CCF-140 dmj
A. Bond
11/20/67

FORMULA ALGOL - A RAPID DESCRIPTION

Formula Algol was the result of adding two new data types to Algol. Objects of type SYMBOL which are lists and objects of type FORM which are formulae. We have, in addition, REAL, INTEGER, BOOLEAN, LOGIC variables.

We shall read the user manual by Jay Earley in the following order:

Suggested reading order:

1. SYMBOLS

   Read the part of Chapter 1 on List Processing, p. AL-3-8 → p. AL-3-12. Then read Chapter 4 on list processing, ignoring any parts which refer to FORM variables. Do a list exercise.

2. FORMULAE

   Read Chapter 1, p. AL-3-1 → p. AL-3-8. Then read Chapter 3 on formula manipulation but leave out the section on Evaluation, p. AL-3-32 → p. AL-3-36. Do a formula exercise.

3. Evaluation of formulae

   Now do p. AL-3-32 → p. AL-3-36. Do evaluation exercise.

Example

A Markov algorithm to clear rational fractions

```
BEGIN FORM A, B, C, D, N;
SYMBOL P,T, RATIONAL;

A ← A:ANY; B ← B:ANY; C ← C:ANY; D ← D:ANY; N ← N:ANY;
T ← /[OPERATOR:*][COMM:TRUE];
P → /[OPERATOR:+][COMM:TRUE];

                    RATIONAL ← [[
(A/B)*(C/D)         .→   (.A*.C)/(.B*.D)          ,
(A/B)/(C/D)         .→   (.A*.D)/(.B*.C)          ,
(A/B) |T| C          →   (.A*.C)/.B               ,
(A/B)/C             .→   .A/(.B*.C)               ,
C/(A/B)             .→   (.B*.C)/.A               ,
(A/B)↑N             .→   .A↑.N / .B↑.N            ,
.XXX →.ZZZ],[
A/B + C/D           .→   (.A*.D + .C*.B)/(.B*.D)  ,
A/B - C/D           .→   (.A*.D - .C*.B)/(.B*.D)  ,
A/B |P| C           .→   (.A + .C*.B)/.B          ,
A/B - C             .→   (.A - .C*.B)/.B          ,
C - A/B             .→   (.C*.B - .A)/.B          ,
.XXX→.ZZZ]];

END
```

SYMBOL variables - Part 1

Construction of lists

Lists can be built up by assignment statements like

A ←[B, C, D];

This statement actually evaluates B, C and D before forming the list. To prevent evaluation (like QUOTE in Lisp) we use a dot, thus after

A ←[·B, ·C, ·D];

the value of A is [B, C, D]. If A has this value and we perform

E ←[A, ·F, ·G];

we get [B, C, D, F G] as the value of E. If

E ←[[A], ·F];

then [[B, C, D], F].

Note that a list of one element is an atom so A ←[3] assigns the value 3 to A. To get a list we use A ←[[3]].

Also that objects on the lefthand side of assignment statement are evaluated if composite objects (symbolic expressions) and not if atoms. This accords with intuition. Thus A ←[·B, ·C] puts [B,C] as value of A.

1ST OF A ←[·C, ·D] puts [C, D] as value of B. Atoms are initialized automatically upon declaration to have themselves as value, e.g., A ←·A;.

Alteration of lists

To alter a list, one must select parts to be altered and then perform some type of editing statement. Selectors can be ordinal or by kind. Examples of ordinal selectors are:

1ST, 2ND, NTH, LAST

NTH BEFORE NTH,

BETWEEN 1ST AND LAST, ALL BEFORE 7TH

kind selectors are

REAL, SUBLIST, ATOM, ANY

These selectors can be nested arbitrarily, thus

1ST REAL BEFORE NTH SUBLIST

INTEGER's are an exception, FIRST 3 means the first 3 elements and FIRST INTEGER 3 mean the first occurrence of the integer 3.

Having selected parts of a list one can alter it by editing statements and we use OF to connect them. Thus,

```
DELETE  1ST  REAL  OF  A
ALTER   2ND  SUBLIST  OF  B  TO  [·C, ·D]
```

Note that a REAL element of a list is an explicit constant like 1.1, whereas an identifier which is declared as REAL is a symbolic object and selected as a FORM. The exception is, if declared as SYMB, then it is a SYMB.

Equality of lists

```
IF  A = B  THEN
```

TRUE if the value of A is exactly equal to the value of B.

Description lists

As well as a list being attached to each atom as its value, there is another kind of list which can be attached. This is an association list.

```
A ←/ [·B : ·C];
```

One can retrieve a value by

```
D → THE ·B OF A;
```

Attributes and values are symbolic objects and must be declared. One can have many different attribute-value pairs on the description list; also, values can be lists of values.

```
APPLES ←/ [·COLOR :·RED, ·GREEN] [·SHAPE : ·ROUND];
```

One can also retrieve a value by COLOR(APPLES), provided the retrieval expression COLOR is an atom. One can alter description lists by editing statements

```
THE  ·COLOR  OF  APPLES  IS  ·BLUE.
     "                   IS NOT ·RED.
     "                   IS ALSO ·BLUE.
```

Note the difference between and necessity for ← and IS. Thus

```
THE ·SHAPE OF APPLES ← ·SQUARE;
```

assigns SQUARE to the variable ROUND.

```
A ← THE ·COLOR OF APPLES;
```

assigns A the value [RED, GREEN].

An attribute, upon assignment or retrieval, can be a composite expression, but it must evaluate to an atom.

Patterns

As well as selecting and altering definite parts of lists, one can look for certain forms of list.

Thus BOOLEAN Z;

S ← [·A, 1, ·B, ·C, [·D, ·E, ·F], ·G, ·A, ·A, ·C];

Z ← S = = [·A, INTEGER, $, ·A, $2]

assigns the value TRUE to Z. The pattern consists of a list with atoms, which match to occurrences of themselves, e.g., A will only match to an occurrence of A, type words like INTEGER, ANY, etc., $N meaning any N objects and $ meaning any number of any types of object. This use of reserved words as data objects is very interesting. One can set up a pattern as a value of some atom, e.g.,

P ← [REAL, $, ·A];

and one can then match, e.g.,

IF S = = P THEN, etc.

Thus, the pattern is subjected to evaluation before the match is applied.

R ← ·S;

F = = R matches the value of F to the atom S

F = = ·R matches the value of F to the atom R.

P ← [REAL, $, R]

assigns the pattern [REAL, $, S] to P.

F = = P

matches the value of F to [REAL, $, S].

One can even match patterns against patterns in which case the single = is needed. This = can be used to match for exact equality for any two lists, in particular for pattern lists. = = REAL matches any real, = REAL matches to the reserved identifier REAL.

In addition to matching on exact atoms, type words and any number of objects, the user can define his own types or classes, using lists.

V ← [·A, ·E, ·I, ·O, ·U];

LET (|VOWEL|) = [X | AMONG (X, V)] defines a class called (|VOWEL|), where VOWEL is a symbol variable, meaning membership of the list V. This can be used in a pattern; thus,

IF S = = (|VOWEL|) THEN.

Actually one needn't only use the system function AMONG, one can use any Boolean procedure or expression to define a class of objects, viz. those for which it returns the value TRUE.

LET (|INOI|) = [X | X > 0 ∧ X > 1]

One can optionally use extractors in any pattern match.

F = = [·I : INTEGER, ·A, $2, ·B : ANY, ·C : (|VOWEL|)]

if matched to [3, A, B, C, D, A], then corresponding values are extracted into the extractors. Thus, I, B, C have values 3, D, A.

Extractors can be present in an assignment of a pattern value

P ← [·I : INTEGER, ·A]

and they are evaluated before use. Thus,

F = = [THE 1ST OF A : INTEGER]

FORM variables - Part 1

Construction of formulae

A formula is a symbolic object whose form is anything like an Algol expression. Thus,

```
X ← (·X + ·Y) ↑ 2 ;
F →3 * SIN(·G) + 2 ;
G ← ·IF ·X > ·Y THEN ·X ELSE ·Y;
E ← PROC ·(·X, ·Y);
B →A· [·X, ·Y];
H ←X ·→ ·Y
```

set up the following values as symbolic entities. (X + Y) ↑ 2, 3 *SIN(G) +2, IF X > Y THEN X ELSE Y, PROC(X, Y), A[X, Y], X ← Y.

If the variables are undotted, they are evaluated, thus after

```
A ← ·B;
C ← SIN(A);
```

A has value B and C has value SIN(B). Real, Integer, Boolean and Logic variables can be used in formulae and evaluate is the usual way. Thus, C ← ·X; A ← (3.142/2); B ← SIN(A) + C;

B has the value 1 + X.

One can use a symbol variable to define a variable which takes as its value, one of +, -, *, etc. It can be assigned values and used in constructing formlae, thus

R ← / [OPERATOR : +, - /] [COMM : TRUE, FALSE, FALSE] [INDEX : J]

J of type integer.
Then if R has value + and E ← .A |<R>| .B;
E has value A + B.

Formula patterns

In a similar way to lists, one can use patterns to analyze the structure of a formula. Patterns can be any formula and can use atoms, type words or classes.

F = = .A ; F = = REAL , F = = ANY
F = = OF (B)

where B is a user defined Boolean procedure of one argument defining a class of objects.

F = = .A + .B , F = = .IF .X THEN .Y ELSE .Z
F = = A. [.X, .Y] , F = = A. (.X, .Y) , F = = .X. ← .Y

In addition, one can define a class of patterns by using a list of patterns $P \leftarrow [P_1, P_2, \ldots P_n]$, then F = = P matches if at least one of $P_1, \ldots P_n$ match F.

One can use a variable operator

F = = .A | R | .B.

Extractors can be used in the usual way.

If the pattern match F = = .A |R| .B is used, it matches R if the binary operator in the formula occurs as a value of the descriptor OPERATOR on the description list of R. The extracted value is not available, but can be deduced from the INDEX which contains an integer which is the order of the operator. Thus, with the assignment in the previous section and F ← .A - .B; Z ← F = = .A |R| .B; Y ← THE INDEX OF R;

Z now has TRUE and Y has 2. Since + has COMM value TRUE, F ← .B + .A; Z ← F = = .A |R| .B; gives TRUE.

In addition to the = = construction which matches the entire formula, we can test for inclusion of a pattern somewhere in the formula using >>.

F >> P matches if F contains a subexpression which matches the pattern P.

In this case, we can use an extractor on F.

A : F >> B : P

Means, if a match occurs, the subexpression of F which matched P is stored as the value of B. The old value of B is substituted for the subexpression in F and this updated F is stored as the value of A.

Markov Algorithms

One can change formula by using productions P → Q. A production is applied to the formula F by matching P = = F, if no match can is applicable, if a match we extract the subexpressions into the expressions and substitute them in the expression Q. This value is then assigned to F. Thus applying

A : ANY * (B : ANY + C : ANY) → .A * .B + .A * .C

to the F there F ← .X ↑ 2 * (.Y + SIN (.Z)) ; extracts X ↑ 2, Y and SIN(Z) into A, B, C and assigns F the value X ↑ 2 * Y + X ↑ 2 * SIN(Z). Productions are set up as schema by assigning lists of them to symbol variables. It can be either a series or parallel schema

S ← [P_1, P_2, ... P_n] or S ← [[P_1, P_2 ... P_n]]

and the schema is applied to a formula F by F ← F. ↓ S. (The dot is only to distinguish from truncation.)

In series processing P_1 is matched by >> to F, i.e., all subexpressions, if it fails, P_2 is tried, and so on.

In parallel processing P_1 is matched by = = to F, then P_2, and so on. If all fail at this level, then the first subexpression of F is tried on all P_1, and so on.

One can mix schema slightly, e.g., in S ← [P_1, [P_2, P_3], P_4],[P_2, P_3] is applied parallelly after P_1 has been applied serially and before P_4 is applied serially. A dot on the → implies termination of the algorithm after application of this production.

SYMBOL variables - Part 2

One can have SYMBOL ARRAY's whose use is obvious. Elements of lists can be of any type including FORM.

Further uses of description lists

Description lists can be attached to variables of type FORM. They can also be attached to sublists of a list giving a local description. Thus,

T ← [F, A/[NUM : 1] , B, C, A/[NUM : 2], D, E];

These local description lists attached to A do not interfere with the global description list of an occurrence of A outside the list attached to T.

Push down and Pop up statements

↓ and ↑ are operators which push and pop any atoms of type SYMBOL. Thus, each atom is like a stack and when it is used in other statements,

the value of it is the value on the top of the stack.

Generalization of For statements

The <for clause> is generalized to

FOR <symbolic expression> ←<for list> DO | PARALLEL FOR <expression> ←ELEMENTS OF <expression> DO and the <for list> to

ELEMENTS OF <symbolic expression> |
ATTRIBUTES OF <symbolic expression>

Thus, the index can take values from the value list or the description list of a SYMBOLIC object. Also, one can have parallel execution as in

PARALLEL FOR [I,J,K] ←ELEMENTS OF [[S], [T], [U]] DO

This means that the 1st element of S, T, U are assigned to I, J, K, respectively and the DO performed, then the 2nd elements, and so on.

Default actions

If any list runs out before another the value NIL is used. This value NIL is used in other exhaustion situations such as looking for the value associated with a non-existent attribute.

Indirect accessing brackets

If A ← .B; B ← .C; C ← .D;

then using A causes it to return B as value. For indirect accessing one can use angular brackets < >.

Thus <A> returns C.
<<A>> returns D.

FORM variables - Part 2

Evaluation of Formulae

The EVAL operator provides additional evaluation capability. Having computed a symbolic formula, we can substitute numerical values for the variables and find the value.

e.g., F ← .X ↑ .Y; Z ← EVAL (X, Y) F(3,2);

substitutes 3 for X and 2 for Y and puts 9 in Z. One can use EVAL to substitute other symbolic values.

e.g., F ← .X ↑ .Y; Z ← EVAL (.X, .Y) F (.A, .B)

gives A ↑ B as value of Z.

EVAL applied to formula of more general Algol like types will execute them. Thus,

```
F ← .IF.B THEN .A ELSE .C;
  EVAL (B) F (TRUE)   will return A.
```

Similarly dotted procedures, dotted arrays or dotted assignment statements are executed by EVAL. During evaluation EVAL does some obvious simplifications like A ↑ 0 → 1.

There are two other related operators, SUBS and REPLACE. SUBS does a straightforward substitution without evaluation or simplification

```
D ← SUBS (X1, .X2) F (Y1, .Y2)
```

REPLACE replaces all atoms in the formula by their current value and evaluates it.

```
F ← .X + .Y * .Z;
Y ← 1; Z ← 2;
Z1 ← REPLACE (F);          gives X + 2.
Z2 ← SUBS (Y, .Z) F (3, 4);  "   X + 12.
```

Thus, EVAL and SUBS only substitute for the objects named in their parameter lists, but REPLACE substitutes for all.

SUBS does no evaluation; just selected substitutions.

REPLACE does evaluation after total substitution.

EVAL does evaluation after selected substitutions.

Note the dots on the r.h.s. of productions; this is because the r.h.s. is evaluated, when it is formed just before use. The match and extraction is then done. REPLACE then operates on the r.h.s. and finally EVAL is applied, all by the system.

Note, also, that the substitution arguments to EVAL and SUBS are, in general, symbolic expressions which are evaluated; the 1st list like (.Y, .Z) above must evaluate to atoms, and that EVAL, SUBS and REPLACE will only replace FORM atoms, not atoms of other types.

Recursive patterns

If an element of a pattern is a pattern, this pattern is matched. Thus, the pattern

```
SUM ← [OF(SUM) + OF(SUM), OF(P)]
```

when matched, F = = SUM, will match all subexpressions. This can be used to extract all the elements of a formula:

```
BOOLEAN PROCEDURE P(X); FORM X;
BEGIN   P ← TRUE; INSERT  X  AFTER LAST OF L; END;
DELETE L;
```

This extracts all the subparts of a composite sum and puts them in the list L.

One can have mixed formula and symbol patterns, thus

S ← [Φ, .A ^ .B, REAL, NF S : ANY THEN ANY ELSE ANY]

is a list pattern.

Assigning a value to an operator class

One cannot say R ← +; R can be assigned a value either by pattern matching or by the following dodge:

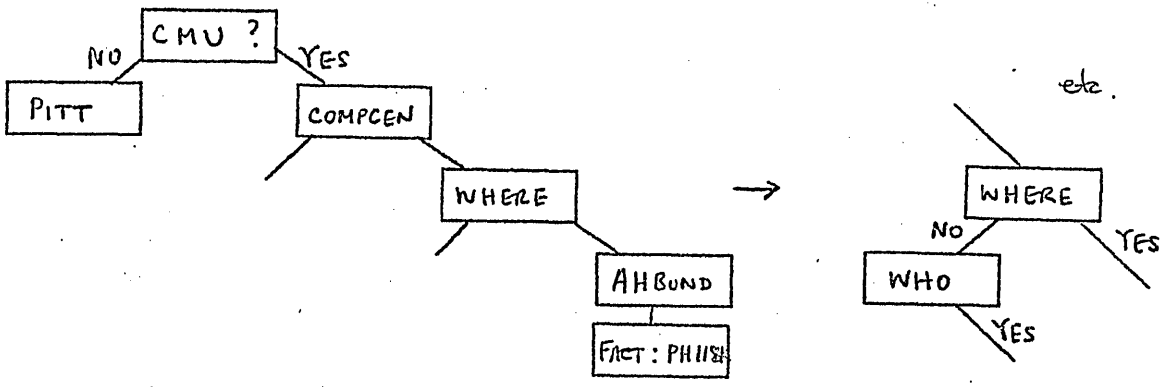
D ← /[OPERATOR : +];
FOR S ← ATTRIBUTES OF D DO R ← The S of D;

Exercises in Formula Algol

1. Write a program to take questions and answers and set up a discrimination net, i.e., a question is a list of properties that the retrieved object must satisfy something like

   [CMU, COMPCENTER, WHERE, AHBOND]

   and a fact is an associated symbol PH118K. The discrimination net is of the form



   and the answer to a question can be retrieved by testing successive elements. When a new fact is added, e.g.,

   [CMU, COMPCENTER, WHO, MANAGER]

   one must test successively until a failure is obtained and then modify the net to add the new fact. Design a data form and a discrimination net to act as an automatic user consultant. One would like to be able to answer questions which cut across the net also, e.g., give a list of all people in room PH118K. As there is no input in Formula Algol, the cards must be compiled in as a symbol array CARD, say.

2. Given a polynomial in form

   $$(a_{1n}x^n + a_{1n-1}x^{n-1} + .. + a_1)(a_{2m}x^m + . \quad)(\quad)$$

   expand it and collect like terms.

3. Given a rational function, clear all fractions.

4. Solve a differential equation by computing the first N terms of a Taylor Series.

5. Transform a formula in the propositional calculus into disjunctive normal form.

6. Write a program to give numerical values to an accuracy e of Legendre Polynomials $P_l^m(x)$ for arbitrary m, l, x, e. You can use any generation method you think suitable.

7. Write a LISP interpreter in formula algol, possibly along the following lines:

```
Represent (A.B) by (A*B)

then define symbol procedures
symbol procedure car (x); symbol x; begin
if x == y: any * any then car ← y else begin print (.UNDEFINED)
    go to halt; end end;
```