FILOMAT (Niš) 9:2 (1995), 315–324
Filomat '94, Niš, October 22-24, 1994
Geometry. Computer Sciences

USAGE OF S-EXPRESSIONS AND PREDICATE EXPRESSIONS IN PROCEDURAL PROGRAMMING LANGUAGES

Tatjana Vukelić and Mirjana Ivanović*

ABSTRACT. An extension of a procedural programming language with S-expressions and predicate expressions is described. Several examples in the field of graph theory, logic and set theory, hash tables, and sparse matrices are presented.

1. Introduction

Procedural programming languages are still the most frequently used programming languages. However, during the last decade many other programming paradigms (functional, logical, relational, etc.) came into the wide usage. Various programming languages and programming styles enable more natural and "simpler" solving of various classes of problems.

Great variety of programming styles lead to development of new programming languages and extensions of existing programming languages. Procedural programming languages are a good base that can be easily extended with new concepts and elements.

To enhance expressiveness of programming language Modula-2 [4], S-expressions [1] and predicate expressions (some forms of predicate formulas) [3] are included into the language. Modula-2 is widely used procedural programming language. It has variety of data types and data structures, supports structured and modular programming style and forces a programmer to write clear and readable code. With proposed extensions, Modula-2 programs are even more readable, shorter and simpler than their equivalents written in "real" Modula-2.

1991 *Mathematics Subject Classification.* 68N15.
*Supported by Grant 0401A of RFNS through Math. Inst. SANU

Although extensions described in this paper are part of the extended Modula-2 (and are implemented by a translator of extended Modula-2 programs into the "real" ones,) similar extensions can be achieved by abstract data type mechanism or by building suitable function libraries (in Modula-2 and other procedural languages.)

In the rest of the paper we shall shortly introduce the basic constructs of extended Modula-2 and then proceed with examples of possible applications.

2. S-expressions and predicate expressions

In this section an S-expression as built-in data type of extended Modula-2 and two new language constructs (predicate expressions and FORALL loop) are presented.

2.1. S-expressions.

S-expressions are basic data structures in some functional programming languages. Using Beckus-Naur form, S-expression is defined as follows:

```
S-exp         = atom | "(" S-exp-list ")"
S-exp-list    = S-exp | S-exp "." S-exp | S-exp S-exp-list
atom          = symbolic-atom | numeric-atom
numeric-atom = integer-atom | real-atom
```

The empty S-expression is denoted as `nil`. The following two conventions hold for S-expressions:

(1) `.nil` can be omitted (i.e., need not be written down), and
(2) `.(` and corresponding `)` can be omitted.

S-expression is a built-in data type of extended Modula-2 and is denoted as `SExp` (but it also can be implemented as an abstract data type [1].) `SExp` is supported with the set of primitive functions, predicates, arithmetic operations and input-output operations.

Examples of possible operations over S-expressions are [2] (for every S-expression e, e_1, and e_2):

(1) **Hd**(e) - returns e_1 if e is of the form: $(e_1 \ . \ e_2)$,
(2) **Tl**(e) - returns e_2 if e is of the form: $(e_1 \ . \ e_2)$,
(3) **Add**(e_1, e_2) - returns the sum of two numerical atoms e_1 and e_2,
(4) **Mul**(e_1, e_2) - returns the product of (numerical atoms) e_1 and e_2,
(5) e_1 :: e_2 - returns a new S-expression of the form $(e_1 \ . \ e_2)$,
(6) e_1 ++ e_2 - appends two S-expressions giving a new one.

An empty S-expression is in extended Modula-2 denoted as `NULL` (i.e., `NULL` is a constant value of the type `SExp`). Some of the built-in functions over

S-expressions could be implemented as operators (function **Add**, for example could be implemented by "overloading" operator +). First experiences show however, that chosen set of functions and operators (as presented in this section) enables best readability of resulting programs.

As an example of programming with S-expression, we quote the implementation of procedure `ListOfPair(e1, e2: SExp): SExp` which returns the following S-expression: `( (e1 e2) )`, i.e. `( (e1 .(e2 .nil)) . nil)`.

```
PROCEDURE ListOfPair(e1, e2: SExp): SExp;
BEGIN
  RETURN (e1 :: (e2 :: NULL)) :: NULL
END ListOfPair;
```

2.2. Predicate expressions.

Predicate expressions are special kind of expressions [3] based on formulas of first order predicate calculus. In an extended Modula-2, they have the following form (given in extended Beckus-Naur form:)

```
PredExp     = PredSym Ident {"," PredSym Ident}
              "|" WhereFrom {"," WhereFrom}
              "WHERE" Condition {"," Condition}.
WhereFrom   = Ident "IN" Domain.
PredSym     = "EVERY" | "EXIST".
Domain      = Ident | Set | Interval | S-exp | Array.
Interval    = "[" LowerBound ".." UpperBound "]".
```

where `Condition` is a standard Modula-2 expression, whose value is a logical truth value. The value of predicate expression has a logical truth value as well. Predicate expression can also be implemented as abstract data types and supported with suitable functions, but then the corresponding programs would be less readable.

A following predicate expression:

```
EVERY x | x IN X WHERE Condition
```

can be read as "is it true that every `x` from `X` fulfills the `Condition`?" This expression returns `TRUE` if for all elements `x` from `X` the value of the (boolean) expression `Condition` is `TRUE`.

A following predicate expression:

```
EXIST x | x IN X WHERE Condition
```

can be read as "is it true that there exists at least one `x` in `X` such that `Condition` is fulfilled?" This expression returns `TRUE` if for at least one element `x` from `X` the value of the (boolean) expression `Condition` is `TRUE`.

Predicate expressions can be used with S-expressions, sets, arrays, and intervals. Arrays and intervals (i.e., subranges) are the same as in "real" Modula-2. Sets are however, more general. The elements of **Set** in extended Modula-2 [3] can be of arbitrary data types (simple or composite) and cardinality of a **Set** is not limited. All the types of set elements must be the same (like in "real" Modula-2.)

2.3. FORALL loop.

Usage of S-expressions and predicate expressions is immense in various areas and in solving of different problems. However, to make this usage simpler and more powerful, we introduced a new kind of **FOR loop** called **FORALL loop**. A new loop could be defined by the following rule of extended Beckus-Naur form:

```
ForAllLoop = "FORALL" Identifier "IN" Domain "DO"
                  Statements
             "END".
```

`Domain` in **FORALL loop** is the same as domain in predicate expression, and `Statements` are all available statements in extended Modula-2, including **FORALL**. Statement

```
FORALL x IN X DO Statements END
```

means that statements inside `FORALL` loop are executed for every element `x` that belongs to S-expression, set, array or interval `X`.

In the next section we proceed with some possible applications of S-expressions and predicate expressions: hash tables, graphs, sets, sparse vectors and matrices. Using S-expressions and predicate expressions, simpler and more readable programs are obtained.

3. Possible applications

3.1. Hash tables.

A hash table is one of the most popular structures for fast data retrieval. It is most often used with dictionaries. A dictionary is presented as a hash table, and consists of n ordered sets. Every set is presented as an S-expression. Hash function is a function that transforms a word into a number between 1 and n. Value of the hash function determines a set that the word belongs to.

Definition of a hash table can be (in extended Modula-2) as follows:

```
CONST  n = 211;
TYPE   HTab = ARRAY [1..n] OF SExp;
```

Procedure Initialize initializes elements of a hash table:

```
PROCEDURE Initialize(VAR HT: HTab);
VAR i: [1..n];
BEGIN
   FORALL i IN [1..n] DO HT[i] := NULL END
END Initialize;
```

Procedure Found checks if a word belongs to a dictionary:

(1) by the hash function HashFun the word is transformed into a hash value (number k)
(2) if the word belongs to the set that contains all words with the same hash value k, function returns TRUE.

In the following procedure we shall assume that the data type String exists and that it is implemented as a fixed-length array of characters.

```
PROCEDURE Found(Word: String; HT: HTab): BOOLEAN;
VAR x: String;
BEGIN
   RETURN EXIST x | x IN HT[HashFun(Word)] WHERE x=Word
END Found;
```

Procedure Store stores a word into a hash table.

```
PROCEDURE Store(Word: String; VAR HT: HTab);
VAR pos: [1..n];
BEGIN
   pos := HashFun(Word);
   HT[pos] := Word :: HT[pos]
END Store;
```

Graphs.

A graph G consists of

(1) set V, whose elements are called nodes and
(2) set of pairs E, whose elements are called edges.

Graph can be defined using adjacency lists. To every node, a list of adjacent nodes is attached. Graph can also be defined as a list of edges. An edge is represented as a pair of nodes, which it connects.

```
TYPE node  = CARDINAL;
     edge  = RECORD c1, c2 : node END;
     Graph = RECORD nodes : SET OF node;
                    edges : SET OF edge
             END;
```

Graph is *connected* if there is a path between every pair of its nodes. We shall assume that function `Path(c1, c2: node): BOOLEAN` returns the value `TRUE` if there is a path between nodes `c1` and `c2`, otherwise returns the value `FALSE`.

Function `Connected` checks if a graph is connected.

```
PROCEDURE Connected(G: Graph): BOOLEAN;
VAR  c1, c2: node;
BEGIN
   RETURN EVERY c1, EVERY c2 | c1 IN G.nodes, c2 IN G.nodes
          WHERE Path(c1,c2)
END Connected;
```

A graph is *complete* if each node is connected to every other node. Procedure `Edge(c1, c2):BOOLEAN` checks if there is an edge incident to nodes `c1` and `c2`. It assumes that if `c1 = c2`, there is an edge between those nodes.

Function `Complete` checks if a graph is complete.

```
PROCEDURE Complete(G: Graph): BOOLEAN;
BEGIN
   RETURN EVERY c1, EVERY c2 | c1 IN G.nodes, c2 IN G.nodes
          WHERE Edge(c1, c2) AND (c1 <> c2)
END Complete;
```

Degree of a node v, is equal to the number of edges that are adjacent to v. Function `Degree` determines the degree of node `v` in the graph `G`.

```
PROCEDURE Degree(v: node; G: Graph): CARDINAL;
VAR  Deg : CARDINAL; E : edge;
BEGIN
   Deg := 0;
   FORALL E IN G.edges DO
      IF (E.c1 = v) OR (E.c2 = v) THEN INC(Deg) END
   END
END Degree;
```

3.2. Sets.

Let us recall that elements of a set in extended Modula-2 can be of arbitrary type and that the number of set types is (conceptually) unlimited. For example, in the following definition:

```
TYPE  SetAnyType = SET OF AnyType
```

where AnyType can be of arbitrary type, including arrays, records and other sets. Procedure SetMember determines whether an element x is a member of a set S.

```
PROCEDURE SetMember(x: AnyType; S: SetAnyType);
VAR  e : AnyType;
BEGIN
   RETURN EXIST e | e IN S WHERE e = x
END SetMember;
```

Procedure SubSet checks whether set s1 is a subset of a set s2.

```
PROCEDURE SubSet(s1, s2: SetAnyType);
VAR  x1, x2: AnyType;
BEGIN
   RETURN EVERY x1, EXIST x2 |
                  x1 IN s1, x2 IN s2 WHERE x1 = x2
END SubSet;
```

3.3. Sparse vectors and matrices.

A sparse vector is a vector that consists mostly of zero elements. It can be presented by S-expression whose elements are ordered pairs. Every pair presents one non-zero element of a sparse vector. The first element of the ordered pair is an index of the element in a vector, and the second is the value of the element. For example, a vector $V = [1\ 0\ 0\ 0\ 0\ 2\ 0]$ is represented by ((1 1) (6 2)).

Procedure MulVec returns a product of a vector v and scalar n.

```
PROCEDURE MulVec(v: SExp; n: INTEGER): SExp;
VAR  res, el: SExp;
BEGIN
   res := NULL;
```

```
   FORALL el IN v DO
      res := res ++ ListOfPair(Hd(el), Mul(Tl(el), n))
   END;
   RETURN res
END MulVec;
```

Procedure SumVec sums two vectors. In this procedure, following procedures will be used:

(1) Find(e,v) which returns a pair in the vector v, whose index is equal to a value e.
(2) Delete(e,v) which deletes a pair e from the vector v.

```
PROCEDURE SumVec(v1, v2: SExp): SExp;
VAR  res, el1, el2 : SExp;
BEGIN
  res := NULL;
  FORALL el1 IN v1 DO
    el2 := Find(Hd(el1), v2);
    IF el2 <> NIL
      res := res ++ ListOfPair(Hd(el1),
                               Add(Tl(el1), Tl(el2)));
      Delete(el2, v2)
    ELSE
      res := res ++ (el1 :: NULL)
    END
  END;
  RETURN res ++ v2;
END SumVec;
```

Procedure VecScPro returns a scalar product of two vectors. In this procedure, the procedure FindVal(n, v) is assumed to return the value of the element with an index n in the vector v.

```
PROCEDURE VecScPro(v1, v2: SExp): SExp;
VAR  res, el1, el2: SExp;
BEGIN
   res := 0;
   FORALL el1 IN v1 DO
      IF EXIST el2 | el2 IN v2 WHERE Hd(el1) = Hd(el2) THEN
         res := Add(res, Mul(Tl(el1), FindVal(Hd(el1), v2)))
```

```
      END
   END;
   RETURN res
END VecScPro;
```

A sparse matrix is a matrix that contains relatively many zero elements. A sparse matrix can be represented by S-expression that consists of ordered pairs. By every pair a row of matrix that has at least one non-zero element is presented. The first element of the pair is the index of a row of matrix, and the second element of the pair is a sparse vector.
Matrix

$$M = \begin{bmatrix} 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix}$$

has the representation $M =$ ((1 ((2 4)))(3 ((1 6) (4 7)))). We shall also assume that function procedure `Transpose(M)` returns transposed matrix of matrix `M`.

Procedure `MatVecPro` multiplies a matrix by a vector. In this procedure, procedure `VecScPro` (defined previously) is used.

```
PROCEDURE MatVecPro(V, M: SExp): SExp;
VAR res, s, TM: SExp;
    val : INTEGER;
BEGIN
   res := NULL;
   TM := Transpose(M);
   FORALL s IN TM DO
      val := VecScPro(V, S);
      IF val <> 0 THEN
         res := res ++ ListOfPair(Hd(s), val)
      END
   END;
   RETURN res
END MatVecPro;
```

The result of multiplying a sparse vector and a sparse matrix is a new sparse vector.

Procedure `MatPro`, multiplies two matrices. In this procedure, procedure `MatVecPro` is used.

```
PROCEDURE MatPro(M1, M2: SExp): SExp;
```

```
VAR  res, v, c, TM, tpro : SExp;
BEGIN
   res := NULL;
   TM := Transpose(M2);
   FORALL v IN M1 DO
      tpro := MatVecPro(v, M2);
      res := res ++ ListOfPair(Hd(v), tpro)
   END;
   RETURN res
END MatPro;
```

4. Conclusion

S-expressions and predicate expressions are included into programming language Modula-2. In a similar way they can be included into other procedural programming languages. Inside a procedural programming language, S-expressions bring elements and concepts of functional programming languages. Elements of functional programming style in procedural programming languages bring advantages of both programming styles in different areas. Programs are shorter, simpler and more readable than in pure procedural programs and more efficient than equivalent functional programs.

A usage of predicate expressions in procedural programming languages brings more concise, clearer and more powerful code. Both extensions contribute to better expressiveness of programs.

Usages mentioned in this paper present only a small part of possibilities which extensions of procedural language bring.

References

[1] Z. Budimac and M. Ivanović, *New Data Type in Pascal* (1989), Proc. of DECUS Europe Symposium, The Hague, Holland, 193-199.

[2] M. Ivanović and Z. Budimac, *Usage of S-expression in Pascal* (1989), Proc. of 11th International Symposium "Computer at the University", Cavtat, 3.18.1-3.18.6.

[3] T. Vukelić and M. Ivanović, *Predicate expressions in procedural programming languages* (to appear).

[4] N. Wirth, *Programming in Modula-2*, fourth edition, Springer Verlag, Berlin, 1988.

University of Novi Sad, Faculty of Science, Institute of Mathematics, Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
E-mail address: {vukelic,mira}@unsim.ns.ac.yu