AN INTRODUCTION TO THE COGENT PROGRAMMING SYSTEM

John C. Reynolds

Applied Mathematics Division

Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois

739-7711

AN INTRODUCTION TO THE COGENT PROGRAMMING SYSTEM*

John C. Reynolds

Applied Mathematics Division

Argonne National Laboratory

Argonne, Illinois

ABSTRACT

The COGENT programming system is a compiler whose input language is designed to describe symbolic or linguistic manipulation algorithms. It represents an attempt to unify the concepts of syntax-directed compilation and recursive list-processing. Basically, a COGENT program is a list-processing program in which the list structures normally represent phrases of one or more object languages. Correspondingly, the COGENT language itself contains two major constructions: productions, which define the object language syntax, and generator definitions, which define list-processing subroutines.

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

In designing the system the major objective has been to unify the concept of syntax-directed compilation(1) with the more general but primitive concept of recursive list-processing(2). This objective is achieved by using the syntax

*Work performed under the auspices of the U.S. Atomic Energy Commission.

of a language to define a mapping between strings of the language and list structures. Given such a mapping, linguistic processes become equivalent to list processes, and a list-processing language becomes a concise vehicle for describing such processes.

Thus a program written in the COGENT language is a list-processing program in which the list structures normally represent phrases of one or more _object_ languages (i.e., the input and output languages to be processed by the program), in a representation determined by the syntax of these languages. Correspondingly, the COGENT language itself contains two major constructions: _productions_, which define the object language syntax, and _generator definitions_, which define list-processing subroutines called generators.

Productions and the String-to-List Mapping

The productions used in COGENT are conceptually identical to the Backus Normal Form.(3) A set of such productions, describing a simple language of polynomials, is illustrated in Table 1. The five metalinguistic symbols of the Backus notation, "::=", "<", ">", "|", and the end-of-line, are transliterated into the conventional keypunch characters "=", "(", ")", ",", and "." . This transliteration creates an ambiguity between metasymbols and object characters which is resolved by requiring that the characters "(" , ")" , "," , and "." must themselves be enclosed in parentheses when used as object characters, as in the fifth line of Table 1.

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

Table 1. Productions describing a simple language of polynomials. Illustrative internal code numbers are shown on the right.

When several alternative constructions are to be given for the same phrase class (we follow Brooker and Morris[4] in using "phrase class" rather than "syntactic unit" to denote a set of syntactically equivalent phrases), a sequence of single-alternative productions may be used, e.g.,

```
(TERM) = (FACTOR).
(TERM) = (TERM)*(FACTOR).
```

or the alternative constructions may be combined into a compound production, e.g.,

```
(TERM) = (FACTOR),(TERM)*(FACTOR).
```

The latter format is merely an abbreviation for the former, and in discussing the meaning of COGENT we will assume that such compounds have been expanded so that all productions are single-alternative.

It is well-known that a set of productions may be used to convert a phrase of object language into a parsing diagram or construction tree. In such a tree, an example of which is given in Figure 1, the nonterminal nodes correspond to phrase classes, the terminal nodes correspond to characters, and the relationship of any nonterminal node to its subnodes corresponds to a particular production. Essentially, the concept of the construction tree provides the mapping between object language phrases and list structures. However, the list structures actually used in COGENT are considerably more compact than construction trees, since the latter contain redundant information already specified by the productions themselves.

When productions are read by the COGENT compiler (after compounds have been expanded into single-alternative productions), they are assigned unique internal code numbers. Given such a set of code numbers, the list structure representing an object phrase may be defined in terms of the construction tree for the phrase as follows: There is a list element corresponding to each nonterminal node of the tree. The first field of this element gives the code number of the production that relates the tree node to its subnodes. The remaining fields, if any, are pointers to the list elements that represent those subnodes of the original node which are themselves nonterminal. Figure 2 gives the list structure corresponding to the construction tree in Figure 1, using the code numbers shown on the right of Table 1. There are no list elements representing terminal tree nodes since the nature of these nodes is defined by the productions themselves.

Two further comments should be made on the use of productions to map object strings into list structures. First, this usage requires that a set of productions must be nonambiguous, i.e., it must define a unique mapping. Except for a minor restriction on the use of empty productions (which define a phrase class to contain the empty string), nonambiguity is the only requirement imposed on productions in COGENT.

Secondly, two different sets of productions defining the same object language may define different mappings. Thus, for example, if the seventh line in Table 1 is replaced by

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

the same object language is defined, but the list structures representing terms are reversed.

The Syntax Analyzer

When a COGENT program is executed, its main routine is always a syntax analyzer compiled from the productions describing the input object language. This analyzer reads strings from the input medium, creates the corresponding list structures, and calls upon generators to process these structures. In the COGENT program itself, this calling of generators is specified by prefixing labels to some of the productions defining the input language.

Normally, the syntax analyzer constructs list structures from the bottom up, so that each list element is created immediately after its last sublist. However, whenever the analyzer is about to create an element corresponding to a labelled production, it calls the generator named by the label and gives this generator as input arguments the sublists of the element that would otherwise be created. When the generator returns control, its result replaces the element that would have been created, and the syntax 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 labels are attached to the following two productions from Table 1:

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

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

Figure 3 illustrates the action of the syntax analyzer while parsing the input string "-(A+B)*DC" according to these productions. First, the generator ADDCOMP will be called and given the two arguments shown in 3a. Next, MULTCOMP will be called and given the arguments shown in 3b, the first of which contains the result of ADDCOMP. The final result of the parse is shown in 3c.

This general method, of associating generators with productions and calling a generator at each syntactic level where the corresponding production is used, is equivalent to syntax-directed compilation as developed by E. T. Irons.(1) Our approach extends beyond Irons' only in the generators themselves, which are capable of conditional analysis of list structures and of calling each other recursively.

Generator Definitions: Constants

Basically, a generator is a subroutine for manipulating list structures and is similar in form to an ALGOL procedure. More precisely, since a generator may have any number of input arguments but only a single result, it is similar to a function procedure with call by value imposed on all arguments.

The essential peculiarity of generators is that the values of their variables are list structures, usually representing phrases of object language. Thus the most distinctive feature of generator definitions is the format of constants, which must denote such list structures.

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 consists of both a phrase class name and a string of object characters, separated by a slash and enclosed in parentheses. Just as in productions, the four object characters "(", ")", "," , and "." must be parenthesized to prevent ambiguity. 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 shown in Figure 2.

The set of constants used in COGENT (and therefore the set of values that variables may assume) is extended to include <u>parametric phrases</u>, i.e., phrases in which one or more subphrases, called <u>parameters</u>, are left unspecified. Such phrases play a central role in COGENT as templates for the synthesis and analysis of list structures.

Parametric phrases are represented by allowing phrase class names to appear in the object string of a constant. For example,

```
    (TERM/(FACTOR)*(FACTOR)*(FACTOR))
```

represents a term containing three parametric factors. In general, each phrase class name in the object string represents a parameter. Associated with each such parameter is a numerical index, which is normally the order of appearance of the parameter in the entire object string. However, indices may also be specified explicitly, e.g.,

```
    (TERM/(FACTOR/1)*(FACTOR/2)*(FACTOR/3))
```

is equivalent to the constant above.

Parametric constants are converted into list structures in a manner similar to that for conventional constants. Each phrase class name behaves syntactically as a phrase of the named class, but is converted into a special list element called a parameter element, which always contains a single field giving the index of the parameter. The list structure corresponding to either of the above constants is shown in Figure 4.

Assignment Statements

The format and meaning of variables, expressions, and simple assignment statements is the same in COGENT as in most programming languages. An expression may be a constant, a variable, or a compound expression; in the latter, a functional notation represents the calling of other generators (i.e., the value of the compound expression is the result of the called generator).

However, in addition to the conventional or direct assignment statement, synthetic and analytic assignment statements are provided to describe the synthesis and analysis of list structures. While these statements are actually general-purpose list-processing operations, they explicitly display the linguistic manipulations which these operations represent. As noted earlier, this is accomplished by using parametric constants as templates.

The synthetic assignment statement has the format

<synthetic assignment statement> ::=

<name> /= <expression$_{template}$>, <expression$_1$>,...,<expression$_n$>.

This statement evaluates all the expressions, creates a copy of the list structure that is the value of the template expression (usually a parametric constant), 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 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 by the value of Y. Thus Z will be given the value (TERM/ABE*BED).

The analytic assignment statement has the format

<analytic assignment statement> ::=

<expression$_{test}$> =/ <expression$_{template}$>, <name$_1$>,...,<name$_n$> .

This statement evaluates both expressions, and then compares, element by element, the list structures which 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 name$_i$. Thus, for example, if Z has the value (TERM/ABE*BED), then the statement

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

will give X the value (FACTOR/ABE) and Y the value (FACTOR/BED).

An additional property of the analytic assignment statement arises because 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 structures do not match, then the analytic assignment statement will fail, 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), 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.(4)

<u>Control Statements and Failure</u>

Any statement in a generator definition may be prefixed with a numerical label or statement number, to allow it to be referenced by control statements in the same generator definition. The most important control statement is the conditional jump statement, with the format:

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

(The character "+" is used to mean "go to".) When control reaches a conditional jump statement, the right-hand assignment statement is executed, and then control jumps to the statement denoted by the statement number IF or UNLESS the assignment statement did not fail.

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 will transfer control to statement 10. But if the value of Z is not a term containing two factors, then control will pass to the next statement without altering X or Y.

Failure, which is the basic control mechanism in COGENT, is a formalization of the concept of an error trap. A failure may originate in three ways:

1) Certain primitive (built-in) generators used to perform tests may fail.
2) An analytic assignment statement will fail if the lists being compared do not match.
3) A control statement written "$FAILURE." is provided which simply fails without taking any other action.

Once a failure occurs, it propagates 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 useful in describing complex processes that 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.

Declarations and Nested Generator Definitions

In addition to statements, a generator definition may contain declarations that control the interpretation of variables and other names within the definition. The usual concept of type, e.g., integer, real, or array, is not meaningful in COGENT since all variables have the same set of possible values. Thus the only function of declarations is to control storage allocation and to specify initialization values. For example,

```
$OWN X,Y = (FACTOR/BED),Z.
```

declares the names X, Y, and Z to be own variables and specifies (FACTOR/BED) as the initial value of Y.

In general, names may be declared to be own variables, local variables, or pseudoconstants. Own variables are similar to ALGOL own variables and are initialized at the beginning of program execution. Local variables are similar to ALGOL non-own variables and are initialized upon each entrance to the generator in which they are declared.

Pseudoconstants are names which are used as abbreviations for constants, and are not variables in the usual sense of possessing dynamically varying values. For example, within the scope of the declaration

```
$PCON TERM3 = (TERM/(FACTOR)*(FACTOR)*(FACTOR)).
```

any appearance of the name TERM3 is equivalent to the constant (TERM/(FACTOR)*(FACTOR)*(FACTOR)).

A complete generator definition has the format:

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

where

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

This format allows generator definitions to be nested and thus provides a limited block structure (in which complete generator definitions are the only blocks). This nesting allows a generator to evaluate or alter variables declared in a higher-level generator.

Generator Elements

In many programs, it is convenient to use variables which take on generators themselves as their values. This capability is provided by a special type of list element called a generator element, which has a single field 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 leads to a more general interpretation of the concepts of generator names and compound expressions. A generator name is actually a type of pseudoconstant which denotes a generator element. 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 a variable, while PLUSCOMP is a generator name. Then

```
    X = PLUSCOMP.
```

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

```
Z = X(Y).
```

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

Similarly, if PROCESS and PLUSCOMP are generator names, then

```
Z = PROCESS(Z, PLUSCOMP).
```

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

```
$GENERATOR PROCESS ((X, G) ...
... Y = G(Y). ... ).
```

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

A more esoteric example is a compound expression such as

```
GSWITCH(X) (Y)
```

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

Primitive Generators: Numerical Operations

A variety of facilities is provided in COGENT by primitive generators, i.e., generators which may be used in a COGENT program without being defined. These facilities include numerical operations, symbol-table maintenance, and output.

Numbers in COGENT are represented by special list elements called number elements, which contain two fields: a mode (integer or floating-point) and a value. The value fields of floating-point numbers have a fixed length, but the value fields of integers have a variable length depending upon the magnitude of the integer. Thus there is no overflow in integer arithmetic, since the result of each operation is dynamically assigned adequate storage. This facility for arbitrarily large integers is particularly useful in algebraic manipulation programs, in which exact fractional coefficients may be used throughout the computation.

Primitive generators are provided for performing the basic arithmetic operations upon number elements. For example, ADD(X, Y) accepts two number elements and produces a new element representing their sum. In general, these arithmetic primitives accept arguments with mixed modes, and produce floating-point results if any of their arguments are floating-point.

Primitive generators are also provided for converting general list structures into number elements. For example, DECCON(X) accepts a list structure and reduces this structure back into a character string in accordance with the string-to-list mapping defined by the object language syntax. This string is then interpreted as a positive decimal integer (ignoring nondigits), and the corresponding number element is returned as the result of DECCON. Similar primitives are provided for octal integers and decimal floating-point numbers.

To facilitate the programming of arithmetic operations, unsigned integer constants are allowed in generator definitions, and are used to denote positive integer number elements.

Identifiers

A general-purpose symbol-table facility is provided by special list elements called _identifier_ elements, which contain packed character strings. These elements serve two purposes:

1) To save storage by allowing phrases whose syntactic substructures have no semantic content to be represented by packed strings rather than parsed list structures.

2) To allow descriptive information to be associated with such phrases.

Identifier elements are grouped into one or more identifier tables, which are distinguished by positive integers called table numbers. Within a single table no two elements can contain the same character string. Thus all references to identifier elements with the same string and in the same table are references to the same element.

Specifically, an identifier element contains four fields: the table number, the character string, the table link name (used to link elements together for table searching and not directly accessible to the programmer), and the association list name. The association list name may be set by the programmer to point to an arbitrary list structure, and is used to associate descriptive information with an identifier.

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 mapped back into 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 return 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. ALIST(I) accepts an identifier element I and returns the association list name from this element.

Additional primitives are provided for iterating over all elements in an identifier table, and for deleting elements from tables.

Output

Output operations in COGENT 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 is usually performed by 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., an integer number element giving the BCD code for the character.

Basically, the list-to-string mapping performed by STANDSCN is determined by the productions that describe the output object language, so that the output operation of character scanning is the inverse of the input operation of syntax analysis. However, the productions do not specify the response of STANDSCN to special list elements such as number elements. Normally, STANDSCN will convert a number element to a free-field decimal digit string, but this response may be altered by the programmer. The programmer also has the option of defining his own character-scanning generator in terms of more basic scanning primitives.

The character-output phase is performed by the character receiver. For printed output, the primitive PUTP(C) is supplied for use as a character receiver. On each call, PUTP accepts an integer number element 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 OUTP() outputs the current line even if the margin limit has not been reached, and then initializes a new line. Similar primitives are provided for BCD and binary card output.

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().
```

Normally, the character-output primitives produce a free-field format, but the programmer has the option of introducing more sophisticated output formats by arbitrarily specifying the response of a character-receiving primitive to special character codes or margin limits.

Implementation

An initial version of COGENT has been coded for the Control Data 3600.(5) Two aspects of this implementation warrant a brief description: the basic method used for syntax analysis, and the data representation and storage allocation mechanism used for list structures.

It is fairly straightforward to convert a set of Backus Normal Form productions into a set of recursive subroutines which collectively form a syntax analyzer. The essential difficulty is that for certain object languages these subroutines will contain ambiguity points, i.e., points at which a conditional branch must be performed although the current state of the analysis is insufficient to determine the branch. In most approaches to syntax analysis, these ambiguity points are handled by "backup": a single control path extending from an ambiguity point is pursued until the analysis is found to be inconsistent with the input string; then the analyzer and input string are backed up to the last-encountered ambiguity point and an alternative path is pursued.

However, in COGENT all paths extending from an ambiguity point are pursued in parallel. The actual analysis program simulates an assembly of independent analyzers, all parsing the same input string according to the same program, but with different internal states. When an analyzer encounters an ambiguity point, it fissions into two or more analyzers; when an analyzer reads an input character inconsistent with its internal state, it vanishes from the assembly.

In performing this simulation, the analysis program alternates between a normal mode, in which the assembly contains a single analyzer, and an ambiguity mode, in which the assembly contains more than one analyzer. In the ambiguity mode the actions of list construction and generator calling that would be performed by a single analyzer are not actually executed; instead, indicators for these actions are stored in a queue associated with the analyzer. When the program switches from the ambiguity mode to the normal mode, the actions stored in the queue of the surviving analyzer are carried out before the analysis continues. Thus, from the user's viewpoint, the analysis program behaves like a single analyzer which, when necessary to resolve an ambiguity, looks ahead at the input string.

The storage allocation mechanism used for list structures follows the basic approach used in LISP.(2) A pushdown stack of consecutive words provides dynamically assigned storage for the variables of the generators, and thus contains the names of all list structures being used in the computation. The list elements themselves are stored in a separate area called list storage, which 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.

However, the representation of list elements differs from LISP in several respects. Most important, a list element is not always a single machine word, but may be an array of consecutive words whose length depends upon the size and number of fields in the element. (Thus list structures in COGENT are akin to plexes, as defined by D. T. Ross.(6)) 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 routine relocates the active list elements after marking them. This routine is based upon a scheme proposed by D. Edwards.(7)

A second feature of the list representation is the use of <u>literal</u> list names to denote certain short but frequently used list elements. These names are not addresses of arrays in list storage, but are direct encodings of the fields of the named elements. The use of literal names for parameters, small integers, and normal elements with no sublists provides substantial savings in both storage space and execution time.

A Programming Example

Table 2 gives a COGENT program for translating algebraic expressions from Polish prefix notation into conventional infix notation. The input to this program is assumed to be a sequence of Polish expressions separated by commas and followed by a period. The Polish expressions use single-letter variable names, the binary operators "+" , "-" , "*" , and "/" , and the unary operator "=" (denoting negation).

The overall program consists of three sections:

1) The primary syntax description, headed by "$PRIMSYN", gives the productions describing the input object language. This section begins with a goal specifier "((SENTENCE)(.))", which specifies the phrase class (SENTENCE) to be the overall goal to be sought by the syntax analyzer, and specifies "." to be the terminal character which must follow the (SENTENCE).

2) The secondary syntax description, headed by "$SECSYN", gives the additional productions needed to describe the output object language.

3) The generator description, headed by "$PROGRAM", gives the generator definitions. Within these definitions the control statement "$RETURN(X)." causes a generator to exit with X as its result. The statement "NORMEXIT()." is a call of a primitive generator which causes a normal program termination.

The program converts a (PEXP) (Polish expression) into an equivalent (EXP) (conventional expression) at each syntactic level where a (PEXP) is recognized. Thus the input arguments of the generators GMULT, GADD, and GNEG are always phrases of the class (EXP). Within these generators, conditional analysis is used to avoid unnecessary parenthezation.

```
$PRIMSYN ((SENTENCE)(.))
       (VAR) = 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.
       (MOP) = *,/.
       (AOP) = +,-.
       (NOP) = =.
  GVAR/ (PEXP) = (VAR).
 GMULT/ (PEXP) = (MOP)(PEXP)(PEXP).
  GADD/ (PEXP) = (AOP)(PEXP)(PEXP).
  GNEG/ (PEXP) = (NOP)(PEXP).
        (PEXPSEQ) = (PEXP),(PEXPSEQ)(,)(PEXP).
   OUT/ (SENTENCE) = (PEXPSEQ).
 $SECSYN
        (FACTOR) = (VAR),(()(EXP)()).
        (TERM) = (FACTOR),(TERM)(MOP)(FACTOR).
        (EXP) = (TERM),-(TERM),(EXP)(AOP)(TERM).
```

NOTE TO EDITOR: This is the first page of a two-page table. The caption is on the next page.

```
$PROGRAM
$GENERATOR GVAR ((X)
    X /= (EXP/(VAR)),X. $RETURN(X). ).
$GENERATOR GMULT((OP,X,Y)
    +1 IF X =/ (EXP/(TERM)),X.
    X /= (TERM/(()(EXP)())),X.
 1/ +2 IF Y =/ (EXP/(FACTOR)),Y.
    Y /= (FACTOR/(()(EXP)())),Y.
 2/ X /= (EXP/(TERM)(MOP)(FACTOR)),X,OP,Y.
    $RETURN(X). ).
$GENERATOR GADD ((OP,X,Y)
    +1 IF Y =/ (EXP/(TERM)),Y.
    Y /= (TERM/(()(EXP)())),Y.
 1/ X /= (EXP/(EXP)(AOP)(TERM)),X,OP,Y.
    $RETURN(X). ).
$GENERATOR GNEG ((OP,X)
    +1 IF X =/ (EXP/(TERM)),X.
    X /= (TERM/(()(EXP)())),X.
 1/ X /= (EXP/-(TERM)),X.
    $RETURN(X). ).
$GENERATOR OUT ((X)
    STANDSCN(X,PUTP). OUTP(). NORMEXIT(). ).
```

Table 2. An illustrate program which translates Polish prefix expressions into conventional infix expressions.

REFERENCES

1. Irons, Edgar T., "A Syntax Directed Compiler for ALGOL 60," Comm. ACM, 4, p. 51 (1961).
2. McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Comm. ACM, 3, p. 184 (1960).
   McCarthy, John, et al., LISP 1.5 Programmer's Manual, M.I.T. Press (1962).
3. 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.
4. Brooker, R. A., and Morris, D., "A General Translation Program for Phrase Structure Languages," Journal ACM, 9, p. 1 (1962).
5. Reynolds, John C., COGENT Programming Manual, ANL-7022, Argonne National Laboratory, March 1965.
6. Ross, D. T., "A Generalized Technique for Symbol Manipulation and Numerical Calculation," Comm. ACM, 4, p. 147 (1961).
7. Edwards, Daniel J., "LISP II Garbage Collector," unpublished.

FIGURE CAPTIONS

Fig. No.	
1	Construction tree for the polynomial "-(A+B)*DC", according to the productions given in Table 1.
2	List structure representing the polynomial "-(A+B)*DC", according to the productions given in Table 1.
3	List structures communicated between the syntax analyzer and generators (see text).
4	List structure representing the parametric constant (TERM/(FACTOR)*(FACTOR)*(FACTOR)).

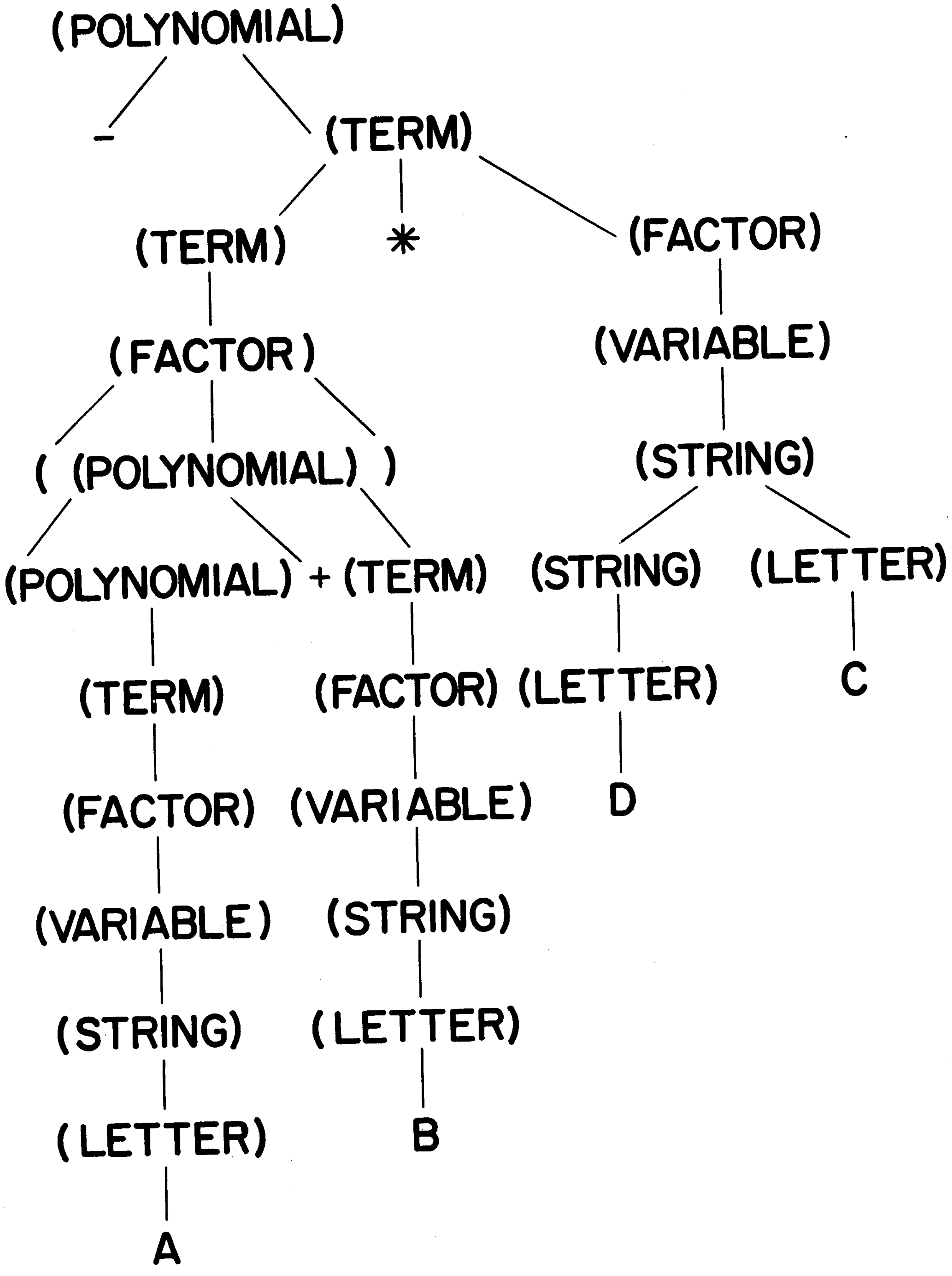
(POLYNOMIAL)
-
(TERM)
(TERM)
*
(FACTOR)
(FACTOR)
(VARIABLE)
((POLYNOMIAL))
(STRING)
(POLYNOMIAL) + (TERM)
(STRING)
(LETTER)
(TERM)
(FACTOR)
(LETTER)
C
(FACTOR)
(VARIABLE)
D
(VARIABLE)
(STRING)
(STRING)
(LETTER)
(LETTER)
B
A

Fig. 1

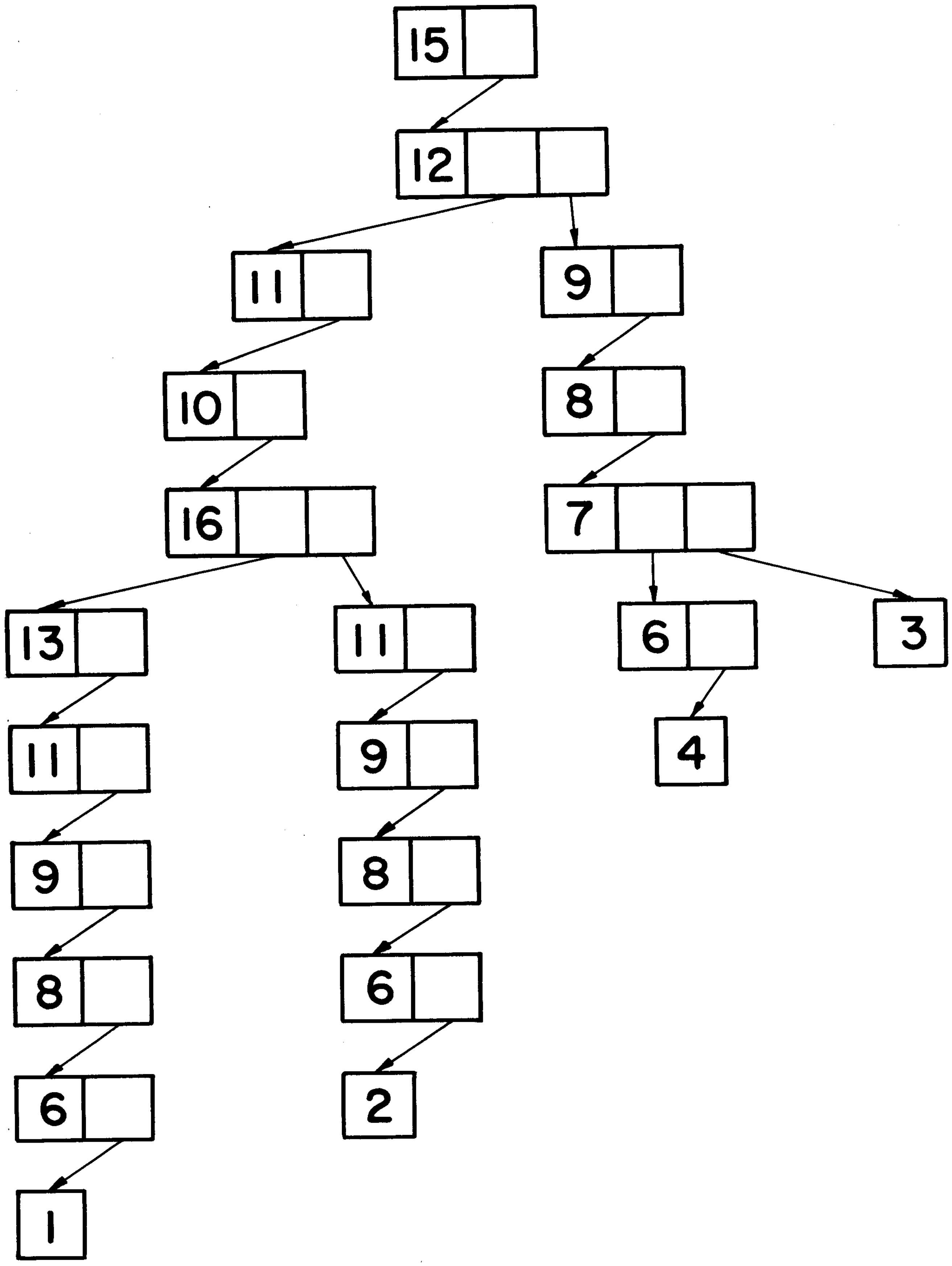

Fig. 2

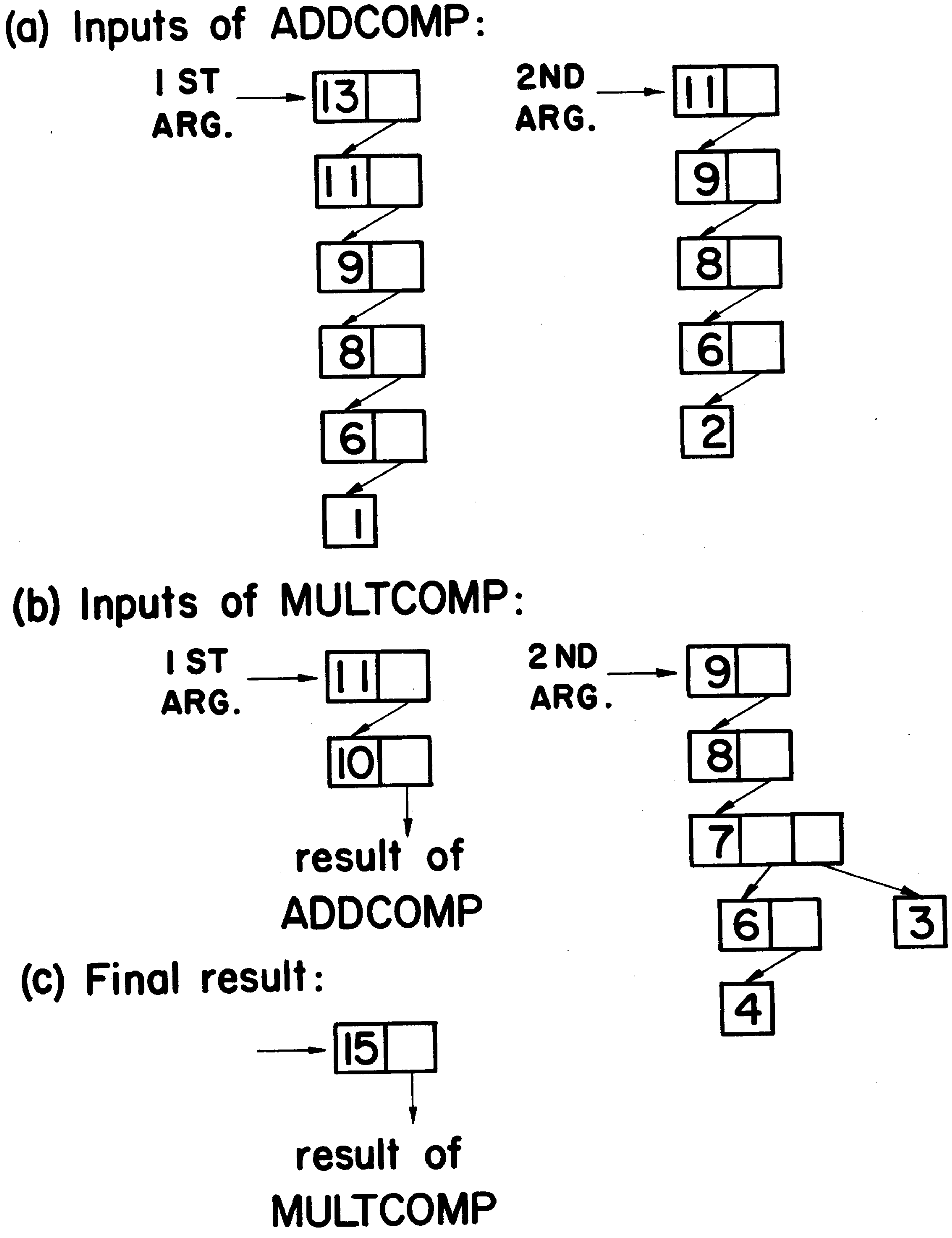
(a) Inputs of ADDCOMP:
1 ST ARG.
13
11
9
8
6
1
2ND ARG.
11
9
8
6
2
(b) Inputs of MULTCOMP:
1 ST ARG.
11
10
result of ADDCOMP
2 ND ARG.
9
8
7
6
3
4
(c) Final result:
15
result of MULTCOMP

Fig. 3

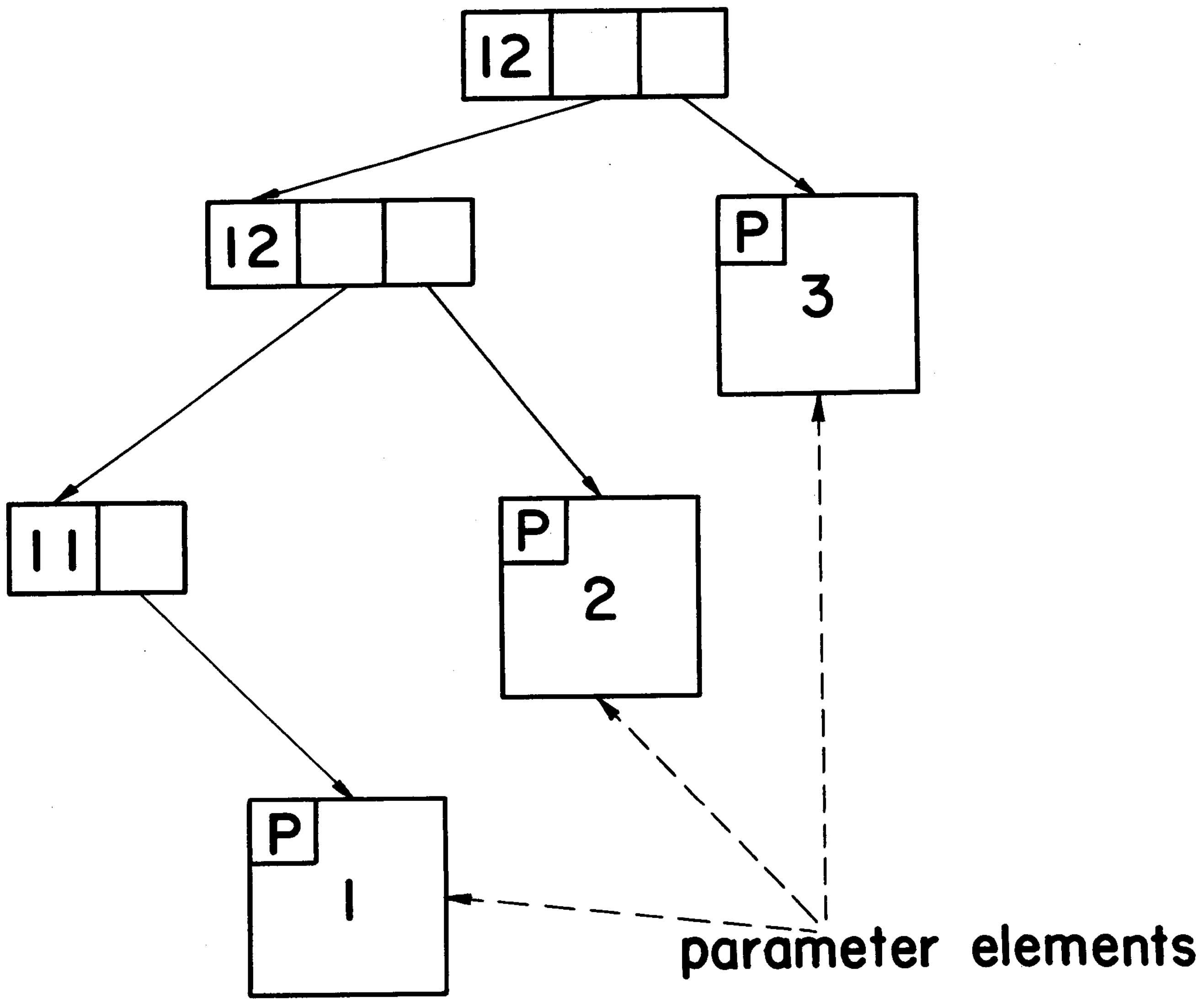

Fig. 4