PUC

Series: Monographs in Computer Science
and Computer Applications
Nº 9/75

AN INCREMENTAL EXTENSION FOR RECURSIVE PROGRAMS AND STRUCTURES IN FORTRAN

by

A.L. Furtado

and

Daniel Schwabe

Departamento de Informática

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

Series: Monographs in Computer Science
and Computer Applications
Nº 9/75

AN INCREMENTAL EXTENSION FOR RECURSIVE PROGRAMS AND STRUCTURES IN FORTRAN *

by

A. L. Furtado

and

Daniel Schwabe

Series Editor: Larry Kerschberg

December, 1975

* This work was supported in part by the Brazilian Government Agency FINEP under contract Nº 244/CT, and developed by the Information Systems Group, Departamento de Informática, PUC/RJ.

ABSTRACT

An extension to FORTRAN is proposed which is incremental in the sense that it comprises three successive stages, each stage requiring the availability of the lower ones. The three stages are: recursion, LISP-like list processing, and run-time compilation of arithmetic expressions that have been created and symbolically manipulated in list representation.

The purpose of the extension is to increase the power of the language by making it suitable for application in areas where recursive programs and structures are indicated. It is therefore an "orthogonal extension" [1], rather than a mere expansion of the syntax.

Also, the syntax itself has been augmented (never changed) as little as possible, trying to find a compromise between the ease of expression and the desire to conform to the original style of the language.

KEYWORDS

FORTRAN, extensible languages, recursion, LISP, list processing, formula manipulation.

RESUMO

Uma extensão ao FORTRAN é proposta, sendo incremental no sentido de que compreende três estágios sucessivos, cada estágio requerendo a disponibilidade dos inferiores. Os três estágios são: recursão, processamento de listas como em LISP, e compilação em tempo de execução de expressões aritméticas que foram criadas e manipuladas simbolicamente como listas.

O objetivo da extensão é aumentar o poder da linguagem tornando-a adequada para aplicação em áreas em que programas e estruturas recursivas são indicadas. É portanto uma "extensão ortogonal" [1], e não uma simples expansão da sintaxe.

Também, a própria sintaxe foi aumentada (nunca mudada) tão pouco quanto possível, tentando achar um equilíbrio entre a facilidade de expressão e a conformidade com o estilo original da linguagem.

PALAVRAS CHAVE

FORTRAN, linguagens extensíveis, recursividade, LISP, processamento de listas, manipulação de fórmulas.

CONTENTS

1. INTRODUCTION

FORTRAN is one of the first high level programming languages to appear, and is perhaps the most widely used.

Since the time of its appearance, the computing field has developed remarkably, giving rise to increasingly sophisticated applications and not surprisingly laying bare many shortcomings of FORTRAN in coping with them . Nevertheless, FORTRAN has retained a considerable appeal to many users, by virtue of its simplicity and efficiency. As a consequence, many extensions to it have been suggested [2].

In this paper we propose an "incremental" extension to FORTRAN, based on our early experiments in extending the IBSYS FORTRAN compiler (on an IBM 7044) [3] .

The extension comprises three levels, each of them using the preceding ones, but being useful on its own right.

Level one - recursion - is available in several widely known programming languages, such as ALGOL, PL/I, SNOBOL, etc. Many algorithms have been formulated in terms of recursive formulas, and can be readily programmed if recursion is supported by the programming language.

Level two - LISP - like list processing - has proved its usefulness especially in non-numerical computation. The combination of block - structured languages (supporting both recursion and iteration) with lists appears to be very convenient (see the LISP 2 proposal [4] and CPL [5]). It may also be argued that LISP-lists are better from a structured programming point of view than the structures plus unrestricted pointers of PL/I.

Level three - run-time compilation of arithmetic expressions - has to do with formula manipulation, a capability that has been added to FORTRAN in the past and more recently to PL/I via the FORMAC pre-processors, and to ALGOL in the FORMULA ALGOL system [6].

Our extension is far less ambitious than FORMAC. LISP-lists are used for representing formulas, which has the advantage of not requiring a special data structure for this purpose. So, at level three the only additional feature is precisely the ability to compile the given or derived formulas and allowing

their direct (as opposed to interpretive) execution. This feature is of special interest to the numerical analyst who may need to evaluate a formula many times, and does not want to incur in the inefficiency of interpretive evaluation as offered by FORMAC. Languages like SNOBOL have also run-time compilation,but with a far more general scope than is needed here; in the present case one has to deal only with arithmetic expressions rather than all statements in the language.

In designing our three-level extension we had in mind the following goals:

a. To make FORTRAN more adequate for writing algorithms in a natural way (recursion) and to cope with additional fields of application (by having LISP-lists and run-time compilation).

b. To allow a relatively easy implementation, either by changing the compiler or by writing a pre-processor.

c. To be consistent with the general style of the language, i.e. we do not wish to create an entirely different language, thereby missing the wide knowledge of FORTRAN by so many users.

In short, we strived for maximum effectiveness - a moderate change resulting in the greatest possible increase in the power of the language.

2. RECURSION

Recursion is a property of many objects we deal with in programming; recursive functions and subroutines are usually the best way to treat recursively defined data types, as for instance lists.

Therefore it would be very useful to have recursive functions and subroutines in FORTRAN. As an example, consider the following formulas for computing the number of combinations of n objects taken m at a time:

$$C\binom{1}{n} = n$$

$$C\binom{n}{n} = 1$$

$$C\binom{m}{n} = C\binom{m-1}{n-1} + C\binom{m}{n-1}, \quad \text{for } m \neq n \text{ and } m \neq 1$$

This leads immediately to the following recursive function:

```
      RECURSIVE INTEGER FUNCTION C (N, M)
          INTEGER  N, M
            IF  (M .EQ. 1)   GO  TO 1
                             GO  TO 2
1           C = N
            RETURN
2           IF  (M .EQ. N)   GO  TO 3
                             GO  TO 4
3           C = 1
            RETURN
4           C = C (N-1, M-1) +  C (N-1, M)
            RETURN
      END
```

Another example is the well-known QUICKSORT method [7] for ordering the elements of a set. The algorithm can be thus described:

```
QUICKSORT (S)
  if #S  = 1 then return
             else begin
                  v = any element of S ;
                  S1= { w∈S  such that w < v } ;
                  S2= { w∈S  such that w = v } ;
                  S3= { w∈S  such that w > v } ;
                  S = QUICKSORT(S1) || S2 || QUICKSORT(S3) ;
                  end
end QUICKSORT
```

which could be programmed in recursive FORTRAN as follows:

```
      RECURSIVE SUBROUTINE Q(I, L)
        INTEGER   I, J, K, L
C ------------------------------------ END OF RECURSION
        IF (I .EQ. L) RETURN
C ------------------------------------ BUILD S1, S2, S3
        CALL SPLIT (I, J, K, L)
C ------------------------------------ IF S1 IS NOT EMPTY, SORT IT
        IF (I .NE. J) CALL Q(I, J-1)
C ------------------------------------ IF S3 IS NOT EMPTY, SORT IT
        IF (K .NE. L) CALL Q(K+1, L)
      END


      SUBROUTINE SPLIT (I, J, K, L)
        INTEGER I, J, K, L, V, M, SS(100), S, N
        COMMON /QUICK/ S(100), N
        J = I
        K = L
        V = IRAND (I, L)

        DO 1 M = I TO L
           SS(M) = S(M)
1       CONTINUE
C ------------------------------------ SS NOW CONTAINS A WORKING COPY OF S(I
        DO 5 M = I TO L
        IF (SS(M) .LT.SS(V)) GO TO 3
        IF (SS(M) .GT.SS(V)) GO TO 4

C ------------------------------------ FORM S1
3       S(J) = SS(M)
        J = J+1
        GO TO 5
```

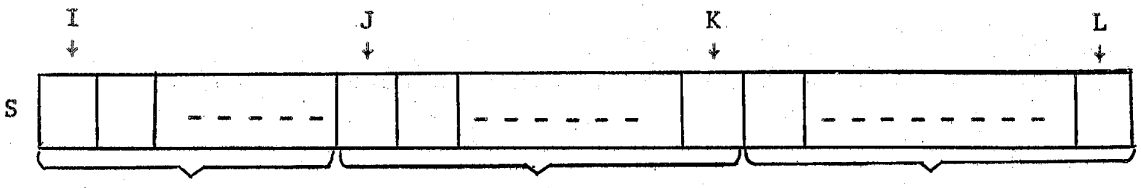
```
C ------------------------------------ FORM S3
4       S(K) = SS(M)
        K = K-1

5     CONTINUE
C ------------------------------------ FORM S2
      DO 6 M = J, K
         S(M) = SS(V)

6     CONTINUE
      RETURN

   END
```

The figure below explains the meaning of I, J, K, L.



S1 ≡ S (I : J-1) S2 ≡ S (J : K) S3 = S (K+1 : L)

Thus, subroutine Q uses I , J, K , L to delimit within S the space occupied by S1, S2, and S3, which are composed by subroutine SPLIT. IRAND is a function that returns a random integer in the range indicated by its arguments. The calling program should include the statement CALL Q(1,N) to start the process.

Returning to the general discussion of recursion, we remark that FORTRAN provides a convenient way for distinguishing three classes of variables: parameters, local variables, and global variables. Parameters are those which appear in the parameters list; global variables are those appearing in COMMON statements; and all other variables are local. Parameters and local variables are handled by the recursion stack, whereas global variables are not; this justifies the use of COMMON in a recursive subprogram simply for assigning a static attribute to a variable, regardless of whether or not the COMMON statement is serving its regular communication purpose (i.e., the COMMON statement may appear in the recursive subprogram and nowhere else).

Less convenient are the restrictions imposed by FORTRAN on the use of arrays with variable dimensions (which may be parameters but not local variables), and the need for a special kind of subprogram for initializing variables in COMMON.

About the moment when the push-down operation on the recursion stack should take place, we have decided for performing the operation upon entrance to the subprogram, and the reverse pop-up operation is performed upon exit. This method avoids problems arising with indirect recursion, which occurs when a chain of calls to subprograms becomes a cycle; the following example illustrates the situation:

```
RECURSIVE SUBROUTINE A RECURSIVE SUBROUTINE B RECURSIVE SUBROUTINE C
          .                      .                      .
          .                      .                      .
          .                      .                      .
      CALL B                 CALL C                 CALL A
          .                      .                      .
          .                      .                      .
          .                      .                      .
```

A decision of pushing down immediately before a recursive call would not work in this case, because the compiler seeing each subroutine in separate would not recognize the above CALLs as forming a cycle; however, since the subroutines are declared as recursive the push-down upon entrance would work.

A consequence of our decision is that no side-effects are possible with parameters - but they are possible with variables in COMMON. This is tantamount to saying that if a subprogram is declared as recursive it will be executed as if it were called by value.

The recursion stack could utilize the unused part of the space allocated to the job. Each entry in the stack would contain the identification of the subprogram, return address, addresses and values of arguments and local variables; for array variables a kind of dope vector would be included, containing: address of base element, number of elements, etc. (plus of course the values of all the array elements - which explains why we were careful to avoid stacking any arrays in the QUICKSORT example).

3. LIST PROCESSING

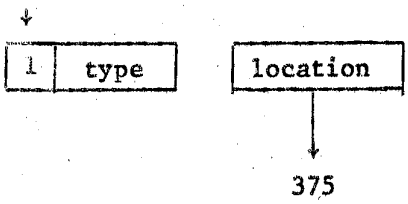
The constant .NIL. and variables of type SYMB are introduced. The latter can be one or two-dimensional arrays.

The following primitives are supplied, with the obvious meaning: HEAD, TAIL, CONS, NULL, ATOM. Another primitive - QUOTE - is used to create atoms * in various ways; let A be a numeric (REAL or INTEGER) FORTRAN variable, with a current value of 375, and let S be a SYMB variable; then, one of the following alternatives could be employed:

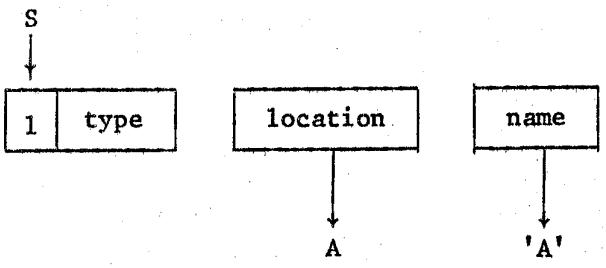
(a) S = QUOTE (375)
(b) S = QUOTE (A)
(c) S = QUOTE (.A)

causing the creation of atoms as shown below:

(a) or (b)



(c)



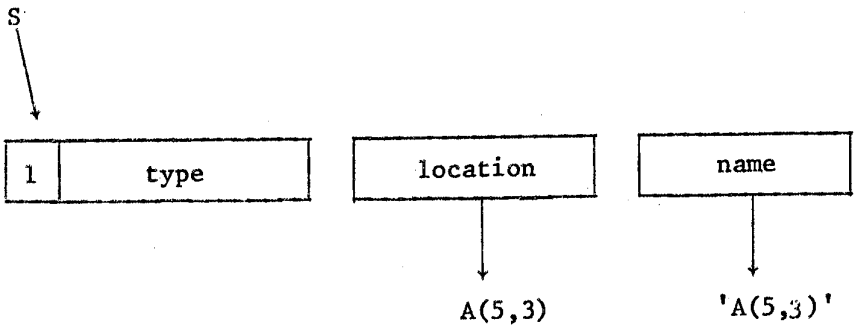
---

* .NIL. is regarded as a predefined atom.

If A is an array, and the INTEGER variable K is currently equal to 3, the statement

(d) S = QUOTE (.A (5, K))

will cause the creation of the atom:



The tag 1 indicates an atom, whereas 0 indicates a list cell. Type has enough bits to indicate both the type of the value whose location is given and whether it is the case of a "named" atom (cases (c) and (d) above).

Equality between atoms is tested by the infix operators .EQV.(equality of values) or .EQN. (equality of names); clearly the latter implies the former *.

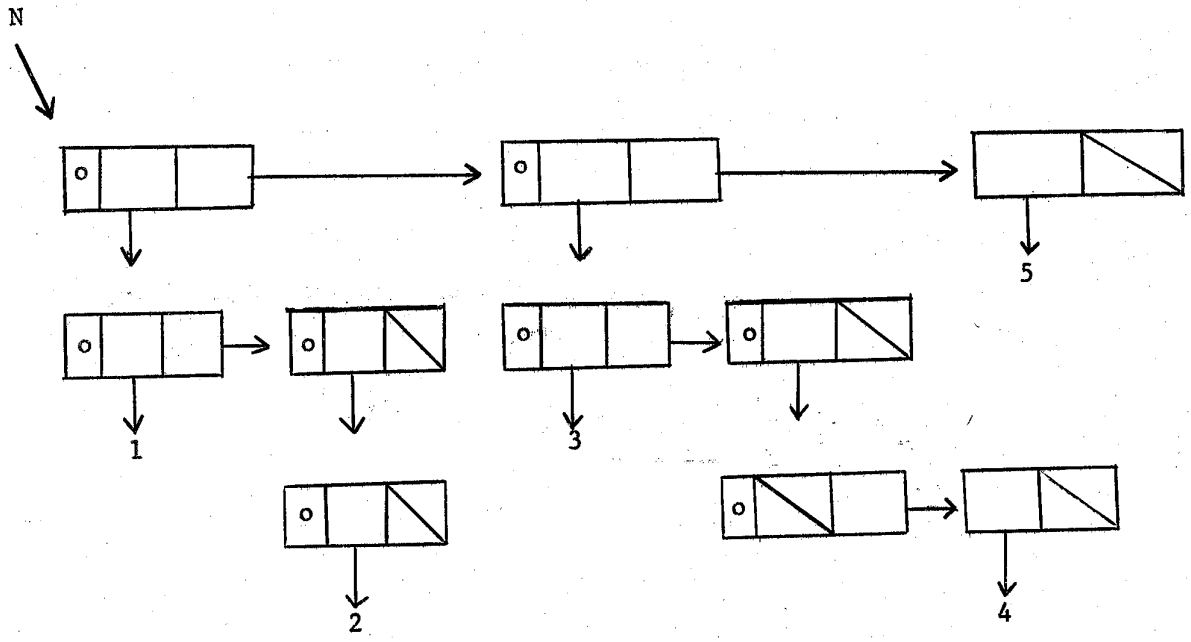
In order to introduce lists, we consider the following program segment:

---

* As a useful convention, we stipulate that .EQN. works exactly like .EQV. if both operands have no names. If one operand only has a name, the result will be .FALSE.

```
      -----
      SYMB T (5), L, M, N, A, B
      INTEGER I
      -----
      DO 1 I = 1,5
           T (I) = QUOTE (I)
1     CONTINUE
      L = CONS (T (1), CONS (CONS (T (2), NIL)))
      M = CONS (T (3), CONS (NIL, T(4)))
      N = CONS (CONS (L,M), T(5))
      ----------------
```

N will be the list ((1 (2)) (3 (( ) 4)) 5), stored as below; for simplicity the full representation of atoms is ommitted:



Also if the statements

```
A = HEAD (N)
B = TAIL (N)
```

then A and B will be, respectively, the list (1(2)) and ((3(( )4))5).

Recursion allows an easy translation into FORTRAN of the usual LISP non-primitives such as EQUAL and APPEND:

```
      RECURSIVE LOGICAL FUNCTION EQUAL  (A, B )
         SYMB A, B
         IF (ATOM (A)  .AND. ATOM (B))  GO  TO 1
                                        GO  TO 2

1           EQUAL = A .EQV. B
            RETURN
2        IF  (ATOM (A) .OR. ATOM (B))   GO  TO 3
                                        GO  TO 4
3           EQUAL = .FALSE.
            RETURN
4        IF  (EQUAL (HEAD (A), HEAD (B))) EQUAL = EQUAL (TAIL (A), TAIL (B)`
         RETURN

      END
```

Note that EQUAL could be reprogrammed in terms of .EQN. instead of .EQV.

```
      RECURSIVE SYMB FUNCTION APPEND (A, B)
         SYMB A, B
         IF (NULL (A))  GO  TO 1
                        GO  TO 2
1           APPEND = B
            RETURN
2           IF (ATOM (A)) GO  TO 3
                          GO  TO 4
3           APPEND = CONS (A, B)
            RETURN
4           APPEND = CONS (HEAD(A), APPEND (TAIL (A), B))
            RETURN
      END
```

The reader will recall that if the lists (1 2) and (3 4) are operated upon by CONS the result will be the list ((1 2) 3 4), whereas APPEND would yield (1 2 3 4).

Two especially useful "utility routines" are the sub-programs for reading a list from input and for printing it out.

Conversion from SYMB to regular FORTRAN objects is ensured by the primitives VALUE and NAME, which are in a sense the inverse of QUOTE. VALUE extracts the value whose location is given in an atom's cell, and constitutes a generic function in that the type of datum it returns will depend on the type information in the atom's cell (which is of course run-time information). NAME extracts the name of the atom (which must have been created via the dotted argument version of QUOTE), and returns it as a string of characters.

As a simple example of application of the list processing extension, we give a sub-program for computing the derivative of an arithmetic expression involving only sums and products and represented as a list in completely parenthesized form. E is the expression, to be differentiated with respect to V.

```
      RECURSIVE SYMB FUNCTION DER (E, V)
        L3(A, B, C) = CONS (A, CONS (B, C))
        SYMB E, V, P, Q, R, L3, PLUS, TIMES
        COMMON / L / PLUS, TIMES, ONE, ZERO

        IF (ATOM (E))  GO TO 1
                       GO TO 2
1       IF (E.EQN. V)  GO TO 3
                       GO TO 4

3           DER = ONE
            RETURN
4           DER = ZERO
            RETURN
2       P = HEAD (E)
        Q = HEAD (TAIL (E))
        R = HEAD (TAIL (TAIL (E)))
        IF  (Q .EQV. PLUS)  GO TO 5
                            GO TO 6
```

```
5           DER = L3 (DER (P), Q, DER (R))
            RETURN
6       IF  (Q .EQV. TIMES)  GO TO 7
                             GO TO 8
7           DER = L3 (L3 (P. Q. DER (R)), PLUS,  L3 (DER (P), Q, R))

            RETURN
8       WRITE (6, 9)
9       FORMAT ('UNEXPECTED OPERATOR')
        STOP
      END
```

The calling program is supposed to read-in or construct E and V. All variables appearing in the expression E should be named, the same going for the variable associated with the atom V. Also, PLUS, TIMES, ONE, and ZERO should be initialized by statements such as

```
INTEGER P
DATA P / '+' /
PLUS = QUOTE (P)
       .
       .
       .
```

Among the implementation considerations the memory allocation for the SYMB cells is the most important. The area allocated to the user's program but not occupied by his code and data could be used (shared with the space for the recursion stack). The links could be actual addresses or indices, if an array organization is used. A garbage-collection process could be invoked whenever the cell-space overflows; when this happens all SYMB declared variables and SYMB temporaries will be used for starting a tagging process for all cells reachable from them (which include cells reachable from their incarnations in the recursion stack); unreachable cells will then be considered as available for re-use.

4. RUN-TIME COMPILATION OF ARITHMETIC EXPRESSIONS

In order to understand how this feature is used, we must go back to the arithmetic expressions represented as lists, introduced in the derivation example of the previous section. Let F be a SYMB variable denoting such a list, and assume that some of its atoms - say X and Y - have been created via the dotted option of QUOTE.

Now, X and Y are also variables in the program (their names and addresses being stored in the atoms, as seen), and their values can be altered via ordinary arithmetic statements. Thus F can be regarded as a function of X and Y.

In fact, it would be easy to write a program for traversing the list F and evaluating the expression for the current values of X and Y. Unfortunately this sort of evaluation - called interpretive evaluation - is grossly inefficient, although it is the only option available for example in FORMAC. The situation is especially serious if the expression is to be evaluated not once but many times, as it is often the case in numerical analysis applications. Of course, if the form of F is known when the program is written the best solution is to write it as a function statement or function subprogram; however we are interested here in expressions that are not known when the program is written, and particularly in expressions that are created as part of the program execution.

Our extension provides variables of type RUN. By calling a built-in subroutine COMPIL, having as one argument a SYMB variable, say F, and a RUN variable, say G, the list corresponding to F is traversed <u>once</u>, and machine code is generated for the arithmetic expression and associated with G. If later in the program G appears in a context where a numerical value is expected, the code is executed thus yielding the result of the expression.

In addition there is a LOGICAL function NORUN that tests a RUN variable and returns .TRUE. if it is <u>not</u> executable (does not correspond to a successfully compiled arithmetic expression), and returns .FALSE. otherwise.

As an example, suppose that we have a SYMB variable F that contains the symbolic expression of a function of three variables, and a SYMB array

D2F that will be the Hessian matrix * for function F, i.e. D2F (I, J) would contain the symbolic expression of $\frac{\partial^2 F}{\partial x_i \partial x_j}$, $1 \leq i, j \leq 3$. The following program would generate a RUN array code for each entry in D2F.

```
      INTEGER I, J
      REAL X(3), V
      RUN REAL H(3,3)
      SYMB F,DF(3), D2F(3,3), DERIV
        .
        .
        .
C ----------------------------- CALCULATE ENTRIES IN THE MATRIX
       DO 2 I = 1,3
C ----------------------------- CALCULATE FIRST ORDER PARTIAL DERIVATIVES
         DF(I) = DERIV(F,X(I))
         DO 1 J = 1,I
C ----------------------------- CALCULATE SECOND ORDER PARTIAL DERIVATIVES
           D2F(I,J)=DERIV (DF(I), X(J))
C ----------------------------- REMEMBER THE MATRIX IS SIMETRIC
           D2F(J,I)=D2F (I,J)
C ----------------------------- GENERATE CODE FOR EACH ENTRY
           H(I,J)= COMPIL(D2F(I,J))
           H(J,I)= H(I,J)
1          CONTINUE
2       CONTINUE
C ----------------------------- EVALUATE D2F(2,3) AT X=(1.,0.7,0.8)
        X(1) = 1.
        X(2) = 0.7
        X(3) = 0.8
        V = H(2,3)
        .
        .
        .
        END
```

---

* Hessian matrices are relevant in the area of function optimization.

The subroutine COMPIL could use a precedence scheme for the compilation of arithmetic expressions. These should include any arithmetic expressions valid in FORTRAN so COMPIL should be able to generate calls to standard functions such as SIN, COS, etc.; this would be achieved by implicitly generating EXTERNAL statements for them (reference to programmer defined functions should also be allowed).

Our experience shows that even a very simple-minded compilation strategy, without any code optimization, leads to a much faster execution than the existing interpretive evaluation schemes.

5. CONCLUSIONS

A considerable number of algorithms have been designed and programs have been written for many interesting applications using recursion and recursive structures - notably LISP-lists - and the proposed extension allows their straightforward translation into FORTRAN programs.

This ease of translation should be compared with the feasible but often difficult adaptation provided by embedding schemes [8].

The present work can be pursued by making use of the introduced features in several ways. As an example, note that variable length character strings can be easily handled using lists, and employing either recursive or iterative methods for their manipulation.

Moreover, without any further change to the syntax, expressions other than mathematical formulas could be compiled, for which it is sufficient to reprogram COMPIL.

REFERENCES

[1] STANDISH, T.A. Extensibility in programming language design. SIGPLAN Notices, 10 (7) : 18-21, July 1975.

[2] SAMMET, J. Roster of programming languages for 1973. SIGPLAN Notices, 9 (11) Nov. 1974.

[3] TOSCANI, S.S. Recursividade em FORTRAN [Recursion in FORTRAN] Rio de Janeiro, PUC, Rio Datacentro, 1969. M. Sc. thesis.

[4] ABRAHAMS, P.W. et alii. The LISPZ programming language and system. In: AFIPS conference proceedings; fall joint computer conference, 1966. v.29 p. 661-76.

[5] BARRON, D.W. et alii. The main features of CPL. Computer Journal, 6 : 134-43, 1963.

[6] PERLIS, A.J. & ITURRIAGA, R. An extension of ALGOL for manipulating formulae. Commun of the ACM, 7 (2) : 127-30, Feb. 1964.

[7] AHO, A.V. et alii. The design and analysis of computer algorithms. Reading, Addison - Wesley, 1974.

[8] BOBROW, D.G. & WEIZENBAUM, J. List processing and extension of language facility by embedding. IEEE Transaction on Computers, 13 (4) 1964.