A DEFINITION OF FORMULA ALGOL†

Alan J. Perlis
Renato Iturriaga††
Thomas A. Standish†††

† The research reported here was supported by the Advanced Research Projects Agency of the Department of Defense under Contract SD-146 to the Carnegie Institute of Technology.

†† Partially supported by the National University of Mexico and the Instituto Nacional de la Investigacion Cientifica.

†††National Science Foundation graduate fellow.

This paper was presented at the
Symposium on Symbolic and Algebraic Manipulation
of the
Association for Computing Machinery, Washington, D.C.
March 29-31, 1966.

ACKNOWLEDGEMENT:

We are grateful to Professor Robert W. Floyd and L. Stephen Coles for numerous helpful suggestions regarding the preparation of the manuscript.

ABSTRACT

Formula Algol is an extension to ALGOL 60 incorporating formula manipulation and list processing. This paper defines a current version of the Formula Algol language which is implemented on the CDC G-20.

1. Contents and General Description

1.1 Contents

6. Special Functions

7. History and Implementation

1.2 GENERAL DESCRIPTION

Formula Algol is an extension of ALGOL 60 [1] incorporating formula manipulation and list processing. The extension is accomplished by adding two new types of data structures: formulae and list structures, and by adding an appropriate set of processes to manipulate them. The control structure of ALGOL 60 is inherited without change. The resulting language is suitable for expressing a class of formula and list structure manipulations. Algorithms may be written to construct at run-time algebraic formulae, Boolean formulae, and list structures. Operations are available which alter or combine formulae and list structures, and which access arbitrary subexpressions. Formulae may be evaluated, substituting numerical or logical values for occurrences of variables contained within. They may be subjected to substitution processes causing the replacement of occurrences of variables by designated formulae. They may be subjected to processes of algebraic or logical transformation defined by sets of algebraic or logical rules in a form akin to Markov algorithms. Predicates are available to determine precisely the structure and composition of any formula constructible, and mechanisms are provided to extract subexpressions of a formula provided its structure is known. Likewise, predicates exist to determine the structure and composition of any list structure constructible, and mechanisms are provided to extract sublists and subexpressions. Numerical, logical, and formula values may be stored as elements in list structures and retrieval mechanisms exist to select them for use as constituents in other processes. Description lists composed of attributes and associated value lists may be attached to list structures and, in particular, to symbol variables, and processes exist for retrieving value lists and for creating, altering, and deleting attribute-value list pairs. Push down stacks of arbitrary depth are

available for the storage of list structures and, in particular, single symbols, and generators are provided in the form of new types of for statements which assign to controlled variables the elements of a single list structure, or alternatively, of several list structures in parallel, for use in an arbitrary process. Several special functions in the form of standard procedures are available for the purposes of creation of names at run-time, testing the current size of the available space list, taking a derivative of a formula with respect to a given formula variable, erasing list structures, and so on. Finally, both arrays and procedures may be defined to have formulae or list structures as values.

2. The form and symbol Declarations

In ALGOL 60 variables may be declared for each possible type of data, e.g. real, integer, Boolean. In Formula Algol, two new data types form and symbol corresponding to formulae and list structures respectively may be used to declare identifiers, arrays, or procedures. That is, lists of identifiers may be declared permitting assignment of formulae or list structures to each as values, arrays may be declared having formulae or list structures as elements, or procedures may be declared whose values are data structures of either of these two types.

When the form and symbol declarators are used in the declaration of simple variables, not only is storage reserved for each variable but a side effect occurs in which the value of each variable is initialized to the name of the variable. Thus, in the declarations form F,G and symbol S,T, the atomic formula names F,G and the atomic symbol names S,T are created and assigned as the values of F,G,S, and T respectively.

An additional property of an identifier X, declared either of type form

or of type symbol, is that X may have a description list associated with it into which attributes and values may be entered and retrieved. Furthermore, if X is of type symbol, X names a push down stack into which may be stored list structures and their degenerate cases, symbols, and data terms.

3. Formula Expressions and Symbolic Expressions

3.1 Formula Expressions

3.1.1 Syntax

<formula expression> ::= <arithmetic expression> |
<Boolean expression> | <an arithmetic expression (Boolean expression) in which some of the primaries (Boolean primaries) have been replaced by formula primaries or in which some operators have been prefixed with a dot>† | <assignment formula> |
<formula expression> <the mark "|"> <identifier>
<the mark "|"> <formula expression>

<formula primary> ::= <array formula> | <procedure formula> |
<transformed formula> | <evaluated formula> | . <identifier> |
<conditional formula>

<array formula> ::= <array identifier> . [<subscript list>]

<procedure formula> ::= <procedure identifier> .
<actual parameter part>

<transformed formula> ::= <identifier> ↓ <schema variable>

<conditional formula> ::= . if <formula expression> then
<formula expression> else <formula expression>

---

† This is a short description of what could be a formal syntactic statement.

<assignment formula> ::= <variable> . ←<formula expression>

<evaluated formula> ::= see section 3.1.4.1

3.1.2 Examples

Y3 ↑ 2 + sqrt (F)

B ∨ ~ C ∧ D

F |T| sin(C)

. if B then C else D

F . ←F + G

A . [I,J]

Taylor . (F,X,N)

. F

eval (X,Y) F (2,3)

F ↓ G

3.1.3 Semantics of Arithmetic, Boolean, Conditional, Procedure, Array, and Assignment Formulae

The process by which the value of a Formula Algol expression is obtained is explained by means of a recursively defined function called VAL. This function does not appear explicitly in the syntax of the source language, rather, it is executed implicitly at run time on each occasion in which the value of an expression is obtained. In the definition of VAL, single quotation marks placed around an expression, e.g. 'f+g', indicate that a formula construction process is to be evoked causing the creation inside the computer at run time of a data structure which represents the expression. Such data structures can be represented in many ways [e.g. trees, Polish prefix chains, etc.]. The choice of such representations is implementation dependent and is not part of the language itself. As a further notational convention in this paper, if Greek letters or syntactic classes are used inside quote marks, they are non-self-referential.

Formula Algol is a strict extension of Algol 60 with regard to values and types. Exactly as in Algol 60, each value has an associated type. In the explanation of the function VAL below, the association of a type with a value is given explicitly by writing an ordered pair of the form (<type>, <value>).

Formal definition of VAL(E) = (TYPE(E), VALUE(E)).

1. E is a constant which is either a <number> or a <logical value>.

   VALUE(E) is the conventional value of a number or a logical value (identical to that given by the Algol Report [1])

   TYPE(E) is set to integer if the number is an <integer> [1]; it is set to real if the number is a <decimal number> [1]; it is set to Boolean if E is a <logical value> [1].

2. E is of the form . <identifier>

   TYPE(E) = symbol if the <identifier> was declared of type symbol, otherwise TYPE(E) = form

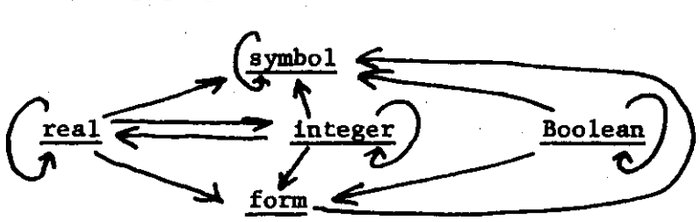
   VALUE(E) = '<identifier>'. This means that an atomic name is constructed inside the computer.

3. E is a variable

   TYPE(E) = declared type of E

   VALUE(E) = value of the last expression, say F, assigned to E by an assignment statement or by an extraction operation (c.f. section 4.1.2)

Such assignments are legal if and only if given E ← F. There is an arrow in the following graph from TYPE(F) to TYPE(E):



4. E is a function designator. Say $P(x_1,\ldots,x_n)$

TYPE(E) = the type that precedes the procedure heading in the declaration of P.

VALUE(E) = the value produced by the call of the procedure P with actual parameters: $VALUE(x_1),\ldots,VALUE(x_n)$, as defined in the Algol report.

5. (Case I) E is a binary expression of the form A <op> B where A and B are formula expressions and

<op> ::= +|-|*|/|↑|<|≤|>|≥|=|≠|∨|∧|⊃|≡

TYPE(E) is defined by the following table:

Type (A) \ Type (B)	real	integer	Boolean	form
real	T1	T1	error	T4
integer	T1	T2	error	T4
Boolean	error	error	T3	T5
form	T4	T4	T5	form

where

T1= { real if <op> is a numeric operator; Boolean if <op> is a relational operator; error otherwise

T2= { integer if <op> is a numeric operator other than /; real if <op> is /; Boolean if <op> is a relational operator; error otherwise

T3= { Boolean if <op> is a logical connective; error otherwise

T4= { form if <op> is either a numeric or relational operator; error otherwise

T5= { form if <op> is a logical connective; error otherwise

If TYPE(E) = real, integer, or Boolean then VALUE (E) = is the number or logic value obtained by carrying out the operation <op> with arguments VALUE(A) and VALUE(B). If TYPE(E) is form, then VALUE(E) is 'α <op> β' where α is VALUE(A) and β is VALUE(B).

(CASE II) E is of the form A . <op> B

TYPE(E) = form

VALUE(E) = 'α <op> β'

where α = VALUE(A) and β = VALUE(B).

Here we observe that the use of . <op> automatically causes, in all cases, the construction of a formula and prevents actual arithmetic or logical operations from being carried out.

6. (Case I) E is a unary expression of the form <op1> A where A is any formula expression and <op1>::= sin|cos|exp|ln|sqrt|arctan|sign|entier|

¬|+|-|abs

TYPE(E) is defined by the following table:

Type(A) \ <op1>	sin,cos,exp ln,sqrt,	sign entier	abs + -	¬
real	real	integer	real	error
integer	real	integer	integer	error
Boolean	error	error	error	Boolean
form	form	form	form	form

If TYPE(E) is real, integer, or Boolean then VALUE(E) is the number or logical value obtained by carrying out the operation <op1> with argument VALUE(A). If TYPE(E) is form then VALUE(E) is the expression '<op1> α' where α = VALUE(A).

(Case II) E is . <op1> A

TYPE(E) = form

VALUE(E) = '<op1> α' where α = VALUE(A)

Examples

Suppose that at a certain point in a source program that F and G have been declared of type form, that X and Y have been declared of type real, that X has been assigned the value 3.2, that Y has been assigned the value 2, that F has been assigned the value G/5, and that G has as its value its own name. Consider the following assignment statements:

a) X ← (X + Y) ↑ 2 ;
b) F ← 3 × sin(G) + (F + X) ↑ Y ;
c) F ← SQRT(F) ;

In statement (a) all variables are numeric. Thus the arithmetic expression (X + Y) ↑2 is evaluated numerically using the current values of X and Y and the result (27.04) is stored as the value of X. In statement (b), the value of F becomes the formula expression '3× sin(G) + (G/5 + 3.2) ↑2'. Finally, statement (c) replaces the value of F by a formula consisting of the SQRT of its current value, viz. 'SQRT (3X sin(G) + (G/5 + 3.2) ↑2)'. If it is desired to reassign the name 'F' to be the value of F, one may execute the assignment statement F ← . F. In general, we may use . F anywhere as a primary in any formula expression in which it is desired to refer to the name rather than the value of F.

7. E is a conditional formula of the form

. if B then A else C

TYPE(E) = form

VALUE(E) = 'if β then α else γ'

where β = VALUE(B), α = VALUE(A), and γ = VALUE(C), furthermore it is required TYPE(B) = form or Boolean, otherwise an error will result.

The conditional action represented by this formula can be executed by applying the eval operator to VALUE(E) (see section 3.1.4). Under evaluation, if eval β is true then the result is eval α; if eval β is false then the result is eval γ, otherwise the result is the conditional formula

'if ρ then α else γ' where ρ = VALUE(eval β) and where α,β, and γ are given above.

8. E is a procedure formula of the form $E = A \,.\, (X_1, X_2, \ldots, X_n)$

TYPE(E) = form

VALUE(E) = 'A . $(\eta_1, \eta_2, \ldots, \eta_n)$'

where η_i = VALUE(X_i),

A is the name of a declared procedure, and

$X_1, X_2, \ldots, X_n$ are formula expressions.

Example

Consider the assignment statement

F ← Taylor . (G,X,N) ;

where F,G,X, and N are of type form. Executing this statement causes the construction of the formula 'Taylor . (η_1, η_2, η_3)' where η_1 = VALUE(G), η_2 = VALUE(X), and η_3 = VALUE(N), and this procedure formula, stored as the value of F, represents a postponed procedure call. Applying the eval operator to F causes the procedure Taylor to be called with an actual parameter list (eval η_1, eval η_2, eval η_3) where the result of the procedure call (which procedure must be a function designator, i.e. must have a value) becomes the value of the expression eval F. Procedure formulae are the means by which representations of procedure calls may be used in constructing formula data structures. If a normal function designator is used as a primary in the construction of a formula, the value resulting from the call of the function designator is used in the construction of the formula instead of the formula representing the procedure call. E.G., F ← Taylor (G,X,N) ; causes the procedure Taylor to be called with actual parameters G,X, and N and causes the resulting value to be stored in F.

9. E is an array formula of the form

$A \,.\, [X_1, X_2, \ldots, X_n]$.

TYPE(E) = form

VALUE(E) = 'A [η_1,η_2, ..., η_n]'

where η_i = VALUE(X_i)

A is name of a declared array, and

X_1,X_2, ..., X_n are formula expressions.

Example

Executing the assignment statement

F ← A . [I,J,K] ;

causes the construction of the formula 'A [η_1,η_2,η_3]' where η_1 = VALUE(I), η_2 = VALUE(J), and η_3 = VALUE(K). This formula represents a postponed array access of the array element A [I,J,K]. Applying the eval operator to F with integer values for I,J, and K (see section 3.1.4) causes the execution of the array access. If a formula is constructed using A [I,J,K] as a primary, then the value of the subscripted variable A [I,J,K] is used in its construction. If, however, a formula representing the subscripted variable itself (in contrast to its value) is desired as a primary in a formula, then the corresponding array formula A . [I,J,K] must be used.

An important application of array formulae is the generation of names dynamically at run-time. Upon entrance to a block containing the declaration form array A[1:N] ; N array elements are created and these become available as names of storage locations for use in the construction of formulae internal to that block. Furthermore, the names of these storage locations may be used in the construction of formulae without any values having been stored into them. Later, values may be assigned to these locations and by means of the evaluation process, the values may be substituted for occurrences of the names of the locations.

10. E is an assignment formula of the form

A . ←B ,

TYPE(E) = form

VALUE(E) = 'A ←β'

where β = VALUE(B)

Example

Consider the assignment statement

F ←G. ←A + B ;

where the type of F is form and where the types of G,A, and B are any type other than symbol or Boolean. Executing this statement causes the construction of the formula 'G ←α' where α = value(A+B), and causes this formula to be assigned to F as a value. Applying the eval operator to F causes eval (α) to be stored as the value of G, and, additionally, the value of the expression eval F becomes eval (α).

3.1.4 Evaluation Rules and Evaluated Formulae

3.1.4.1 Syntax

<evaluated formula> ::= eval<variable> |

eval <bound variable list> <formula expression> <value list> |

subs (<formula expression list>) <formula expression>

<value list> |

replace (<formula expression>)

<value list> ::= (<actual parameter list>) |

(<the mark "<" > <variable> <the mark ">" >) |

<bound variable list> ::= (<formula expression list>) |

(<the mark "<" > <variable> <the mark ">" >)

<formula expression list> ::= <formula expression> |

<formula expression list>, <formula expression>

3.1.4.2 Examples

eval F
eval (X,Y,Z) F (3,4,2.5)
subs (X,Y) F (A.[N] + 6, B.(M,Q)/T)
replace (F)

3.1.4.3 Semantics

We may think of formulae as abstractions of computations. By manipulating formulae, we alter the computations they represent. At some point in the execution of a program, we may wish to carry out the computation represented by a formula. To do this, we could substitute values for occurrences of variables appearing by name only in a formula, and these values will be combined according to the computation expressed by the formula resulting in an evaluated formula. In order to accomplish the above, we have the eval operator.

If we have a formula consisting of names of formula variables joined by arithmetic operators, then, if we assign each of the formula variables a numerical value, the result of the evaluation of the formula will be a number. Hence, the evaluation of an arithmetic formula by complete substitution of numbers for formula variables is a computation carrying the set of numbers substituted into a number. Analogously, substitution of Boolean values for formula variables in a Boolean formula produces a Boolean value.

On the other hand, we need not substitute arithmetic or Boolean values for formula variables, but rather, we can substitute other formulae. This, in this case, evaluation of the formula, instead of producing a single value, expands it to an enlarged formula. Hence, eval may be used to construct formulae.

A third use of eval is that of producing trivial simplifications in a formula without altering its value and without substitution. This is done

according to the following table:

Simplifications of eval

A ↑ 0 → 1	A × 0 → 0	commutative
A ↑ 1 → A	A × 1 → A	
A ↑ -1→ 1/A	A × -1→ -A	
A ↑ -n→ 1/A↑n	A × -n→ -(A × n)	
A / 1 → A	A + 0 → A	
A /(-1)→ -A	A + (-n) → A - n	
A /(-n)→ -(A/n)	0 + A → A	
0 / A → 0	(-n) + A → A - n	
(-n) / A → -(n/A)	A - 0 → A	
	A - (-n) → A + n	
	0 - A → -A	
	(-n) - A → -(n + A)	

X ∨ true → true
X ∧ true → X
X ∨ false → X
X ∧ false → false

} commutative.

Whenever an expression contains two numeric (Boolean) arguments joined by an arithmetic (logical) operator, they are combined by eval into a numeric (Boolean) result according to the operation expressed by this operator.

A final use of eval is to carry out the array access or procedure call indicated by an array formula (see section 3.1.3) or a procedure formula (see section 3.1.3), or to carry out the assignment of a value or the choice of a value indicated by an assignment formula (see section 3.1.3) or a conditional formula (see section 3.1.3).

These uses of eval are usually combined. Thus evaluation of a formula may produce partial expansion, and some trivial simplification simultaneously. Note: All substitutions are carried out simultaneously.

I. The substitution operation

The operation subs, which evokes a substitution process, is defined as follows:

Consider a statement of the form

$$D \leftarrow \underline{subs}\ (X_1, X_2, \ldots, X_m)\ F\ (Y_1, Y_2, \ldots, Y_n) \qquad (1)$$

where $n \geq 1$ and $m \geq 1$.

Then

(a) F must be a formula expression.

(b) If TYPE(F) is numeric or Boolean or if VALUE(F) is a number or logical constant, then the effect of 1 is precisely that of $D \leftarrow F$.

(c) If TYPE(F) = form and VALUE(F) is a formula, then D will have the value obtained by substituting Y_i for each occurrence of X_i in a copy of VALUE(F), provided VALUE(X_i) is an atomic formula variable. This substitution is performed for each $i \leq m$.

(d) Y_i may be an expression of any type except symbol.

II. The evaluation process

The evaluation operator eval is defined as follows: Consider a statement of the form

$$D \leftarrow \underline{eval}\ (X_1, X_2, \ldots, X_m)\ F\ (Y_1, Y_2, \ldots, Y_n) \qquad (1')$$

Then rules (a), (b), (c), and (d) above apply without change. In addition, the resulting formula is simplified according to the table above and executing eval F where F is an assignment formula, a procedure formula or an array formula, respectively, causes the assignment to be executed, the procedure to be called, and the array element to be accessed respectively.

When evaluating a conditional formula, only if the Boolean formula in the if clause yields a Boolean value will the conditional action represented

by the formula be executed.

III. The function replace

The function designator replace(F) where F is a formula expression produces a formula which is obtained from F by replacing every atomic variable in F by its currently assigned value and by applying eval to the result. The atomic variables used in the formula F must be declared either locally or globally to the block in which replace(F) is executed.

3.2 Symbolic Expressions

3.2.1 Syntax

<symbolic expression> ::= <variable> | <function designator> |
<selection expression> | <value retrieval expression> |
<the mark "<" > <symbolic expression> <the mark ">" >

3.2.2 Examples

S
Select (L,N)
3 rd of S
color (apple)
< S >
<< S >>
last N of indexlist (M th of S)

3.2.3 Semantics

A symbolic expression is a rule for computing either a single symbol or a list as a value. This occurs according to the following rules.

1. If S is a variable declared of type symbol, the value of S is the current contents of S. When an identifier is declared of type symbol, its contents are initialized to contain the name of S (this is not true for subscripted variables). Thus, after declaration and until destroyed by an assignment statement, by a push down statement, or by an extraction, the value of S is the name of S. Executing the special assignment statement S ← .S restores the name of S to be the value of S. If a list has been stored as the contents of S by an assignment statement, then the value of S is the list. If the contents of S has been

pushed down or popped up the value of S is the symbol or list at the current top of the push down stack. If the contents is empty, the value of S is the symbol nil.

2. If S is a function designator resulting from the declaration of a symbol procedure, the value of S is that assigned to the procedure identifier by executing the body of the procedure declaration using actual parameters given in the function designator call.

3. If S is a selection expression (see section 3.2.6), then the value of S is a part of some symbolic data structure selected according to the selection rules set forth in section 3.2.6.

4. If S is a value retrieval expression, then the value of S is a function of an ordered pair of symbols (T,U) consisting of the value list associated with the attribute U on the description list attached to T (see section 3.2.5).

5. If S is a symbolic expression of the form <T>, where T is a symbolic expression, the value of T is first computed and if the result is a single atomic symbol, say 'V', the value of S is the contents of 'V' otherwise the result is a run-time error. The angular contents brackets may be nested arbitrarily many times to provide arbitrarily many levels of indirect access.

The sub-language for list processing is so arranged that anywhere an atomic symbol occurs in a statement or an expression it may be replaced by a symbolic expression which when evaluated yields an atomic symbol as a result. Further, anywhere a list may occur in a statement, it may be replaced by a symbolic expression which when evaluated yields a list as a result.

3.2.4 Lists

3.2.4.1 Syntax

<list> ::= <list element> | <list>, <list element>

<list element> ::= <expression> | <list expression> <description list>

<symbolic expression> <description list>

<list expression> ::= [<list>]

<expression> ::= <arithmetic expression> | <Boolean expression> |

<formula expression> | <formula pattern> |

<symbolic expression> | <list pattern> | <list expression>

3.2.4.2 Examples

[X + sin(Y), false, [A,B,C], F - G]
[A,E,I,O,U]

3.2.4.3 Semantics

Symbols may be concatenated into a list by writing them one after another, and by separating them with commas. This list may be assigned as the contents of another symbol by executing an assignment statement. E.g. Vowel ← [A,E,I,O,U] ; In addition to symbol variables, any expression except a designational expression may be written as an element of a list and its value will be entered. For example, let X,Y, and Z be formula variables, let A,B, and C be Boolean variables, let U,V, and W be real variables, and let R,S, and T be symbol variables. Then the assignment statement

S ← [X + sin(Y), 3 + 2×U, if B then R else T, [R,T,R], -36] ;

when executed causes each expression on the right to be evaluated, and the list of values to be sotred into the contents of S. Automatic data term conversion results from storing non-symbolic values into lists. The second from the last item in the above list is the quantity [R,T,R]. This becomes a sublist of the list stored into S. Hence, the expression stored into S is, in reality, a list structure. It is further possible for certain of the elements of a list to bear local description lists (see section 3.2.5.3).

3.2.5 Description Lists

3.2.5.1 Syntax

<description list> ::= / <attribute value list>

<attribute value list> ::= <attribute value segment> |

<attribute value list> <attribute value segment>

<attribute value segment> ::= [<attribute> : <list>] |

[<attribute> : <empty>]

<value retrieval expression> ::= <identifier> (<symbolic expression>) |

the <attribute> of <symbolic expression>

<attribute> ::= <symbolic expression> | <formula expression>

3.2.5.2 Examples

Description lists

/[types: mu, pi, rho][color: green][processed: true]
/[properties: continuous, differentiable]

Value Retrieval Expressions

color (apple)

the ancestor of the leftrelative of <S>

3.2.5.3 Semantics

A description list is a sequence of attributes and values. An attribute may be any atomic symbol or any formula. The value of any type of expression except a designational expression may be used as a value. Each attribute is followed by a list of values associated with it. This value list may contain more than one member, it may contain only one member, or it may be empty. A description list may be attached to one of three types of objects:

1. A variable declared of type symbol for which there are two cases (a) global attachment, and (b) local attachment.
2. A variable declared of type form.
3. A sublist of a list.

Assignment statements are used to construct and to attach description lists. For example, assuming that all variables involved have been declared of type

symbol, the statements

```
S ←/[types: mu, pi, rho][ancestors: ortho1, para5][color: green] ;   (1)
T ←[F, A/[mark:1], B,C, A/[mark:2], D,E] ;                          (2)
```

assign respectively a description list to S and a list as the contents of T. The description list attached to S is globally attached meaning that it is permanently bound to S for the lifetime of the variable S which lifetime is determined by the ALGOL block structure in which S occurs. In the list assigned as the value of T, the symbol A occurs twice in the second and fourth positions. The description lists attached to these two separate occurrences of A are attached locally meaning that the separate occurrences of a given atomic symbol within a list have been given descriptions which interfere neither with each other nor with the global description list attached to A if such should occur, and that the attributes and values of a given local description list are accessible only by means of symbolic expressions accessing the particular occurrence of the symbol to which the given local description list is attached.

In the following examples, suppose F is a variable declared of type form and that all other variables involved are variables declared of type symbol.

```
F ←/[properties: continuous, differentiable] ;                      (3)
V ←[A, [B,C]/[processed: true],A, [B,C]/[processed: false],A] ;     (4)
```

In example (3), a description list is attached to a formula. In example (4), the list assigned to be the contents of V has two identical sublists [B,C] in the second and fourth positions having different local description lists.

Value lists stored in description lists are retrieved by means of value retrieval expressions. To accomplish retrieval two arguments must be supplied:

(1) an attribute consisting of an atomic symbol or a formula, and

(2) an atomic symbol or a position in a list structure having a description list.

The attribute is then located on the description list if it is present and its associated value list, if any, becomes the value of the retrieval expression. If there is no description list, or if there is a description list but the attribute does not appear on it, or if the attribute does appear on it but has an empty value list, then the value of the retrieval expression is the symbol nil. Thus, in examples (1) and (2) above, the value retrieval expressions color (.S), mark (2 nd of T), and mark (3 rd of T) have the values green, 1, and nil respectively. The construction, the color of .S, accomplishes the same function as color (.S) but is slightly more versatile in that any symbolic or formula expression may be used to calculate the attribute whereas only identifiers may be used for the attribute in the form <identifier> (<symbolic expression>). Thus, for example, the expression the 3 rd of types(.S) of <P>. is a legal value retrieval expression whose attribute is calculated by selecting the third element of the list which is the value of the expression types (.S). In example (1) above, if it is desired to access attributes of the global description list of the second element A, instead of accessing only its local description list, then the element must be selected by means of a selection expression, stored into a variable, and an indirect access of the global description list performed. E.g. X ← 2 nd of T; P ← mark (X); are two statements which, in the case of example (2) above, extract the name 'A' and store it as the value of X, and which then access the value list associated with the attribute 'mark' on the global description list of the value of X (i.e., the global description list of A). The result is stored in P.

3.2.6 Selection Expressions

3.2.6.1 Syntax

<selection expression> ::= <selector> of <symbolic expression>

<ordinal suffix> ::= st | nd | rd | th

<ordinal selector> ::= <arithmetic primary>␣<ordinal suffix> | last

<elementary position> ::= <ordinal selector> | <ordinal selector>
<class name> | <ordinal selector> <expression> |
<ordinal selector> <augmented type> | <ordinal selector> integer
<arithmetic primary>

<position> ::= <elementary position> | <arithmetic primary>␣
<ordinal suffix> before <elementary position> |
<arithmetic primary>␣<ordinal suffix> after <elementary position>

<selector> ::= between <position> and <position> | all after <position> |
all before <position> | first <unsigned integer> |
last <unsigned integer> | <position> | all <expression> |
all <augmented type> | all <class name>

<augmented type> ::= real | integer | Boolean | form | symbol
sublist | text | atom | any

3.2.6.2 Examples

3 rd of S
last of S
N th real of S
last sublist of S
last [A,B,C] of S
5 th (|trigfunction|) of S
N th before last Boolean of S
all symbol of S
last 3 of S

3.2.6.3 Semantics

Selection expressions are formed by composing selector operators with symbolic expressions. A symbolic expression is first evaluated producing a symbolic data structure as a value. A selector operator is then applied to the resulting symbolic data structure to gain access to a part of it. Assume first that the symbolic data structure, S, on which a selector operates is a simple list. Then

1. An ordinal selector refers to an element of this list either by numerical position, i.e., the n th element, or by designating the last element. E.g. 3 rd of S, last of S.

2. An elementary position refers to an element of this list by designating it (a) as the n th or last instance of an augmented type, e.g., n th real, last sublist, (b) as the n th or last instance of an expression, e.g., n th (F + G), last [A,B,C], (c) as the n th or last instance of a member of a class, which class may be defined as consisting of any arbitrary Boolean test on an element (see section 4.2.6), e.g., 5 th (|trigfunction|), last (|vowel|), (d) or as the n th or last, i.e., by ordinal selection.

3. A position refers to an element of this list either by designating its elementary position or by designating it as the n th before or the n th after some elementary position.

4. A selector refers to an element by its position or else designates one of the following sublists of the list

   a) The sublist between two positions, e.g., between 3 rd and 7 th of S.

   b) The sublist consisting of all elements before or after a given position, e.g., all after 3 rd symbol of S, all before last real of S.

   c) The sublists consisting of the first n elements or the last n elements, e.g., first 3 of S, last k of S.

   d) The sublists consisting of (i) all instances of a given expression, e.g., all F of S, (ii) all instances of a given augmented type, e.g., all real of S, (iii) all instances of elements which are members of a given class, e.g., all (|trigfunction|) of S. The elements of the sublists so composed occur in the same order that they occur in the list from which they are selected.

Selectors may be compounded to access sublists and their elements. Suppose the statement S ← [A, [X,X, [A,A], X], A] has been executed. Then the expression 2 nd of S is a list valued symbolic expression with the list [X,X, [A,A], X] as value, whereas the expression 3 rd of 2 nd of S has the list [A,A] as value, and whereas the expression last of 3 rd of 2 nd of S has the single atomic symbol A as value.

It is possible for selectors to refer to elements or sublists which do not exist. For example, suppose the statement S ← [A,B,C]; has been executed. Then the expression 5 th of S refers to an element which doesn't exist. The value of such an expression is the symbol nil. Similarly, the expression first 5 of S refers to a sublist which doesn't exist. The value of this expression is the list [A,B,C,nil,nil]. Generally, the rule is (1) if a selection expression refers to a single element which doesn't exist, the value of the expression is the symbol nil, and (2) if a selection expression refers to a sublist required to contain more elements that are available in the list structure being accessed, then the symbol nil is repeatedly appended to the end of the insufficient structure until it is of requisite length.

4. Predicates for Formulae and List Structures

4.1 Formula Patterns

4.1.1 Syntax

<formula pattern> ::= <formula expression> == <formula pattern structure> |
<formula expression> >> <formula pattern structure> |
<extractor> <formula expression> >> <extractor> <formula pattern structure>

<extractor> ::= <variable> :

<formula pattern structure> ::= <a formula expression in which some of the primaries may have been replaced by pattern primaries and some of the operators may have been replaced by operator classes> †

<formula pattern primary> ::= <type> | atom | any | of (<variable>) | of (<procedure identifier>) | (<formula pattern structure>) | <extractor> <formula pattern primary>

<operator class> ::= <the mark "|"> <operator class name> <the mark"|">

<operator class name> ::= <variable>

<operator class assignment> ::= <operator class name> ← /[operator: <operator list>] <operator attribute list>

<operator list> ::= <operator> | <operator list>, <operator>

<operator attribute list> ::= <empty> | [comm: <logical value list>] | [index:<variable>] | [comm:<logical value list>][index:<variable>]

<logical value list> ::= true | false | <logical value list>, true | <logical value list>, false

4.1.2 Semantics

A computation may construct a formula whose structure cannot be predicted in advance or a situation may arise in a program where it is desired to discriminate among various formulae in a given class depending on their various properties. For this a mechanism is needed to determine precisely the structure of any given formula. Formula patterns are used for this purpose and they constitute a set of predicates over the class of formula data structures. These formula patterns are sufficient in the sense

---

† This is a short description of what could be a formal syntactic statement.

that whatever constructions are used to create a formula, the process may be reversed by the choice of a sequence of predicates. Furthermore, a given formula pattern may be used to represent a class of possible formulae, and any formula may be tested for membership in this class.

In the definition of a formula pattern, a formula expression, F, is compared with a formula pattern structure, P, to determine one of two things (1) corresponding to the construction F==P, whether the expression F is an exact instance of the formula pattern structure P or, (2) corresponding to the construction F>>P, whether the formula expression F contains an instance of the formula pattern structure P. Both constructions F==P and F>>P are Boolean expressions having values true or false.

The Construction F==P

The formula expression F is defined recursively to be an exact instance of the formula pattern structure P as follows:

1. If P is a type word: real, integer, Boolean, form, or symbol, then F==P is true if and only if the value of F is a real number, an integer, a logical value, a formula, or a list structure respectively.

2. If P is the reserved word atom, then F==P is true if and only if the value of F is either a number, a logical value, or an atomic formula name.

3. If P is the reserved word any, then F==P is always true.

4. If P is the construction of (<variable>) where the variable, say S, must be declared of type symbol, and where S has been assigned as a value, a list of formula pattern structures, say $[P_1, P_2, \ldots, P_n]$, then F==P is true if and only if $F==P_1 \vee F==P_2 \vee \ldots \vee F==P_n$ is true.

5. If P is the construction of (<procedure identifier>) where the procedure identifier names a Boolean procedure with one formal parameter specified of type form, for example, Boolean procedure B(X); form X; <procedure body>, then F==P is true if and only if the procedure call B(F) yields the value true.

6. If P is of the form A_1 <op>$_1$ B_1, then F==P is true if and only if (a) F is of the form A_2 <op>$_2$ B_2, (b) A_2==A_1, (c) B_2==B_1, and (d) if <op>$_1$ is a single operator then <op>$_2$ must be identical to <op>$_1$ whereas if <op>$_1$ is an operator class, then <op>$_2$ must be a member of <op>$_1$ as defined below. Similarly, for unary operators, if P is of the form <op>$_1$ B_1, then F==P is true if and only if (a) F is of the form <op>$_2$ B_2 and conditions (c) and (d) above are true.

7. If respectively P is of the form
   (a) <array identifier> . [S_1,S_2, ..., S_n]

   (b) <procedure identifier> . [S_1,S_2, ..., S_n]

   (c) <variable> . $\leftarrow S_1$

   (d) . if S_1 then S_2 else S_3

   where S_1,S_2, ..., S_n are formula pattern structures, then

   F==P is true if and only if respectively
   (a) F is an array formula with the same array identifier as P and with a subscript list whose successive elements are instances of S_1,S_2, ..., S_n.
   (b) F is a procedure formula with the same procedure identifier as P and with an actual parameter list whose successive elements are instances of S_1,S_2, ..., S_n.
   (c) F is an assignment formula with the same left part variable as P and with a right hand expression which is an instance of S_1.
   (d) F is a conditional formula of the form . if B then C else D and B==S_1, C==S_2, and D==S_3.

Extractors

Assume for some P and some F that F==P is true. If an extractor is used in P preceding a formula pattern primary, then the subexpression in F which matches the formula pattern primary preceded by the extractor is assigned as the value of the variable found to the left of the colon in the extractor.

Operator Classes and Commutative Instances

Before an operator class is used in a formula pattern, it must be defined. The definition is accomplished by an operator class assignment which assigns to a variable, which must be declared of type symbol, a description list of the form

/[operator: <operator list>] <operator attribute list>

Suppose R is a variable declared of type symbol for which the following operator class assignment has been executed:

R ← /[operator: +, -, /][comm: true, false, false][index: J]

where J must be a variable declared of type integer and where operator, comm, and index are reserved words used for special attributes. Let P be a formula pattern structure having the form

A_1 | R | B_1 .

Then F==P is true if and only if (a) F is of the form A_2 <op>$_2$ B_2, and (b) one of the two following conditions hold:

(i) A_2==A_1, B_2==B_1, and <op>$_2$ is a member of the operator value list found on the description list of R. In the specific case above, this list is [+,-,/].

(ii) B_2==A_1, A_2==B_1, and <op>$_2$ is a member of the list of operators obtained from the operator value list by deleting those operators whose corresponding logical values in the logical value list following the attribute comm are false. (In the specific case above, this reduced operator list is the list consisting of the single operator +). Thus commutative instances about + are considered, but not commutative instances about -, or /. Note that [comm: true, false, false] need not appear on the description list of R at all in which case no commutative instances of any operator will be considered.)

If F==P is true, the integer variable used as value of the attribute index will be set to an integer denoting the position of $<op>_2$ in the operator value list. (In the specific case above, J is set to 1,2, or 3 according to whether $<op>_2$ was +,-, or / respectively.) The operator $<op>_2$ is stored as a data term as the value of R. Later the construction |[R]| can be used in an expression in place of an operator, and the operator $<op>_2$ extracted during the previous matching will be used in the construction of the formula data structure that the expression represents. Alternatively, R may be assigned any operator by the assignment statement R ←<operator> ; and |[R]| may be used in the same fashion.

The Construction F>>P

The formula pattern F>>P is true if F contains a subexpression, say S, (which may be equal to F itself) such that S==P is true. A recursive process is used to sequence through the set of sub-expressions of F for successive testing against the formula pattern structure P. The sequencing has the properties that if two sub-expressions S_1 and S_2 are both instances of P, then if S_2 is nested inside S_1, then S_1 will match P first, and if neither S_1 nor S_2 is nested inside the other, then if S_1 occurs to the right of S_2 in a linearized written form of S, then S_1 is recognized before S_2.

The formula pattern A:F>>B:P in which extractors precede the right and left hand sides of the formula pattern has the following meaning. First, F>>P is tested. If the result is true, then (a) the sub-expression of F which matches P is stored as the value of B, and (b) a formula is constructed consisting of F with the sub-expression matching P replaced by the previous value of B (viz. the value B had before the assignment described in (a) took place). This formula is stored as the value of A.

4.1.3 Examples

4.1.3 Examples

Example 1. Let A,B,X,Y, and Z be declared of type form, and let R be declared of type real. Suppose that the statement

X ← 3 × sin(Y) + (Y - Z) / R + 2 × R ;

has been executed. Consider the statement:

if X>> A:integer × B: sin(form) then Z ← 2 × B + A ;

Since the pattern X>>A:integer × B: sin(form) is true, the assignment Z ← 2 × B + A will be executed, assigning as the value of Z the formula 2 × sin(Y) + 3 because A has the value 3 and B has the value sin(Y).

Example 2. Let X be of type symbol, A,B,Y,M,T,G, and P be of type form, and D be of type Boolean. Then executing the statements:
X ← [real, integer, Boolean] ; G ← Y + 8×(M - T) ;
P ← form + A: of(X) × B:form ; D ← G==P; causes D to be set to true because the pattern G==P is true, and causes A to be set to 8 and B to be set to M - T.

Example 3. Suppose we execute the statements

F ← 2 ×(sin (X↑2 + Y↑2) + cos (X↑2 - Y↑2)) / 5; G ← sin(form) + cos(form) ;

where all variables used are of type form. Then A:F>>T:G is a pattern with value true. The value of T will replace the first instance of G in F, i.e., the expression sin(X↑2 + Y↑2) + cos(X↑2 - Y↑2) (this being the first sub-expression matching the pattern G according to the sequencing priorities defined above). A is assigned the expression 2 × T / 5. Thus A is the same as F with the first sub-expression of F matching G replaced by the value of T.

Example 4. Assume all variables in the following sequence of declarations and statements are of type form.

```
Boolean procedure HASX(F); value F; form F; HASX ← F>>X;
    G ← (X↑2 + 3) ↑2 × (Y - 1) ; F ← A:of(HASX) × B:(any-1) ;
    T ← G==F ;
```

Then T is set to true, A is set to (X↑2 + 3) ↑2, and B is set to Y-1. Here we see that any Boolean procedure may be used in a formula pattern to test the properties of a matching sub-expression of a formula. The full generality of Boolean procedures is thus delivered.

4.2 List Patterns

4.2.1 Syntax

<list pattern> ::= <symbolic expression> == [<list pattern structure>] |
    <list expression> == [<list pattern structure>] |
    <symbolic expression> == <list expression> |
    <symbolic expression> == <symbolic expression> |
    <list expression> == <list expression>

<list pattern structure> ::= <list pattern primary> |
    <list pattern structure>, <list pattern primary>

<list pattern primary> ::= $ | $ <arithmetic primary> | <expression> |
    <class name> | <augmented type> | <list pattern primary>
    <description list> | <extractor> <list pattern primary> |
        [<list pattern structure>]

<extractor> ::= <variable> :

4.2.2 Semantics

List patterns are predicates for determining the structure of lists. List patterns use the mechanisms found in COMIT [5] to test whether a linear list is an instance of a certain linear pattern. The list pattern structure describes the pattern being tested for, and is composed of a sequence of list pattern primaries separated by commas. The symbols $ and $ n may be used as list pattern primaries with the same significance as in

COMIT (viz. $ stands for any arbitrary number of consecutive arbitrary elements and $ n stands for n consecutive arbitrary elements). If a symbolic expression is used as a list pattern primary, its value is first computed, and if that value is a list, each element of the list becomes one of the consecutive list pattern primaries in the list pattern structure. Other kinds of elements introduced below may also become list pattern primaries.

A list pattern compares a list (determined by either (1) a list expression, or (2) a list valued symbolic expression) to a linear pattern (described by either (1) a list expression, (2) a list valued symbolic expression, or (3) a list pattern structure) to see if the list is an instance of the pattern. The list pattern is a Boolean primary with values true and false and thus may be combined with other Boolean expressions by means of logical operators.

4.2.3 Examples

Example 1. Suppose the statement S ← [A,B,C,D] has been executed, where all variables involved have been declared of type symbol, and where the values of A,B,C, and D are their respective names. Consider the statement

if S == [$1, B, $] then T ← [T,B] else T ← [T, last of S] ;

Since the contents of S, which is the list [A,B,C,D] is an instance of the pattern [$1, B, $] (which is read "a single arbitrary constituent, followed by a B, followed by any arbitrary number of arbitrary constituents"), the list pattern S == [$1, B, $] is true. Therefore, T ← [T,B] is executed, which has the effect of appending a B to the end of the list stored as the value of T.

As with the formula pattern structures used both as predicates and selectors for formulae, list pattern structures may function not only as predicates but also as selectors. The same mechanism is used to accomplish this. If any list pattern primary in a list pattern structure is preceded by a variable declared of type symbol followed by a colon, then the corresponding element in the list being tested, in the event there is a match, becomes the value of that symbol variable. The value may be accessed at any later point in the program.

Example 2. As in the previous example, suppose the statement S ← [A,B,C,D] has been executed where all variables are symbols and where A,B,C, and D have as values their respective names. Then, executing the statement

if S == [T:$2, V:$2] then S ← [V,T] ;

changes the contents of S to be the list [C,D,A,B]. Furthermore, the contents of T has as its value the list [A,B], and V has as its value the list [C,D].

4.2.4 Equality Tests

If we have two symbolic expressions, we may test whether their values are equal by means of the relation <symbolic expression> == <symbolic expression>. The values of the symbolic expressions may be single symbols, lists of symbols, formulae, or values of any other type. Naturally, if the values of the two symbolic expressions are non-conformable data structures, the result of the predicate will be false. Similarly, two list expressions may be tested for equality, as may a symbolic expression and a list expression.

4.2.5 Testing for Types

A single valued symbolic expression having a value whose type is unknown may be used in the list pattern <symbolic expression> == <augmented type> in order to determine the type. An augmented type is either real, integer, Boolean, form, symbol, sublist, text, atom, or any. Here the type text is

assigned to any Formula Algol reserved word entered in quotation marks as an element of a list. E.g., S ← ['if',B, 'then', C] where 1 st of S == text is true and where 3 rd of S == text is true. The type atom is true for atomic formulae, numbers, and logical values, and type any is true for any arbitrary element not of type symbol.

4.2.6 Testing for Membership in a Class

Class Definitions

4.2.6.1 Syntax

<class name> ::= (|<symbolic expression>|)

<class primary> ::= <class name> | [<class expression>]

<class secondary> ::= <class primary> | ~ <class primary>

<class factor> ::= <class secondary> | <class factor> ∧ <class secondary>

<class expression> ::= <class factor> | <class expression> ∨ <class factor>

<class definition> ::= let <class name> = [<formal parameter>
<the mark "|"> <Boolean expression>] | let <class name> =
<class expression>

4.2.6.2 Semantics and Examples

Sets may be defined by means of class definitions. For example, suppose the statement V ← [A,E,I,O,U] has been executed. Then the statement let (|vowel|) = [X | Among (X,V)] ; defines the set of all vowels where Among(P,Q) is a Boolean procedure which is true if P is an element of the list contained in Q, and false otherwise. Suppose, now, that having sometime previously executed the statement S ← [A,B,C], we execute the statement

if 1 st of S == (|vowel|) then delete S ;

The list pattern 1 st of S == (|vowel|) will be evaluated by first computing the value of the expression 1 st of S, which is the symbol A, and second by

substituting A for the formal parameter X in the class definition of (|vowel|). This results in the Boolean procedure Among (A,V) being executed, the value of which is true. Thus, A is a member of the class (|vowel|), and the list pattern 1 st of S == (|vowel|) is true. This causes the statement delete S to be executed which erases the value of S.

Class definitions may consist of Boolean combinations of other defined classes. E.g., let (|A|) = (|B|) ∧ (|C|) ; is legal provided class (|B|) and class (|C|) are elsewhere defined. Another example of a class definition would be let (|empty|) = [X|false] ; This defines the empty set. Note: Any arbitrary Boolean expression including a Boolean procedure call may be used to define a class. Thus the full generality of Boolean procedures is delivered.

Class definitions may be used as list pattern primaries in list pattern structures. When this is done, the element matching the class definition is tested for membership in the class. If the result is true, the list pattern structure continues to be matched against the list being tested. If the result is false, the list pattern structure fails to match the list being tested. E.g. S == [D, (|vowel|), $] is a legal list pattern which tests the list which is the value of S to see if it is of the form D, followed by a vowel, followed by an arbitrary number of arbitrary constituents. Similar to formula pattern structures, list pattern structures may be stored as the value of a variable for use in list patterns. E.g., the statements S ← [$1, B, $] ; T ← [A,B,C,D] ; if T == S then go to exit ; are equivalent to the statement if [A,B,C,D] == [$1, B, $] then go to exit ;

Subpatterns are permitted as list pattern primaries so that list structures may be tested. E.g., [A, $2, symbol, [B,$],D] is a legal list pattern structure.

5. Other Kinds of Statements and Expressions

5.1 Push Down and Pop Up Statements

5.1.1 Syntax

<push down operator> ::= ↓ | <push down operator> ↓

<pop up operator> ::= ↑ | <pop up operator> ↑

<push down statement> ::= <push down operator> <symbolic expression>

<pop up statement> ::= <pop up operator> <symbolic expression>

5.1.2 Examples

↓ S

↓↓↓S

↑ S

↑↑ S

↓ 3 rd of indexlist(.S)

5.1.3 Semantics

The contents of any variable declared of type symbol is a push down stack. The value of a variable consists of the current contents of the topmost level of the push down stack. Assignment statements using symbol variables on the left replace the current contents of the topmost level of its push down stack. Applying a single push down operation, ↓, to the name of such a variable pushes down each level of the stack making the topmost level (level 0) empty and replacing the contents stored at level k with the contents stored previously at level k-1, for k = 1,2, ..., maxlevel + 1. The empty topmost level may then acquire a value as its contents by means of the execution of an appropriate assignment statement. A lower level of the push down stack is inaccessible to the operation of extracting contents until the execution of a pop up statement restores it to the topmost level. Applying a single pop up operator, ↑, to the name of a variable destroys the contents of the topmost level (level 0) and replaces

the contents stored at level k with the contents stored previously at level k + 1, for k = 0, 1, ..., maxlevel-1. A push down operator (pop up operator) consisting of n consecutive occurrences of a single push down operator (pop up operator) has the same effect as n consecutive applications of a single push down operator (pop up operator). A push down operator (pop up operator) is applied to a symbolic expression by (1) evaluating the symbolic expression and (2) determining if the result is an atomic symbol or not. (3) If not, nothing is done. (4) If so, the operator is applied to the push down stack named by the atomic symbol as described above. Whatever structure occupies the contents of a symbol variable, S, may become the contents of a lower level of the push down stack in S by application of the push down operator to S. In particular, list structures may be stored in the push down stack in S.

5.2 Additional Types of For Statements

5.2.1 Syntax

<for list element> ::= ... | <symbolic expression>

elements of <symbolic expression> |

attributes of <symbolic expression>

<for clause> ::= ... | for <symbolic expression> ← <for list> do |

parallel for [<formal parameter list>] ←

elements of [<symbolic expression list>] do |

parallel for [<symbolic expression>] ←

elements of [<symbolic expression>] do

5.2.2 Examples

```
for list elements
    S
    attributes of S
    elements of S
```

for clauses

for S ← 1, true, F+G, <T>, last of T1 do
for S ← elements of <T> do
for S ← attributes of T do
parallel for [I,J,K] ← elements of [[S], [T], [U]] do

5.2.3 Semantics

We may wish to generate the elements of a list or the attributes of a description list one by one in order to assign them to the controlled variable in a for statement. For this purpose, the for list elements, attributes of S, and elements of S, are introduced. Here, attributes on the description list of the value of S, which must be an atomic symbol, are generated in the order that they occur by attributes of S, and elements of S, generates the successive elements of the list which is the value of S. In the former case, S may be any symbolic expression with an atomic symbol as value. In the latter case, S may be any list valued symbolic expression. Successive elements generated are assigned to the controlled variable given in the for clause.

Parallel generation is also permissible. For example: if S ← [A,B,C], T ← [D,E] and U ← [F,G,H,I] have been executed where the variables A through I have as values their respective names, then executing the statement

```
parallel for [I,J,K] ← elements of [ [S], [T], [U] ] do
        L ← [L,I,J,K] ;
```

causes the following to happen. First, all first elements of the lists contained in S,T, and U respectively are generated and placed in the contents of the controlled variables I,J, and K respectively. Control then passes to the body of the parallel for statement and returns when finished with its execution. On the second cycle, all second elements of S,T, and U are

generated and placed in the controlled variables I,J, and K respectively. Control then passes to the statement following the do and returns. On the third cycle, all third elements are generated, on the fourth cycle, all fourth elements are generated, and so on. If any list runs out of elements before any of its neighbors, the symbol nil keeps getting generated as the n th element of that list whenever n exceeds the number of elements on the list. The parallel generation stops on the first cycle before the symbol nil would be generated from all lists. The number of controlled variables is arbitrary but must be the same as the number of lists designated in the symbolic expression list.

List valued symbolic expressions may be used to supply lists of controlled variables and lists of lists to generate in parallel, as, for example, in the construction

parallel for [V] ← elements of [W] do

where the statements V ← [I,J,K] and W ← [[S], [T], [U]] have been executed previously. At the end, L should contain [L,A,D,F,B,E,G,C, nil, H, nil, nil, I].

5.3 Editing Statements and Description List Editing Statements

5.3.1 Syntax

<editing statement> ::= insert <list expression> <insertion locator part> <symbolic expression> | delete <selector part> of <symbolic expression> | delete <symbolic expression> | alter <selector part> of <symbolic expression> to <expression> | <description list editing statement>

<insertion locator> ::= before <position> of | after <position> of

<insertion locator list> ::= <insertion locator> | <insertion locator list>, <insertion locator>

<insertion locator part> ::= <insertion locator> |

(<insertion locator list>)

<selector list> ::= <selector> | <selector list>, <selector>

<selector part> ::= <selector> | (<selector list>)

<description list editing statement> ::= the <symbolic expression>

of <symbolic expression> <is phrase> <expression>

<is phrase> ::= is | is not | is also

5.3.2 Semantics

Editing statements are used to transform, permute, alter, and delete elements of lists. The insert construction causes a list structure given by a list expression to be inserted at the list of places in a given list specified by an insertion locator part. The list on which insertion is to be performed is obtained by evaluating the symbolic expression found as the last item in the construction. All the insertions take place simultaneously. The first delete construction given in the syntax equation for editing statements above performs simultaneous deletions of a list of parts within the list obtained by evaluating the symbolic expression. The list of parts to be deleted is specified by the selector part in accord with the semantics of selectors. The second delete construction deletes the symbolic expression and is equivalent to an erase command. The alter construction is the same as the first delete construction except it replaces each item of the list of parts deleted with an arbitrary expression.

5.3.3 Examples

Suppose S ← [X,A,A,X] has been executed. Then the statement: insert Y before last of S; changes the value of S to look like [X,A,A,Y,X]. Similarly, the statement: insert [[Y,Z]] (after 1 st of, before last of) S ; changes the value of S to look like [X, [Y,Z], A,A, [Y,Z], X]. The statement: delete 3 rd before last of S; alters the value of S to look like

[A,A,X], and delete all A of S; causes the value of S to be changed to [X,X]. In a similar vein, the statement: alter all A of S to [[C,C]]; changes the value of S to look like [X, [C,C], [C,C], X].

5.3.4 Description List Editing Statements

Description list editing statements add or delete values on description lists. They supplement the role performed by assignment statements in this regard. Suppose that S ←/ [types: mu,pi,rho] [color:red] has been executed. Then, if the statement: the color of S is green ; is executed, the value of the attribute 'color' on the description list of S is replaced with the new value 'green'. This yields the altered description list / [types: mu,pi,rho] [color:green] as a result. On the other hand, the statement: the color of S is also green ; could be executed. Instead of replacing the color 'red' with the value 'green', the latter statement appends the value 'green' to the value list following the attribute 'color'. This yields the description list / [types: mu,pi,rho] [color: red,green] as a result. Finally, description list editing statements may be used to delete values from value lists of a specific attribute. Executing the statement: the types of S is not pi; alters the above description list to be of the form /[types: mu,rho] [color: green].

5.4 Transformed Formulae

5.4.1 Syntax

<transformed formula> ::= <formula expression> ↓ <schema variable>

<schema variable> ::= <variable>

<schema assignment> ::= <schema variable> ← [<schema>]

<schema> ::= <schema element> | <schema>, <schema element>

<schema element> ::= <variable> | <single production> |

<parallel production>

<single production> ::= <formula pattern structure>[1] → <formula expression> |

<formula pattern structure>[1]. → <formula expression>

<parallel production> ::= [<parallel elements>]

<parallel elements> ::= <variable> | <single production> |

<parallel elements>, <variable> |

<parallel elements>, <single production>

5.4.2 Examples

Transformed Formula

F ↓ S

Single Production

A: form × (B:form + C:form) → .A × .B + .A × .C

Schema Assignment

S ← [P1 → R1, [P2 → R2, P3 → R3], P4 → R4]

A complete example is given at the end of the discussion of the semantics.

5.4.3 Semantics

This section uses the concepts of description lists and formula patterns discussed in sections 3.2.5 and 4.1 respectively.

Let F and G be formulae, and let P be a formula pattern. The application of the production P → G to the formula F is defined as follows:

1. If F==P is false (see section 4.1.2) then the application is said to fail.

2. If F==P is true, then the application is said to succeed, and F is transformed into the value of the expression replace(G). As explained in section 4.1.2, if F==P is true, and if P contains extractors, sub-expressions of F matching corresponding parts of P are

---

1. For the definition of <formula pattern structure>, see section 4.1.1.

assigned as values of the extractors. Furthermore, if the names of the extractor variables are used as atoms in the construction of G, then executing the procedure replace(G) substitutes these extracted sub-expressions for occurrences of the names of the extractor variables causing as a result a rearrangement of the sub-expressions of F into the form expressed by the structure of G.

For example, the distributive law of multiplication over addition may be executed as a transformation by applying the production

A: any × (B: any + C:any) → .A × .B + .A × .C (1)

to a given formula. Suppose F ← X↑2 × (Y + sin(Z)). Then applying the production (1) to F will result in the extraction of the sub-expressions X↑2, Y, and sin(Z) into the variables A,B, and C respectively, and will cause the replacement of the atomic names A,B, and C occurring on the right hand side of (1) with these sub-expressions resulting in the transformation of the value of F into the formula X↑2 × Y + X↑2 × sin(Z).

A schema is a set of transformation rules. Each rule is either a single production or a list of single productions defining a parallel production. Variables occurring in a schema must have single productions as values. Expressions of the form F ↓ S are formula primaries, and thus may be used as constituents in the construction of formulae. The value of such a formula primary is a formula which results from applying the productions of the schema S to the formula F according to one of the two possible sequencing modes explained as follows. Sequencing modes give the order in which productions of a given schema, S, are applied to a given formula, F, and to its sub-expressions. The two sequencing modes differ in the order in which a given production will be applied to different sub-expressions of F, and in the conditions defining when to stop.

1. One-by-one sequencing

One-by-one sequencing corresponds to a syntactic construction of the form S ← [P_1,P_2, ..., P_n]. For j ← 1 step 1 until n, production P_j is applied to F. If the application of P_j succeeds, P_j's transformation is applied to F and control returns to the first production, P_1, which is reapplied to the result. If P_j fails to apply to F, it is applied recursively to each sub-expression of F. Therefore, production P_k is applied to F if and only if production P_{k-1} is not applicable either to F itself or to any sub-expression of F. This sequencing will stop either when no production can be applied to F, or any of its sub-expressions, or when a production containing .→ has been executed.

2. Parallel sequencing

Parallel sequencing corresponds to a syntactic construction of the form S ← [[P_1,P_2, ..., P_n]]. Here for j ← 1 step 1 until n production P_j is applied to F. If the application of P_j fails, production P_{j+1} is applied to F, and so on up to P_n. If all single productions of a parallel production fail at the topmost level of F, then the whole sequence is applied recursively to the main sub-expressions of F. Thus, in parallel sequencing, each one of the productions is applied at level k of the formula F only if all productions have failed at level k-1. The termination condition is reached when all productions fail at the bottom level of F or when a production containing .→ has been executed.

In general, a schema will have a combination of both sequencing modes.

The schema variable, S, has to be declared of type symbol. Optionally, a description list may be associated with S. If the special attribute index occurs in the description list of S, then when the transformation has been completed, the value of an integer variable used as the value of the attribute

index is set to 0 if no transformation took place, i.e., no production was applicable to F. The variable is set to 1 if at least one transformation took place and exit occurred because no further production of S was applicable. Finally, the variable is set to 2 if a production containing .→ was applicable. The following complete example of a schema clears fractions in arithmetic expressions.

```
begin  form F,X,A,B,C ; symbol S,P,T ;
     A ← A:any ; B ← B:any ; C ← C:any ;
     P ← / [operator: +] [comm: true] ; T ← / [operator: ×] [comm: true] ;
S ← [A ↑ (-B) → 1 / .A↑.B ,
     A |P| (B/C) → (.A × .C + .B) / .C ,
     A |T| (B/C) → (.A × .B) / .C ,
     A - B/C → (.A × .C - .B) / .C ,
     B/C - A → (.B - .A × .C) / .C ,
     A / (B/C) → (.A × .C) / .B ,
     (B/C) / A → .B / (.C/.A) ,
     (B/A) ↑ C →[.B ↑ .C / .A ↑ .C] ;
F ← (X + 3/X) ↑2 / (X - 1/X) ;
PRINT (F ↓ S)  end
```

The above program will print X × (X↑2+3) ↑2/(X↑2×(X↑2-1)).

6. Special Functions

The following special functions are available:

Derv(F,X) which takes the derivative of a formula F with respect to the formula variable X.

Replace(F) defined in section 3.1.4.3.

Empty(L) which is a Boolean procedure having the value true if the contents of the symbol variable L is empty, and having the value false otherwise.

Mark(F) which is a function designator whose value is the value of F but which marks F with a special bit. Thus, for the expression Mark(F) + G, the value is '$\alpha + \beta$' where α = VALUE(F) with a special bit attached, and where β = VALUE(G).

Test(F) which is a Boolean function designator whose value is true if F is marked and false otherwise.

Clear(F) which is a function designator whose value is VALUE(F) but which has the special marker bit cleared.

Create(N) which is a symbol function designator whose value is a list of N created variables with names given by a numeric code.

Eradl(S) which erases the description list attached to the symbol S.

Length(L) which is an integer function designator having as value the number of elements in the topmost level of the list which is the value of L. This special function is included as a tightly coded routine for the sake of efficiency.

Cells which is an integer primary whose value at any time is the number of cells remaining on the available space list.

7. History and Implementation

Formula Algol has been under development at Carnegie Institute of Technology for three years since January 1963, and has undergone continual evolution and expansion since that date. In August, 1963, an interpretive version was running and was reported at the Working Conference on Mechanical Language Structures in Princeton, N. J. [2]. The present version of Formula Algol has been implemented as a compiler on the CDC G21 computer [6]. Its syntax analyzer is written as a set of Floyd-Evans productions [3],[4], its code generators are written in the notation of Feldman's Formal Semantic Language [7], and its run-time routines are written in machine code for the purpose of constructing, testing, and manipulating formulae and list structures at run-time. A standard linked list memory scheme has been used.

REFERENCES

[1] Naur, P. et.al., Revised Report on the Algorithmic Language ALGOL 60, Communications of the ACM, Vol. 6, pp. 1-17, (January 1963).

[2] Perlis, A. J. and Iturriaga, R., An Extension to ALGOL for Manipulating Formulae, Communications of the ACM, Vol. 7, p. 127, (February 1964).

[3] Floyd, R. W., A Descriptive Language for Symbol Manipulation, Journal ACM, Vol. 8, p. 579, (1961).

[4] Evans, A., An ALGOL 60 Compiler, Annual Review in Automatic Programming, Vol. 4, Pergammon Press.

[5] Yngve, V. H., COMIT Programmers Reference Manual, The M.I.T. Press, (September 1961).

[6] Iturriaga, R., Standish, T. A., Krutar, R. A., and Earley, J. C., Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula Algol Compiler, to appear in Proceedings Spring Joint Computer Conference 1966, Spartan Books.

[7] Feldman, J. A., A Formal Semantics for Computer Languages, Doctoral Dissertation, Carnegie Institute of Technology, (1964).

[8] Feldman, J. A., A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler, Communications of the ACM, Vol. 9, p. 3, (January 1966).