DEC/|$FLOAT/| $NOP/| $NOTRAN/

<label sequence> ::= <empty>|<label sequence><label>

<label> ::= <name>/

<goal specifier sequence> ::= <goal specifier>|
<goal specifier sequence>,<goal specifier>

<goal specifier> ::= <goal><terminator sequence>

<goal> ::= <phrase class name>

<terminator sequence> ::= <terminator>|
<terminator sequence><terminator>

<terminator> ::= <object character representative>

1. Special Labels

The general format for productions involves several features which were not discussed in Chapter I. Special labels are used to alter the form of the list structure produced by the syntax analyzer (and similarly to alter the list structures created from constants in generator definitions). The four character-packing special labels $OCT/, $DEC/, $FLOAT/, and $IDENT, n/ cause the analyzer to convert a phrase of object language into a single number or identifier element, instead of the usual list structure representing the construction tree for the phrase.

Thus when a production has the special label "$IDENT, n/", any phrase that is partitioned by this production will be converted by the analyzer into an identifier element, in the identifier table indicated by n, which contains the characters of the phrase (i.e., the string of output codes for these characters). Similarly, when a production has the special label $OCT/, $DEC/, or $FLOAT/, any phrase partitioned by the production will be converted into a number element obtained by converting the characters of the

phrase as a positive octal integer, decimal integer, or floating-point number. The list elements produced by the character-packing special labels are similar to the results of the primitive generators OCTCON, DECCON, FLOATCON, and IDENT, but the special labels allow these elements to be produced directly from the input string, without going through the intermediary of a construction-tree type of list structure.

For example, consider the polynomial object language defined in Chapter I. If the production that defines variables is given the special label

$IDENT,1/ (VARIABLE) = (STRING). (8")

then variables will be converted into identifier elements in identifier table 1, instead of extended list structures. Thus when (8") is used, either the input string "-(A+B)*DC" or the equivalent constant (POLYNOMIAL/ -(()A+B())*DC) will be converted into the list structure shown in Figure 6, as opposed to the structure in Figure 2.

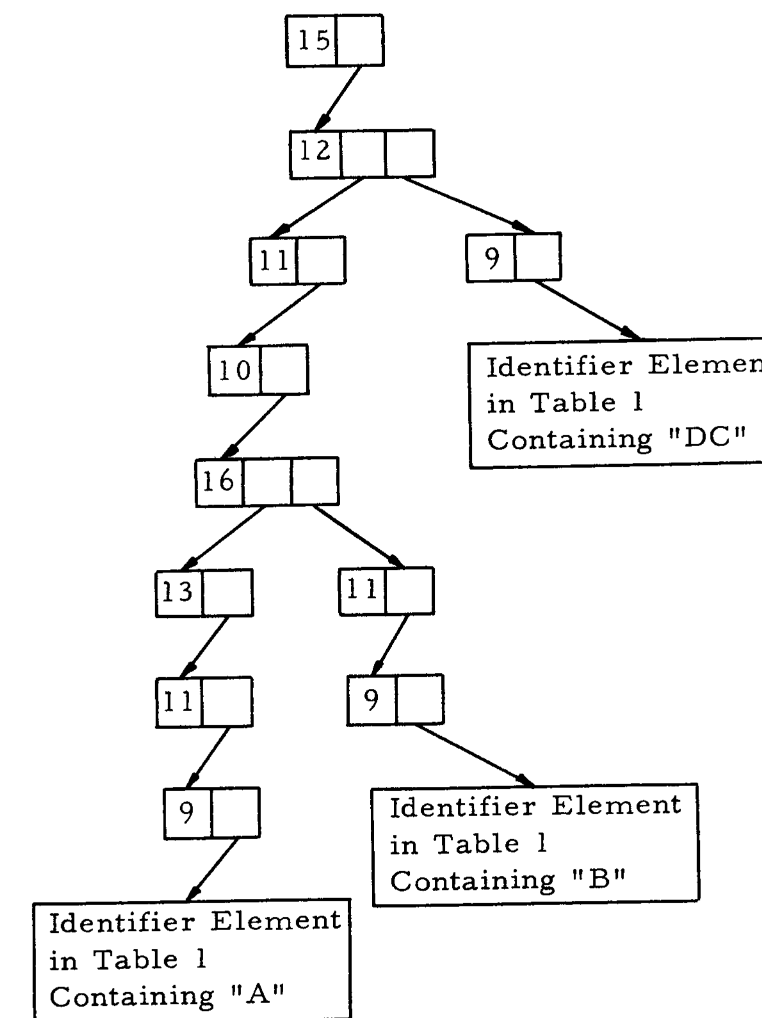


Fig. 6. List Structure for "-(A+B)*DC" Using the Special Label "$IDENT,1/"

The special labels $NOTRAN/ and $NOP/ have a more subtle effect. Normally (in the absence of a special label), when the syntax analyzer recognizes a phrase partitioned by a particular production, one of two actions occurs: (a) If the production does not have a regular label, a single list structure will be produced by creating a new head element containing the code number of the production and pointers to sublists representing immediate subphrases of the recognized phrase. (b) If the production has a regular label, a new list element will not be created by the analyzer; instead, the lists representing subphrases will be given directly to the appropriate generator, and the result of this generator will represent the recognized phrase.

The special label $NOTRAN/ is used to force the creation of a new list element as in (a), even in the presence of one or more regular labels. Then if a generator is indicated by a regular label, it will receive as input a single list structure representing the entire phrase, rather than structures representing the immediate subphrases. For example, if

ADDCOMP/ (POLYNOMIAL) = (POLYNOMIAL)+(TERM).

were used in our polynomial-describing production set, then during the analysis of "-(A+B)*DC" the generator ADDCOMP would receive the two arguments shown in Figure 3a. But if

ADDCOMP/ $NOTRAN/ (POLYNOMIAL) = (POLYNOMIAL)+(TERM).

were used, the generator ADDCOMP would receive the single argument shown in Figure 7.

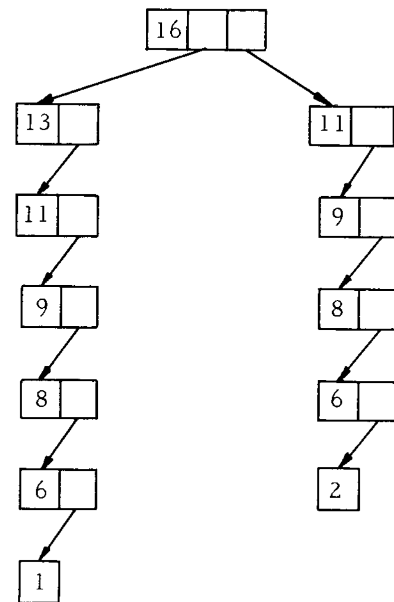


Fig. 7. List Structure Given to a Generator When the Special Label $NOTRAN/ Is Used

The special label $NOP/ is used to suppress the creation of a new list element by the syntax analyzer, even in the absence of regular labels. When the analyzer recognizes a phrase partitioned by a production with the special label $NOP/, the list structure representing the immediate subphrase of this phrase will be taken directly to represent the phrase itself. Implicit in this definition is the assumption that the phrase has only a single immediate subphrase; thus the $NOP/ special label may only be attached to a production containing exactly one phrase class name in its construction string.

The use of $NOP/ can produce a substantial compression in the list structures that represent object phrases, without complicating the programming of manipulations of these structures. As a rule of thumb, it is convenient to attach $NOP/ to any production that has a construction string containing a single phrase class name and no object character representatives, unless the production defines a subphrase of some phrase that is to be represented by an identifier or number list element. When $NOP/ is used in this manner, the resulting list structures will still be decomposed into the correct strings of object language by the primitive character-scanning generators. In addition, since $NOP/ affects the conversion of constants into list structures as well as the action of the syntax analyzer, the list structures denoted by constants will be compressed in the same manner as the structures produced by the analyzer.

As an example of the compression of list structures that may be obtained by a judicious use of $NOP/ and the character-packing special labels, consider the following version of the productions for our polynomial language:

	(LETTER) = A, B, C, D, E.	(1"-5")
	(STRING) = (LETTER),(STRING)(LETTER).	(6"-7")
$IDENT,1/	(VARIABLE) = (STRING).	(8")
$NOP/	(FACTOR) = (VARIABLE).	(9")
	(FACTOR) = (()(POLYNOMIAL)()).	(10")
$NOP/	(TERM) = (FACTOR).	(11")
	(TERM) = (TERM)*(FACTOR).	(12")
$NOP/	(POLYNOMIAL) = (TERM).	(13")
	(POLYNOMIAL) = +(TERM),-(TERM).	(14"-15")
	(POLYNOMIAL) = (POLYNOMIAL) +(TERM).	(16")
	(POLYNOMIAL) = (POLYNOMIAL) -(TERM).	(17")

The list structure representing the polynomial "-(A+B)*DC" (or the equivalent constant) according to these productions is shown in Figure 8. A comparison with Figure 2 shows the degree of compression achieved.

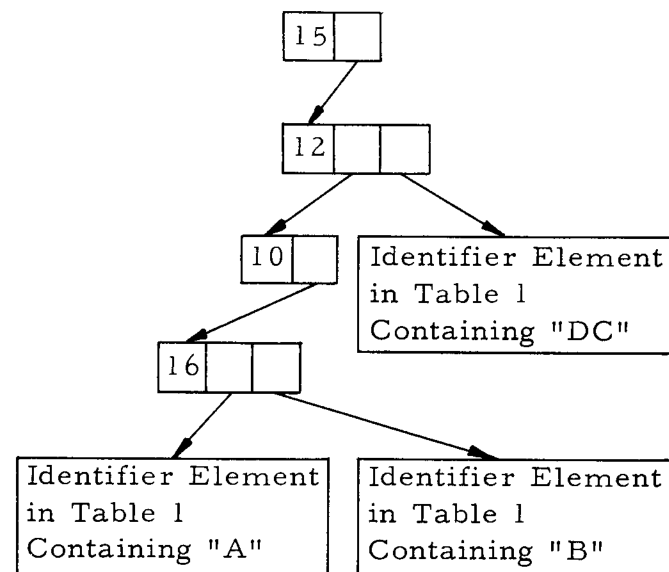


Fig. 8. Compressed List Structure for "-(A+B)*DC" Using the Special Labels $NOP/ and $IDENT,1/

2. Multiple Labels

In addition to permitting special labels, the format of productions allows the use of more than one regular label. If a production has several regular labels, then when the analyzer recognizes a phrase partitioned by this production, a succession of generators will be called, as indicated by the label sequence from right to left. The first (rightmost) generator will receive arguments representing the immediate subphrases (or whatever list structures are indicated by the special label). Thereafter each generator will receive as input the result of the previous generator, and the result of the last generator will be taken to represent the phrase. All labels in a label sequence except the rightmost must denote generators with a single input argument. Labels (as opposed to special labels) only affect the syntax analyzer and are ignored during the conversion of constants into list structures.

3. Forcing Markers

A final feature of productions is the forcing marker. The presence of an asterisk immediately preceding the resultant of a production indicates to the syntax analyzer that the production is to be given preference when necessary to resolve an ambiguity in the object language. The meaning of the forcing marker will be described more precisely in the discussion of the syntax analyzer. It is ignored in the conversion of constants and in the construction of character-scanning tables.

4. Preliminary Processing of Productions

As a COGENT program is read by the COGENT system, each production (either primary or secondary) is subjected to the following preliminary processing:

a. Compound productions (containing more than one construction string) are reduced to the equivalent simple productions. For each construction string, a separate simple production is created containing this construction string, plus the label sequence, special label, forcing marker and resultant which appear in the compound production.

b. If a (simple) production contains the same resultant and construction string as some previously read production, it replaces the earlier production. More specifically, the new label string, special label, and forcing marker replace the corresponding entities of the earlier production, but the production code number assigned to the earlier production is retained.

c. If a production does not match any previously read production, it is assigned a unique production code number. These code numbers are assigned in increasing order, beginning with one.

The syntax analyzer is compiled as soon as the primary syntax description has been read, so that only primary productions affect the structure of the analyzer. On the other hand, the compilation of the tables that govern character scanning, and the conversion of constants into list structures is controlled by the combined set of primary and secondary productions. Thus in general, a COGENT program specifies two distinct sets of productions: a primary set specified by the primary syntax description, and a total set specified by the combined primary and secondary descriptions. When the entire secondary description is empty, these two sets are identical.

5. Formal Definition of Syntactic Concepts

In a properly written COGENT program, both the primary and total sets of productions must satisfy several requirements that arise from the nature of the syntax analyzer and the system routine that converts constants. To state these requirements precisely, we must first define a number of concepts concerning productions and syntax. In all of these definitions, we assume a fixed set of productions, which may be either the primary or total set.

First we must define the notion that a string of object characters (and possibly parameters) is a phrase belonging to a particular phrase class. This concept was defined in Chapter I by introducing construction trees, but we now give a more rigorous definition which does not involve tree-structure considerations.

For this purpose, we consider strings consisting of zero or more elements, which may be object characters, phrase class names, or parameters. A distinction is made between phrase class names and parameters, although a particular parameter is always associated with the phrase class named within the parameter. A string that does not contain any phrase class names is called a *ground* string.

Given two strings S and S', S' is called an *extension* of S if S' may be obtained from S by replacing a single phrase name in S by either (a) the construction string of some production which defines this phrase class name, or (b) a parameter containing this phrase class name. If the phrase class name being replaced is the last (rightmost) phrase class name in S, then S' is called a *right extension* of S. Then a sequence of strings $S^0 \ldots S^f$ is called a *construction* of S^f from the phrase class named N if S^0 is a single-element string containing N, and if each S^{i+1} $(i \geq 0)$ is a right extension of S^i, and if S^f is a ground string.

Now suppose we are given a ground string, which may be either a string of input characters read by the syntax analyzer (i.e., obtained from a string of input character codes in accordance with the appropriate character definitions), or else a string of characters and/or parameters appearing in a constant. We say that this string is a *phrase of the phrase class named N* if there exists a construction of the string from the phrase class named N. If there are two or more distinct constructions of the string from N, then the string is said to be an *ambiguous* phrase.

The concept of a construction may also be used to define the concepts of partitioning and of subphrases. If S^1 is obtained from S^0 by replacing the phrase class name in S^0 by the construction string of a production, then this production is said to *partition* the phrase S^f.

On the other hand, consider any appearance of a phrase class name in any S^i except S^0. The successive right extensions which carry S^i into S^f will map this appearance of a phrase class name into a substring of S^f; the substring is then said to be a *subphrase* of S^f. If the appearance is in S^1, then the corresponding substring is an *immediate* subphrase of S^f.

As an example, we give the construction of the string "-(A+B)*DC" from the phrase class (POLYNOMIAL):

(POLYNOMIAL)
-(TERM)
-(TERM)*(FACTOR)
-(TERM)*(VARIABLE)
-(TERM)*(STRING)

-(TERM)*(STRING)(LETTER)
-(TERM)*(STRING)C
-(TERM)*(LETTER)C
-(TERM)*DC
-(FACTOR)*DC
-(()(POLYNOMIAL)())*DC
-(()(POLYNOMIAL)+(TERM)())*DC
-(()(POLYNOMIAL)+(FACTOR)())*DC
-(()(POLYNOMIAL)+(VARIABLE)())*DC
-(()(POLYNOMIAL)+(STRING)())*DC
-(()(POLYNOMIAL)+(LETTER)())*DC
-(()(POLYNOMIAL)+B())*DC
-(()(TERM)+B())*DC
-(()(FACTOR)+B())*DC
-(()(VARIABLE)+B())*DC
-(()(STRING)+B())*DC
-(()(LETTER)+B())*DC
-(()A+B())*DC

A construction is simply a formalization of the "construction tree" introduced in Chapter I. As can be seen by comparing the above construction with Figure 1, each extension in the construction corresponds to a nonterminal node of the tree and is performed by replacing the phrase class name of the node by the names and characters of the subnodes. On the other hand, if the construction is read backwards, the sequence of extensions becomes a sequence of reductions which correspond, in order, to the successive states of a syntax analyzer while analyzing the string S^f.

We must also define several relationships between phrase classes that are determined by the set of productions. A phrase class named X is said to *span* a phrase class named Y if either X and Y are the same, or if there exists a sequence of phrase class names X, X', ..., Y, beginning with X and ending with Y, in which each name after the first appears in the construction string of some production defining the preceding member of the sequence. Essentially, X spans Y if either X and Y are the same, or if a phrase of the class Y may be a subphrase of a phrase of the class X.

A phrase class is said to be null-containing if the empty string may be a phrase of the class. The set of null-containing phrase classes is the limit of the sequence

$$\xi^0 \subseteq \xi^1 \subseteq \xi^2 \subseteq \ldots$$

where ξ^0 is the set of all phrase classes which are resultants of productions with an empty construction string, and each ξ^{i+1} is the union of ξ^i with the set of phrase classes which are resultants of productions with construction strings containing only phrase classes in ξ^i (and no characters).

A phrase class Y is an immediate head of a phrase class X if there exists a production defining X in which Y is the first element of the construction string. Y is an immediate null-preceded head of X if there exists a production defining X in which Y is an element of the construction string other than the first, and all preceding elements are the names of null-containing phrase classes. Y is a head of X if there exists a sequence X, X', ... , Y of two or more phrase classes, beginning with X and ending with Y, in which each class after the first is an immediate head of the preceding class. Y is a null-preceded head of X if there exists a sequence X, X', ..., Y of two or more phrase classes, beginning with X and ending with Y, in which each class after the first is either an immediate head or an immediate null-preceded head of its predecessor, and one such class is an immediate null-preceded head of its predessor. Essentially, Y is a head of X if a phrase of Y may be a first subphrase of a phrase of X, and Y is a null-preceded head of X if a phrase of Y may be a subphrase of a phrase of X in which there are one or more preceding subphrases, all of which are empty strings.

6. Restrictions on Production Sets

We may now state the requirements imposed on both the primary and the total sets of productions:

a. The syntax determined by the set of productions must be unambiguous; i.e., there must not exist any string of object characters and/or parameters which is an ambiguous phrase of any phrase class. Actually, this requirement may be relaxed slightly for the total (but not the primary) set of productions. For the total set, all possible phrases need not be unambiguous; only the phrases that actually appear in constants in the COGENT program must be unambiguous.

This requirement is inherent in the basic nature of COGENT, since an ambiguous syntax would give an ambiguous mapping of input strings into list structures. However, the programmer has the responsibility of being sure that the productions he writes do not permit ambiguities. Certain gross ambiguities will cause the COGENT system to

reject a COGENT program, but when the ambiguities are more subtle, a syntax analyzer will be successfully compiled that will operate correctly unless it actually reads an input string that is an ambiguous phrase. Ideally, one would like to have a system that would reject any set of productions that defined an ambiguous syntax, but this goal cannot be achieved without seriously restricting the class of acceptable object languages. Indeed, it can be proven that no algorithm can be constructed that will accept an arbitrary set of productions and determine whether they define an unambiguous syntax.(7)

As an example of productions that determine an ambiguous syntax, consider

(STRING) = (LETTER),(STRING)(STRING).

These productions would not be allowed, since any (STRING) containing three or more (LETTER)'s would have more than one construction. On the other hand, either

(STRING) = (LETTER),(STRING)(LETTER).

or

(STRING) = (LETTER),(LETTER)(STRING).

is allowable (although the former is preferable, since it minimizes the push-down storage used by the storage recovery algorithm).

b. The following restriction is placed on the use of character-packing special labels: If X is any phrase class name appearing in the construction string of a production with a character-packing special label, then no phrase class name that is spanned by X may be the resultant of a production with a character-packing special label.

Thus, for example, the production

$IDENT,1/ (STRING) = (STRING)(LETTER).

is not allowed, since (STRING) appears in the construction string of a production with a character-packing special label and is also the resultant of such a production. On the other hand,

(STRING) = (STRING)(LETTER).
$IDENT,1/ (VARIABLE) = (STRING).

is allowable since (STRING) does not span (VARIABLE).

The necessity for this restriction arises from the internal operation of the syntax analyzer. When the analyzer is about to recognize the first character of a phrase that is partitioned by a production with a character-packing special label, the analyzer switches from a "list" mode of operation, in which list elements are created and generators are called, to a "character" mode of operation, in which object characters are simply accumulated in a string. After the complete phrase is recognized, the analyzer switches back to the list mode. Essentially, the restriction on character-handling special labels is intended to prevent the analyzer from attempting to switch to a mode that it is already in.

c. No phrase class may be a null-preceded head of itself. Thus for example,

(DUMMY) =.

(STRING) = (LETTER),(DUMMY)(STRING)(LETTER).

is prohibited, since (STRING) is an (immediate) null-preceded head of itself.

This restriction is necessary because, upon reaching the beginning of a phrase of a class that is a null-preceded self-head, the syntax analyzer cannot determine the number of empty phrases to recognize without looking ahead at the character string. (The difficulty is not that the analyzer is faced with several alternatives, which can only be resolved by look-ahead, but that the number of alternatives is infinite.)

d. As stated earlier, all productions with the special label $NOP/ must have construction strings in which the number of phrase classes is exactly one.

F. The Syntax Analyzer

1. Parsing

Fundamentally, the syntax analyzer is a recursive subroutine which accepts as arguments a phrase class name called the goal, and one or more object characters called terminators, and which parses the input string according to these arguments. The operation of parsing involves finding a substring S of the input string which is a phrase of the class named by the goal, which begins with the current character code, and which is immediately followed by one of the terminator characters. If such a substring is found, the analyzer advances the input medium past the substring (so that the new current character code is an input code of one of the terminators), and produces as the result of the parse a list structure representing S. This result may be defined as follows:

a. If S is partitioned by a production P whose special label is empty, or is $NOP/ or $NOTRAN/, then for each phrase class name X_i in the construction string of P (in order from left to right), the corresponding immediate subphrase of S is parsed with X_i as a goal. The ordered sequence R consisting of the results of these parses is formed.

b. If P has an empty label sequence and an empty special label, or if P has the special label $NOTRAN/, then a normal list element is created containing the production code number of P and pointers to the members of R. R is replaced by (a single-member sequence containing) this list element.

c. If P has the special label $OCT/, $DEC/, or $FLOAT/, then the string S is converted to a number element as a positive octal, decimal, or floating-point number, and this element is made the only member of R. The conversion follows the rules given in the next chapter (pp. 94-96) for the primitive generators OCTCON, DECCON, and FLOATCON, as applied to a sequence of the output codes for the characters in S.

d. If P has a special label of the form $IDENT,n/, then an identifier element with table number n containing the string S (as represented by the sequence of output codes for each character) is made the only member of R. If $n \neq 0$, identifier table n is searched for an already-existing element containing S, and a new element is created only if the search fails. If $n = 0$, no search is made, and an element containing n and S is created but not actually added to any table. When a new identifier element is created, its association list name is normally set to the dummy element. (More generally, it is set to the value of the internal variable ial, as described in Chapter III, pp. 93-94.)

e. For each label of the label sequence of P, in order from right to left, the generator indicated by the label is called and given the members of R as its input arguments, and the result of this generator replaces R.

f. The final value of R is taken as the result of the parse. Note that since the special label $NOP/ must only be used in productions whose construction strings contain one phrase class name, the final value of R will always contain a single member.

2. Treatment of Ambiguities

Essentially, the syntax analyzer is a recursive routine which reads and tests character codes from the input medium, creates list structures, and calls generators. At various points in its operation, the analyzer will encounter conditional branches to different control paths corresponding to different parsings of the input string. Frequently these

branches will be determined by testing the current character code, which always represents the first unparsed character in the input string. However, at certain points the value of the current character code will be inadequate to determine the branch, so that the analyzer must look ahead at further characters; such branch points are called ambiguity points.

From an abstract viewpoint, the look-ahead problem is solved by allowing the analyzer to fission into two or more analyzers which simultaneously pursue the different control paths extending from an ambiguity point. Thus, in general, the analysis process is carried out by an assembly of abstract analyzers, each of which may fission into two or more analyzers upon encountering an ambiguity point, or may fail (i.e., vanish from the assembly) upon testing an input character incompatible with the parse being pursued. Ultimately, if all the abstract analyzers fail before completing a parse with respect to the desired goal, then the input string is not a well-formed phrase of the goal class; if just one analyzer completes a successful parse, then the input string is well-formed; and if more than one analyzer completes a successful parse, then the input string is ambiguous.

The real syntax analyzer simulates an assembly of abstract analyzers by alternating between two modes: a normal mode, corresponding to an assembly containing a single analyzer; and an ambiguity mode, corresponding to an assembly containing several analyzers. In the ambiguity mode, a linked list of abstract analyzer states is maintained, and each state in turn is advanced from the reading of one character code to the next, so that the abstract analyzers are synchronized by the reading of the input string. A further property of the ambiguity mode is that the actions of list construction and generator calling which would be performed by a single abstract analyzer are not actually executed; instead, indicators for these actions are stored in a "recognition queue" which is part of the analyzer state. When all but one of the abstract analyzers fail, the actions indicated by the recognition queue of the successful analyzer are carried out, and the real analyzer returns to the normal mode.

From the viewpoint of the user, the syntax analyzer appears to contain an input processor which reads character codes and passes them on to the parsing routine. Normally this processor stays a single character code ahead of the parsing routine, but at ambiguity points, it runs further ahead, reading as many character codes as are necessary to resolve a "local" ambiguity.

3. Forcing Markers

The relation between the real syntax analyzer and an assembly of abstract analyzers allows a more precise meaning to be given to the forcing markers that may appear in productions. When the analyzer is in the normal mode, forcing markers are ignored. But in the ambiguity mode, if an abstract analyzer recognizes a phrase that is partitioned by a production with an asterisk forcing marker, then this abstract analyzer will immediately force all other abstract analyzers to fail, its recognition queue will be executed, and the real analyzer will return to the normal mode. Thus a forcing marker will cause ambiguities to be resolved by giving preference to the marked productions.

To illustrate the use of forcing markers, consider a simple assembly language in which an instruction may consist of either the letters "BCD" followed by a (CHARACTER STRING), or else any alphanumeric operation code except "BCD" followed by an (ADDRESS FIELD). Such an object language could be defined by the productions:

(OP CODE) = (LETTER),(OP CODE)(LETTER),(OP CODE)(DIGIT).

*(BCD CODE) = BCD.

(INSTRUCTION) = (OP CODE)(ADDRESS FIELD).

(INSTRUCTION) = (BCD CODE)(CHARACTER STRING).

Upon reading the characters "BCD", the analyzer will be forced to resolve an ambiguity by ignoring the interpretation of "BCD" as an (OP CODE).

4. Goal Specifiers

When the syntax analyzer is entered on the main level of a COGENT program, it must be given a goal that specifies the phrase class for which it is to search, and a set of terminators that specify the possible input characters which may follow the phrase being sought. This information is provided by the goal specifier sequence, which appears at the beginning of the primary syntax description. Each goal specifier gives a phrase class name indicating a goal, followed by one or more object character representatives indicating terminators. When the syntax analyzer is initially called, it is given the leftmost goal specifier; but after the analyzer returns from this initial call, it may be called again an arbitrary number of times and given any of the specifiers in the specifier sequence.

At any point in the execution of a COGENT program, either a single goal specifier is active, or else all specifiers are inactive. Initially, the first (leftmost) specifier is active, but the active specifier may be changed at any time by calling the primitive generator SETIVGOL(n), where n is an integer which indicates the specifier to be made active by giving its position in the specifier sequence, from left to right. By calling SETIVGOL(0), all specifiers may be made inactive.

Although the active goal specifier may be reset at any point in a COGENT program, its effect always occurs on the main level of the program, which consists of repeated calls of the syntax analyzer with the current active specifier as its argument. When the program begins execution, the analyzer is called and given the goal and terminators indicated by the first specifier. If the input string is well-formed and not ambiguous, the analyzer will parse it and will exit with the input string advanced so that the current character code is the first character beyond the parsed phrase, i.e., one of the terminators. When such an exit occurs, the analyzer is immediately called again and given the current active specifier; if no specifier is active, the entire program is terminated.

Three points should be noted: First, the active specifier that is set by SETIVGOL has no effect until the next main call of the syntax analyzer. Secondly, for each main-level call of the analyzer after the initial call, the first character of the phrase to be parsed will be one of the terminators for the previous call. Finally, the last character read by the entire program will be a single character code following the last parsed phrase and will be one of the terminators for the last main-level call of the analyzer.

G. Constants

<parameter> ::= (<open phrase class name>)|
(<open phrase class name>/<positive integer>)

<object string> ::= <empty>|
<object string><object character representative>|
<object string><parameter>

<constant> ::= (<open phrase class name>/<object string>)|
<positive integer>

Constants are used in the generator description to denote list structures which are fixed at the time a COGENT program is written and are never altered as the program runs. A constant that is simply a positive integer denotes a one-element list structure consisting of the appropriate integer number element. A constant containing a phrase class name followed by an object string denotes the list structure obtained by parsing the object string with respect to the goal specified by the phrase class name.

This parsing of object strings is performed when the program is compiled by the COGENT system, but it is closely analogous to the action of the syntax analyzer while the program is being executed. Basically, the parsing of constants differs from the parsing of input strings by the

analyzer in three respects: the total production set is used instead of the primary set, labels (but not special labels) are ignored, and parameters in the object string are converted into parameter elements in the list structure.

Given a constant containing a phrase class name N and an object string S, which must be an unambiguous phrase of the class N, the list structure denoted by the constant is defined as follows:

1. If S has a single member which is a parameter containing the phrase class name N, then the resulting list structure is a single parameter element. If the parameter contains a positive integer, then this will be taken as the index of the parameter element. Otherwise, the index is the order of appearance of the parameter among all parameters in the entire constant, as read from left to right.

2. If S is partitioned by a production P whose special label is empty, or is $NOTRAN/, then the resulting list structure is headed by a normal element containing the production code number of P, followed by the names of sublists representing the immediate subphrases of S, as obtained by parsing these subphrases with respect to the phrase class names in the construction string of P.

3. If P has the special label $NOP/, the resulting list structure is the structure obtained by parsing the single immediate subphrase of S with respect to the single phrase class name in the construction string of P. (If P has a $NOP/ special label, it must have a single phrase class name in its construction string.)

4. If P has the special label $OCT/, $DEC/, or $FLOAT/, then the resulting list structure is a single number element, obtained by converting S as an octal, decimal, or floating-point number. The conversion follows the rules given in the next chapter (pp. 94-96) for the primitive generators OCTCON, DECCON, and FLOATCON, as applied to a sequence of the output codes for the characters in S. Parameters in S cause an error warning, but are otherwise ignored.

5. If P has the special label $IDENT,n/, then the resulting list structure is a single identifier element, with table number n, containing a string of the output codes for each character in S. Parameters in S are ignored but cause error warnings. If $n \neq 0$, identifier table n is searched for an already-existing element (which would be part of the list structure for a previously compiled constant) containing the desired string, and a new element is created only if the search fails. If $n = 0$, the set of previously compiled tableless identifier elements is searched, and a new element is created only if a previous tableless identifier with the desired string does not exist. All identifier elements created by parsing constants have their association list names initially set to the dummy element, unless otherwise specified by identifier declarations.

The list structures created by parsing constants must never be altered during the execution of a COGENT program. The only allowable exception to this rule is the resetting of the association list names of identifier elements within these structures.

H. The Generator Description: Nesting of Generator Definitions

```
<generator description> ::= $PROGRAM<main declaration sequence>
     <main generator definition sequence>

<main declaration sequence> ::= <declaration sequence>

<main generator definition sequence> ::=
     <generator definition sequence>

<generator definition sequence> ::= <empty>|
     <generator definition sequence>< generator definition>

<generator definition> ::= $GENERATOR <name>
     ((<input variable sequence>)
     < declaration sequence><generator definition sequence>
     < statement sequence>).

< declaration sequence> ::= <empty>|
     <declaration sequence><declaration>

< statement sequence> ::= <empty>| < statement sequence>
     < statement>|<statement sequence>< statement label>

<statement> ::= <assignment statement>|< control statement>
```

The generator description gives a set of generator definitions, each of which specifies a list-processing subroutine called a generator. In general, each generator accepts an arbitrary (zero or more), but fixed, number of input arguments and either produces a single result or else fails without producing any result. However, in addition to producing a result or failing, a generator may influence the course of a computation through one or more side effects, such as:

1. Writing on the output media.
2. Resetting the value of a global variable, i.e., a variable associated with another generator, or with the entire program.

3. Creating or erasing identifier elements.
4. Resetting the association list of an identifier element.
5. Altering an existing list structure (as opposed to creating a new structure) by means of a primitive generator such as REPLACE.

The syntax of the generator description allows generator definitions to be nested; i.e., any generator definition may contain within itself the definition of one or more subgenerators. The value of this nesting feature lies in the ability of a subgenerator to evaluate and reset variables associated with the larger generator that contains it.

Several definitions are needed to clarify the relationships between nested generators. If the definition of generator B is a member of a generator definition sequence that is an immediate subphrase of the definition of generator A, then B is an immediate subgenerator of A. More generally, B is a subgenerator of A if it is an immediate subgenerator of A or an immediate subgenerator of an immediate subgenerator of A, etc. Finally, if a generator is defined by a member of the main generator definition sequence, then it is not a subgenerator of any other generator, and is called a main generator.

The nest structure requires a careful specification of the relationship between an appearance of a name within a generator definition and the declaration or input variable sequence that controls this appearance. Consider an appearance of a name within a generator definition (i.e., in a statement in the definition, or perhaps in an initialization specifier of a declaration in the generator definition). If the name is defined by some declaration in this generator definition (and if the declaration appears before the appearance of the name), then the appearance of the name is controlled by this declaration. If the name occurs in the input variable sequence of the generator definition, then the appearance of the name is controlled by the input variable sequence. But if the name is not defined by a declaration and does not occur in the input variable sequence, then the appearance of the name is said to be global to the generator definition.

Now suppose that an appearance of a name is global to the definition of a generator B, and that B is an immediate subgenerator of a generator A. If the name is defined by a declaration or input variable sequence in the definition of A, then this declaration or sequence controls its appearance. Otherwise, the appearance is global to A as well as to B.

Thus to find the controlling declaration or input variable sequence for an appearance of a name, one searches upward through the nest of generator definitions until the first applicable declaration or sequence is found. However, an appearance of a name may not be controlled by any declaration

or input variable sequence in the entire nest; i.e., the appearance may be global to a main generator definition. In this case, if the name is defined by a main declaration, then its appearance is controlled by this main declaration. But if the name is not defined by a main declaration, then its appearance is implicitly declared to be a generator name representing a main generator or a primitive generator. (Subgenerator names cannot be implicitly declared, but must appear in generator declarations, as described in the next section.)

In addition to appearances of names in generator definitions, the appearances of names in labels within productions may also be controlled by declarations. An appearance of a name in a production label behaves in the same manner as a name that is global to a main generator; if the name is defined by a main declaration, the label is controlled by this declaration; otherwise the label is implicitly declared to be a main or primitive generator name. This is the only situation in which a declaration may control a name that appears before the declaration itself. Note that a main declaration may be used to define a production label to be a variable name instead of a generator name, providing that the named variable always has a generator element as its value. In this case, the current value of the variable will specify a generator to be called by the syntax analyzer.

To illustrate the relation of nested generator definitions and declarations, consider the following skeleton of a complete COGENT program:

```
$PRIMSYN ... BETTY/ ... DON/ ...
$PROGRAM  $OWN SAM,DON.  $PCON BILL = (FACTOR/ABC).
$GENERATOR BETTY ((BOB)  $GEN SUE, MAE.
    $GENERATOR SUE ((SAM)  $LOCAL TOM.
    ... TOM = ADD( MAE( BILL), SAM). ... ).
    $GENERATOR MAE ((JOE)  $OWN BOB.
    ... BOB = BETTY(SAM). ... ).
... DON = MAE.  SAM = BETTY( SUE(BOB)). ... ).
```

In line 1:

BETTY is implicitly declared to be a main generator name.

DON is controlled by the own declaration in line 2.

In line 5:

TOM is controlled by the local declaration in line 4.

ADD is global to SUE and BETTY and is implicitly declared to be a primitive generator name.

MAE is global to SUE and is controlled by the generator declaration in line 3.

BILL is global to SUE and BETTY, and is controlled by the pseudoconstant declaration in line 2.

SAM is controlled by the input variable sequence in line 4.

In line 7:

BOB is controlled by the own declaration in line 6.

BETTY is global to MAE and BETTY and is implicitly declared to be a main generator name.

SAM is global to MAE and BETTY and is controlled by the own declaration in line 2.

In line 8:

DON is global to BETTY and is controlled by the own declaration in line 2.

MAE is controlled by the generator declaration in line 3.

SAM is global to BETTY and is controlled by the own declaration in line 2.

BETTY is global to BETTY and is implicitly declared to be a main generator name.

SUE is controlled by the generator declaration in line 3.

BOB is controlled by the input variable sequence in line 3.

I. Declarations and Input Variable Sequences

<declaration> ::= <local declaration>|<own declaration>|
<pseudoconstant declaration>|<generator declaration>|
<identifier declaration>

<local declaration> ::= $LOCAL<normal declaration string>.

<own declaration> ::= $OWN<normal declaration string>.

<pseudoconstant declaration> ::= $PCON<normal declaration string>.

<generator declaration> ::= $GEN<generator declaration string>.

<identifier declaration> ::= $IDA<identifier declaration string>.

<normal declaration string> ::= <name><initialization specifier>|
<normal declaration string>,<name><initialization specifier>

<generator declaration string> ::= <name>|
<generator declaration string>,<name>

<identifier declaration string> ::= <constant><initialization specifier>|<identifier declaration string>,<constant>
<initialization specifier>

<initialization specifier> ::= <empty> | = <constant> | = <name>

<input variable sequence> ::= <empty>|
<nonempty input variable sequence>

<nonempty input variable sequence> ::= <name>|
<nonempty input variable sequence>,<name>

An *initialization specifier* indicates an initial value to be given to a variable, pseudoconstant, or identifier association list. An empty specifier always denotes the dummy list element. A specifier containing a constant denotes the value of the constant. When a specifier contains a name, the name must be a pseudoconstant, and the specifier denotes the value of the constant associated with the pseudoconstant.

1. Local Declarations

A *local declaration* gives a sequence of names, each followed by an initialization specifier, and declares these names to be *local* variables of the generator whose definition contains the declaration. The initialization specifiers indicate the initial values to which these local variables will be set each time the generator is entered. Local declarations must not appear in the main declaration sequence of a COGENT program.

To describe the behavior of local variables, it is useful to formalize the concept of a *calling chain*. At any instant in the execution of a COGENT program, the state of the program may be described by a sequence of generators, called the *calling chain*, in which each member has been called by its predecessor and has in turn called its successor. The recursive capability of the generators allows this chain to contain more than one appearance of the same generator. In this situation the local variables of a generator will be allocated separate storage areas for each appearance

of the generator in the calling chain, but only the area corresponding to the last appearance of this generator will be active, i.e., available for evaluation and resetting by the generator and its subgenerators. Thus when a generator is called, a new area for its local variables is allocated and made active, and is initialized as indicated by the initialization specifiers for the variables. When the generator exits, this area is released, and the area associated with the previous occurrence of the generator in the calling chain is made active.

If there is no occurrence of some generator in the calling chain, then there is no storage area allocated for the local variables of this generator, and these variables are undefined. This fact leads to a restriction on the calling of subgenerators. Suppose that B is a subgenerator of A, and that B contains a global appearance of a name that is controlled by a local declaration in A. Then B can only be called if A already appears in the calling chain, since otherwise the global appearance will refer to an undefined variable.

2. Input Variable Sequences

An *input variable sequence* gives a sequence of names that are declared to be *input* variables of the generator whose definition contains the input variable sequence. Input variables behave in the same manner as local variables, except that when storage is allocated for the variables upon entrance to the generator, the variables are initially set to the input arguments of the generator.

The number of arguments for a generator is determined by the number of names in its input variable sequence. (An empty sequence indicates a generator with no arguments.) When a call of the generator is indicated by a compound expression, the assignment of arguments to input variables is determined by an ordered correspondence between the input variable sequence of the generator and the subexpression sequence of the compound expression. Similarly, when a call of the generator is indicated by the rightmost label of a production with no special label, the assignment of arguments to variables is determined by an ordered correspondence between the input variable sequence of the generator and the phrase class names in the construction string of the production.

The restriction on the calling of subgenerators containing global appearances of local variables also applies to subgenerators containing global appearances of input variables.

3. Own Declarations

An *own declaration* gives a sequence of names, each followed by an initialization specifier, and declares these names to be *own* variables of

the generator whose definition contains the declaration. The initialization specifiers indicate the initial values to which these own variables will be set at the beginning of the execution of the entire COGENT program.

The own variables of a generator are assigned a single, permanently allocated storage area. Thus when the calling chain contains several occurrences of the same generator, the own variables of this generator are the same for all occurrences, and a resetting of an own variable by one occurrence will affect the value for all other occurrences. In addition, when a generator exits, the values of its own variables are not lost, but remain available to later calls of the same generator.

An own declaration may also appear in the main declaration sequence. In this case, the names in the declaration are declared to be universal variables, i.e., own variables of the entire program. A universal variable may be referenced by any generator definition, or by a production label (if its value is an appropriate generator element). Any resetting of a universal variable is effective throughout the program.

4. Pseudoconstant Declarations

A pseudoconstant declaration gives a sequence of names and initialization specifiers and declares each name to be a pseudoconstant representing the value of the following initialization specifier. Any appearance of a name controlled by a pseudoconstant declaration is completely equivalent to a constant with the value represented by the pseudoconstant. Thus a name controlled by a pseudoconstant declaration must never appear in a context in which a constant would be meaningless, e.g., in an assignment statement that assigns a value to this name.

An initialization specifier in a pseudoconstant declaration may itself contain a pseudoconstant controlled by a previous or higher-level declaration. Thus several pseudoconstants may be chained together to represent the same constant. A pseudoconstant declaration may be used in the main declaration sequence to define pseudoconstants that are referenced throughout the entire program.

5. Generator Declarations

A generator declaration gives a sequence of names and declares each name to be a generator name, which is a type of pseudoconstant representing a generator element. The generator declaration is similar to the pseudoconstant declaration, except that it does not directly specify the value of the pseudoconstants that it declares; instead, it indicates that these values will be specified by a sequence of generator definitions immediately following the current declaration sequence.

Whenever a generator has one or more immediate subgenerators, the declaration sequence in its definition must include a generator declaration giving the names of these subgenerators. Thus each subgenerator name must appear twice, once in a generator declaration in the declaration sequence, and once at the beginning of a generator definition in the definition sequence following the declaration sequence. (This use of two separate syntactic entities for declaring and defining generator names is peculiar to COGENT. In the analogous construction in ALGOL, a single entity, the procedure declaration, simultaneously declares that a name represents a procedure and defines this procedure. This type of construction has been avoided in COGENT to simplify the compiling process for COGENT programs.)

Now suppose that B is an immediate subgenerator of A. Then the name of B must be declared in the declaration sequence of A, so that appearances of this name (with this meaning) can occur only in A or its subgenerators. But although references to the name of B are restricted to this region of the program, calls of B may occur from outside the region since, for example, the generator element for B can be made the value of a universal variable. The only restriction on calls of a subgenerator (as opposed to references to its name) is the restriction given above on global appearances of the names of local or input variables.

A generator declaration may be used in the main declaration sequence to declare that certain names represent main generators. Normally this is unnecessary, since in the absence of a declaration these names will be implicitly declared as the names of main or primitive generators. But occasionally the programmer may wish to give to a main generator some name that is also the name of a primitive generator. In this situation, an implicit declaration will be ambiguous, and the name must be given in a main generator declaration.

The general rules by which the system distinguishes between main and primitive generator names are as follows:

a. If a name is not controlled by a declaration and does not appear at the beginning of any main generator definition, it is taken to be a primitive generator name. If such a name does match any of the names of primitives available in the COGENT system, then the programmer must supply his own machine-language subroutine for the generator.

b. If a name is not controlled by a declaration but appears at the beginning of some main generator definition, and if the name does not match any of the primitive names known to the system, then the name is taken to be a main generator name.

c. If a name is controlled by a main generator declaration, it is taken to be a main generator name, even if it matches a primitive name

known to the system. In this case, the name must also appear at the beginning of some main generator definition in the program.

d. If a name is not controlled by a declaration, and appears at the beginning of some main generator definition, and also matches some primitive name known to the system, then the meaning of the name is ambiguous.

6. Identifier Declarations

The final type of declaration is the identifier declaration, which gives a sequence of constants, each followed by an initialization specifier. Each constant must denote a list structure consisting of a single identifier element; the declaration causes the association list name of this identifier element to be initialized, when the program begins execution, to the value indicated by the corresponding initialization specifier. If an identifier element appears in the list structure denoted by any constant in the program, but does not appear in any identifier declaration, then its association list will be initialized to the dummy element when the program begins execution.

Identifier declarations describe identifier elements, rather than names, and therefore do not control the appearances of any names in a program. Because of this, an identifier declaration may appear in any declaration sequence, including the main sequence, and its meaning is independent of its position in the program.

In general, declarations and input variable sequences are subject to the following restrictions: Within the input variable sequence and declaration sequence of a single generator (exclusive of its subgenerators), the same name must not be declared more than once. Similarly, within the main declaration sequence, the same name must not be declared more than once.

J. Expressions and Assignment Statements

<expression> ::= <name>|<constant>|<compound expression>

<compound expression> ::= <head expression>(<expression sequence>)

<head expression> ::= <expression>

<expression sequence> ::= <empty>|<nonempty expression sequence>

<nonempty expression sequence> ::= <expression>|
<nonempty expression sequence>,<expression>

<assignment statement> ::= <direct assignment statement>|
<synthetic assignment statement>|<analytic assignment statement>

<direct assignment statement> ::= <name> = <expression>.|
<compound expression>.

<synthetic assignment statement> ::=
<name>/=<template expression><synthesis string>.

<template expression> ::= <expression>

<synthesis string> ::= <empty>|<synthesis string>,<synthesis item>

<synthesis item> ::= <empty>|<expression>

<analytic assignment statement> ::=
<test expression>=/<template expression><analysis string>.

<test expression> ::= <expression>

<analysis string> ::= <empty>|<analysis string>,<analysis item>

<analysis item> ::= <empty>|<name>

1. Expressions

Expressions are used within statements to indicate values, which are obtained during the execution of the program by the process of evaluating the expression. When an expression is a constant, the indicated value is the value of the constant; when an expression is a name, the indicated value is the current value of the variable represented by the name, or the permanent value of the pseudoconstant represented by the name.

The evaluation of a compound expression causes the generator containing the compound expression to call other generators, or to call itself recursively. Specifically, a compound expression is evaluated as follows:

a. The head expression and the expressions in the expression sequence are evaluated. The order in which these evaluations are performed is not defined and will be chosen by the COGENT programming system to

optimize the machine code compiled from the COGENT program. If the evaluation of the head expression or any member of the expression sequence fails, then the evaluation of the entire compound expression fails, without evaluating further subexpressions or performing step b. If failure does not occur, then the value of the head expression must be a generator element, and the number of items in the expression sequence must equal the number of input arguments for the generator indicated by this generator element.

b. The generator indicated by the value of the head expression is called and given the values of the members of the expression sequence as input arguments. If this generator fails, then the evaluation of the compound expression fails. Otherwise, the result of this generator is taken as the value of the compound expression.

2. Direct Assignment Statements

A direct assignment statement containing a name on the left and an expression on the right is executed by evaluating the expression on the right and assigning its value to the variable named on the left. (The name on the left must not be a pseudoconstant.) If the evaluation of the expression fails, then the statement fails without affecting the value of the left-hand variable.

A direct assignment statement containing only a compound expression is executed by evaluating the compound expression and ignoring its value. If the evaluation fails, the statement fails.

3. Synthetic Assignment Statements

A synthetic assignment statement is executed as follows:

a. The template expression and all expressions in the synthesis string are evaluated. The order of evaluation is undefined and will be chosen to optimize code. If the evaluation of any of these expressions fails, then the statement fails without evaluating further expressions or executing step b.

b. An instantiated copy of the value of the template expression is formed and made the value of the variable named on the left of the statement. (This name must not be a pseudoconstant.)

The instantiated copy Y of an arbitrary list structure X is defined recursively as follows:

a. If X is headed by a normal element, then Y is a structure headed by a normal element with the same production code number, followed by the names of instantiated copies of the sublists of X.

b. If X is an identifier, number, generator, or dummy element, then Y is the same element as X.

c. If X is a parameter element with index i, and the ith item in the synthesis string (from left to right, counting empty items) is an expression, then Y is the value of this expression. If the ith item is empty, then Y is the same element as X.

In any synthetic assignment statement, the number of items in the synthesis string must always be greater than or equal to the largest index in any parameter element of any list structure that may ever occur as the value of the template expression.

Two properties of the synthetic assignment statement should be noted. First, if an item in the synthesis string is empty, or if the value of an expression in the string contains a parameter element, then the result of the statement may contain parameter elements. When such an element arises from an empty item in the synthesis string, then its index is the same as the corresponding element in the template value. When such an element arises from a parametric-valued expression in the string, then its index is the same as the corresponding parameter in the value of this expression.

Secondly, a synthetic assignment statement may be used to cause a parameter associated with some phrase class to be replaced by a phrase of a different class. The result of such an operation is said to have mixed syntax. Although structures with mixed syntax cannot be represented by constants, their creation and use are permissible in a COGENT program. The action of various list-processing operations on such structures is always well-defined when the structures are considered explicitly as lists, but the operations can no longer be considered directly as operations upon phrases of object language, without regard to the underlying list representation.

4. Analytic Assignment Statements

An analytic assignment statement is executed as follows:

a. The test and template expressions are evaluated. The order of evaluation is undefined and will be chosen to optimize code. If the evaluation of either expression fails, the statement fails without evaluating further expressions or executing step b or c.

b. The value of the test expression is compared with the value of the template expression as described below. If the comparison fails, then the statement fails without executing step c. If the comparison succeeds, it will "associate" a unique list structure with each name that appears in the analysis string.

c. The unique list structure associated with each name in the analysis string is made the value of the variable represented by this name. (The names in the string must not be pseudoconstants.)

The comparison of a test value X with a template value Y is executed as follows:

a. If Y is headed by a normal element, then the comparison fails unless X is headed by a normal element with the same production code number and number of components. The sublists of X are compared with the sublists of Y, and the comparison fails if any of these subcomparisons fail.

b. If Y is an identifier, generator, or dummy element, then the comparison fails unless X is the same element.

c. If Y is a number element, then the comparison fails unless X is a number element with the same mode and value.

d. If Y is a parameter with index i, the comparison does not fail, and if the ith item in the analysis string (from left to right, counting empty items) is a name, then X is associated with this name.

The following restriction is imposed on analytic assignment statements: If the ith item in the analysis string is not empty, then every list structure that may ever occur as the value of the template expression must contain exactly one parameter element with index i. This restriction insures that the comparison of list structures will always associate a unique structure with each name in the analysis string.

An analytic assignment statement with an empty analysis string may be used to compare two list structures completely, providing that the template structure never contains parameter elements. In particular, such a statement may be used to compare numbers for equality.

K. Statement Labels, Control Statements, and the Flow of Control

<statement label> ::= <statement number>/

<statement number> ::= <positive integer>

<control statement> ::= <unconditional jump statement>|
<conditional jump statement>| <return statement>|
<fail statement>

<unconditional jump statement> ::= +<statement number>.

<conditional jump statement> ::=
+<statement number>UNLESS<assignment statement>|
+<statement number>IF<assignment statement>

<return statement> ::= $RETURN(<expression>).| $RETURN.

<fail statement> ::= $FAILURE.

Statement labels may be inserted at arbitrary points in a statement sequence. The only restrictions on the use of these labels are:

1. A statement sequence must not contain two or more statement labels with the same statement numbers. (Statement numbers are compared as numerical values, not character strings.)

2. Within a statement sequence, for every statement number appearing in a jump statement, the same number must also appear in a statement label.

3. Zero must not be used as a statement number.

A statement number is said to denote the first statement following the unique label containing this number. If no statement follows this label, the number denotes the end of the statement sequence.

The control statements serve two purposes: The unconditional and conditional jump statements cause control jumps between statements within the same statement sequence (i.e., in the same generator). The return and fail statements cause the generator containing these statements to exit. The exact behavior of the control statements is best described by giving the rules that govern the flow of control through a statement sequence:

1. After a generator has been entered and its local and input variables have been initialized, control passes to the first statement of its statement sequence.

2. When control passes to an assignment statement that is not imbedded in a conditional jump statement, the statement is executed, and if it does not fail, control passes to the next statement in sequence. If the assignment statement fails, the entire generator containing the statement fails without further execution.

3. When control passes to an unconditional jump statement, it is passed on to the statement denoted by the statement number in the jump statement.

4. When control passes to a conditional jump statement containing UNLESS, the assignment statement imbedded in the jump statement is executed. If the assignment statement fails, control passes to the statement denoted by the statement number in the jump statement. If the assignment statement does not fail, control passes to the next statement in sequence.

5. When control passes to a conditional jump statement containing IF, the assignment statement imbedded in the jump statement is executed. If the assignment statement does not fail, control passes to the statement denoted by the statement number in the jump statement. If the assignment statement fails, control passes to the next statement in sequence.

6. When control passes to a return statement containing an expression, the expression is evaluated. If the evaluation does not fail, the generator containing the statement returns with the value of the expression as its result. If the evaluation fails, the generator fails without further execution.

7. When control passes to a return statement that does not contain an expression, the generator containing the statement returns with the dummy element as its result.

8. When control passes to a fail statement, the generator containing the statement fails without further execution.

9. If control passes to the end of a statement sequence, either from the last statement in the sequence or by means of a jump to a label after the last statement, then the generator containing the statement sequence returns with the dummy element as its result.

CHAPTER III

PRIMITIVE GENERATORS AND INTERNAL VARIABLES

This chapter describes the primitive generators available in the initial version of the COGENT system. It is expected that additional primitives will be added to the system in the future. In addition, the programmer may introduce his own primitives by supplying the appropriate machine-language subroutines.

Many of the primitives described in this chapter have restrictions on the type or nature of their input arguments. If these restrictions are violated, these primitives may cause the entire program to terminate, or they may have an unpredictable effect on the course of the program.

A. Internal Variables

Internal variables provide communication between primitive generators. They are similar to own variables in that their storage is permanently allocated (i.e., never lost, or reallocated during recursion), and they are initialized at the beginning of the execution of the entire program. The distinctive characteristic of internal variables is that their usage is fixed by the system rather than specified by the user. Furthermore, internal variables cannot be referenced directly by names in generator definitions. To emphasize the latter property, we will denote internal variables in this chapter by underlined uncapitalized names such as rem, which obviously cannot appear as names in a COGENT program.

Each internal variable is associated with one or more primitive generators which set or evaluate it in the course of their operation. For example, rem is set by DIVIDE(X, Y) to the remainder of dividing X by Y (providing X and Y are integers). However, for each internal variable there are also two primitive generators whose only purpose is to set or evaluate the internal variable. Thus SETIVREM(X) is a generator which gives rem the value X, and IVREM() is a generator whose result is the value of rem. These standard setting and evaluating generators provide a facility for defined (programmer-written) generators to communicate with the internal variables, despite their inability to reference these variables directly.

Each internal variable has a standard initial value to which it is set when a program begins execution. Since the operation of many primitives is conditioned by the values of internal variables, the standard initial values specify standard modes of operation for the primitives. For example, when the primitive generator ALIST finds an association list which is the dummy element, the value of the internal variable dal is taken to specify a generator to be called. Since the initial value of dal is a generator element pointing to

the primitive generator FAILURE, the standard mode of operation is for ALIST to fail upon encountering a dummy association list. But the programmer may introduce another mode of response by defining his own generator and setting dal to a generator element referring to this generator.

Each internal variable also has a limited set of legal values. The primitive generators for setting the internal variables must not be used to set a variable to an illegal value.

1. Properties of the Internal Variables

The following list gives the name, usage, initial value, and legal values of all internal variables. For each variable, all primitives (except the standard setting and evaluating primitives) that affect or use the variable are indicated. (We will frequently say that an internal variable specifies a generator or has a generator as its initial value; in such situations, the actual value is a generator element denoting the appropriate generator.)

rem (remainder). When the primitive generator DIVIDE(X, Y) is executed and both X and Y are integer number elements, then this internal variable is set to the remainder of dividing X by Y.

Initial value: 0 Legal values: any list structure.

Reset by: DIVIDE.

dal (dummy association list). This variable specifies a no-argument generator which is called by ALIST whenever ALIST encounters an association list name that is a dummy element.

Initial value: FAILURE. Legal values: generator elements.

Used by: ALIST.

chr (character receiver). This variable specifies a one-argument generator which accepts output codes for object characters during character-scanning operations.

Initial value: FAILURE. Legal values: generator elements.

Used by: NORMSCN, IDENTSCN, FDINTSCN, FOINTSCN, XDINTSCN, XOINTSCN.

Reset but restored by: STANDSCN.

lsr (list receiver). This variable specifies a one-argument generator which accepts list names of sublists during character-scanning operations.

Initial value: STNDSCN1. Legal values: generator elements.

Used by: NORMSCN.

Reset but restored by: STANDSCN.

nir (negative integer receiver). This variable specifies a one-argument generator which accepts negative integer number elements during character-scanning operations.

Initial value: FAILURE. Legal values: generator elements.

Used by: FDINTSCN, FOINTSCN, XDINTSCN, XOINTSCN.

fdl (field length). This variable specifies a field length (positive integer number element) for use during the character scanning of fixed-field integers.

Initial value: 0 Legal values: integers, $0 \leq \underline{fdl} \leq 1023_{10}$.

Used by: XDINTSCN, XOINTSCN.

lzc (leading zero code). This variable specifies an output code (positive integer number element) to represent leading zeros in the character scanning of fixed-field integers.

Initial value: 0 Legal values: integers, $0 \leq \underline{lzc} \leq 373_8$.

Used by: XDINTSCN, XOINTSCN.

inr (integer receiver). This variable specifies a one-argument generator which accepts integer number elements during character-scanning operations.

Initial value: FDINTSCN. Legal values: generator elements.

Used by: STNDSCN1.

flr (floating-point receiver). This variable specifies a one-argument generator which accepts floating-point number elements during character-scanning operations.

Initial value: FAILURE. Legal values: generator elements.

Used by: STNDSCN1.

isn (internal scanner). This variable specifies a two-argument generator to be used for character scanning during the production of identifiers and numbers.

Initial value: STANDSCN. Legal values: generator elements.

Used by: IDENT, CIDENT, DECCON, OCTCON, FLOATCON.

ial (initial association list). This internal variable gives the value to be assigned to the association list names of newly created identifier elements. It is not only used by the primitive generator IDENT, but also by the syntax analyzer when controlled by a $IDENT,n/ special label.

Initial value: dummy element. Legal values: any list structure.

Used by: IDENT, syntax analyzer.

por (P-medium overflow receiver). This variable specifies a one-argument generator which is called by PUTP when an attempt is made to place an output code larger than 77_8 in the current P-medium (printed output medium) record image.

Initial value: FAILURE. Legal values: generator elements.

Used by: PUTP, STANDPMC, PUTPC.

pmr (P-medium margin). This variable specifies the first character position in the P-medium record image to the right of the field in which characters are to be placed.

Initial value: 120_{10}. Legal values: integers, $1 \le \underline{pmr} \le 136_{10}$.

Used by: PUTP, STANDPMC, PUTPC.

pin (P-medium index). This variable specifies the character position in the current P-medium record image where the next character is to be placed by PUTP.

Initial value: 1 Legal values: integers, $1 \le \underline{pin} \le \underline{pmr}$.

Used by: PUTP, STANDPMC, PUTPC.

Reset by: PUTP, STANDPMC, PUTPC, OUTP.

pmc (P-medium margin controller). This variable specifies a one-argument generator which is called by PUTP when $\underline{pin} = \underline{pmr}$.

Initial value: STANDPMC. Legal values: generator elements.

Used by: PUTP, STANDPMC, PUTPC.

pno (P-medium logical unit number). This variable gives an integer number element indicating the logical unit number (as recognized by the 3600 control system, SCOPE) of the output device used for the P-medium and also used for comments and error messages by various system routines.

Initial value: 61_{10} (Standard SCOPE output tape).

Used by: OUTP, STANDPMC, DUMPV, DUMP1, DUMPALL, various system routines which output messages. Legal values: integers, $1 \le \underline{pno} \le 80_{10}$.

cor (C-medium overflow receiver). This variable specifies a one-argument generator which is called by PUTC when an attempt is made to place an output code larger than 77_8 in the current C-medium (card output medium) record image.

Initial value: FAILURE. Legal values: generator elements.

Used by: PUTC, STANDCMC, PUTPC.

cmr (C-medium margin). This variable specifies the first character position in the C-medium record image to the right of the field in which characters are to be placed.

Initial value: 73_{10}. Legal values: integers, $1 \le \underline{cmr} \le 81_{10}$.

Used by: PUTC, STANDCMC, PUTPC.

cin (C-medium index). This variable specifies the character position in the current C-medium record image where the next character is to be placed by PUTC.

Initial value: 1 Legal values: integers, $1 \le \underline{cin} \le \underline{cmr}$.

Used by: PUTC, STANDCMC, PUTPC.

Reset by: PUTC, STANDCMC, PUTPC, OUTC.

cmc (C-medium margin controller). This variable specifies a one-argument generator which is called by PUTC when $\underline{cin} = \underline{cmr}$.

Initial value: STANDCMC. Legal values: generator elements.

Used by: PUTC, STANDCMC, PUTPC.

cno (C-medium logical unit number). This variable gives an integer number element indicating the logical unit number (as recognized by SCOPE) of the output device used for the C-medium.

Initial value: 62_{10} (Standard SCOPE punched output tape).

Used by: OUTC, STANDCMC. Legal values: integers, $1 \le \underline{cno} \le 80_{10}$.

bmr (B-medium margin). This variable specifies the first bit position in the B-medium (Binary card output medium) record image to the right of the field in which binary numbers are to be placed.

Initial value: 1 Legal values: integers, $1 \le \underline{bmr} \le 961_{10}$.

Used by: PUTB.

bwl (B-medium word length). This variable specifies the number of bits to be used for the numbers being written in the current B-medium record image.

Initial value: 1 Legal values: integers, $1 \le \underline{bwl} \le 960_{10}$.

Used by: PUTB.

bin (B-medium index). This variable specifies the bit position in the current B-medium record image of the high-order bit of the next number to be written in this image by PUTB.

Initial value: 1 Legal values: integers, $1 \le$ bin $\le$ bmr.

Used by: PUTB.

Reset by: PUTB, OUTB.

bmc (B-medium margin controller). This variable specifies a one-argument generator to be called by PUTB when bin + bwl > bmr, i.e., when the number to be written in the B-medium record image runs beyond the right margin.

Initial value: FAILURE. Legal values: generator elements.

Used by: PUTB.

bno (B-medium logical unit number). This variable gives an integer number element indicating the logical unit number (as recognized by SCOPE) of the output device used for the B-medium.

Initial value: 62_{10} (Standard SCOPE punched output tape).

Used by: OUTB. Legal values: integers, $1 \le$ bno $\le 80_{10}$.

ino (input medium logical unit number). This internal variable is not used by a primitive generator, but is referenced by the syntax analyzer (via the input editor). It gives an integer number element indicating the logical unit number of the input device from which the analyzer reads character strings to be parsed. When the value of ino is changed, the next record read by the analyzer will be obtained from a new input device, but any characters remaining in the current input record will be processed before the next record is read.

Initial value: 60_{10} (Standard SCOPE input tape).

Used by: Input editor. Legal values: integers, $1 \le$ ino $\le 80_{10}$.

gol (goal). This internal variable is not used by a primitive generator, but is evaluated whenever the syntax analyzer is called on the main level of the COGENT program. If the value of gol is i > 0, then the syntax analyzer is called to search for the goal indicated by the ith goal specifier in the primary syntax description; but if the value of gol is zero, then the entire program terminates.

Initial value: 1 Legal values: integers, $0 \le$ gol $\le 1023_{10}$.

Used by: Syntax analyzer.

2. Standard Primitives for the Internal Variables

The following are the standard primitive generators for setting internal variables.

SETIVREM(X) sets rem.
SETIVDAL(X) sets dal.
SETIVCHR(X) sets chr.
SETIVLSR(X) sets lsr.
SETIVNIR(X) sets nir.
SETIVFDL(X) sets fdl.
SETIVLZC(X) sets lzc.
SETIVINR(X) sets inr.
SETIVFLR(X) sets flr.
SETIVISN(X) sets isn.
SETIVIAL(X) sets ial.
SETIVPOR(X) sets por.
SETIVPMR(X) sets pmr.
SETIVPIN(X) sets pin.
SETIVPMC(X) sets pmc.
SETIVPNO(X) sets pno.
SETIVCOR(X) sets cor.
SETIVCMR(X) sets cmr.
SETIVCIN(X) sets cin.
SETIVCMC(X) sets cmc.
SETIVCNO(X) sets cno.
SETIVBMR(X) sets bmr.
SETIVBWL(X) sets bwl.
SETIVBIN(X) sets bin.
SETIVBMC(X) sets bmc.
SETIVBNO(X) sets bno.
SETIVINO(X) sets ino.
SETIVGOL(X) sets gol.

Each of these generators accepts a single argument X and sets the appropriate internal variable to the value of X. The dummy element is returned as the result of these generators. These generators must not be used to assign illegal values to internal variables.

The following are the standard primitives for evaluating internal variables:

IVREM( ) evaluates rem.

IVDAL( ) evaluates dal.

IVCHR( ) evaluates chr.

IVLSR( ) evaluates lsr.

IVNIR( ) evaluates nir.

IVFDL( ) evaluates fdl.

IVLZC( ) evaluates lzc.

IVINR( ) evaluates inr.

IVFLR( ) evaluates flr.

IVISN( ) evaluates isn.

IVIAL( ) evaluates ial.

IVPOR( ) evaluates por.

IVPMR( ) evaluates pmr.

IVPIN( ) evaluates pin.

IVPMC( ) evaluates pmc.

IVPNO( ) evaluates pno.

IVCOR( ) evaluates cor.

IVCMR( ) evaluates cmr.

IVCIN( ) evaluates cin.

IVCMC( ) evaluates cmc.

IVCNO( ) evaluates cno.

IVBMR( ) evaluates bmr.

IVBWL( ) evaluates bwl.

IVBIN( ) evaluates bin.

IVBMC( ) evaluates bmc.

IVBNO( ) evaluates bno.

IVINO( ) evaluates ino.

IVGOL( ) evaluates gol.

Each of these generators has no input arguments, and returns as its result the current value of the appropriate internal variable.

B. Testing Primitives

Each of these primitive generators performs a test upon its input arguments and fails unless the test is satisfied. If the test is satisfied, the primitive returns the dummy element as its result. All the testing primitives except LARGER accept any list structures as input arguments.

NORMTEST(X). Fails unless X is a normal list element.

IDENTEST(X). Fails unless X is an identifier element.

NUMTEST(X). Fails unless X is a number element.

PARATEST(X). Fails unless X is a parameter element.

GENTEST(X). Fails unless X is a generator element.

DUMTEST(X). Fails unless X is the dummy element.

FIXTEST(X). Fails unless X is an integer number element.

FLOATEST(X). Fails unless X is a floating-point number element.

POSTEST(X). Fails unless X is a positive (integer or floating-point) number element. An integer or floating-point zero is always positive.

ZEROTEST(X). Fails unless X is an integer or floating-point number element with the value zero.

NEGTEST(X). Fails unless X is a negative (integer or floating-point) number element.

EQLIT(X, Y). Fails unless X and Y are the same list name, i.e., unless they name identical (not merely similar) list structures.

LARGER(X, Y). X and Y must both be number elements. LARGER fails unless the value of X is larger than the value of Y. If both X and Y are integers, then they are compared as integers. If either X or Y is a floating-point number, the other argument is converted to floating-point if necessary, and the comparison is performed in floating-point. Integer-to-floating-point conversion is discussed in Section E below on arithmetic primitives.

C. List-handling Primitives

PRODCODE(X). X must be the name of a normal list element. PRODCODE returns an integer number element giving the production code number contained in this normal element.

SIZE(X). X must be the name of a normal element. SIZE returns an integer number element giving the number of name-components (components other than the production code number) in this normal element. X must not have more than 1023_{10} name-components.

SUBLIST(X,N). X must be the name of a normal element with at least one name-component, and N must be an integer element such that $1 \leq N \leq 1023_{10}$ and such that N is smaller than or equal to the number of name-components in X. SUBLIST returns the name which is the Nth name-component in the element named by X.

SUBLIST1(X). Equivalent to SUBLIST(X,1).

SUBLIST2(X). Equivalent to SUBLIST(X,2).

REPLACE(X,Y,N). X must be the name of a normal element with at least one name-component, and N must be an integer element such that $1 \leq N \leq 1023_{10}$ and such that N is smaller than or equal to the number of name-components in X. Y may be any list structure. REPLACE alters the normal element named by X by replacing the contents of its Nth name-component by the value of Y. The result of REPLACE is the dummy element.

REPLAC1(X,Y). Equivalent to REPLACE(X,Y,1).

REPLAC2(X,Y). Equivalent to REPLACE(X,Y,2).

The REPLACE primitives are unique in that they alter an existing normal element, as opposed to creating new elements. Three consequences of this fact should be kept in mind when using these primitives:

1. The normal element altered by REPLACE may be a member of several list structures, each of which may be the value of several variables. Thus REPLACE may effectively alter the values of variables that are not its arguments by altering the structures named by these variables.

2. REPLACE may be used to create cyclic list structures, i.e., structures that are sublists of themselves. Cyclic structures are permissible in COGENT but must be used with caution. In particular, the processes of list synthesis and analysis and of character scanning may never terminate when applied to cyclic structures. On the other hand, the storage-recovery algorithm is capable of handling such structures, and the primitive dumping generators may be used to print them out.

3. The REPLACE primitives must never be used to alter list structures that are the values of constants. The violation of this rule will have unpredictable consequences.

PINDEX(X). X must be a parameter element. PINDEX returns an integer number element giving the value of the index of X.

D. Dummy Primitives

The action of each of these generators is trivial. Except for NOP, these generators may have any number of arguments (including zero), and their effect is independent of their arguments.

FAILURE. Always fails.

NORESULT. Always returns the dummy element as its result.

ZERO. Always returns an integer zero as its result.

NOP. Must have one or more input arguments. It always returns the value of its last (rightmost) input argument.

E. Arithmetic Primitives

The input arguments of each of these generators must be number elements, and the result will also be a number element. Except for POWER2 and POWER10, if all of the input arguments are integers, then the generator will perform an integer arithmetic operation and return an integer result, but if any argument is a floating-point number, then any other argument that is an integer will be converted to floating-point, a floating-point operation will be performed, and a floating-point result will be returned.

ADD(X,Y). Returns X + Y.

SUB(X,Y). Returns X - Y.

INCREASE(X). Returns X + 1.

DECREASE(X). Returns X - 1.

NEG(X). Returns -X.

ABS(X). Returns $|X|$.

MULT(X,Y). Returns X x Y.

DIVIDE(X,Y). Returns X/Y. If X and Y are integers, the internal variable rem will be set to an integer number element giving the remainder of dividing X by Y. The remainder will have the same sign as X (except that a remainder of zero will always be positive) and will satisfy $0 \leq |rem| < |Y|$ and $X = (X/Y) \times Y + rem$. If Y is an integer or floating-point zero, DIVIDE will fail.

POWER2(X,N). X may be an arbitrary number element, but N must be an integer such that $-1023 \leq N \leq 1023$. POWER2 will return $X \times 2^N$, with the

same mode as X. If X is an integer and N is negative, the sign of the result will be the sign of X (except that zero will always be positive), and the magnitude of the result will be the largest integer smaller than or equal to $|X \times 2^N|$.

POWER10(X,N). X must be a floating-point number element, and N must be an integer such that $-1023 \leq N \leq 1023$. POWER10 will return a floating-point number element representing $X \times 10^N$.

The following remarks pertain to all arithmetic primitives as well as to the integer-to-floating-point conversions that may be performed by other primitives such as LARGER or FLOATCON:

1. If exponent underflow occurs while a primitive is executing a floating-point arithmetic operation, the result of the operation is taken to be a floating-point zero.

2. If exponent overflow occurs while a primitive is executing a floating-point arithmetic operation or an integer-to-floating-point conversion, the primitive will fail.

3. If, during an integer-to-floating-point conversion, the integer is too large to have an exact floating-point representation, it will be rounded.

4. There is no negative zero in COGENT; i.e., both the integer zero and the floating-point zero are unique and positive. (However, the floating-point zero is distinct from the integer zero.)

F. Identifier-handling Primitives

SETA(I,X). I must be an identifier element. The association list name of I is set to the value of X. The dummy element is returned.

ALIST(I). I must be an identifier element. The result of ALIST is the association list name contained in I, unless this name is the dummy element. If it is the dummy element, then the no-argument generator specified by the internal variable dal is called. If this generator fails, then ALIST fails; otherwise the result of this generator is returned as the result of ALIST. Since the initial value of dal is FAILURE, ALIST will fail upon encountering a dummy association list, unless the value of dal is reset.

TABLENO(I). I must be an identifier element. TABLENO returns an integer number element giving the table-number component of I. Normally this number indicates the identifier table containing the identifier, but a table number of zero indicates an identifier that is not in any table.

FIRSTID(N). N must be a positive integer number element denoting an identifier table. FIRSTID returns the first identifier element in this table. If identifier table N is empty, or if N = 0, FIRSTID fails.

NEXTID(I). I must be an identifier element. NEXTID returns the next identifier element in the same table as I. If I is the last element in the table, or if I has a table number of zero, then NEXTID fails.

FIRSTID and NEXTID allow the programming of searches through an identifier table. For example, the following statements will search identifier table one for an identifier element with an integer zero as its association list:

```
   +2  UNLESS X = FIRSTID(1).

1/+3  IF ALIST(X) =/ 0.

   +1  IF X = NEXTID(X).

2/ ....
```

If the desired identifier element is found, control will go to statement 3, with the desired element as the value of X. Otherwise control will go to statement 2.

It should be emphasized that the ordering of an identifier table cannot be controlled by the programmer. In particular, if an identifier element in some table is created or erased while the same table is being searched, it is unpredictable whether the creation or erasure will occur before or after the current position of the search.

ERASID(I).

ERASIT(N). These primitives are used to erase identifier elements, i.e., to delete them from identifier tables. The process of erasing an identifier element is defined as follows:

1. If the element has a table number of zero, no action is taken.

2. If the element is a constant identifier element, i.e., if it appears within the list structure denoted by any constant in the COGENT program, no action is taken. (Constant identifier elements cannot be deleted from tables.)

3. If the element has a nonzero table number and is not a constant identifier element, then it is removed from the identifier table that contains it, and its table number component is set to zero.

Note that when an element is erased it does not vanish, but simply becomes a tableless identifier.

The primitive ERASID(I) accepts an identifier element I and erases it, as defined above. The dummy element is returned.

The primitive ERASIT(N) accepts a positive number element N denoting an identifier table and erases all identifier elements in this table. The dummy element is returned. If N = 0, ERASIT has no effect; but if N $\neq$ 0, all identifiers in the table except constant identifiers are deleted.

G. Character-scanning Primitives

In general, character scanning is a process which reduces a list structure to a string of object characters, as represented by a sequence of output codes. Character scanning serves three purposes in COGENT:

1. To produce a string of object characters to be written on the output media.

2. In the primitive generators IDENT and CIDENT, to produce a character string for an identifier element.

3. In the primitive generators DECCON, OCTCON, and FLOATCON, to produce a character string to be converted into a number element.

At first, it would appear that the reduction of an arbitrary list structure to a character string is completely defined by the total set of productions. But an arbitrary list structure may contain nonnormal elements such as number elements, whose conversion into characters is not defined by the productions. Thus the process of character scanning requires the establishment of certain conventions regarding the reduction of nonnormal elements.

These conventions may be specified by the programmer in several ways. Most simply, he may use the character-scanning primitives STANDSCN or STNDSCN1, whose operation assumes a standard set of conventions. If these standard conventions are undesirable, they may be altered by resetting the internal variables inr and flr which determine the treatment of number elements by STANDSCN and STNDSCN1. Finally, if these primitives cannot be altered to perform the desired scan, the programmer may replace them by his own scanning generator, written in terms of more basic scanning primitives. If such a special scanning generator is made the value of the internal variable isn, it will be used for character scanning by the primitive generators that create identifiers and numbers.

In general, as a primitive scanning generator produces a sequence of output codes, it communicates each code by calling another generator called the character receiver, and giving the output code to this generator as a single integer argument. Thus a scanning primitive must always be informed of the character receiver it is to use. Some of the primitives accept a generator element indicating the character receiver as one of their input arguments, but most of the primitives obtain their character receiver as the value of the internal variable chr.

1. Basic Character-scanning Primitives

We first discuss the basic character-scanning primitives. Each of these generators processes a single list element and sends the resulting character codes to a character receiver. The generator NORMSCN, which processes a normal element, also calls a list receiver to process sublists of the normal element.

NORMSCN(X). X must be a normal list element. NORMSCN examines the production code number of this element and obtains from the character-scanning table the production denoted by this code number. It then sequences through the construction string of this production, from left to right. When an object character representative is encountered in the construction string, the generator that is the value of the internal variable chr is called and given as its single argument the output code for the character representative (as an integer number element). When the ith phrase class name is encountered in the construction string, the generator that is the value of lsr (list receiver) is called and given the ith sublist of X as its single argument.

If any of the generators called by NORMSCN fails, then NORMSCN fails without further execution. Otherwise NORMSCN returns the result of the last generator it calls or, if it does not call any generator, the dummy element.

The character-scanning table used by NORMSCN is compiled from the total set of productions and gives an encoding of the construction string of each production. It is not affected by labels, special labels, or forcing markers.

IDENTSCN(X). X must be an identifier element. IDENTSCN sequences through the character string component in this identifier element, from left to right. For each character in the string, the generator that is the value of the internal variable chr is called and given as its single argument the output code for the character (as an integer number element).

The result of IDENTSCN is the dummy element. If any call of the character receiver fails, then IDENTSCN fails without further execution.

FDINTSCN(X). (free field decimal integer scan). X must be an integer number element. If X is negative, then the generator that is the value of

the internal variable nir (negative integer receiver) is called and given X as its single argument. When this generator fails or returns a result, FDINTSCN fails or returns the same result without further execution. (The initial value of nir is the primitive generator FAILURE.)

If X is positive, FDINTSCN constructs a digit string giving a decimal representation of the integer X. The length of this string will depend on X and will be such that the leftmost digit is nonzero unless X is zero, in which case the string will be the single digit zero. FDINTSCN then sequences through this string from left to right. For each digit, the generator that is the value of chr is called and given the output code for the digit (as an integer number element). The output codes are taken as the codes specified by the character definitions for the object character representatives 0, 1, ..., 9.

If X is positive and the character receiver does not fail, then FDINTSCN will return the dummy element as its result. If the character receiver fails, FDINTSCN fails without further execution.

FOINTSCN(X). (free field octal integer scan). This generator is identical to FDINTSCN, except that an octal, rather than decimal, representation is produced.

XDINTSCN(X). (fixed field decimal integer scan). X must be an integer number element. If X is negative, then the generator that is the value of the internal variable nir is called and given X as its single argument. When this generator fails or returns a result, XDINTSCN fails or returns the same result without further execution. (The initial value of nir is the primitive generator FAILURE.)

If X is positive, XDINTSCN constructs a digit string giving a decimal representation of the integer X in which exactly n digits appear, where n is the value of the internal variable fdl (field length). (The initial value of fdl is zero, so that fdl should be reset to a positive integer number element before calling XDINTSCN.) If X is too large to be represented by n digits, i.e., if X is larger than $10^n - 1$ (or if fdl = 0), then XDINTSCN fails without further execution. Otherwise, XDINTSCN sequences through the digit string from left to right. For each digit, the generator that is the value of chr is called and given the appropriate output code (as an integer number element).

The output codes are taken as the codes specified by the character definitions for the object character representatives 0, 1, ..., 9, except that all leading zeros are replaced by the output code that is the value of the internal variable lzc (leading zero code), whose initial value is zero. A zero digit is defined to be a leading zero if it appears to the left of the leftmost nonzero digit in the string. (When X is zero, all digits except the rightmost in the string are leading.)

If the character receiver fails, XDINTSCN fails without further execution. If XDINTSCN does not fail, it returns the dummy element.

XOINTSCN(X). (fixed field octal integer scan). This generator is identical to XDINTSCN, except that an octal, rather than decimal, representation is produced. The largest representable integer is $8^n - 1$.

DFLTSCN1(X,N). (decimal floating-point number scan). X must be a positive, nonzero, floating-point number element. N must be an integer number element such that $1 \leq N \leq 28$. DFLTSCN1 produces a representation of the value of X in the form:

$$X \cong f \times 10^e,$$

where $-308 \leq e \leq 308$, $1/10 \leq f < 1$, and f is an N-digit decimal fraction:

$$f = .f_1 f_2 \ldots f_N$$

Rounding is used in producing f.

The result of DFLTSCN1 is an integer number element giving the value of e. The digits $f_1 \ldots f_N$ are not passed on to a character receiver, but are placed in a storage buffer region which may be accessed by the generator FLTSCN2.

OFLTSCN1(X,N). (octal floating-point number scan). This generator is similar to DFLTSCN1 except that an octal representation of X is produced. This representation has the form

$$X \cong f \times 2^e,$$

where $-1023_{10} \leq e \leq 1023_{10}$, $1/2 \leq f < 1$, and f is an N-digit octal fraction. Rounding is used in producing f, unless N = 28, in which case the representation is exact.

FLTSCN2 (PC, CR, PR). This generator produces the digits left in the storage buffer region by the last execution of DFLTSCN1 or OFLTSCN1. PC must be an integer number element such that $0 \leq PC \leq N$, where N is the number of digits left in the buffer (i.e., the second argument given to DFLTSCN1 or OFLTSCN1 when it was previously called). CR must be a character-receiving generator, and PR must be a generator with no arguments.

FLTSCN2 first calls the character receiver CR the number of times specified by the integer PC, supplying successively as input arguments the digits $f_1 \ldots f_{PC}$ (as represented by the output codes specified by the character definitions for the object character representatives 0, 1, ..., 9).

Next, FLTSCN2 calls the no-argument generator PR. Then FLTSCN2 again calls CR repeatedly, supplying the digits $f_{PC+1}, \ldots, f_N$. If the generator CR or PR fails, then FLTSCN2 fails without further execution. Otherwise, FLTSCN2 returns the dummy element.

Note that the character receiver used by FLTSCN2 is specified by the input argument CR, rather than by an internal variable. The purpose of calling the generator indicated by PR is to allow a decimal point to be inserted in the character string after the PCth digit.

FLTSCN2 cannot be used recursively; i.e., the generators specified by CR and PR must not call FLTSCN2 (or DFLTSCN1 or OFLTSCN1) directly or indirectly.

N3600SCN(X, CR, IR). (3600 number scan). This generator is intended for use when the output of a COGENT program is itself a 3600 program using the same internal number representation as the COGENT program. X must be a number element, CR must be a character-receiving generator, and IR must be a generator with no input arguments.

If X is represented by a literal list name, i.e., if X is an integer between -1023_{10} and 1023_{10} inclusive, then N3600SCN fails. Otherwise, N3600SCN sequences through the machine words of the element X, as it is actually represented in storage. For each word,

a. The generator that is the value of IR is called.

b. The generator that is the value of CR is called 16_{10} times, and given a sequence of digit output codes which form a positive octal integer equal to the actual contents of the machine word.

If CR or IR fails, then N3600SCN fails without further execution. Otherwise, N3600SCN returns the dummy element.

2. Higher-level Character-scanning Primitives

We now consider the standard character-scanning primitives STANDSCN and STNDSCN1, which reduce an entire list structure to a character string, using standard conventions for the handling of nonnormal elements:

STNDSCN1(X). This primitive is exactly equivalent to a generator with the following definition:

```
$GENERATOR STNDSCN1((X)

+1 UNLESS NORMTEST(X).  $RETURN(NORMSCN(X)).

1/ +2 UNLESS IDENTEST(X).  $RETURN(IDENTSCN(X)).

2/ +3 UNLESS FIXTEST(X).  $RETURN(IVINR()(X)).

3/ FLOATEST(X).  $RETURN(IVFLR()(X)). ).
```


The purpose of STNDSCN1 is to convert an entire list structure X into a character string and to pass the output code for each character in this string onto the character-receiving generator specified by the internal variable chr. In normal usage, it is assumed that whenever STNDSCN1 is called, the value of the internal variable lsr, which specifies a list receiver for the primitive NORMSCN, will be a generator element denoting STNDSCN1 itself (such an element is the initial value of lsr).

Assuming that STNDSCN1 is the value of lsr, then STNDSCN1 will call itself recursively (through NORMSCN) to process sublists of its arguments. In this case, the sequence of output codes X' produced by STNDSCN1 when applied to a list structure X may be defined recursively as follows:

a. If X is headed by a normal list element, and S is the construction string of the production denoted by the code number in this element, then X' is obtained from S by replacing each object character representative by its output code, and replacing each phrase class name by the sequence of output codes produced from the corresponding sublist of X.

b. If X is an identifier element, then X' is the sequence of output codes in the character-string component of X.

c. If X is an integer number element, X' is the sequence of output codes produced by calling the generator that is the value of the internal variable inr (integer receiver) and giving this generator X as its single argument. If this generator fails, then STNDSCN1 fails.

The initial value of inr is the primitive generator FDINTSCN, and the initial value of nir (used by FDINTSCN) is the primitive generator FAILURE. Thus if these internal variables are not altered, positive integer elements in the list structure given to STNDSCN1 will be converted into free-field decimal digit strings, while the occurrence of any negative integer element will cause STNDSCN1 to fail.

d. If X is a floating-point number element, X' is the sequence of output codes produced by calling the generator that is the value of the internal variable flr (floating-point receiver), and giving this generator X as its single argument. If this generator fails, then STNDSCN1 fails.

The initial value of flr is the primitive generator FAILURE. Thus if this internal variable is not altered, the occurrence of any floating-point number element in the list structure given to STNDSCN1 will cause STNDSCN1 to fail.

e. If a parameter, generator, or dummy element occurs in the list structure given to STNDSCN1, then STNDSCN1 will fail.

Essentially, the programmer may alter the response of STNDSCN1 to number elements by changing the values of inr, flr, or other internal variables. In particular, he may define his own generators for scanning numbers and set the appropriate internal variables to generator elements which refer to these generators.

STANDSCN(X, CR). This primitive is exactly equivalent to a generator with the following definition:

```
$GENERATOR STANDSCN((X, CR) $LOCAL CHRSAVE, LSRSAVE.

CHRSAVE = IVCHR( ). SETIVCHR(CR).

LSRSAVE = IVLSR( ). SETIVLSR(STNDSCN1).

+1 UNLESS STNDSCN1(X).

SETIVCHR(CHRSAVE). SETIVLSR(LSRSAVE). $RETURN.

1/ SETIVCHR(CHRSAVE). SETIVLSR(LSRSAVE). $FAILURE. ).
```

The action of STANDSCN is identical to STNDSCN1, except that the character receiver is specified by a second input variable instead of the internal variable chr, and the value of the internal variable lsr does not need to be set up before calling STANDSCN. The response of STANDSCN to number elements may be altered by resetting internal variables in the same manner as with STNDSCN1.

It should be noted that STANDSCN alters the values of the internal variables chr and lsr, but always restores these variables before exiting, even if failure occurs.

A generator element denoting STANDSCN is the initial value of the internal variable isn (internal scanner), which specifies the scanning generator to be used by the primitives IDENT, CIDENT, DECCON, OCTCON, and FLOATCON while converting list structures into identifiers or numbers. The programmer may change the value of isn to denote a scanning generator which he has written himself, in order to change the scanning used to produce identifiers and numbers. Such a scanning generator must accept two arguments (as does STANDSCN): the list to be scanned, and a generator element denoting the character receiver to be used.

3. Examples of Character Scanning

We give two examples to illustrate the use of the primitive character-scanning generators. As the first example, we assume that it is desired to extend the standard scanning conventions to allow the scanning of negative and floating-point numbers. These quantities are to be reduced to character strings as follows:

a. A positive nonzero floating-point number X is to be reduced to a string of the form

f.ffffff ± eee

where f and e are decimal numbers such that $X = f \times 10^e$ and $1 \leq f < 10$. A zero floating-point number is to be reduced to the string "0.0".

b. A negative integer or floating-point number X is to be reduced to a string of the form

(- ....)

where the dots represent the reduction of the magnitude of X.

The following generator definition defines a generator to scan floating-point numbers. This generator accepts a single argument, which must be a floating-point number element, and passes the resulting characters to the character receiver specified by the internal variable chr. (We assume that 33_8, 74_8, 40_8, 34_8, 0, and 20_8 are the output codes for the characters ".", "(", "-", ")", "0", and "+", respectively.)

```
$GENERATOR FLOATSCAN ((X) $GEN DP. $LOCAL CR,E.

    $GENERATOR DP (( ) CR(33B). ).

    CR = IVCHR().

    +1 UNLESS NEGTEST(X). CR(74B). CR(40B).

    FLOATSCAN(NEG(X)). CR(34B). $RETURN.

    1/ +2 UNLESS ZEROTEST(X).

    CR(0). CR(33B). CR(0). $RETURN.

    2/ E = DECREASE(DFLTSCN1(X, 8)). FLTSCN2(1, CR, DP).
```

+3 UNLESS NEGTEST(E).

CR(40B). XDINTSCN(NEG(E)). $RETURN.

3/ CR(20B). XDINTSCN(E).).

The following generator definition defines a generator to scan negative integers. It also accepts a single argument, which must be a negative integer number element, and uses the character receiver specified by the internal variable chr.

$GENERATOR NEGINTSCAN ((X) $LOCAL CR.

CR = IVCHR().

CR(74B). CR(40B). FDINTSCN(NEG(X)). CR(34B).).

Now FLOATSCAN must be made the value of the internal variable flr so that it will be called by STNDSCN1 when a floating-point number element is encountered. Similarly NEGINTSCAN must be made the value of the internal variable nir so that it will be called by FDINTSCN (which is in turn called by STNDSCN1) when a negative integer is encountered. Also, the internal variable fdl must be set to 3 to control the operation of XDINTSCN while processing the e-field of a floating-point number. Thus the three statements

SETIVFLR(FLOATSCAN). SETIVNIR(NEGINTSCAN).
SETIVFDL(3).

would be included in the COGENT program at a point where their execution would precede all character scanning.

As a second example, we assume that the standard scanning conventions for number elements are to be used, but that it is desired to cause all identifier elements to be reduced to character strings which are preceded and followed by blanks. This cannot be accomplished by using STANDSCN or STNDSCN1, so that a special scanning generator must be defined to replace STANDSCN (we assume that 60_8 is the output code for a blank):

$GENERATOR SPECSCAN ((X, CR)

$GEN SS1. $LOCAL CHRSAVE, LSRSAVE.

$GENERATOR SS1 ((X)

+1 UNLESS NORMTEST(X). NORMSCN(X). $RETURN.

1/ +2 UNLESS IDENTEST(X). IVCHR()(60B).

IDENTSCN(X). IVCHR()(60B). $RETURN.

2/ +3 UNLESS FIXTEST(X). IVINR()(X). $RETURN.

3/ FLOATEST(X). IVFLR()(X).).

CHRSAVE = IVCHR(). LSRSAVE = IVLSR().

SETIVCHR(CR). SETIVLSR(SS1).

+1 UNLESS SS1(X).

SETIVCHR(CHRSAVE). SETIVLSR(LSRSAVE). $RETURN.

1/ SETIVCHR(CHRSAVE). SETIVLSR(LSRSAVE).

$FAILURE.).

This generator is written to be similar to STANDSCN except for the treatment of identifiers.

To cause identifiers to be delimited by blanks in output strings, the defined generator SPECSCAN would simply be used in place of the primitive STANDSCN in the generator definition for the appropriate output generator. However, it might also be desirable for identifier strings to be delimited by blanks when they are substrings of larger identifiers, i.e., when identifier elements already appear in the list structure passed to IDENT to be converted into an identifier element. In this case, the statement

SETIVISN(SPECSCAN).

would be placed in the program so that it would be executed before IDENT. This statement would then cause IDENT (and also CIDENT, DECCON, OCTCON, and FLOATCON) to use SPECSCAN instead of STANDSCN to perform character scanning.

H. Identifier-creating Primitives

IDENT(X, N). X may be an arbitrary list structure, and N must be a positive integer number element denoting an identifier table. IDENT converts X into a character string and produces an identifier element in table N containing this character string. The following steps are performed:

1. The generator specified by the internal variable isn is called and given X as its first argument. Its second argument is a special

character-receiving generator (which is internal to IDENT and not available as a separate primitive) which accumulates the sequence of output codes that it receives in a special character buffer area. If the generator specified by isn fails, then IDENT fails without further execution. Since the initial value of isn is STANDSCN, the standard character-scanning conventions will be used, unless isn or the internal variables affecting STANDSCN or its subgenerators are reset.

The generator specified by isn must not call IDENT, CIDENT, DECCON, OCTCON, or FLOATCON directly or indirectly, since these generators all use the same character buffer area. The size of this buffer area is limited, so that the sum of the total number of output codes placed in the buffer plus the number of output codes that are larger than or equal to 75_8 must not exceed 1016_{10}. If this restriction is violated, an error message will be printed and the entire COGENT program will terminate.

2. If $N \neq 0$, the identifier table denoted by N is searched for an identifier element containing a character string that matches the string in the character buffer area. If such an identifier element is found, IDENT returns with this element as its result.

3. If N = 0, or if the search in step 2 fails, an identifier element is created containing the string in the buffer area and the table number N. The association list is set to the value of the internal variable ial (initial association list), whose initial value is the dummy element. If $N \neq 0$, the new identifier element is placed in table N and returned as the result of IDENT. If N = 0, the element is returned but not placed in any table.

The identifier construction performed by the special label $IDENT,n/ follows the above steps, except that in step 1 no character scan is performed, and the character buffer is filled directly by output codes from the syntax analyzer. If the buffer size is exceeded, a terminating error will occur.

CIDENT(X, N). This generator is similar to IDENT, except that if step 3 is reached, CIDENT fails without creating a new identifier element. Thus CIDENT will convert X into an identifier element in table N only if such an element already exists in the table.

I. Number-creating Primitives

DECCON(X). X may be an arbitrary list structure. If X is a number element, it is converted into an integer number element which is returned as the result of DECCON. Otherwise, DECCON converts X into a character string and produces a positive integer element giving the value of this string interpreted as a decimal number. The following steps are performed:

1. If X is an integer number element, DECCON returns immediately with X as its result. If X is a floating-point number element, an integer number element is created whose sign is the same as X (except that zero is always positive) and whose magnitude is the largest integer smaller or equal to the magnitude of X, and DECCON returns with this integer element as its result.

2. The generator specified by the internal variable isn is called and given X as its first argument. Its second argument is a special character-receiving generator (which is internal to DECCON and not available as a separate primitive) which accumulates the sequence of output codes it receives in a special buffer area. If the generator specified by isn fails, DECCON fails without further execution. Since the initial value of isn is STANDSCN, the standard character-scanning conventions will be used, unless isn or the internal variables affecting STANDSCN or its subgenerators are reset.

The generator specified by isn must not call IDENT, CIDENT, DECCON, OCTCON, or FLOATCON directly or indirectly, since these generators all use the same character buffer area. The size of this buffer area is limited, so that the sum of the total number of output codes placed in the buffer plus the number of output codes that are larger than or equal to 75_8 must not exceed 1024_{10}. If this restriction is violated, an error message will be printed, and the entire COGENT program will terminate.

3. DECCON sequences through the output codes in the character buffer. This sequence of codes is interpreted as a positive decimal integer, and the corresponding integer number element is returned as the result of DECCON. Output codes are interpreted as digits according to the character definitions for the object character representatives 0, 1, ..., 9. Output codes that do not represent digits are ignored.

OCTCON(X). This generator is similar to DECCON, except that octal rather than decimal conversion is used; i.e., in step 3, the sequence of output codes is interpreted as a positive octal integer.

FLOATCON(X). X may be an arbitrary list structure. If X is a number element, it is converted into a floating-point number element which is returned as the result of FLOATCON. Otherwise, FLOATCON converts X into a character string and produces a positive floating-point number element giving the value of this string interpreted as a decimal number. The following steps are performed:

1. If X is a floating-point number element, FLOATCON returns immediately with X as its result. If X is an integer number element, it is converted into a floating-point number element, and FLOATCON returns with this floating-point number as its result.

2. Same as step 2 for DECCON.

3. FLOATCON converts the sequence of output codes in the buffer area into a decimal integer in the same manner as DECCON. This integer is then converted into a floating-point number. Finally, if any output code that does not represent a digit appears in the buffer, this floating-point number is divided by 10^n, where n is the number of digits following the last such nondigit. (Thus the last nondigit is interpreted as a decimal point.) FLOATCON then exits with the resulting floating-point number as its result.

Integer-to-floating-point conversion was discussed in Section E on arithmetic primitives. If such a conversion produces an exponent overflow, FLOATCON will fail. On the other hand, if division by 10^n produces an exponent underflow, FLOATCON will return a floating-point zero.

The number construction performed by the special labels $DEC/, $OCT/, and $FLOAT/ causes the buffer area to be filled with output codes directly from the syntax analyzer. Then an integer or floating-point number element is created as in step 3 for DECCON, OCTCON, or FLOATCON, respectively. If the buffer size is exceeded, or if a floating-point overflow (with $FLOAT/) occurs, a terminating error will occur.

J. Output Primitives

The output facilities in COGENT are inevitably more machine-dependent than the rest of the language. To minimize this dependency, output is largely described in terms of its ultimate forms, such as punched cards or print lines, without reference to intermediate media such as magnetic tape or disks.

Output is described in terms of <u>output media</u>, each of which is characterized by a particular ultimate form, and is handled by a separate set of primitive generators. At all times during the execution of a COGENT program, there is associated with each output medium a <u>current record image</u>. Certain output primitives cause output codes or other numbers to be written in the current record image; other generators cause the image to be transmitted to an output device and then to be reinitialized.

Three output media are defined in the initial COGENT system:

1. The P (Printed Output) medium. The current record image for P contains 135_{10} character positions, numbered 1 to 135 (from left to right), plus a carriage control character. Each position holds a character code between 0 and 77_8, and the relation between these codes and the corresponding print characters is established by the internal BCD representation of the 3600. When the current record image is initialized, all character positions and the carriage control character are set to blanks (60_8). (A blank carriage control character causes a print line to be printed immediately after the preceding line.)

When a print record is to be sent to a printer with less than 135 character positions, the extra right-hand character positions must contain blanks.

2. The C (Card Output) medium. The current record image for C contains 80_{10} character positions, numbered 1 to 80 and corresponding to the 80 columns of a punched card, from left to right. Each position holds a character code between 0 and 77_8, and the relation between this code and the corresponding column punches is established by the internal BCD representation of the 3600. When the current record image is initialized, all character positions are set to blanks (60_8).

3. The B (Binary Card Output) medium. The current record image for B contains 960_{10} bit positions, numbered 1 to 960 and corresponding to the individual hole positions on a punched card, as shown in Figure 9. The bit 1 indicates that a hole is to be punched; the bit 0 indicates that it is not to be punched. When the current record image is initialized, all bits are set to zero except bits 10_{10} and 12_{10}, which are set to one to cause the standard 7-9 punch (indicating a binary card) to occur in column one of the card.

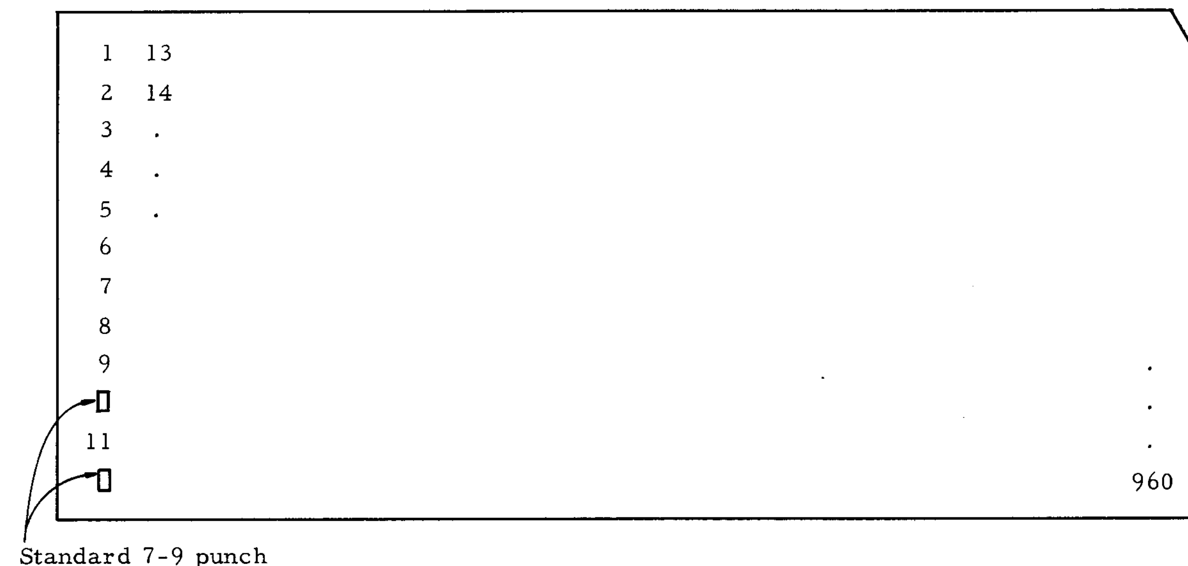


Figure 9. Numbering of Hole Positions for the B (Binary Card Output) Medium

It is expected that additional output media, with associated primitives and internal variables, may be added to the COGENT system in the future. The programmer may also define his own output media by coding the appropriate primitives in machine language.

1. P-medium Output Primitives

PUTP1(C, P). C must be an integer (number element) between 0 and 77_8, and P must be an integer between 1 and 135_{10}. PUTP1 places the value of C in the character position indicated by P of the current P-medium record image. The dummy element is returned.

PUTP(C). This generator is intended for use as a character receiver. Normally, successive calls of PUTP will place characters in successive positions of the current P-medium record image, but arbitrary actions may be specified to occur either when an oversized character is received or when a specified character position (margin) is reached.

C must be an integer between 0 and 1023_{10}. The following steps are performed:

a. If C is larger than 77_8, then the generator specified by the internal variable por (P-medium overflow receiver) is called and given C as its only argument. When this generator fails or returns a result, then PUTP fails or returns the same result without further execution.

b. If the value of the internal variable pin (P-medium index) is equal to the value of the internal variable pmr (P-medium margin), then the generator specified by the internal variable pmc (P-medium margin controller) is called and given C as its only argument. When this generator fails or returns a result, then PUTP fails or returns the same result without further execution.

c. The value of C is placed in the character position indicated by the value of pin. Then the value of pin is increased by one, and PUTP returns the dummy element as its result.

OUTP( ). The current P-medium record image is sent to the output device indicated by the value of the internal variable pno (P-medium logical unit number), which must be an integer giving a logical unit number recognized by the 3600 control system SCOPE. The initial value of pno is 61_{10}, which represents the standard output tape. When the record has been outputted, a new current record image is created and initialized, pin is set to one, and the dummy element is returned as the result of OUTP.

SPACEP( ). The carriage control character in the current P-medium record image is set to 0. This causes a blank line to precede the printing of this record image. The dummy element is returned.

EJECTP( ). The carriage control character in the current P-medium record image is set to 1. This causes a page to be ejected before the printing of this record image. The dummy element is returned.

STANDPMC(C). This primitive is provided as the initial value of pmr. It is exactly equivalent to a generator defined by

$GENERATOR STANDPMC((C) OUTP(). $RETURN(PUTP(C)).).

The initial values of the internal variables associated with the P-medium are:

por:	FAILURE
pin:	1
pmr:	120_{10}
pmc:	STANDPMC
pno:	61_{10}

If these variables (except pin) are not reset, then a character scan using PUTP as a character receiver will produce printed output in a free-field format, with 119 characters per line. In this case, the execution of PUTP(C) will perform the following steps:

a. If C is larger than 77_8, then PUTP will fail without further execution.

b. If pin = 120_{10}, then the current record image will be written on the standard printed output tape, a new current record image will be created and initialized, and pin will be reset to 1.

c. The value of C will be placed in the character position indicated by the value of pin. Then pin will be increased by one, and PUTP will return with the dummy element as its result.

It should be noted that, in addition to the records produced by OUTP, the output of the primitive dump generators, as well as comments and error messages produced by various system routines, are written by the output device indicated by pno. Thus system messages will be interspersed with the regular output on the P-medium.

2. C-medium Output Primitives

PUTC1(C, P). C must be an integer (number element) between 0 and 77_8, and P must be an integer between 1 and 80_{10}. PUTC1 places the value of C in the character position indicated by P of the current C-medium record image. The dummy element is returned.

PUTC(C). This generator is intended for use as a character receiver. Normally, successive calls of PUTC will place characters in successive positions of the current C-medium record image, but arbitrary actions may be specified to occur either when an oversized character is received or when a specified character position (margin) is reached.

C must be an integer between 0 and 1023_{10}. The following steps are performed:

a. If C is larger than 77_8, then the generator specified by the internal variable cor (C-medium overflow receiver) is called and given C as its only argument. When this generator fails or returns a result, then PUTC fails or returns the same result, without further execution.

b. If the value of the internal variable cin (C-medium index) is equal to the value of the internal variable cmr (C-medium margin), then the generator specified by the internal variable cmc (C-medium margin controller) is called and given C as its only argument. When this generator fails or returns a result, PUTC fails or returns the same result without further execution.

c. The value of C is placed in the character position indicated by the value of cin. Then the value of cin is increased by one, and PUTC returns the dummy element as its result.

OUTC( ). The current C-medium record image is sent to the output device indicated by the value of the internal variable cno (C-medium logical unit number), which must be an integer giving a logical unit number recognized by SCOPE. The initial value of cno is 62_{10}, which represents the standard punched output tape. Then a new current record image is created and initialized, cin is set to one, and the dummy element is returned as the result of OUTC.

STANDCMC(C). This primitive is provided as the initial value of cmc. It is exactly equivalent to a generator defined by

$GENERATOR STANDCMC((C) OUTC(). $RETURN(PUTC(C)).).

The initial values of the internal variables associated with the C-medium are:

cor: FAILURE

cin: 1

cmr: 73_{10}

cmc: STANDCMC

cno: 62_{10}

If these variables (except cin) are not reset, then a character scan using PUTC as a character receiver will produce cards in a free-field format, with 72 characters per card. In this case, the execution of PUTC(C) will perform the following steps:

a. If C is larger than 77_8, then PUTC will fail without further execution.

b. If cin = 73_{10}, then the current record image will be written on the standard punched output tape, a new current record image will be created and initialized, and cin will be reset to 1.

c. The value of C will be placed in the character position indicated by the value of cin. Then cin will be increased by one, and PUTC will return with the dummy element as its result.

3. Simultaneous P- and C-medium Output Primitives

PUTPC(C). This generator is intended for use as a character receiver for writing on both the P and C media simultaneously. It is exactly equivalent to a generator defined by

$GENERATOR PUTPC((C) PUTP(C). $RETURN(PUTC(C)).).

4. B-medium Output Primitives

PUTB1(N, P, L). N, P, and L must be integers such that $1 \le P \le 960_{10}$, $1 \le L \le 961 - P$, and $N \ge 0$. Bit positions P to P + L - 1 of the current B-medium record image are set to a binary representation of the value of N, with position P containing the most significant bit. The dummy element is returned. If $N \ge 2^L$, it is reduced modulo 2^L.

PUTB(N). The action of this generator depends upon three numerical quantities which are specified by internal variables: bin (B-medium index), bwl (B-medium word length), and bmr (B-medium margin). N must be an integer such that $N \ge 0$.

If bin + bwl > bmr, then PUTB calls the generator specified by the internal variable bmc (B-margin controller) and gives this generator N as its single argument; when this generator fails or returns a result, PUTB fails or returns the same result without further execution. Otherwise, bit positions bin to bin + bwl - 1 of the current B-medium record image are set to a binary representation of the value of N, with the bin-th position containing the most significant bit. Then the value of bin is replaced by bin + bwl, and PUTB returns with the dummy element as its result. If $N \ge 2^{bwl}$, it is reduced modulo 2^{bwl}.

OUTB( ). The current B-medium record image is sent to the output device indicated by the value of the internal variable bno (B-medium logical unit number), which must be an integer giving a logical unit number recognized by SCOPE. The initial value of bno is 62_{10}, which represents the standard punched output tape. Then a new current record image is created and initialized, bin is set to one, and the dummy element is returned as the result of OUTB.

No standard margin-controlling primitive is provided for the B-medium. It is assumed that the programmer will define his own controlling generator if the medium is to be used. The initial values of the pertinent internal variables are:

bmr: 1

bwl: 1

bin: 1

bmc: FAILURE

bno: 62_{10}

so that PUTB will simply fail unless these variables are reset.

K. Dump Primitives

The primitive dump generators are provided for debugging purposes and allow list structures to be written on the P-medium in a fixed format which explicitly displays their structural characteristics. In the output of these generators, each list element is represented by one or more print lines giving the name and components of the element. Nonliteral names are given as absolute machine addresses. The same element is not printed more than once, even if it appears several times in the list structures being dumped.

The dump primitives always write on the output device indicated by the internal variable pno, but they do not alter the current P-medium record image. They all return the dummy element as their result.

DUMPV(X). The value of X is printed out.

DUMP1( ). The names and values of all local, input, and own variables associated with the generator that calls DUMP1 are printed out.

DUMPALL( ). The names and values of all local, input, and own variables associated with all generators in the calling chain, plus all entries in the identifier tables, along with their association lists, are printed out.

L. Tape-control Primitives

Unlike other input-output facilities, the tape-control primitives explicitly describe the handling of magnetic tape, rather than an ultimate form such as cards or print lines. Each tape-control primitive accepts as its single argument an integer number element giving a logical unit number (as recognized by SCOPE), which must refer to a magnetic tape unit. The primitives all return the dummy element as their result.

The exact action taken by each of these primitives corresponds to the SCOPE tape control request with the same name.

BSPR(LUN). Backspace one record.

BSPF(LUN). Backspace one file.

REWIND(LUN). Rewind to load point.

UNLOAD(LUN). Rewind and unload.

SKIP(LUN). Skip to end-of-file.

MARKEF(LUN). Mark end-of-file.

M. Exit Primitives

NORMEXIT( ).

ABEXIT( ). The calling of either of these primitive generators causes an immediate termination of the entire COGENT program. NORMEXIT causes a normal termination, while ABEXIT causes an abnormal termination.

In general, there are five possible causes of program termination:

1. If the syntax analyzer returns on the main level of the program, and the value of the internal variable gol is zero, a normal termination occurs.

2. If NORMEXIT is called, a normal termination occurs.

3. If ABEXIT is called, an abnormal termination occurs.

4. If a primitive generator or system routine detects some erroneous condition, such as an ill-formed or ambiguous input string, the failure of a generator called by the syntax analyzer, or an illegal argument for a primitive generator, an abnormal termination occurs.

5. If some type of storage (list storage, pushdown stack, or syntax stack) is exhausted, an abnormal termination occurs.

CHAPTER IV

AN ILLUSTRATIVE COGENT PROGRAM: SYMBOLIC DIFFERENTIATION

As an extended example of COGENT programming, we give in this chapter a complete program for symbolic differentiation. The input read by this program is assumed to be a sequence of sentences followed by an end-of-file, where each sentence is an arithmetic expression followed by a period. The expressions are built up from positive decimal integers and variables, combined by the operations +, -, *, /, ** (exponentiation), and by functional forms. (The usual precedence of operations, as in FORTRAN, is assumed.) Expressions may also contain subexpressions of the form:

```
$D(<variable>, <expression>)
```

denoting the derivative of an expression with respect to a variable.

The output to be produced by the program is an equivalent sequence of sentences in which the derivative forms have been removed by carrying out the indicated differentiations symbolically.

A. Syntax Description

The following character and syntax descriptions are used:

```
$CHARDEF ($EOF) = (101)101.
$PRIMSYN ((SENT SEQ)($EOF))
                 (LETTER) = A, B, C, D, E, F, G, H, I, J, K, L, M,
                            N, O, P, Q, R, S, T, U, V, W, X, Y, Z.
                 (DIGIT) = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
                 (DIGIT STR) = (DIGIT), (DIGIT STR)(DIGIT).
                 (NAME STR) = (LETTER), (NAME STR)(LETTER),
                              (NAME STR)(DIGIT).
      $DEC/ (POS INT) = (DIGIT STR).
  $IDENT, 1/ (VARIABLE) = (NAME STR).
  $IDENT, 2/ (FUNCT NAME) = (NAME STR).
      $NOP/ (PRIMARY) = (POS INT), (VARIABLE).
            (PRIMARY) = (()(EXP)()).
            (PRIMARY) = (FUNCT NAME)(()(EXP SEQ)()).
      DIFF/ (PRIMARY) = $D(()(VARIABLE)(,)(EXP)()).
```



```
      $NOP/ (FACTOR) = (PRIMARY).
            (FACTOR) = (FACTOR)**(PRIMARY).
      $NOP/ (TERM) = (FACTOR).
            (TERM) = (TERM)*(FACTOR), (TERM)/(FACTOR).
      $NOP/ (EXP) = (TERM), +(TERM).
            (EXP) = -(TERM), (EXP)+(TERM), (EXP)-(TERM).
      $NOP/ (EXP SEQ) = (EXP).
            (EXP SEQ) = (EXP SEQ)(,)(EXP).
       OUT/ (SENT) = (EXP)(.).
     START/ (SENT SEQ) = .
  NORESULT/ (SENT SEQ) = (SENT SEQ)(SENT).
  $PROGRAM $OWN DVAR.
```

The character description is used to define a special object character representative for an end-of-file, which is needed as a terminator in the goal-specifier sequence of the primary syntax description. (The input code produced by the standard input editor when an end-of-file is encountered is 101_8.)

In the primary syntax description, character-packing special labels are used to convert positive decimal integers into integer number elements, and to convert variables and function names into identifier elements in identifier tables 1 and 2, respectively. To compress the list structures produced by the syntax analyzer, the special label $NOP/ is attached to each production whose construction string contains a single phrase class name and no characters (except for productions that define phrases that are to be converted into identifier or number elements, rather than extended list structures). The special label $NOP/ is also attached to the production (EXP) = +(TERM), where it acts to delete the unnecessary plus sign; i.e., (EXP) = +(TERM) is treated in the same manner as (EXP) = (TERM).

The actual process of differentiation is performed by the generator DIFF(X, Y), which accepts a variable X and an expression Y, and produces as its result a primary representing the derivative of Y with respect to X, in which the $D-type of subexpression no longer occurs. Since DIFF is called at each syntactic level where the $D-form occurs, the $D-forms will be elminated as they are encountered by the analyzer, so that the argument Y of DIFF will already have any $D-forms within it removed by previous applications of DIFF. Thus the syntax analyzer itself decomposes multiple differentiations into a sequence of single differentiations which are carried out by repeated calls of DIFF.

On the sentence level, each sentence, which will already have been translated to remove $D-forms, is processed by the generator OUT, which reduces the sentence to a string of object characters and writes this string on the printed output medium. At the beginning of the program, an empty sentence sequence will be recognized, and the generator START will be called; the only purpose of this generator is to set the internal variable gol to zero (so that when a sentence sequence terminated by an end-of-file has been parsed, the program will terminate). Thereafter, whenever a sentence sequence is recognized, it is processed by the primitive NORESULT, which produces the dummy element and thereby discards the list structure of the already processed sequence.

Since the output object language is the same as the input object language, the secondary syntax description is empty; i.e., the primary and total sets of productions are the same.

The generator description begins with a main declaration which establishes a single universal own variable, DVAR.

B. Precedence

Expressions, terms, factors, and primaries are all phrases that represent algebraic combinations of integers and variables. We will call all of these phrases general expressions. Most of our program will consist of generators that synthesize general expressions from simpler general expressions. In these processes, it is frequently necessary to enclose such expressions in parentheses or to remove enclosing parentheses. We first consider the precedence rules that determine when such operations are necessary, and define two generators for performing the operations.

Because of our use of the special label $NOP/, a general expression of one phrase class may be represented by a list structure whose head element refers to a production defining another phrase class. Thus a phrase of the class (EXP) may be represented by any of the following types of list structure:

0: Expression-headed, i.e., headed by a list element containing the code number of a production whose resultant is (EXP).

1: Term-headed, i.e., headed by a list element containing the code number of a production whose resultant is (TERM).

2: Factor-headed, i.e., headed by a list element containing the code number of a production whose resultant is (FACTOR).

3: Primary-headed, i.e., either headed by a list element containing the code number of a production whose resultant is (PRIMARY), or else consisting of a single integer number element or an identifier element in Table 1.

Similarly, a (TERM) may be represented by a term-headed, factor-headed, or primary-headed list structure, a (FACTOR) may be represented by a factor-headed or primary-headed structure, and a (PRIMARY) must be represented by a primary-headed structure.

We define the precedence number of an (EXP) to be 0, of a (TERM) to be 1, of a (FACTOR) to be 2, and of a (PRIMARY) to be 3. On the other hand, we define an expression-headed list structure to be 0-headed, a term-headed structure to be 1-headed, a factor-headed structure to be 2-headed, and a primary-headed structure to be 3-headed. We then have the following precedence rule: A general expression with precedence number i may be represented by a j-headed list structure if $j \geq i$. Conversely, a j-headed structure may be taken to represent a general expression with precedence number i if $i \leq j$.

Now suppose we have a list structure X representing a general expression of one phrase class, and we wish to produce a structure representing an equivalent general expression of another phrase class. If X is j-headed, and the desired phrase class has precedence number i, then the structure X may be used directly if $i \leq j$. But if $i > j$ then a new structure must be synthesized representing the original general expression enclosed in parentheses.

To perform this parenthesizing when it is required by the precedence rule, we define the generator:

```
$GENERATOR CLOTHE((X, N)
        +3 IF X =/ (EXP/(EXP)+(TERM)),,.
        +3 IF X =/ (EXP/(EXP)-(TERM)),,.
        +3 IF X =/ (EXP/-(TERM)),.
        +1 UNLESS N =/ 1.  $RETURN(X).
     1/ +3 IF X =/ (TERM/(TERM)*(FACTOR)),,.
        +3 IF X =/ (TERM/(TERM)/(FACTOR)),,.
        +2 UNLESS N =/ 2.  $RETURN(X).
     2/ +3 IF X =/ (FACTOR/(FACTOR)**(PRIMARY)),,.
        $RETURN(X).
     3/ X /= (PRIMARY/(()(EXP)())),X.  $RETURN(X).  ).
```

CLOTHE(X, N) accepts a list structure X representing any general expression, and an integer $N > 0$ giving a precedence number. It produces a structure representing an equivalent general expression with precedence number N. The original expression is parenthesized only if necessary. The case N = 0 is not considered, since parenthesizing is never necessary when N = 0.

A second generator is needed for removing parentheses. The generator

```
$GENERATOR STRIP ((X)
     +1 UNLESS X =/ (PRIMARY/(()(EXP)())),X.
1/ $RETURN(X). ).
```

accepts a list structure representing any general expression and produces a structure representing an equivalent general expression in which enclosing parentheses have been removed if they exist.

C. Algebraic Operations

Before discussing the main generator DIFF itself, we consider a set of generators used by DIFF to combine general expressions algebraically. Given any general expressions X and Y, the generators SUM(X, Y), NEGATIVE(X), PRODUCT(X, Y), QUOTIENT(X, Y), and POWER(X, Y) produce general expressions representing X+Y, -X, X·Y, X/Y, and X^Y, respectively. We will consider PRODUCT in detail; the remaining generators are sufficiently similar to be given without further discussion.

The following generator will accept two list structures representing general expressions and produce a structure representing the product of these expressions:

```
$GENERATOR PRODUCT ((X, Y)
     X /= (TERM/(()(EXP)())*(()(EXP)())), X, Y.
     $RETURN(X). ).
```

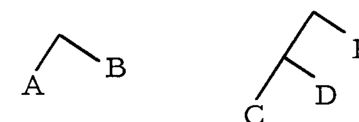
Such simple generators can be used for algebraic operations in a symbolic differentiation program. However, the output of such a program, although mathematically valid, would be too complicated to be readable. To produce a practical program, we must use algebraic operation generators that perform at least a limited amount of algebraic simplification.

In the first place, unnecessary parenthesization should be avoided; i.e., X or Y should only be parenthesized if this is required by the precedence rule. The generator CLOTHE may be used for this purpose:

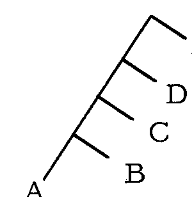
```
$GENERATOR PRODUCT ((X, Y)
     X /= (TERM/(TERM)*(FACTOR)), CLOTHE(X,1), CLOTHE(Y,2).
     $RETURN(X). ).
```


However, this version of PRODUCT still produces unnecessary parenthesization when Y is term-headed. For example, if X is (TERM/A*B) and Y is (TERM/C/D*E), then this generator gives (TERM/A*B*(()C/D*E())), while (TERM/A*B*C/D*E) would be desired.

The avoidance of unnecessary parentheses when Y is term-headed is a nontrivial problem, as can be seen by considering the actual list structures involved. In the example just given, X and Y have the schematic structures:



while the desired result has the structure:



Such a structure must be created by recursion. The simplest method is to have PRODUCT, when Y is term-headed, strip off the right-most factor in Y, call itself recursively to form the product of X with the remaining part of Y, and then call itself (or QUOTIENT) recursively to multiply (or divide) this result by the stripped-off factor:

```
$GENERATOR PRODUCT ((X,Y)  $LOCAL T1.
     +1 UNLESS Y =/ (TERM/(TERM)*(FACTOR)), Y, T1.
     $RETURN(PRODUCT(PRODUCT(X,Y), T1)).
  1/ +2 UNLESS Y =/ (TERM/(TERM)/(FACTOR)), Y, T1.
     $RETURN(QUOTIENT(PRODUCT(X,Y), T1)).
  2/ T1 /= (TERM/(TERM)*(FACTOR)), CLOTHE(X,1),
     CLOTHE(Y,1).
     $RETURN(T1). ).
```

Note that CLOTHE(Y, 1) may be used instead of CLOTHE(Y, 2) in statement 2, since Y cannot be term-headed if this statement is reached.

In addition to the avoidance of unnecessary parentheses, several other types of algebraic simplification are desirable. It is particularly important to delete unnecessary appearances of zeroes and ones in the

results of the algebraic operation generators. For the generator PRODUCT, this involves performing the following replacements:

X*0 → 0; 0*Y → 0; X*1 → X; 1*Y → Y.

These simplifications are especially important in symbolic differentiation, since in their absence the general expressions produced by DIFF will usually contain a profusion of redundant zeroes and ones (since DIFF recursively decomposes the expression being differentiated into its constituent variables and integers, whose individual derivatives are either zero or one). Thus, for example, if the algebraic operation generators do not delete unnecessary zeroes and ones, the translation of "$D(X, X*Y)" would be "1*Y+0*X" instead of simply "Y".

A further type of useful simplification is the distribution of negative signs, i.e., the replacements:

(-X)*Y → -X*Y; X*(-Y) → -X*Y; (-X)*(-Y) → X*Y.

When the removal of redundant zeroes and ones and the distribution of negative signs are incorporated into the generator PRODUCT, the final definition becomes:

```
$GENERATOR PRODUCT ((X,Y)  $LOCAL T1.
        +1 UNLESS X =/ 0.  $RETURN(0).
     1/ +2 UNLESS Y =/0.  $RETURN(0).
     2/ +3 UNLESS X =/1.  $RETURN(Y).
     3/ +4 UNLESS Y =/1.  $RETURN(X).
     4/ +5 IF X =/ (EXP/-(TERM)), X.
        +6 UNLESS Y =/ (EXP/-(TERM)), Y.
     5/ $RETURN(NEGATIVE(PRODUCT(X,Y))).
     6/ +7 UNLESS Y =/ (TERM/(TERM)*(FACTOR)), Y, T1.
        $RETURN(PRODUCT(PRODUCT(X,Y), T1)).
     7/ +8 UNLESS Y =/ (TERM/(TERM)/(FACTOR)), Y, T1.
        $RETURN(QUOTIENT(PRODUCT(X,Y), T1)).
     8/ T1 /= (TERM/(TERM)*(FACTOR)), CLOTHE(X,1),
              CLOTHE(Y,1).
        $RETURN(T1).  ).
```

It should be emphasized that this version of PRODUCT does not approach the full degree of algebraic simplification that is possible. In particular, integer factors are not multiplied out, and identical factors are not coalesced into powers. Thus, for example, if X is (TERM/2*X) and Y is (TERM/3*X), then the result of PRODUCT(X, Y) will be (TERM/2*X*3*X), while the simpler result (TERM/6*X**2) might be desirable. The whole concept of algebraic simplification is rather relative and imprecise, and the amount and type of simplification performed by this version of PRODUCT has been chosen somewhat arbitrarily.

The remaining algebraic operation generators are defined below. As with PRODUCT, the extent of the algebraic simplifications performed by these generators has been chosen somewhat arbitrarily.

```
$GENERATOR SUM ((X,Y)  $LOCAL T1.
        +1 UNLESS X =/ 0.  $RETURN(Y).
     1/ +2 UNLESS Y =/ 0.  $RETURN(X).
     2/ +3 UNLESS Y =/ (EXP/(EXP)+(TERM)), Y, T1.
        T1 /= (EXP/(EXP)+(TERM)), SUM(X,Y), T1.  $RETURN(T1).
     3/ +4 UNLESS Y =/ (EXP/(EXP)-(TERM)), Y, T1.
        T1 /= (EXP/(EXP)-(TERM)), SUM(X,Y), T1.  $RETURN(T1).
     4/ +5 UNLESS Y =/ (EXP/-(TERM)), Y.
        T1 /= (EXP/(EXP)-(TERM)), X, Y.  $RETURN(T1).
     5/ T1 /= (EXP/(EXP)+(TERM)), X, Y.  $RETURN(T1).  ).

$GENERATOR NEGATIVE ((X)  $LOCAL T1.
        +1 UNLESS X =/ 0.  $RETURN(0).
     1/ +2 UNLESS X =/ (EXP/(EXP)+(TERM)), X, T1.
        T1 /= (EXP/(EXP)-(TERM)), NEGATIVE(X), T1.
              $RETURN(T1).
     2/ +3 UNLESS X =/ (EXP/(EXP)-(TERM)), X, T1.
        T1 /= (EXP/(EXP)+(TERM)), NEGATIVE(X), T1.
              $RETURN(T1).
     3/ +4 UNLESS X =/ (EXP/-(TERM)), X.  $RETURN(X).
     4/ X /= (EXP/-(TERM)), X.  $RETURN(X).  ).
```

```
$GENERATOR QUOTIENT ((X,Y)  $LOCAL T1.
        +1 UNLESS X =/ 0.  $RETURN(0).
    1/ +2 UNLESS Y =/ 1.  $RETURN(X).
    2/ +3 IF X =/ (EXP/-(TERM)), X.
        +4 UNLESS Y =/ (EXP/-(TERM)), Y.
    3/ $RETURN(NEGATIVE(QUOTIENT(X,Y))).
    4/ +5 UNLESS Y =/ (TERM/(TERM)*(FACTOR)), Y, T1.
        $RETURN(QUOTIENT(QUOTIENT(X,Y), T1)).
    5/ +6 UNLESS Y =/ (TERM/(TERM)/(FACTOR)), Y, T1.
        $RETURN(PRODUCT(QUOTIENT(X,Y), T1)).
    6/ T1 /= (TERM/(TERM)/(FACTOR)), CLOTHE(X,1),
              CLOTHE(Y,1).
        $RETURN(T1).  ).

$GENERATOR POWER ((X,Y)
        +1 UNLESS Y =/ 1.  $RETURN(X).
    1/ X /= (FACTOR/(FACTOR)**(PRIMARY)), CLOTHE(X,2),
            CLOTHE(Y,3).
        $RETURN(X).  ).
```

D. The Main Generator DIFF

We now come to the main generator of our program, DIFF(X, Y), which accepts two list structures X and Y representing a variable and an expression, and produces a structure representing a primary giving the derivative of Y with respect to X. The actual process of differentiation is carried out by a second generator DIFF1; the action of DIFF is merely to call DIFF1 with Y as its only argument, and then to call CLOTHE to convert the general expression produced by DIFF1 into a primary.

The purpose of using a second generator in this manner is to allow DIFF1 to obtain the identifier element representing the variable of differentiation without having to pass this element to DIFF1 as an input argument on each recursive level. This is accomplished by setting the universal own variable DVAR, upon entrance to DIFF, to the name of the variable of differentiation.


```
$GENERATOR DIFF ((X,Y)
    DVAR = X.  $RETURN(CLOTHE(DIFF1(Y), 3)).  ).
$GENERATOR DIFF1 ((Y)  $LOCAL Z, T1.
    +1 UNLESS Y =/ DVAR.  $RETURN(1).
1/  +2 IF NORMTEST(Y).  $RETURN(0).
2/  +3 UNLESS Y =/ (EXP/-(TERM)), Y.
    $RETURN(NEGATIVE(DIFF1(Y))).
3/  +4 UNLESS Y =/ (EXP/(EXP)+(TERM)), Y, Z.
    $RETURN(SUM(DIFF1(Y), DIFF1(Z))).
4/  +5 UNLESS Y =/ (EXP/(EXP)-(TERM)), Y, Z.
    $RETURN(SUM(DIFF1(Y), NEGATIVE(DIFF1(Z)))).
5/  +6 UNLESS Y =/ (TERM/(TERM)*(FACTOR)), Y, Z.
    $RETURN(SUM(PRODUCT(DIFF1(Y), Z),
        PRODUCT(DIFF1(Z), Y))).
6/  +7 UNLESS Y =/ (TERM/(TERM)/(FACTOR)), Y, Z.
    $RETURN(SUM(QUOTIENT(DIFF1(Y), Z),
        NEGATIVE(QUOTIENT(PRODUCT(DIFF1(Z), Y),
        POWER(Z,2))))).
7/  +8 UNLESS Y =/ (FACTOR/(FACTOR)**(PRIMARY)),
        Y, Z.
    T1 /= (TERM/LOG(()(EXP)())*(FACTOR)**
        (PRIMARY)), STRIP(Y), Y, Z.
    $RETURN(SUM(PRODUCT(PRODUCT(DIFF1(Y), Z),
        POWER(Y, SUM(Z, (EXP/-1)))), PRODUCT(DIFF1(Z),
        T1))).
8/  +9 UNLESS Y =/ (PRIMARY/(()(EXP)())), Y.
    $RETURN(DIFF1(Y)).
9/  Y =/ (PRIMARY/(FUNCT NAME)(()(EXP SEQ)())), Y, Z.
    $RETURN(ALIST(Y)(Z)).  ).
```

The first action of DIFF1 is to compare its input argument Y with the variable of differentiation specified by the universal variable DVAR. If these structures match, then the integer 1, representing dX/dX, is returned. Otherwise, if Y is not a list structure headed by a normal element (i.e., if it is an integer or some variable other than the variable of differentiation), then the integer 0 is returned.

If Y is headed by a normal list element, then DIFF1 tests for each of the possible normal elements that can head Y. When the head element is identified, its sublists are obtained, DIFF1 is called recursively to differentiate the sublists, and the undifferentiated and differentiated sublists are combined by using the algebraic operation generators into a structure representing the derivative of Y, which is returned as the result of DIFF1. The various cases are treated according to the following differentiation formulas:

$$(-Y)' = -Y'$$
$$(Y+Z)' = Y'+Z'$$
$$(Y-Z)' = Y'-Z'$$
$$(Y \cdot Z)' = Y' \cdot Z + Z' \cdot Y$$
$$(Y/Z)' = Y'/Z - Z' \cdot Y/Z^2$$
$$(Y^Z)' = Y' \cdot Z \cdot Y^{Z-1} + Z' \cdot \log(Y) \cdot Y^Z$$
$$(Y)' = Y'$$

The final step of DIFF1 is reached if Y represents a functional expression. In this case, the action to be taken depends upon the function name, and a different calculation must be specified for each function name that may occur in the input object language. Of course, functional expressions could be treated by having DIFF1 test for each function name and branch to the appropriate calculation. However, it is advantageous to use a less direct approach which allows the program to be extended easily to handle more function names.

In this approach, each function name, which will be an identifier element in table 2, is assumed to have as its association list a generator element denoting a generator for differentiating the corresponding function. When DIFF1 finds a functional expression, it merely calls the generator denoted by the association list of the function name and gives this generator as its single argument the expression sequence within the functional expression. The result of this generator is then returned as the result of DIFF1.

We define function-differentiating generators for the four function names SIN, COS, EXP, and LOG. Note that each definition contains an identifier declaration which initializes the association list of the appropriate function name to the desired generator element. Also note that since SIN, COS, EXP, and LOG all take expression sequences containing single expressions, and since the production (EXP SEQ) = (EXP) has a $NOP/ special label, the input argument of each of these generators is a list structure that directly represents a general expression, rather than an expression sequence whose subphrases are expressions.


```
$GENERATOR SINDIF ((Y)  $IDA (FUNCT NAME/SIN) = SINDIF.
    $LOCAL T1.
    T1 /= (PRIMARY/COS(()(EXP SEQ)())), Y.
    $RETURN(PRODUCT(DIFF1(Y), T1)).  ).
$GENERATOR COSDIF ((Y)  $IDA(FUNCT NAME/COS) = COSDIF.
    $LOCAL T1.
    T1 /= (PRIMARY/SIN(()(EXP SEQ)())), Y.
    $RETURN(NEGATIVE(PRODUCT(DIFF1(Y), T1))).  ).
$GENERATOR EXPDIF ((Y) $IDA (FUNCT NAME/EXP) = EXPDIF.
    $LOCAL T1.
    T1 /= (PRIMARY/EXP(()(EXP SEQ)())), Y.
    $RETURN(PRODUCT(DIFF1(Y), T1)).  ).

$GENERATOR LOGDIF ((Y)  $IDA (FUNCT NAME/LOG) = LOGDIF.
    $RETURN(QUOTIENT(DIFF1(Y), Y)).  ).
```

These generators correspond to the following differentiation formulas:

$$\sin(Y)' = Y' \cdot \cos(Y)$$
$$\cos(Y)' = -Y' \cdot \sin(Y)$$
$$\exp(Y)' = Y' \cdot \exp(Y)$$
$$\log(Y)' = Y'/Y$$

E. Additional Generators

The generator OUT accepts a list structure representing a general expression, removes enclosing parentheses if they exist, synthesizes a sentence from the expression, and writes this sentence on the printed output medium. Each sentence appears in a free-field format beginning at the left of a new line, and the sentences are separated by blank lines. Since the only number elements appearing in the list structures given to OUT are positive integers, the primitive STANDSCN is used for character scanning without resetting any internal variables.

```
$GENERATOR OUT ((X)
    X /= (SENT/(EXP)(.)), STRIP(X).
    SPACEP(). STANDSCN(X, PUTP). OUTP(). ).
```

The generator START is called at the beginning of the program. Its only effect is to reset the internal variable gol to zero, so that the program will terminate when a sentence sequence followed by an end-of-file has been parsed.

```
$GENERATOR START(() SETIVGOL(0). ).
```

REFERENCES

1. Irons, Edgar T., A Syntax Directed Compiler for ALGOL 60, Comm. ACM 4, p. 51 (1961).

   Mayoh, B. H., Iron's Procedure DIAGRAM, Comm. ACM 4, p. 284 (1961).

   Ledley, R. S., and Wilson, J. B., Automatic-Programming-Language Translation Through Syntactical Analysis, Comm. ACM 5, p. 145 (1962).

   Irons, Edgar T., Towards More Versatile Mechanical Translators, Working Paper No. 47, Communications Research Division, Institute for Defense Analyses, Princeton, New Jersey.

2. Brooker, R. A., and Morris, D., An Assembly Program for a Phrase Structure Language, The Computer Journal 3, p. 168 (1960).

   Brooker, R. A., and Morris, D., Some Proposals for the Realization of a Certain Assembly Program, The Computer Journal 3, p. 220 (1961).

   Brooker, R. A., Morris, D., and Rohl, J. S., Trees and Routines, The Computer Journal 5, p. 33 (1962).

   Brooker, R. A., and Morris, D., A General Translation Program for Phrase Structure Languages, Journal ACM 9, p. 1 (1962).

   Rosen, Saul, A Compiler-Building System Developed by Brooker and Morris, Comm. ACM 7, p. 403 (1964).

3. McCarthy, John, Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I, Comm. ACM 3, p. 184 (1960).

   Woodward, P. M., and Jenkins, D. P., Atoms and Lists, The Computer Journal 4, p. 47 (1961).

   McCarthy, John, et al., LISP 1.5 Programmer's Manual, M.I.T. Press (1962).

4. Backus, J. W., The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, Proceedings of the International Conference on Information Processing, UNESCO, Paris, June 1959, p. 125.

5. Ross, D. T., A Generalized Technique for Symbol Manipulation and Numerical Calculation, Comm. ACM 4, p. 147 (1961).

6. Edwards, Daniel J., LISP II Garbage Collector, unpublished.

7. Floyd, Robert W., On Ambiguity in Phrase Structure Languages, Comm. ACM 5, p. 526 (1962).