PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 2/71

PUCMAT - A PROGRAMMING MODULE FOR ARITHMETIC PATTERN-MATCHING

by

Antonio Luz Furtado

Computer Science Department - Rio Datacenter

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

UC 31449-4

PUCMAT - A PROGRAMMING MODULE FOR ARITHMETIC PATTERN-MATCHING

Antonio Luz Furtado
Associate Professor
Computer Science Department
PUC / RJ

This paper was submitted for publication elsewhere.

Series Editor: Prof. A. L. Furtado

ABSTRACT

An effort towards a convenient notation for pattern-matching as applied to formula manipulation of arithmetic expressions led to the definition of a compact set of statements.

These are being experimented with, at present, under the form of an independent programming language (PUCMAT) which is compiled into an extended FORTRAN IV language via the COMCOM [1] software-writ - ing system.

However PUCMAT will be better utilized as a module of some high level language at a later stage.

PUCMAT is very much indebted to FORMULA ALGOL [2], COMIT [3], SNOBOL [4], CONVERT [5], AXLE [6] and SCHATCHEN [7] .

1. BASIC CONCEPTS:

Suppose one wishes to indicate a transformation from the expression x^n into $ax^n + b$. A natural notation would be

$$x^n \rightarrow ax^n + b$$

Now, if instead of restricting ourselves to the elements x and n we want to express: take a generic expression composed of any element raised to any other element and transform it into a times the expression plus b, we could write

$$e_1^{e_2} \rightarrow ae_1^{e_2} + b$$

It is clearly convenient to represent generic elements such as e_1 and e_2 using a different set of symbols, to avoid any confusion with elements such as a and b that stand for themselves. This introduces what is known as prototype notation, which applied to our example together with usual programming language representation would yield

```
$1**$2    →    A * $1 ** $2 + B
```

Prototypes can be devised to indicate classes of elements. Here we shall consider:

$V - variable

$C - constant

$A - atomic operand (variable or constant)

$Ø - operator

$F - function

$E - expression

$ - universal prototype, matching any "fragment" [5]; (exemplified at the end of this section).

Take:

(1) $V ** $C

We say that the expression

(2) X ** 5

"matches" (1) in the sense that (2) is an instance of the generic expression or "pattern" (1).

While the match is attempted, an "associative set" can be constructed in order to exhibit the correspondence between the proto - types in (1) and specific elements in (2); in our example

(3) (($V X) ($C 5))

A "transformation rule" is specified by taking, say

(4) A * $V ** $C + B

which is called a "skeleton", and forming the sequence pattern→skeleton; in our example

(5) $V ** $C → A * $V ** $C + B

We say that a transformation rule (5) is "applied" to an expression (2), meaning that:

a. a match between (1) and (2) is tried ((1) and (2) being scanned from left to right);

b. as the match proceeds (3) is constructed;

c. if the match fails no further action is taken;

d. if the match succeeds a new expression is created as a copy of (4) except for the prototypes, which are replaced by their corresponding elements in (3).

Thus (5) applied to (2) gives

A * X ** 5 + B

A few remarks should be added about the prototypes. Possibly different occurrences are distinguished by numbering; $V1 - $V2, for instance, would match A - B.

When the non-occurrence of an element does not invalidate the match, the number part of the prototype should begin with zero;

$V01 - $V2 would match A - B but it would also match - B.

The "universal" prototype $ is matched as exemplified below:

expression: A + B - C

pattern: $1 - $2

associative set: (($1 (A + B)) ($2 C))

that is, the pattern element " - " delimits the range of $1, as the end of the expression delimits $2; in general, non-universal prototypes or end of expression delimit the recognition of a universal prototype.

2. INFORMAL DESCRIPTION OF THE PROGRAMMING MODULE:

The programming module is composed of three statements:

2.1 - The DEFINE Statement:

Its purpose is to allow the user to create new prototypes.

Since prototypes are symbols that stand for classes, the creation is done by specifying the symbol and then attaching to it the criterium for class membership.

The criterium is either:

- enumeration of members;
- a logical predicate that is true for any member;
- a combination of the preceding criteria in a logical expression, using the operators AND, OR, NOT

There are two built-in logical predicates: HAS, FREEOF. User coded predicates are allowed.

Besides the criterium for class membership two other specifications may be attached:

- the COMMUT attribute, provided that the new prototype corresponds to a class of operators;

- the COLLECT mechanism, through which during the application process new members are added to the initial enumeration (which could be empty).

Example of the DEFINE statement:

```
DEFINE  $P   (+ , *)  COMMUT,
        $Q   (A, B, C, D),
        $R   ()  COLLECT ($C1),
        $S   FREEOF (X) AND (HAS(Y) OR HAS(Z));
```

(NOTE: successful matches to patterns containing $C1 will cause the associated elements to be added to $R).

2.2 - The set-of-rules Statement:

In most cases transformation rules should be grouped together to provide for different cases in a same transformation process. If one is differentiating with respect to x, for instance, the cases of: x, a variable independent of x, sums, products, etc. have to be considered.

The set-of-rules statement is a labeled set of transformation rules, which are numbered.

The first rule is always tried first, but what happens next, for the first and the other rules, depends on:

- the success of the match with the rule;
- whether the new expression is different from the original one (whether a transformation effectively occurred).

Attached to each rule there are the success and the failure exits. The former is taken if both conditions above are fulfilled. Each exit may indicate a transfer to another rule in the set (by the number of the rule) or termination (by an asterisk). Of course in case of success the application will proceed (by the rule indicated in the success exit) with the transformed expression rather than the original one.

The success of the match may cause another action before any exit is taken. The skeleton part of the rule may refer to the whole set-of-rules (recursion) or to other sets-of-rules. In both cases the label of the set-of-rules to be invoked is indicated between dots.

Thus the statement gives the user a very tight control over the sequence of the process, extending the original Markovian algorithm [2]. The need for this has been recognized elsewhere [15]

Invoked sets-of-rules that are not present in the job stream are assumed to be available from library. This feature has a few implications:

- the user is relieved from rewriting commonly used sets-of-rules, unless he wants atypical rules;

- a library set-of-rules can be coded using any technique (not necessarily a pattern-matching technique) and even any language (not necessarily PUCMAT, provided that compatibility and communication is ensured in the implementation);

- non-transformational "functions" can be invoked in place of sets-of-rules as for example COMB [8] that would be required for a binomial expansion.

Classes attached to user-defined prototypes can be altered by several ways, including the COLLECT machinery which gives rise to the concept of variable rules. Different transformations are specified by such rules, according to the current members of attached classes.

Example of the set-of-rules statement:

```
DIFF: 1. X   →   1   //   *  2,
      2. $A  →   0   //   *  3,
      3. $1 + $2  →  .DIFF.($1) + .DIFF.($2) // * *;
```

2.3 - The Application Statement:

This statement causes a set-of-rules to be applied to an arithmetic expression.

Three kinds of arithmetic entities are considered:

- formal constants - these are symbols that stand for themselves, such as A, +, (,), etc.;

- formal variables - these are the "names" (pointers) of arithmetic expressions; when appearing to the rigth of the " ← " sign they are delimited by dots;

- formal classes - these are the classes corresponding to and indicated by prototypes.

An arithmetic expression can be composed of any combination of these entities in the same expression as for instance in A * (.X. + B) - $Z(2).

However a decision must be taken on whether to "unravel" [9] or not non-constant symbols before the application statement delivers the expression to the set-of-rules. (For a very elegant treatment of unraveling see [10]).

Our decision was:

- automatic unraveling for all formal variables, which means that they will all be replaced by the expressions they are naming;

- automatic unraveling of prototypes only if followed by an index, as in $X(3), the index being used to indicate which member of the corresponding class will be taken; as opposed to the immediate unraveling of formal variables, the unraveling of prototypes takes place at each moment a match is tried, thus considering the current members of the class.

An expression containing unraveled prototypes is regarded as a "family" of expressions. Note that patterns and skeletons are also such families, and thus an application could be used to generate new patterns and skeletons from previous ones.

Coming back to the more restricted concept of variable rule, consider the rule $X → $Y. Since the prototype in the skeleton is not included in the pattern it will remain unraveled after a successful match. The user is allowed to introduce a function for a controlled unraveling, e. g.: $X → .LAST.($Y).

After the application is performed the application statement assigns to a formal variable or to a prototype the resulting new expression (if no transformation occurred, this will be the original expression unchanged).

Also the transformation path, i. e. the concatenation of all successfully applied rules (if any) is assigned to a special kind of name.

What is concatenated, for each rule, is the label of the set-of-rules to which it belongs followed by the rule number. Brackets are used to indicate a "deeper" level along the path, wherever a recursive call or a call to other sets-of-rules occur.

It is convenient to remark here that a set-of-rules could be looked at as a graph (whose nodes are graphs in the case of invocation of the same or other set-of-rules). By the same reasoning, an application causes the expression to describe a path (whose nodes can in turn be paths) along the set-of-rules graph.

The transformation path serves to:

- indicate whether any or none of the rules succeeded, being empty in the latter case;

- display the "history" of the application;

- define a very interesting "equivalence" between arithmetic expressions: two or more arithmetic expressions are said to be equivalent under a chosen set-of-rules if and only if their transformation paths along it are equal.

Of course transformation paths could also be investigated for isomorphism or any kind of similarity, inclusion, etc.

Example of an application statement, noting that the first identifier will name the new expression and the second one the transformation path:

```
Y, M ← SIMPL(A + 0 * .B. * 1);
```

3. BNF DESCRIPTION OF THE PROGRAMMING MODULE:

<define statement>:: = define <sequence of definitions>;

<sequence of definitions>:: = <definition> | <sequence of definitions>, <definition>

<definition> :: = <prototype> <logical expression>

<prototype> :: = $| $ || <unsigned integer> | $ || <letter> | $ || <letter> || <unsigned integer>

<logical expression> :: = <logical operand> | <logical expression> <binary logical operator> <logical operand>

<binary logical operator> :: = and | or

<logical operand> :: = not <logical expression> | (<logical expression>)| <logical primary>

<logical primary> :: = <set> collect <set> | <predicate> <set> | <set> commut | <set>

<set> :: = ( ) | (<sequence of members>)

<sequence of members> :: = <member> | <sequence of members>, <member>

<member> :: = <arithmetic expressior - 1>

<arithmetic expression - 1> :: = <element - 1>| <arithmetic expression -1>
<element - 1>

<element - 1> :: = <operator> | <identifier> | <unsigned number> |
<prototype> | (<arithmetic expression - 1>) | ( )

<operator> :: = + |-|*|/|**| =

<predicate> :: = has |freeof| <user defined predicate>

<user defined predicate> :: = <identifier>

<set -of-rules statement> :: = <label> : <sequence of rules>;

<label> :: = <identifier>

<sequence of rules> :: = <rule> | <sequence of rules>, <rule>

<rule> :: = <rule number> . <pattern> → <skeleton> // <exit - 1>
<exit - 2>

<rule number> :: = <unsigned integer>

<pattern> :: = <arithmetic expression -1>

<skeleton> :: = <arithmetic expression -2>

<arithmetic expression - 2 >:: = <element - 2>| <arithmetic expression -2>
<element - 2>

<element - 2> :: = <arithmetic expression - 1> | <invocation> |(< arith-
metic expression -2>)

<invocation> :: = .<identifier> . (<arithmetic expression - 2>)

<exit - 1> :: = <unsigned integer>

<exit - 2> :: = <unsigned integer>

<application statement> :: = <name> , <path> ← (<argument>);

<name> :: = <identifier> | <prototype>

<path> :: = <identifier>

<argument> :: = <arithmetic expression - 3>

<arithmetic expression - 3> :: = <element - 3> | < arithmetic expression - 3><element - 3>

<element - 3> :: = <arithmetic expression - 1> | <formal variable> | <subscripted prototype> | (<arithmetic expression - 3>)

<formal variable> :: = . <identifier>.

<subscripted prototype> :: = <prototype> (<unsigned integer>)

Definitions of identifier, letter, unsigned number, and unsigned integer are the usual ones [14] . The symbol || means concatenation.

It might be noted that the definitions of the three kinds of arithmetic expressions allow any sequence of arithmetic elements, checking only for unbalanced brackets; this permits special forms of expressions, such as Polish forms.

4. SOME EXAMPLES:

In the five following examples, two features of the present implementation are included:

a. The *BEGMAT and *ENDMAT control cards;

b. Comments, delimited as in PL/I by /* and */.

We also note that the application statement causes the print-out of the transformed expression and transformation path.

Example 1

```
*BEGMAT

/*      DIFFERENTIATION WITH IMMEDIATE SIMPLIFICATION OF RESULTING
        SUB-EXPRESSIONS     */

/*      DEFINES COMMUTATIVE  +  AND  *
        AND THE SET OF VARIABLES DEPENDING ON X  */

DEFINE $P    (+)  COMMUT,
       $T    (*)  COMMUT,
       $X    (Y, Z, W);

/*      COMBINED DIFFERENTIATION AND SIMPLIFICATION RULES  */
```

```
DER:  1. X              → 1                              // *  2,
      2. $X             → D($X)                          // *  3,
      3. $A             → 0                              // *  4,
      4. $1 + $2        → .DER.($1)+.DER.($2)            // 8  5,
      5. $01 - $2       → .DER.($01)-.DER.($2)           // 10 6,
      6. $1 * $2        → $1*.DER.($2)+$2*.DER.($1)      // 13 7,
      7. D($)           → D(D($))                        // *  *,
      8. $1 $P 0        → $1                             // *  9,
      9. $1 + $1        → 2*$1                           // *  *,
     10. - 0            → 0                              // * 11,
     11. $ - 0          → $                              // * 12,
     12. $1 - $1        → 0                              // *  *,
     13. $1 $T 0 $P $2  → $2                             //14 16,
     14. $ $T 0         → 0                              // * 15,
     15. $ $T 1         → $                              // *  *,
     16. $1 $T 1 $P $2  → $1 + $2                        // 16 9;

/*   APPLICATION   */

Y, M ← DER (  -A * X + Z);

/*   RE-APPLICATION TO OBTAIN SECOND DERIVATIVE  */

Z, K ← DER (.Y.);

*ENDMAT
```

Result of first application: -A + D(Z)

Path: (DER4(DER6(DER1)(DER5()(DER3)DER10) DER13 DER15)(DER2))

```
Result of second application:  D(D(Z))
Path: (DER4 (DER5 () (DER3) DER10) (DER7) DER8)
```

Example 2 [2]

```
*BEGMAT

/*    CLEARING OF FRACTIONS  */

/*    DEFINES COMMUTATIVE  +  AND  *    */

DEFINE   $P  (+)  COMMUT,

         $T  (*)  COMMUT;

/*    THE RULES REDUCE FRACTIONS TO A SIMPLER FORM */

CLEAR:  1.  $1 ** (-$2)     →  1/$1 ** 2                    // 1  2,
        2.  $1 $P $2/$3     →  ($1 * $3 + $2)/$3            // 1  3,
        3.  $1 $T ($2/$3)   →  ($1 * $2)/$3                 // 1  4,
        4.  $1 - $2/$3      →  ($1 * $3 - $2)/$3            // 1  5,
        5.  $2/$3 - $1      →  ($2 - $1 * $3)/$3            // 1  6,
        6.  $1/($2/$3)      →  ($1 * $3)/$2                 // 1  7,
        7.  ($2/$3)/$1      →  $2/($3 * $1)                 // 1  8,
        8.  ($2/$1) **$3    →  $2 ** $3/$1 ** $3            // 1  9,
        9.  $1 $Ø $2        →  .CLEAR.($1) $Ø .CLEAR.($2)   // 1  *;

 /*     APPLICATION    */

Y, M    ← CLEAR ( (X + 3/Y) ** 2/ (Z - 1/W)   );

*ENDMAT
```

```
Result of application: W * (X * Y + 3) * * 2 / (Y * * 2 *(Z * W-1))

Path: (CLEAR9 (CLEAR9 (CLEAR2 CLEAR9 (CLEAR9(CLEAR9 () ()) ()) ()) ()
      CLEAR8 CLEAR9 (CLEAR9(CLEAR9(CLEAR9 () ()) ()) ()) (CLAER9 () ()))
      (CLEAR4 CLEAR9(CLEAR9 (CLEAR9 () ()) ()) ()) CLEAR6 CLEAR9 (CLEAR3
      CLEAR9 (CLEAR9 () (CLEAR9 (CLEAR9 (CLEAR9 () ()) ()) ())) (CLEAR9
      () ())) (CLEAR9 (CLEAR9 () ()) ()) CLEAR7 CLEAR9 (CLEAR9 () (CLEAR9
      (CLEAR9 (CLEAR9 () ()) ()) ())) (CLEAR9 (CLEAR9 () ()) (CLEAR9
      (CLEAR9 () ()) ())))
```

Example 3 [11]

```
*BEGMAT

/*      SOLUTION OF AN EQUATION FOR A SINGLE OCCURRENCE OF X  */

/*      DEFINES COMMUTATIVE + AND *        */

DEFINE  $P   (+)  COMMUT,

        $T   (*)  COMMUT,

        $X  HAS(X);

/*      THE RULES GIVE INVERSE OPERATIONS AND FUNCTIONS  */
```

```
SOLVX: 1.  $X $T $1 = $2 →  $X = $2/$1                        // 1  2,
       2.  $X $P $1 = $2 →  $X = $2 - $1                      // 1  3,
       3.  $X -  $1 = $2 →  $X = $2 + $1                      // 1  4,
       4.  $1 -  $X = $2 →  $X = $1 - $2                      // 1  5,
       5.  $X /  $1 = $2 →  $X = $2 * $1                      // 1  6,
       6.  $1 /  $X = $2 →  $X = $1/$2                        // 1  7,
       7.  $X ** $1 = $2 →  $X = $2 ** (1/$1)                 // 1  8,
       8.  $1 ** $X = $2 →  $X = ALOG($2)/ALOG($1)            // 1  9,
       9.  EXP($X)  = $1 →  $X = ALOG($1)                     // 1 10,
      10.  ALOG($X) = $1 →  $X = EXP($1)                      // 1 11,
      11.  SQRT($X) = $1 →  $X = $1 ** 2                      // 1 12,
      12.  ATAN($X) = $1 →  $X = SIN($1)/COS($1)              // 1 13,
      13.  SIN($X)  = $1 →  $X = ATAN($1/SQRT(1-$1**2))       // 1 14,
      14.  COS($X)  = $1 →  $X = ATAN(SQRT((1-$1**2)/$1))     // 1  *;


/*    APPLICATION    */

Y, M ← SOLVX(K ** 2 + ALOG(M + SIN(( X ** 3 - K)/(H + 4)* M ** 5)
            ** N - K) * M = P);

*ENDMAT

Result of application: 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)

Path: (SOLVX2  SOLVX1  SOLVX10  SOLVX3 SOLVX2  SOLVX7  SOLVX13  SOLVX1
      SOLVX5  SOLVX3  SOLVX7)
```

Example 4

```
*BEGMAT

/*    INTEGRATES A SIMPLE EXPRESS   */

/*    DEFINES COMMUTATIVE  *

      THE SET OF  +  AND -

      AND EXPRESSIONS NOT CONTAINING  X  */

DEFINE  $T  (*)  COMMUT,

        $S  (+, -)     ,

        $I  FREEOF(X)  ;

/* INTEGRATION RULES  */

INTG:  1.  X          → X ** 2/2                              // *  2,
       2.  X * * $I   → X ** ($I+1)/($I+1)                    // *  3,
       3.  $A         → $A * X                                // *  4,
       4.  $1 $S $2   →.INTG.($1) $S .INTG.($2)               // *  5,
       5.  SIN(X)     → -COS(X)                               // *  6,
       6.  COS(X)     → SIN(X)                                // *  7,
       7.  $1 * $2    →.SIMP.($1 *.INTG.($2)-.INTG.(.SIMP.
                        (.INTG.($2) *.DER.($1))))             // *  *;

/*     DIFFERENTIATION  RULES   */

DER:   1.  X → 1     // * 2 ,
       2.  $A → 0    // * * ;
```

```
/*  SIMPLIFICATION RULES  */

SIMP:  1.  $1  $T  0      →   0           //*  2,
       2.  $1  $T  1      →   $1          //*  3,
       3.  $1  - - $2     →   $1 + $2     //*  *;

/*  APPLICATION  */

Y, M  ← INTG(X ** (A + B) - X * COS(X)    );

*ENDMAT

Result of application: X **(A + B + 1)/(A + B + 1)-(X * SIN(X) + COS(X))

Path:(INTG4  (INTG2)(INTG7((INTG6)(((INTG6)(DER1)SIMP2)INTG5)SIMP3)))
```

Example 5

```
*BEGMAT

/*     EXTRACTION OF CONSTANT COEFFICIENTS AND POWERS OF A POLYNOMIAL IN
       X          */

/*     DEFINES THE CLASSES OF COEFFICIENTS AND POWERS
       AND THE SET OF  +  AND  -                */

DEFINE $K  ()   COLLECT ($S01 $C1),
       $P  ()   COLLECT ($S02 $C2),
       $S  ( + , -);

/*     RULE FOR DELETING EACH TERM      */
```

```
POL:  1.  $S01  $C1 * X ** $S02  $C2  $0 → $0  // 1  *;

/*    APPLICATION   */

Y, M ← POL ( 2 * X ** 3 - 3 * X ** - 5 + 7 * X ** 2 + 9 * X ** - 4 - 11 *
          X ** 12);

/*    PRINTS $K  AND  $P   */


$K ← $K;
$P ← $P;

*ENDMAT
```

Result of application: ()

Path: (POL1 POL1 POL1 POL1 POL1)

Members of $K: (2, - 3, + 7, +9, - 11)

Members of $P: (3, - 5, 2 , - 4 , 12)

NOTE: The statements $K← $K and $P← $P are instances of the "extension" statements (to be described at section 5) that do not strictly belong to PUCMAT; the instances given here serve only as a device to print the classes associated to $K and $P, without altering them.

5. IMPLEMENTATION CONSIDERATIONS:

PUCMAT is implemented as an independent language during the present experimental stage.

Programs in PUCMAT are compiled into an extended $IBFTC FORTRAN IV compiler (IBM - 7044) via a compiler-compiler [1].

This extended FORTRAN IV allows recursion and LISP-like list-processing. An important library facility is an assembly-coded lexical scanner that helps considerably the matching process.

The most interesting point in the compilation strategy is that sets-of-rules become recursive function sub-programs. So the calling mechanisms, communication through parameters and COMMON areas and insertion in the systems library all use FORTRAN ordinary facilities. If PL/I or ALGOL had been chosen as host language the sets-of-rules would be procedures.

All such sub-programs call the recursive sub-program MATCH (to perform the pattern-matching activities) and are called by it whenever a successful rule indicates a transfer to the same set-of-rules or to another one. This constitutes an indirectly recursive scheme.

Another point is the generated call to a sub-program CANON, in order to convert all expressions, including patterns and skeletons, to a canonical form.

The user is free to introduce his own CANON, thus superseding the built-in one, which might be necessary in certain cases, since

canonical forms are frequently only "locally" adequate [12] .

The current built-in CANON performs only complete parenthetization. It seems desirable to introduce a unique parenthetization scheme and lexicographic ordering, to cope with the problems of associative operators and equal non-contiguous operands.

One feature that is still unimplemented at the time of this writing is the combination of predicates and class enumerations into logical expressions. Consideration is also being given to the following features, that might give the user an even tighter control over the matching sequence, thus increasing the speed of the process:

- graph-directed application - given a large set-of-rules the user could take advantage from his knowledge that his expressions would only need a few of the rules, to specify a smaller graph over the nodes of the original one; suppose for instance that he wants to use the set-of-rules of example 1 of section 4 but his expressions contain only the variable X, variables independent of X, constants, and sums;he could write:

```
G:  1.  //  *  3,
    3.  //  *  4,
    4.  //  8  *,
    8.  //  *  9,
    9.  //  *  *;

Y, K ← DER/G(A + 5 + X + C);
```

- path - directed application - if the user knows that all his expressions are equivalent under the set- of- rules he can specify the transformation path and use it for sequence control; taking example 3, of section 4 suppose all expressions are equivalent to A + B * (N / X + 2) = R; then one could write:

```
P ⟵ 2, 1, 2, 6;

Y, M ⟵ SOLVX/.P. (A + B * (N / X + 2) = R);
```

A host language allowing on-line modification, expansion or reduction of procedures in an interactive environment would be particularly suitable for programs with "learning" and "forgetting" features. One possibility would be the deletion of rules that do not belong to the transformation path of any of several submitted expressions.

As we aim at a future implementation of PUCMAT as an extension (module) of a high-level language we have studied a few cases of interaction of PUCMAT and other features even in the present implementation.

With this purpose we introduced an "extension" statement into PUCMAT.

This has either the form

```
formal variable or prototype ⟵ formal expression;
```

or

formal variable or prototype ⟵ function (formal arguments);

where "function" may be any FORTRAN coded function sub-program.

Most experiments where done with set-theoretic functions (intersection, union, difference), using the LISP-like processing feature, and taking COLLECTed classes of prototypes as arguments, as in

$X ⟵ UNION ($Y, $Z);

Here, unraveling of prototypes is automatic; they are replaced by their associated classes.

Of even greater interest will be the combination of formal and numerical processing.

An effort in another direction will apply pattern-matching to the manipulation of algebraic structures (groups, rings, etc.).

6. CONCLUSIONS:

Our experience indicates that PUCMAT notation achieves a good degree of simplicity and naturalness to express the transformation of arithmetic expressions.

Both flexibility and efficiency are increased by giving the user a considerable degree of control over the sequence, through the success and failure exits, explicit recursive calls and calls to other sets-of-rules.

Note also that treating sets-of-rules as functions (or procedures) allows a flexible and efficient combination with functions that do not employ pattern-matching techniques whenever this is convenient, and even with auxiliary non-transformational functions; it also permits the creation of a library of sets-of-rules.

The concept of transformation path and its use to test for relationships between expressions under sets-of-rules appear to be a promising research tool.

This speaks for the desirability of adding the small set of PUCMAT statements and underlying processes to a high level language, on the sole condition that the latter already offers recursion and list-processing.

AKNOWLEDGEMENTS - the present research was made possible thanks to Prof. A. von Staa's COMCOM [1] system and his lexical scanner and output routines, and also to Prof. S. S. Toscani's implementation of recursion on the $IBFTC compiler [13].

REFERENCES

1. Staa, A. von - The COMCOM software-writing system - series: Monographs in Computer Science and Computer Applications - 8/70. Pontifícia Universidade Católica do Rio de Janeiro - 1970.

2. Earley, F. - FORMULA ALGOL Manual-Carnegie-Mellon University 1967.

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

4. Desautels, E. J. and Smith, D. K. - An Introduction to the String Manipulation Language SNOBOL - ibidem.

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

6. Cohen, K. and Wegstein, J. H. - AXLE: An Axiomatic Language for String Transformations - Comm. of the ACM, vol. 8 nº 11, pp. 657 - 661. November 1965.

7. Moses, J. - Symbolic Integration - MIT - 1967

8. Tobey, R. et al - PL/I FORMAC Interpreter - IBM Manual 360 D - 03.3.004 - 1967.

9. Sammet, J. E. - Formula Manipulation by Computer - in Advances in Computers- 8 - edited by Alt, F. L. and Rubinoff, M. - Academic Press - 1967.

10. Engelman, C. - MATHLAB 68 - Proc. IFIP Congress 68 - Edinburgh, August, 1968.

11. Raphael,B. et al- A Brief Survey of Computer Languages for Symbolic and Algebraic Manipulation - in Symbol Manipulation Languages and Techniques - edited by Bobrow, J. G. North-Holland - 1968.

12. Caviness, B. F. - On Canonical Forms ans Simplification - JACM , vol. 17 nº 2, April 1970 - pp. 385 - 396.

13. Toscani, S.S. - Recursividade em FORTRAN - M. Sc. Thesis - Pontifícia Universidade Católica do Rio de Janeiro 1969.

14. Naur, P. (ed.) - Revised Report on the Algorithmic Language ALGOL 60 - Comm. of the ACM, vol. 6 nº 1, pp. 1 - 17 January - 1963.

15. Galler, B. A. and Perlis, A.J. - A View of Programming Languages - Addison - Wesley 1970.