PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 4/70

A LOW-LEVEL MATHEMATICAL PATTERN - MATCHING ALGORITHM

by

Antonio Luz Furtado

Computer Science Department - Rio Datacenter

CENTRO TÉCNICO CIENTÍFICO
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

A LOW - LEVEL MATHEMATICAL PATTERN - MATCHING ALGORITHM

by

Antonio Luz Furtado
COMPUTER SCIENCE DEPARTMENT
PUC - RIO DE JANEIRO

ABSTRACT:

A number of questions concerning pattern-matching are discussed with the aid of an algorithm that is now operational for the IBM-7044, under a modified $IBFTC FORTRAN IV compiler.

The parttern-matching processes discussed here are classified as low-level, in the sense that they exhibit a score of very detailed mechanisms that in general are quite understandable for a programmer but inconvenient for direct use by a mathematician.

Such mechanisms, however, indicate how higher-level facilities could be constructed from them.

Some features were taken from several existing systems and a few of them are original. Experience is still needed to confirm their usefulness and to suggest further improvements and extensions.

A member of examples are scattered through the following sections and four of them are more throughly expounded at section 4.

1 - CONCEPTS AND ELEMENTS:

Mathematical pattern-matching is defined as the comparison of a model (pattern) to an expression in order to determine whether the expression or some of its sub-expressions is an instance of the pattern.

When the match succeeds the desired outcone may be:

a- to simply indicate the successful exit;

b- the decomposition of the expression into the elements that correspond to each of the elements of the pattern;

c- the transformation of the expression, according to some replacement rule;

d- this transformation could be used to generate new patterns and new replacement rules;

The input to a pattern-matching process consists of:

a- expression;

b- pattern;

c- replacement rule;

d- user-defined prototypes;

e- processing mode (see section 2).

The output elements are:

a- a logical value that indicates success or failure of the match;

b- extracted constituents;

c- transformed expression;

d- new pattern and new replacement rule.

While the expression may contain any of the usual mathematical entities (constants, variables, operators, functions) the pattern and the replacement rule may contain, besides these, a special cathegory of entities named prototypes.

A prototype is a symbol that stands for any menber of a class of mathematical symbols. We list the system-defined prototypes, indicating the class that each stands for:

$V	variables
$C	constants
$A	atoms (variables or constants)
$E	expressions
$Ø	operators
$F	functions
$	anything

The user can define other prototypes and include them in the list of user definitions.

Definitions involve either the enumeration of the members of the class that is to correspond to the defined prototype, or a function indicating what property must be satisfied by a member of the class.

As examples consider:

a - ($Z (W Y Z))

b - (HX 'HAS' (X))

The prototype $Z will match any of the variables W, Y, Z; one application thereof would be to specify those variables in a differentiation problem (with respect to X) that depend on X.

The prototype $HX will match any expression containing X, including X alone.

Functions (indicated by single quotes delimiter) are activated in three different moments of the matching process, moment b corresponding to their use to define class membership:

a - before the recognition - ex:

($ ØC (+ *) ' CØMMUT')

the effect is that a pattern that contains an occurrence of $ØC will be flagged so that it will match if + or * is the operator, with no regard to the order in which the operands appear.

b- during the recognition - ex: the function 'HAS' exemplified above.

c- after the recognition - ex:

```
( $J    'INTER'  ($1  $2))
```

supposing the match is done with the purpose of extracting elements, there will correspond to $J the set intersection of all elements corresponding to $1 and $2.

At the moment all functions are system-defined; they are: COMMUT, HAS, FREEOF, INTER, UNION, DIFFER (the latter three being set-theoretic) and RMATCH. The use of RMATCH will be illustrated through an example in section 2.

Different occurrences of a prototype in a pattern and replacement rule are numbered. Thus, take a pattern to indicate: the variable X followed by + followed by any variable followed by - followed by any variable possibly different from the previous one. This pattern would be written as:

```
( X + $V1 - $V2)
```

If the second and the last variables must be the same, the pattern will become:

```
( X + $V1 - $V1)
```

In some cases the non-occurrence of an element must be allowed. To express

the case where both binary and unary minus would match, the pattern would be:

($V01 - $V2)

where the numbering portion of the first prototype is preceded by zero.

Prototypes beginning with a double dollar sign can only be used in the replacement rules. They stand for the original element from the expression. Both what is meant by original and the need for this feature will be explained in section 2 where the structural scheme is presented.

Patterns and replacement rules may be atomic or non-atomic. In the latter case then are lists; they will sometimes have sub-lists which we call sub-patterns.

In turn a set of patterns and a set of replacement rules are organized as lists. Thus, we have a list of patterns and a list of replacement rules.

Exs:

atomic pattern	$1
non-atomic pattern	($1 + X)
pattern with sub-patterns	(($1 + $2) ** 2)
list of patterns	(($1 + 0) (0 + $1) ($1 + $1)...)

Within a list of patterns one may refer to other patterns by their ordinal

number in the list.

If the last example above would contain in the second pattern a reference to the third, it would become:

```
(($1 + 0) ((0 + $1)// 3 *) ($1 + $1) ... )
```

The meaning of such reference and of the asterisk following the number 3 will be explained in the next section.

2 - PROCESSES:

In the preceding section we discussed the objects that appear in pattern-matching. We shall now show how they operate.

The basic algorithm, taking only one expression, one pattern and one replacement rule, goes as follows:

a - the first element of the pattern is analyzed; then:

- if it is not a prototype it stands for itself, which means that an exact copy of it must be found as the first element of the expression;
- if it is a prototype of the form $V, $C, $A, $E, $Ø, $F the first element of the expression must be of the corresponding type;
- if it is a prototype of the form $ the second element of the pattern is immediately analyzed, and will be matched to the first element of the expression;
- if it is a user-defined prototype action will be taken according to its definition (see previous section);
- if it is a sub-pattern the whole algorithm is applied recursively .

b - the first element of the expression is analyzed to see whether it corresponds to the previously analyzed element of the pattern.

c - if the match succeeds, then:

- if the pattern element is a prototype it will form, together with the corresponding element in the expression, an ordered pair to be added to a list of found elements; then the match will proceed with the next pattern element and the next expression element;
- if the pattern element is not a prototype the match simply proceeds as above.

d - if the match fails the process terminates, unless;

- the pattern element is a prototype allowing non-occurrence;
- the previous pattern element was of type $; in this case the expression element will be added to what will be put in the found elements list in correspondence to that occurrence of $; the match proceeds with the second element of the expression but with the same prototype that failed.

e - coming back to the case when a match succeeds, a test is made of whether a previous $ prototype was detected; then the concatenation of all previous unmatched expression elements will finally be put in the found list; if nothing was found the match fails unless that $ allowed non-occurrence.

f - if the end of pattern and the end of expression are simultaneously reached the overall match succeeds, otherwise it fails; if the

last pattern symbol is a $ prototype then the remaining of the expression is assigned to it.

g - the last phase is the replacement, if the user has specified that he wants it; the replacement consists of copying the replacement rule changing only the prototypes by what corresponds to them in the found elements list; note that this list might be called extracted elements list, thus accomplishing the purpose of decomposition of the given expression.

As an example consider:

```
given expression:        (A + B - X + Y)
pattern:                 ($1 - X + $V1)
found elements list:     (($1 (A + B)) ($V1  Y))
replacement rule:        ($1 - X * DELTA)
transformed expression: (A + B - X * DELTA)
```

The same example with a slight change illustrates the case of non - -occurrence allowed:

```
given expression:        (- X + Y)
pattern:                 ($01 - X + $V1)
found elements list:     (($V1 Y))
```

replacement rule: ($01 - X * DELTA)
transformed expression: (- X * DELTA)

Note that the unmatched prototype was disregarded in the creation of the transformed expression.

We now consider the more interesting problem where a list of patterns and the list of corresponding replacement rules are given.

There are two schemes for traversing the expression:

a - the linear scheme;
b - the structural scheme;

The linear scheme is the first to be described:

a - the first pattern in applied to the expression;
b - if the match succeeds the transformation in done (if required);
c - if the match fails, the next pattern is tried;
d - the overall match fails if no pattern succeeds, the end of the list of patterns being reached.

The procedures in case of success or failure prescribe how the system normally handles the problem. The system can be superseded by having one pattern refer to one or two others in the same list.

To do this we enclose the pattern in an extra pair of brackets, the closing one being followed by the symbol //, and this by two ordinal numbers.

The first number indicates the pattern where the match is to be retried in case of success; the second number shows where to retry in case of failure.

An asterisk indicates that the normal system action is to be taken. Its need arises from the requirement that when one transfer is specified the other one must be also included.

A number greater than the number of patterns in the list forces the process to terminate, so that no other pattern is tried.

Now we come to the structural scheme.

We begin by defining a fragment as any part of an expression. It differs from the intuitive concept of sub-expression in that no requirements are made as to a meaningful conbination of operators and operands.

Thus, from the expression ((A + B) * (X / Y)) the following objects (among others) could be taken:

```
                    A
                    +
                    B
                  + B
              (A + B)
                    X
                    /
                    Y
                  / Y
              (X / Y)
            * (X / Y)
((A + B) * (X / Y))
```

One way of obtaining fragments is through the use of the CAR and CDR functions defined in the LISP 1.5. language, CAR takes the beginning of a list (in this case, of an expression) and CDR takes its continu ation (in other words: what remains after the suppression of the beginning).

If CAR and CDR were applied to ((A + B) * (X /Y)), according to the structural scheme, the fragments above would be taken, in that order. It is clear that among such fragments will be all true sub-expressions, provided that the expression is completely parenthesized.

Another function from LISP 1.5 is needed to explain the structural scheme - CONS - which creates a list from two elements (atomic or not)

whose CAR part is the first element and the CDR part is formed from the second element.

The structural scheme goes as follows:

a - the linear scheme is applied to CAR (EXPR); call the transformed fragment A;

b - the linear scheme is applied to CDR (EXPR), giving B;

c - the linear scheme is applied to EXPR with no transformation specified; a special list of found elements is created and then its prototypes are put in the double dollar sign form;

d - the linear scheme is applied to CONS (A,B) and the specified transformations are done.

The list obtained at phase a is sometimes necessary and so it is made available for phase d. An example of this is the following very limited differentiation application:

```
given expression:          ((A + X)* B)
list of patterns:          (X $A1 ($1 + $2) ($1 * $2))
list of replacement rules: (1  0  ($1 + $2) (($$1 * $2)+($$2 * $1)))
```

```
application to A gives 0
application to X gives 1
application to (A + X) produces (($$1 A)($$2 X))
application to (0 + 1) gives (0 + 1)

application to B gives 0
application to ((A + X) * B)  produces (($$1((A + X)))($$2(B)))
```

When the final application is done to ((0 + 1) * 0) we have:

```
pattern:                ($1 * $2)
found elements list:    (($1((0 + 1)))($2(0))($$1((A + X)))($$2(B)))
replacement rule:       (($$1 * $2) + ($$2 * $1))
transformed expression: (((A + X) * 0) + (B * (0 + 1)))
```

Of course the $$ feature is also useful whenever a temporary or a tentative transformation was done and at a later stage one wishes to restablish the original form.

Both the linear and structural schemes will be applied just once to the full expression (linear scheme) or to each fragment (structural scheme).

However a restart in case of success may be specified by the user. If this happens in the linear scheme all patterns are re-applied to the expression until no pattern succeeds any more. In the structural scheme

all patterns are reapplied to the fragment where success was obtained, in a structural way (beginning by the lowest order sub-fragments); similarly re-application stops only when no pattern matches the fragment.

Decomposition, which we also call extraction has a broader sense when re-application is allowed. An overall extraction list is formed by concatenating the contents of each list of found elements.

The example below illustrates this and the concept of processing mode, referred to in section 1:

```
given expression:            (X**2 + B + X**3 + 7)
list of patterns:            (($01 X** $C1 $02))
list of replacement rules: ($02)
processing mode: linear scheme, re-application if success
list of extracted elements:  (($01(+ B +))($C1(2 3))($02(+ 7)))
transformed expression:      (+ 7)
```

The example shows one way of obtaining all exponents of X appearing in an expression. It also exemplifies how transformation can be used to help extraction; it is clear that each recognized power of X is removed prior to the re-application, thus allowing the extraction of the next exponent, until no powers of X stay.

Parallel to the operation of the list of patterns and the list of replacement rules, runs another programming guideline which is the list of user definitions.

The existing definition features, as we saw in section 1, can be activated at three different moments. It might be said furthermore that definitions of. the first two moments have a declarative nature, and that they may appear in any order; the third type might be classified as executable and in this case order in which such definitions appears is relevant.

Consider the problem of finding a quadratic, if any, in a expression.

Let the expression be:

$$(x^2 + a - b^2 + y^2 + z^2 + 2\,xz + 2\,xt + c)$$

It is clear from inspection that the quadratic is $x^2 + z^2 + 2xz$. Note that the following pattern will not isolate the quadratic:

```
($V1 **2  $01 + $V2 **2 $02 + 2*$V1*$V2 $03)
```

the first reason being that the system will expect to find $V2 **2 immedi ately after the first plus sign, which instead is followed by a.

The conbined effect of the list of definitions and the list of patterns below will solve the problem:

```
list of definitions: (($K 'INTER' ($V1 $V2))($L 'INTER' ($V1 $V3))

                     ($M 'RMATCH'((($K + $L)**2))))*
```

```
list of patterns: (($V1**2  $02)(2*$V2*$V3  $02)
                  ($01+$V1**2  $02)($01+2*$V2*$V3  $02)
                  ($01 + $02))

list of replacement rules: ($02  $02  $02  $02  $02)
```

processing mode: linear scheme with re-application in case of success.

In the list of extracted elements there will be, after the application of the patterns:

```
( ... ($V1(X Y Z)) ... ($V2(X X))($V3(Z T)))
```

The list of definitions is then executed, thus adding to the list of extracted elements the following ordered pairs, in successive steps:

```
($K X)
($L Z)
($M (((X + Z)**2)))
```

The reasoning behind this is: one would choose as $K a variable that:

- appears in a sub-expression of the form $V1**2
- and appears in a sub-expression of the form 2*$V2*$V3

The intersection function applied to $V1 and $V2 gives all variables that simultaneously satisfy both conditions.

Similarly one selects candidates for $L.

Finally the function RMATCH executes a simple match using only the elements in the list of definitions, giving in association with $M the desired final result.

Three examples follow where special techniques in programming with patterns are displayed:

a - deferred substitution

given expression: ((A+(B + C))*X)

list of patterns: (($V1+$V2)($V1+$E1)($E1*$V3))

list of replacement rules: (($V1*$V3+$V2*$V3)($V1*$V3 + $E1)$E1)

processing mode: structural, no re-application

transformed expression: (A * X + (B * X + C * X))

note that the following transformations occur:

(B + C) → (B * $V3 + C * $V3)

(A + (B * $V3 + C * $V3)) → (A * $V3 + (B * $V3 + C * $V3))

((A + (B * $V3 + C * $V3)) * X) → (A * X +(B * X + C * X))

where only at the last transformation X is associated with $V3 by the

application of patterns ($E1 * $V3); at this last stage the list of found elements contains:

```
((($E1((A * $V3 + (B * $V3 + C * $V3))))($V3 X))
```

and a replacement of $V3 takes place inside the list of found elements giving:

```
((($E1((A * X + (B * X + C * X))))($V3 X))
```

the simpler pattern (($1 + $2) * $V1) and the replacement rule (($1 * $V1) + ($2 * $V1)) will succeed only if the structural scheme is combined with re-application, as one may verify.

b - post-fixing the operators

given expression: (A - B + X + C -D + X + E - E)

list of patterns: ((+ $V1 $01 $V1 + $02)
($V1 $01 $Ø1 $V2 $02 $V2 $03)
($Ø1 $V1 $1 $E1 $02)
($Ø1 $V1 $01))

list of replacement rules: (((2 * $V1) + $V1 $V1)
($V1 $$02 $01 $$03)
($Ø1 $V1 $1 $E1 $02)
($V1 $Ø1 $01))

processing mode:structural, no re-application

transformed expression: (A - B + C -D + (2 * X) + E - F)

note that the following transformations occur, and that we take advantage of the fragment concept, and of the double dollar sign feature:

(- F) → (F -)
(+ E F -) → (E + F -)
(+ X E + F-) → (X + E + F -)
(- D X + E + F-) → (D - X + E + F -)
(+ C D - X + E + F -) → (C + D - X + E + F -)
(+ X C + D - X + E + F -) → ((2 * X) + X X) - see first pattern
(B (2 * X) + X X) → (B + C - D +(2 * X) + E - F)
(-B + C -D + (2 * X) + E -F) → (-B + C -D +(2* X) + E - F)
(A - B + C - D + (2 * X) + E - F) - no change

the point in post-fixing the operators is seen in the sixth transformation; one looks for a variable (X) that occurs twice in the expression, preceded by a plus sign, but the simple pattern:

($1 + $V1 $2 + $V1 $3)

will not succeed because the system will produce the list of found elements

(($1 (A - B))($V2 X))

and then it will expect to find a second occurrence of X immediately after the next plus sign but it finds c and the match fails. However by post-fixing the plus sign that precedes the second X this plus sign will be sought for only after the second occurrence of X is detected.

c - marking

given expression: (A - B + X ** 2 + C - D + Y ** 2 + E -F + 2 *X*
Y + G - H)

list of patterns: (($1 .. + $V1 ** 2 $02..+$V2**2 $03 ..+2*$V1*
$V2 * $04)(+$V1 **2 $01)(+2*$V1*$V2 $01))

list of replacement rules: (($1 $02 $03 $04 +($V1 + $V2)**2))
(.. + $V1 **2 $01)(..+ 2*$V1*$V2 $01))

processing mode: structural, no re-application

transformed expression: (A - B + C - D + E -F + G - H +(X + Y)**2)

the following transformations occur, where the special symbol .. is used to mark the plus signs that come before the desired sub- expressions, thus avoiding that a plus sign not fulfilling this condition will arrest the search:

(+ 2* X * Y + G - H) → (.. + 2 * X * Y + G - H)
(+ Y ** 2 + E - F .. + 2 * X * Y + G - H) → (.. + Y ** 2 + E - F
..+ 2* X * Y + G - H)

```
(+ X ** 2 + C - D .. + Y ** 2 + E - F .. + 2 * X + Y + G - H) →
(.. + X ** 2 + C - D .. + Y ** 2 + E - F .. + 2 * X * Y + G - H)
(B .. + X ** 2 + C - D .. + Y ** 2 + E - F..+2 * X + Y + G - H)→
(B + C - D + E - F + G - H + (X + Y) ** 2)
(A - B + C - D + E - F + G - H + (X + Y) ** 2 ) - no change
```

These special techniques are only suggested here as possible mechanisms to helps solving certain problems. They would be too clumsy to be handled directly by the user: however they could be generated by a macro- pattern expression, a concept that will be further treated at section 3.

3 - ENVIRONMENT:

The FORTRAN IV compiler that was used for the present work is a modified version of the $IBFTC for the IBM - 7044.

It allows, the RECURSIVE statement and the list assignment statement.

The RECURSIVE statement will cause all sub-programs so declared to be called through a routine which does the proper stacking and unstacking of arguments and return points.

Recursion is required by the system to perform the following actions:

a - decomposition of the given expression into fragments, in case the structural scheme is used;

b - to select from their lists each pattern and corresponding replacement rule and apply them, one such pair in its turn, until the lists are exchausted or a termination condition occurs;

c - when a particular pattern contains sub-patterns, in order to perform the basic algorithm upon the sub-patterns;

d - to retry a match if a failure occrus with commutative patterns; a recursive call is done trying to match the first operand in the expression with the second in the pattern, and similarly with the

second operand in the expression with the first in the pattern; clearly this implies that patterns with several levels of sub--patterns containing commutative operators are allowed, on the only condition of complete parenthetization.

The list-assignment statement assigns to the variable on the left of the equal sign the machine address (pointer) of the head of the list written on the right side. The statement is distinguished from the ordinary assignment by using the delimiter *. Exs:

```
Z = * (( A B ) C ) *

X = * ( A + (B * C )) *
```

Note that the second example is an expression. Indeed, in this system all formula manipulation is done through a LISP-like list-processing package.

The list-processing system should include garbage collection to get rid of all intermediate results that are no longer needed. Although one such process is already available we still heve not included it in the pattern--matching algorithm, which will be done soon.

Another important process that is used is a lexical scanner that recognizes variables, constants, operators, etc.

In the near feature the compiler will be modified further to include the

FORMAL statement. This will be used to associate the hollerith names of variables and functions with their machine addresses or entry points respectively, so establishing communication between the usual FORTRAN numerical evaluation statements and the symbolic handling of formulas.

A probable future extension of this research will be to develop the concept of macro-patterns, in order to obtain a notation that will be more natural for the mathematician and less dependent on the knowledge of programming techniques.

This objective of naturalness has prevented us from requiring the user to present his input in any canonical form. However many algorithms would be more efficient and programming would be made easier if a suitable canonical form were required.

A compromise solution is to include the transformation into canonical form in the macro-pattern expansion phase, leaving it therefore to be automatically handled by the system.

Another future development will be the ability to reference a whole list of patterns from a pattern located in a different list. This will be possible after implementation of the FORMAL statement. A possible use would be: let P1 be a list of patterns for differentiation and P2 a list of patterns for integration; the latter would include immediate integrals and the scheme for integration by parts, which would involve a function-like notation:

```
P2 = * ( ... ( $1 * $2) ... ) *
R2 = * ( ... ( $$1 * $2 - 'P2'(($2 * 'P1'($$1))) ... )*
```

where the indicated replacement rule in R2 is the familiar

$$\int udv = uv - \int vdu$$

the complete parenthetization being omitted for readability.

The ability to reference whole lists of patterns is also important to relieve the user's programming burden, because many commonly used lists could be system-defined and made available, as if they were a package of library routines.

4 - EXAMPLES:

Four complete examples are given in this section.

The first one is a differentiation with respect to X, where the only operations considered are addition, subtraction (unary minus is allowed) and product; each sub-expression that is differentiated is immediately simplified in order to avoid the rapid growth of results.

The processing mode is the structural mode with no re-application.

Each step of the execution causes the listing, in this order, of:

- the sub-expression
- the pattern that matched it
- the corresponding replacement rule
- the list of found elements
- the transformed sub- expression

The reader may note that a second call to the sub-program MATCH, where the EXPR (original expression) were replaced by M (resulting derivative) would yield the second derivative, since the patterns and replacement rules make provision for that.

```
      RECURSIVE MATCH
      COMMON/LSTC/IPD,AREA(10000)
      COMMON/MATC/SF,LLF,IREP,LDEF
      IPD=-10000
      LDEF=*(($P(+)'COMMUT')($T(*)'COMMUT')($DX(W Y Z)))*
      EXPR=*(((-A)*X)+Z)*
      CALL OULST(EXPR)
      PRINT 100
      PATT=*( /*1*/X  /*2*/$DX1  /*3*/$A1  /*4*/(($1+$2)//8 *)
     , /*5*/(($01-$2)//10 *)  /*6*/(($1*$2)//13 *)
     , /*7*/((DER $1)//* 17)  /*8*/($1 $P1 0)  /*9*/(($1+$1)//* 17)
     , /*10*/(-0)  /*11*/($1-0)  /*12*/(($1-$1)//* 17)
     , /*13*/((($1 $T1 0) $P1 $2)//14 16)  /*14*/($1 $T1 0)
     , /*15*/(($1 $T1 1)//* 17)  /*16*/((($1 $T1 1) $P1 $2)//16 9)  )*
      REPL=*( /*1*/1  /*2*/(DER($DX1,X))  /*3*/0  /*4*/($1+$2)
     , /*5*/($01-$2)  /*6*/(($$1*$2)+($$2*$1))  /*7*/(DER((DER $1),X))
     , /*8*/$1  /*9*/(2*$1)  /*10*/0  /*11*/$1  /*12*/0  /*13*/$2
     , /*14*/0  /*15*/$1  /*16*/($1 $P1 $2)  )*
      M=MATCH(EXPR,PATT,REPL)
      PRINT 100
  100 FORMAT(/////)
      CALL OULST(M)
      STOP
      END
```

```
(((- A)* X)+ Z)
```

```
A
$A1
 0
(($A1 A))
0
```

```
(- 0)
($01 - $2)
($01 - $2)
(($2(0))($$2(A)))
(- 0)
```

```
 (- 0)
  (- 0)
 0
 ()
 0


 X
 X
 1
 ()
 1


 (0 * 1)
 ($1 * $2)
 (($$1 * $$2)+($$2 * $1))
 (($1(0))($2(1))($$1((- A)))($$2(X)))
 (((- A)* 1)+(X * 0))


 (((- A)* 1)+(X * 0))
  (($1 $T1 0)$P1 $2)
$2
(($2(((- A)* 1)))($T1 *)($1(X))($P1 +))
((- A)* 1)


($1 $T1 1)
$1
(($T1 *)($1((- A))))
 (- A)


Z
$DX1
(DER($DX1 , X))
(($DX1 Z))
(DER(Z , X))
```

```
((- A)+(DER(Z , X)))
($1 + $2)
($1 + $2)
(($1((- A)))($2((DER(Z , X))))($$1(((- A)* X)))($$2(Z)))
((- A)+(DER(Z , X)))


((- A)+(DER(Z , X)))
```

The second example clears fractions in arithmetic expressions.

The processing mode is again the structural scheme with no re-application. If re-application were allowed, the pattern

(($1 / $2) / ($3 / $4))

would not be necessary since the patterns

($1 /($2 / $3))

(($2 / $3) / $1)

would produce the same effect. However it is an obvious advantage to avoid re-application which is a time-consuming procedure.

```
      RECURSIVE MATCH
      COMMON/LSTC/IPD,AREA(10000)
      COMMON/MATC/SF,LLF,IREP,LDEF
      IPD=-10000
      LDEF=*(($P(+)'COMMUT')($T(*)'COMMUT'))*
      EXPR=*(((X+(3/Y))**2)/(Z-(1/W)))*
      CALL OULST(EXPR)
       PRINT 100
      PATT=*(($1**(-$2))($1 $P1 ($2/$3))($1 $T1 ($2/$3))($1-($2/$3))
     ,(($2/$3)-$1)(($1/$2)/($3/$4))($1/($2/$3))(($2/$3)/$1)
     ,(($2/$1)**$3))*
      REPL=*((1/($1**$2))((($1*$3)+$2)/$3)(($1*$2)/$3)((($1*$3)-$2)/$3)
     ,(($2-($1*$3))/$3)(($1*$4)/($2*$3))(($1*$3)/$2)($2/($3*$1))
     ,(($2**$3)/($1**$3)))*
      M=MATCH(EXPR,PATT,REPL)
      PRINT 100
  100 FORMAT(/////)
       CALL OULST(M)
       STOP
      END
```

```
(((X +(3 / Y))** 2)/(Z -(1 / W)))


(X +(3 / Y))
($1 $P1($2 / $3))
((($1 * $3)+ $2)/ $3)
(($P1 +)($1(X))($2(3))($3(Y))($$P1 +)($$1(X))($$2(3))($$3(Y)))
(((X * Y)+ 3)/ Y)


((((X * Y)+ 3)/ Y)** 2)
(($2 / $1)** $3)
(($2 ** $3)/($1 ** $3))
(($2(((X * Y)+ 3)))($1(Y))($3(2)))
((((X * Y)+ 3)** 2)/(Y ** 2))


(Z -(1 / W))
```

```
($1 -($2 / $3))
((($1 * $3)- $2)/ $3)
 (($1(Z))($2(1))($3(W))($$1(Z))($$2(1))($$3(W)))
(((Z * W)- 1)/ W)


(((((X * Y)+ 3)** 2)/(Y ** 2))/(((Z * W)- 1)/ W))
(($1 / $2)/($3 / $4))
(($1 * $4)/($2 * $3))
(($1((((X * Y)+ 3)** 2)))($2((Y ** 2)))($3(((Z * W)- 1)))($4(W)))
(((((X * Y)+ 3)** 2)* W)/((Y ** 2)*((Z * W)- 1)))


(((((X * Y)+ 3)** 2)* W)/((Y ** 2)*((Z * W)- 1)))
```

The third example shows how to evidence a single occurrence of X originally appearing on the left side of an expression. The replacement rules are essentially the inverse operations or inverse functions with respect to the patterns.

Here the processing mode is linear, with re-application.

Since the expression is completely parenthesized, at each step there is on the left: one operand one operator and a second operand; or a function with its argument; or (at the end of the process) only X.

```
      RECURSIVE MATCH
      COMMON/LSTC/IPD,AREA(10000)
      COMMON/MATC/SF,LLF,IREP,LDEF
      IPD=-10000
      IREP=3
      LDEF=*(($DX 'HAS'(X)))*
      EXPR=*(((K**2)+((ALOG((M+((SIN((((X**3)-K)/(H+4))*(M**5)))**N))
     , -K))*M))=P)*
       CALL OULST(EXPR)
      PRINT 100
      PATT=*((($DX1*$1)=$2)(($1*$DX1)=$2)(($DX1+$1)=$2)(($1+$DX1)=$2)
     , (($DX1-$1)=$2)(($01-$DX1)=$2)(($DX1/$1)=$2)(($1/$DX1)=$2)
     , (($DX1**$1)=$2)(($1**$DX1)=$2)((EXP $DX1)=$1)((ALOG $DX1)=$1)
     , ((SQRT $DX1)=$1)((ATAN $DX1)=$1)((SIN $DX1)=$1)((COS $DX1)=$1))*
      REPL=*(($DX1=($2/$1))($DX1=($2/$1))($DX1=($2-$1))($DX1=($2-$1))
     , ($DX1=($2+$1))($DX1=($01-$2))($DX1=($2*$1))($DX1=($1/$2))
     , ($DX1=($2**(1/$1)))($DX1=((ALOG $2)/(ALOG $1)))
     , ($DX1=(ALOG $1))
     , ($DX1=(EXP $1))($DX1=($1**2))($DX1=((SIN $1)/(COS $1)))
     , ($DX1=(ATAN($1/(SQRT(1-($1**2))))))
     , ($DX1=(ATAN(SQRT(1-($1**2)))/$1)))*
      M=MATCH(EXPR,PATT,REPL)
      PRINT 100
  100 FORMAT(/////)
       CALL OULST(M)
       STOP
      END
```

```
(((K ** 2)+((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))
  ** N))- K))* M))= P)
```

```
(((K ** 2)+((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))
  ** N))- K))* M))= P)
(($1 + $DX1)= $2)
($DX1 =($2 - $1))
(($1((K ** 2)))($DX1(((ALOG(M +((SIN((((X ** 3)- K)/(H + 4))*
  (M ** 5)))** N))- K))* M)))($2(P)))
(((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K))
```

```
  * M)=(P -(K ** 2)))


(((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K))
  * M)=(P -(K ** 2)))
(($DX1 * $1)= $2)
($DX1 =($2 / $1))
(($DX1((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))
  - K))))($1(M))($2((P -(K ** 2)))))
((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K))=
  ((P -(K ** 2))/ M))


((ALOG((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K))=
  ((P -(K ** 2))/ M))
((ALOG $DX1)= $1)
 ($DX1 =(EXP $1))
(($DX1(((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K)))
  ($1(((P -(K ** 2))/ M))))
(((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K)=(EXP((P
  -(K ** 2))/ M)))


(((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))- K)=(EXP((P
  -(K ** 2))/ M)))
(($DX1 - $1)= $2)
($DX1 =($2 + $1))
(($DX1((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))))
  ($1(K))($2((EXP((P -(K ** 2))/ M)))))
((M +((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))=((EXP((P -
  (K ** 2))/ M))+ K))


((M + ((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N))=((EXP((P -
  (K ** 2))/ M))+ K))
(($1 + $DX1)= $2)
($DX1 =($2 - $1))
(($1(M))($DX1(((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N)))
  ($2(((EXP((P -(K ** 2))/ M))+ K))))
(((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N)=(((EXP((P -
  (K ** 2))/ M))+ K)- M))
```

```
(((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))** N)=(((EXP((P -
  (K ** 2))/ M))+ K)- M))
(($DX1 ** $1)= $2)
($DX1 =($2 **(1 / $1)))
(($DX1((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))))($1(N))
  ($2((((EXP((P -(K ** 2))/ M))+ K)- M))))
((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))=((((EXP((P -(K ** 2))
  / M))+ K)- M)**(1 / N)))


((SIN((((X ** 3)- K)/(H + 4))*(M ** 5)))=((((EXP((P -(K ** 2))
  / M))+ K)- M)**(1 / N)))
((SIN $DX1)= $1)
($DX1 =(ATAN($1 /(SQRT(1 -($1 ** 2))))))
(($DX1(((((X ** 3)- K)/(H + 4))*(M ** 5))))($1(((((EXP((P -
  (K ** 2))/ M))+ K)- M)**(1 / N)))))
(((((X ** 3)- K)/(H + 4))*(M ** 5))=(ATAN(((((EXP((P - (K ** 2))
  / M))+ K)- M)**(1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))
  + K)- M)**(1 / N))** 2))))))


(((((X ** 3)- K)/(H + 4))*(M ** 5))=(ATAN(((((EXP((P -(K ** 2))
  / M))+ K)- M)**(1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))
  + K)- M)**(1 / N))** 2))))))
(($DX1 * $1)= $2)
($DX1 =($2 / $1))
 (($DX1((((X ** 3)- K)/(H + 4))))($1((M ** 5)))($2((ATAN(((((EXP((
  P -(K ** 2))/ M))+ K)- M)**(1 / N))/(SQRT(1 -((((EXP((P -
  (K ** 2))/ M))+ K)- M)**(1 / N))** 2)))))))
((((X ** 3)- K)/(H + 4))=((ATAN(((((EXP((P -(K ** 2))/ M))+ K)
  - M)**(1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**
  (1 / N))** 2))))/(M ** 5)))


((((X ** 3)- K)/(H + 4))=((ATAN(((((EXP((P -(K ** 2))/ M))+ K)
  - M)**(1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**
  (1 / N))** 2))))/(M ** 5)))
(($DX1 / $1)= $2)
($DX1 =($2 * $1))
(($DX1(((X ** 3)- K)))($1((H + 4)))($2(((ATAN(((((EXP((P -
  (K ** 2))/ M))+ K)- M)**(1 / N))/(SQRT(1 -(((((EXP((P -
```

```
   (K ** 2))/ M))+ K)- M)**(1 / N))** 2))))))/(M ** 5)))))
 (((X ** 3)- K)=(((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**
   (1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))
   ** 2)))))/(M ** 5))*(H + 4)))


 (((X ** 3)- K)=(((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**
   (1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))
   ** 2)))))/(M ** 5))*(H + 4)))
 (($DX1 - $1)= $2)
  ($DX1 =($2 + $1))
 (($DX1((X ** 3)))($1(K))($2((((ATAN(((((EXP((P -(K ** 2))/ M))+
   K)- M)**(1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**
   (1 / N))** 2)))))/(M ** 5))*(H + 4)))))
 ((X ** 3)=((((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))/
   (SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))** 2)))))
   /(M ** 5))*(H + 4))+ K))


 ((X ** 3)=((((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))/
   (SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))** 2)))))
   /(M ** 5))*(H + 4))+ K))
 (($DX1 ** $1)= $2)
 ($DX1 =($2 **(1 / $1)))
 (($DX1 X)($1(3))($2(((((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**
   (1 / N))/(SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))
   ** 2)))))/(M ** 5))*(H + 4))+ K))))
 (X =(((((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))/
   (SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))** 2)))))
   /(M ** 5))*(H + 4))+ K)**(1 / 3)))


 (X =(((((ATAN(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))/
   (SQRT(1 -(((((EXP((P -(K ** 2))/ M))+ K)- M)**(1 / N))** 2)))))
   /(M ** 5))*(H + 4))+ K)**(1 / 3)))
```

The fourth examples is a simple application of integration by parts (see the end of section 3).

The FORMAL statement is simulated through a sub-routine with the same intended effect.

There are three lists of patterns (and corresponding replacement rules) containing just a few integration, differentiation and simplification rules.

Here the processing mode is linear with no re-application, while at the end of section 3 we showed how this would be done by using the structural scheme with no re-application.

Really, the ability of referencing a whole list of patterns from one particular replacement rule, transforms the linear scheme into a special kind of structural scheme. The differences between this and the standard structural scheme are:

a- the standard scheme is applied to all fragments in any case, while the special one is applied only where the explicit function-like indication is done, and it goes down for as many levels as these indications lead to still other ones in a chain; otherwise the scheme stays linear.

b- the standard scheme works in a bottom-up fashion, and the special one in a top-down fashion

Note that when the replacement rule refers to the corresponding list of patterns we have another case where the algorithm is recursive. In the present example the integration by parts replacement rule refers thrice to the integration patterns.

```
      RECURSIVE MATCH
       COMMON/LSTC/IPD,AREA(10000)
      COMMON/MATC/SF,LLF,IREP,LDEF
      IPD=-10000
      IREP=2
      CALL FORMAL(2HPI,PI,RI,2HPD,PD,RD,2HPS,PS,RS)
      EXPR=*(X*(COS(X)))*
      CALL OULST(EXPR)
      PRINT 100
      PI=*(X $A1 (SIN(X)) (COS(X)) ($1*$2))*
      RI=*(((X**2)/2) ($A1*X) (-(COS(X))) (SIN(X))
     , (($1*('PI'($2)))-('PI'(('PS'((('PI'($2))*('PD'($1)))))))))*
      PD=*(X $A1)*
      RD=*(1 0)*
      PS=*(($1 *1)(1*$1)($1*0)(0*$1))*
      RS=*($1 $1 0 0)*
      L=MATCH(EXPR,PI,RI)
      PRINT 100
  100 FORMAT(/////)
      CALL OULST(L)
       STOP
      END
```

```
(X *(COS(X)))
```

```
(X *(COS(X)))
($1 * $2)
(($1 *('PI'($2)))-('PI'(('PS'((('PI'($2))*('PD'($1)))))))
(($1(X))($2((COS(X)))))
((X *('PI'((COS(X))))-('PI'(('PS'((('PI'((COS(X)))*('PD'(X)))))))
```

```
(COS(X))
(COS(X))
(SIN(X))
()
(SIN(X))
```

```
(COS(X))
(COS(X))
(SIN(X))
()
(SIN(X))


X
X
1
()
1


((SIN(X))* 1)
($1 * 1)
$1
(($1((SIN(X)))))
(SIN(X))


(SIN(X))
(SIN(X))
(-(COS(X)))
()
(-(COS(X)))


((X *(SIN(X)))-(-(COS(X))))


((X *(SIN(X)))-(-(COS(X))))
```

5 - OTHER SYSTEMS:

A few systems were surveyed to look for useful features and to decide on notational problems.

Such systems could be classified as:

a - general pattern.matching languages that are also suitable for other applications such as linguistic work; this class includes COMIT and SNOBOL;

b - embedded packages for mathematical pattern-matching; CONVERT, METEOR and SCHATCHEN were considered;they are all embedded in the LISP 1.5. language;

c - extended languages; here FORMULA ALGOL (and the present system) may be included.

The concept of prototype notation derives from COMIT. Also the use of a double slash to separate the go-to part of the pattern is taken from COMIT.

From SNOBOL we adopted the success and failure exits in the go-to part. A remarkable feature of the SNOBOL language is its backtracking algorithm;

this together with the ANCHOR and UNANCHOR modes or the pattern-valued function POS(n) (of SNOBOL 4) are powerful techniques still not equaled by anything in our system.

We followed the CONVERT system in its use of functions to help or to modify the matching.

The LOOP feature is an interesting feature of SCHATCHEN also not implemented in our system. It allows to find a match even when the sought for pattern elements are non contiguous and false partial recognitions occur.

It can be used when there exists a set of variables in the pattern which are mutually interrelated; the LOOP feature will in turn try all choices of variables until either one is found that matches the pattern or all choices have been tried.

The idea of organizing patterns and replacement rules as lists is also found in FORMULA ALGOL but the schemes for going through them and through an expression are different from those we adopted.

Our feeling is that FORMULA ALGOL is the best of these systems in the sense of naturalness of mathematical notation, although it is to regret that a COMIT-like prototype notation is not included; CONVERT is probably the most powerful system even if it is not so readable as FORMULA ALGOL.

6 - CONCLUSIONS:

The experience so far obtained with this system indicates that pattern--matching in general is a desirable tool in formula manipulation.

It defines transformations in a way that is easy and natural for use by mathematically-minded people.

The programming intricacies that arise in basic pattern handling could very likely be eliminated by a macro-pattern facility. Something as

```
($1**2 + $2**2 + 2*$1*$2 $PM1 $3) → (($1 + $2)**2 $PM1 $3)
```

where $PM1 would be either plus or minus, would find a second degree binomial regardless of the order of the elements and false partial recognitions, and if the remaining terms are related by plus or minus they would be collected in $3.

The system itself would expand this to produce all auxiliary patterns, functions, etc.

The ability to allow references between lists of patterns would also add power to the system.

Besides providing a natural notation, pattern-matching introduces the table-

-driven concept into formula manipulation, since it gives the user total control over transformations, as opposed to the built-in transformations that are available in other formula manipulation systems.

A satisfactory system should really include both features, where:

a - the more usual transformations would be provided in built-in procedures, aiming at maximum efficieny and readiness for use;

b - whenever atypical or less usual transformations were desired, the user would specify them through patterns, thus fulfilling the aim of maximum freedom and flexibility.

7 - BIBLIOGRAPHY:

- Yngve, V. H. - "COMIT as an IR Language" - "Programming Systems and Languages" - edited by Rosen, S. - Mc Graw-Hill - 1967.

- Desautels. E. J. and Smith, D. K. - "An Introduction to the String Manipulation Language SNOBOL" - "Programming Systems and Languages" Edited by Rosen, S. - Mc Graw-Hill - 1967.

- Earley, F.- "Formula ALGOL Manual" - Carnegie - Mellon University - 1967.

- Guzman, A. and Mc Intosh, H. V. - "CONVERT" - Comm. of the ACM, vol. 9, nº 8, August 1966, pp. 604-615.

- Moses, J. - "Symbolic Integration" - MIT - 1967. (see this for a description of the SCHATCHEN system).

- Bobrow, D. G. - "METEOR: A LISP interpreter for string transformation" - "The Programming Language LISP: Its Operation and Applications"- edited by Berkeley, E. C. - Information International - 1964.