FILOMAT (Niš) 9:2 (1995), 325–332
Filomat '94, Niš, October 22-24, 1994
Geometry. Computer Sciences

IMPLEMENTATION OF PREDICATE EXPRESSIONS IN PROCEDURAL PROGRAMMING LANGUAGES

Tatjana Vukelić and Dušan Kamenov

ABSTRACT. *Predicate expressions in a procedural programming language are based on sentences of predicate calculus of first order. The usage of predicate expressions in procedural languages leads to shorter, more effective and more readable programming code, and also decreases number of loops and local variables in procedures and programs. Predicate expressions in programming languages could be used with array, set and interval data types. Elements of array or set could be simple or complex data type. In this paper, definition and implementation of predicate expressions in procedural programming language Modula-2 is presented. Areas of usage are logic, set theory, graph theory, pattern recognition and others.*

1. Introduction

Many statements, particularly in mathematics, are of the form "x satisfies α", where x belongs to set D and α is relation relevant to the elements of set D [1].

Statement "For every $x \in D$, $\alpha(x)$" is an example of a mathematical statement. Symbolically,

$$(\forall x \in D)\alpha(x), \quad \text{or shorter,} \quad (\forall x)\alpha(x)$$

denotes this kind of statement. The part $(\forall x)$ is called universal quantifier.

Statement "Exists $x \in D$, $\alpha(x)$" is also often used in mathematical sentences. Symbolically, this kind of sentence can be presented as

$$(\exists x \in D)\alpha(x), \quad \text{or shorter,} \quad (\exists x)\alpha(x)$$

The part $(\exists x)$ is called existentional quantifier.

1991 *Mathematics Subject Classification.* 68N15.

It is possible to use more than one quantifier in a single sentence. For example,

$$(\forall x)(\exists y)\alpha(x,y)$$

is a valid sentence of predicate calculus.

In the further text, symbols and will be referred to as predicate symbols.

These sentences can be efficiently implemented and used in procedural programming languages. The implementation shown in following chapters is an extension of Modula-2 programming language [2]. In further text, this extension will be called EM2.

Predicate expression in procedural programming languages has the form of list of quantifiers (predicate symbols followed by variables). After that, domains of the variables which follow predicate symbols in quantifiers are stated. At the end stands condition of predicate expression, which is presented by list of boolean expressions. Expressions are separated by commas.

The equivalent symbols for predicate symbols $\forall$ and $\exists$ in EM2 are `EVERY` and `EXIST`, respectively.

Syntax of predicate expressions can be presented by following rules of EBNF:

```
#PredicateExpr =  PredicateSymbol Identifier
     { "," PredicateSymbol Identifier } "|"
     Identifier "IN"  Range { "," Identifier "IN" Range }
     "WHERE" Expression { "," Expression }.
#PredicateSymbol = "EXIST" | "EVERY".
#Range = Array | Set | Interval.
#Array = Identifier.
#Set = Identifier | "{" [ Member {"," Member} ] "}".
#Interval = "[" LowerBound ".." UpperBound "]".
```

Existing set data type in M2 language can also be extended. In Modula-2, elements must be simple data type. In EM2, this restriction is no longer valid; elements can be any data type - single or complex. Elements of set are not ordered, and an element can appear in set several times. For example, $\{1,2,1,3\}$ is a valid set in EM2.

Range of the variables in quantifiers must be finite. Range of those variables is determined by standard Modula-2 data types array or interval, or by new data type set.

Example 1.1. Some simple predicate expressions are:

```
VAR
     a, b   : BOOLEAN;
     array  : ARRAY [1..10] OF CARDINAL;
     set    : SET OF CHAR;
```

```
x       :  CHAR;
y       :  CARDINAL;

        .  .  .

a := EXIST x | x IN set WHERE x < 'f';
b := EVERY y | y IN array WHERE y < 10;
```

The first assignment could be read: "if exist character `x`, which belongs to set `set` and `x` is before character 'f' in the ASCII table, then `a` becomes true, else `a` becomes false". The second assignment could be read: "if for every `y`, where `y` is an element of array `array`, condition `y < 5` is satisfied, then `b` becomes true, else `b` becomes false". Simpler, if every element of array `array` is less than 5 then `b` becomes true, else `b` becomes false.

It is possible to combine more than one quantifier in one predicate expression. Let's see an example. Suppose that `set1` and `set2` are set of cardinals, `a` is type boolean, and `x` and `y` are type cardinal.

Example 1.2.

```
a := EVERY y, EXIST x | y IN set1, x IN set2  WHERE x = y;
```

Simply said, if for every `y` exist `x`, `y` belongs to `set1`, `x` belongs to `set2`, equation `x = y` is satisfied, then `a` becomes true, else `a` becomes false.

2. Implementation of predicate expressions

Predicate expressions can be implemented in procedural programming languages in many different ways. One of them, which is based on translation of predicate expressions to equivalent code in Modula-2 programming language, is presented in further text. To make the translation simpler, some constructions of Modula-2 can be extended. Therefore, following statements are defined:

(1) `FORALL x IN X` (a kind of loop)
(2) `NEXT x`

`FORALL x IN X` means that statements inside loop are executed for all elements of an interval, set or array signed by `X`.

`NEXT x` determines the next element of `X`.

Interval and array are ordered. Theoretically, the order of the elements of a set is irrelevant. But in the computer memory a set is ordered and it is possible to take it's elements one after another.

Example 2.1. Structure

```
FORALL x IN [1..10] DO
      Write(x);
NEXT x
```

is equivalent to

```
x := 1;
WHILE x <= 10 DO
      Write(x);
      x := x + 1
END;
```

For simplicity, predicate expressions with one and two quantifiers will be discussed first. After that, we'll make a generalization of translation for any number of quantifiers.

2.1. Predicate expressions with one quantifier.

In general, predicate expression with one quantifier has the form:

$$PS\ x \mid x \texttt{ IN } X \texttt{ WHERE } condition$$

where predicate symbol is denoted by PS. Following cases are possible:

(*a*) EVERY x | x IN X WHERE *condition*

First, suppose that condition is satisfied for every x from X, and suppose that predicate expression has the truth value true. If x that does not satisfy the condition *condition* is found in for-all loop, then whole predicate expression gets truth value false.

```
EV_x := TRUE;
FORALL x IN X DO
      IF NOT condition THEN
            EV_x := FALSE
      END
NEXT x;
Result := EV_x
```

(*b*) EXIST x | x IN X WHERE *condition*

First, suppose that there is no x from X that satisfies the condition and suppose that whole predicate expression has the truth value false. If x that satisfy the condition condition is found in for-all loop, then the predicate expression gets the truth value true.

```
EX_x := FALSE;
FORALL x IN X DO
      IF condition THEN
```

```
            EX_x := TRUE
        END
    NEXT x
    Result := EX_x;
```

2.2. Predicate expressions with two quantifiers.
In general, predicate expression with two quantifiers has the form:

PS x, PS y | *x* IN *X*, *y* IN *Y* WHERE *condition*

where predicate symbol is denoted by *PS*. Following cases are possible:

(*a*) EXIST *x*, EXIST *y* | *x* IN *X*, *y* IN *Y* WHERE *condition*
In this case, construction 2.1.(b) will be used.

```
EX_x := FALSE;
FORALL x IN X DO
      EX_y := FALSE;
      FORALL y IN Y DO
            IF condition THEN
                  EX_y := TRUE
            END
      NEXT y;
      EX_x := EX_x OR EX_y
NEXT x;
Result := EX_x;
```

(*b*) EVERY *x*, EVERY *y* | *x* IN *X*, *y* IN *Y* WHERE *condition*
Construction 3.1.(a) is used in this case.

```
EV_x := TRUE;
FORALL x IN X DO
      EV_y := TRUE;
      FORALL y IN Y DO
            IF NOT condition THEN
                  EV_y := FALSE
            END
      NEXT y;
      EV_x := EV_x AND EV_y
NEXT x;
Result := EV_x;
```

(*c*) EXIST *x*, EVERY *y* | *x* IN *X*, *y* IN *Y* WHERE *condition*
Construction 3.1.(a) and 3.1.(b) are used combined in this case.

```
EX_x := FALSE;
FORALL x IN X DO
      EV_y := TRUE;
      FORALL y IN Y DO
            IF NOT condition THEN
                  EV_y := FALSE;
            END
      NEXT y;
      EX_x := EX_x OR EV_y
NEXT x
Result := EX_x;
```

(*d*) EVERY x, EXIST y | x IN X, y IN Y WHERE *condition*
Similar to case 3.2.(c),

```
EV_x := TRUE;
FORALL x IN X DO
      EX_y := FALSE;
      FORALL y IN Y DO
            IF condition THEN
                  EX_y := TRUE
            END
      NEXT y;
      EV_x := EV_x AND EX_y
NEXT x;
Result := EV_x;
```

2.3. Predicate expressions with any number of quantifiers.
Predicate expressions are analyzed from left to right. For every quantifier there is a for-all loop and one boolean variable that starts with EX_, if quantifier is EXIST, EV_, if quantifier is EVERY. Boolean variable gets its value before entering the for-all loop. Its value is FALSE in case of EXIST quantifier, and TRUE in case of EVERY quantifier.
Inside for-all loop two cases are possible:

(*1*) If quantifier is the last of the quantifiers in predicate expression, then inside for-all loop is an IF statement:
(*a*) If the quantifier is EXIST quantifier then it is following IF statement:

```
IF condition THEN
      EX_ident := TRUE
END;
```

(*b*) If the quantifier is EVERY quantifier then it is following IF statement:

```
IF NOT condition THEN
        EV_ident := FALSE
END;
```

NEXT *ident* follows the IF statement.

(*2*) If the quantifier is not the last one in predicate expression then inside the for-all loop are statements that matches quantifiers that come after current quantifier (this part could be implemented by recursion). After that follows:

(*a*) If current quantifier is EVERY then

```
EV_ident := EV_ident AND EV_ident2          (1) or
EV_ident := EV_ident AND EX_ident2          (2)
```

Statement (1) if the quantifier after current quantifier is of form EVERY ident2, statement (2) if the quantifier after current quantifier is of form EXIST ident2.

(*b*) If current quantifier is EXIST then

```
EX_ident := EX_ident OR EV_ident2           (3) or
EX_ident := EX_ident OR EX_ident2           (4)
```

Statement (3) if the quantifier after current quantifier is of form EVERY ident2, statement (4) if the quantifier after current quantifier is of form EXIST ident2.

After this statement follows NEXT ident statement.

3. An example of usage of predicate expressions

Predicate expressions can be used in solving different classes of problems. One of the areas of usage is mathematical logic.

Definition 3.1. Proposition $P(p, q, \dots)$ that has the truth value true for any truth values of their variables is called **tautology**.

A procedure which determines if a expression is a tautology could be as follows:

```
PROCEDURE Tautology;
VAR
      a, b:  BOOLEAN;
BEGIN
      IF EVERY a, EVERY b | a IN [FALSE..TRUE],
            b IN [FALSE..TRUE] WHERE a OR b OR NOT b THEN
                  WriteStr(" Expression is a tautology ")
      ELSE
            WriteStr(" Expression is NOT a tautology ")
      END
END Tautology.
```

The result of this program will be "Expression is a tautology" because expression is a tautology.

4. Conclusion

Predicate expressions have wide usage in many areas of computer science. Their great power is in area of mathematics. They allow short, readable and concise presentation of different definitions and theorems. They also have wide usage in pattern recognition. Combined with sets, they are powerful tool for fast and natural solving of different problems. Their usage decreases number of loops and local variables to minimum required, which makes the programming code shorter and more readable.

The future of predicate expressions can also be found in functional and logical programming languages. Predicate expressions are a step closer to human-like way of thinking in programming languages.

References

[1] Milić, Svetozar, *Elementi matematičke logike i teorije skupova*, A-Š delo, Beograd, 1991.

[2] Wirth, Niklaus, *Programiranje na jeziku Modula-2*, Dragon, Beograd, 1990.

Seljačkih buna 25, 21000 Novi Sad
E-mail address: {vukelic,ikamenov}@unsim.ns.ac.yu