PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 1/74

PATTERN MATCHING FOR STRUCTURED PROGRAMMING

by

A. L. Furtado
and
A. S. Pfeffer

Computer Science Department - Rio Datacenter

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

PATTERN MATCHING FOR STRUCTURED PROGRAMMING *

A. L. Furtado
Associate Professor
Computer Science Department

and

A. S. Pfeffer
Master in Computer Science
Computer Science Department
PUC/RJ

Series Editor: Prof. S. M. dos Santos July/1974

* This work was first published in the Proceedings of the "Seventh Asilomar Conference on Circuits, Systems and Computers"
Pacific Grove - California - USA - November 1973

ABSTRACT

A programming module for adding a simple string pattern-matching capability to PL/I is presented.

It is argued that pattern-matching is compatibe with control structures that conform to the principles of structured programming [1]

1. AN OVERVIEW OF P/PL/I

P/PL/I is an extension to PL/I that adds to this language a string pattern-matching capability. Patterns are provided as an additional data type. Three categories of patterns may be distinguished:

a) lexical patterns-these correspond to "primitive" classes such as: letters, digits, blanks, special characters, integers, character string constants (between single quotes), identifiers;

b) syntactical patterns-classes defined by concatenation and disjunction of other classes, variables, and constants;

c) semantic patterns-classes defined by means of predicates; i.e. procedures returning '0' B or '1' B.

For lexical patterns the pattern-matching process is very simple (implemented essentially as a finite state automaton) [2]. The concept of semantic patterns appears in [3].

The definition of a pattern may involve the pattern concatenation (.), disjunction (|), assignment (underscore), and restriction (::) operators. Unless brackets are used concatenation has a higher priority than disjunction. Assignment has the same meaning as immediate assignment in SNOBOL 4, and restriction is used to introduce a predicate as an additional condition for membership in a given class.

Right recursion is supported in the definition of patterns as well as repetition factores.

Some examples of patterns are given below:

```
<P> ::= 'A' | 'B' | 'C'

<Q> ::= ',' . <# ID>

<R> ::= 'D' | 'E' . <R>

<S> ::= (3:*)  P

<T> ::= <P>_Z

<U> ::= <T> . '+' . Z

<V> ::= <T>:: F(Z)
```

Clearly the notation resembler an extend BNF [4,2]. In the definition of <S> the repetition factor indicates 3 or more occurrences of <P>. If the lower bound were zero then <S> would also allow for the non - occurrence of <P>.

In the last two patterns note that:

a) the "+" must be followed by exactly the same substring that precedes it; so "B+B" belongs to <U>, but "B+A" does not:

b) <V> is the same as <T> restricted by (i.e., satisfying in addition the requirements of) the predicate F; in this exemple the value of Z that appears as the argument of F would have been determined when matching <T> (see the definition of <T>)

The pattern-matching and replacement process can now be described by:

$$A \;//\; \langle P \rangle \rightarrow W(X) \;//\; B$$

which reads: "if the contents of a string variable A satisfy the requirements of pattern <P> then report success and replace the contents of string variable B by the value of the PL/I string expression W(X)".

In fact this is what happens when the total mode is enable; in the anchored and unanchored modes, defined as in SNOBOL 4, [5,6], if a sub - string of A satisfies P then the match sueceeds and the value of B becomes the value of A with the matching substring replaced by W(X). We have written W(X) to stress the fact that assignments may have oecurred during the matching phase to string variables that may be present in the PL/I string expression.

The replacement action may be omitted. In this case we are only concerned with the success or failure of the match (and possibly assignments specified in the patterns).

A special variable ##S is set to '0' B or '1' B depending on the outcome of the matching phase.

Among the other special variable the cursor variables are particularly important. They indicate the position of the character in A that is under examination at a given moment in the matching phase.

2. SOME CHARACTERISTICS OF THE IMPLEMENTATION

The implementation uses the standardPL/I pre-processor to convert P/PL/I into valid PL/I.

The two main statements are the pattern definition statement and the pattern-matching statement. The former is shown for the patterns <T> and <R> of the preceding section:

```
##PAT(<T>) ::= <P>_Z);
##PAT(##RECURS(<R> ::= 'D' |'E' . <R>));
```

##PAT is declared as a pre-processor function, and so is ##MAT bellow, which is used for the pattern-matching statement:

```
##MAT(M//<T> → ',' || Z || ',' //N);
```

The replacement part may be omitted:

```
##MAT(M // <T>);
```

In either case the cursor variable for this statement will be reset to 1 before the statement is executed. However by writing:

```
##MAT(M; ##  // <T>);
```

the cursor variables will keeps its previous value which will indicate the position of the first character in M to be examined. Also the user has explicit control over the process; if one writes:

```
##M @ = 5;
##MAT(M, ## //<T>);
```

the fifth character of M will be starting point for the matching. The final value of the cursor variable is 1 plus the leath of the string if the matching failed, or 1 plus the position of the last character in the string that matched the pattern if the matching succeded.

The pre-processor translates the ##MAT statement into a call to a sub-routine. A pattern-matching expression that returns '0'B or '1'B (besides setting ## S), and can be used in any context where a bit - valued expression is expected, can be specified by using ## MAT ##. The pre - processor will translate ## MAT ## into a function call.

In the present implementation all variables appearing in a pattern-matching statement must be declared in a ## DCL statement.

Besides handling P/PL/I statements, the pre-processor program appends to the generated PL/I text a number of run-time routines that will perform the pattern-matching process. It also introduces a binary tree [7] where the structure of the patterns will be represented, and a number of auxiliary tables (for a graph representation of patterns see [8].

As one could expect, if pattern <A> appears in the definition of pattern <B>, then the tree correesponding to <A> is a sub-tree of the tree representing <B>.

Edges in the tree represent either a concatenation, being traversed as long as the matching succeeds, or disjunction, being taken in case of failure. The matching algorithm takes care of bachtracking and right recursion (left recursion is not supported in the present implementation).

The auxiliary tables supply additional information including address and lenght of string variables involved, bounds of the repetition factors, entry points of user-defined predicates, etc.

The pre-processor program and the run-time routines and tables constitute the P/PL/I module. The module runs in an IBM 370/165 under a supervisor. Other modules (for graph-processing [9], list processing, and additional control statements) are also implemented, and through the supervisor several modules can be appended to the same PL/I program.

A simple P/PL/I program is given below. A number of cards containing one or more PL/I-like statements are read in. The purpose of the program is to detect all 'simple arithmetic statements' that assign values to any variable included in a given list, and print out the whole program with generated comments indicating the variables to which such an assignment is being made. Here a simple arithmetic statement consists of a single scalar variable followed by an equal sign followed by an arithmetic expression containing onlu additions and subtractions of variables and integer constants; there are no blanks inside the statements.

```
Ex:  PROC OPTIONS(MAIN);
       DCL TAB(100) CHAR(31) VAR;
       DCL V CHAR(31) VAR;
       DCL (I,NT,NC) BIN FIXED;
       #DCL (CARD CHAR(80));
       #PAT (<SAS>::=<VAR>. '=' .<SAEXP>. ';');
       #PAT (<VAR>::= <# ID> V:: INCL(V));
       #PAT (<SAEXP>::= <TERM>.<TRMS>);
       #PAT (<TERM>:: = <#ID> | <#NB>);
       #PAT (<TRMS>:: = (0:*)<TRMS1>);
       #PAT (<TRMS1>:: = ('+' | '-'). <TERM>;
```

```
DCL INCL ENTRY(CHAR(*)VAR)RETURNS(BIT(1));
INCL: PROC(X) RETURNS(BIT(1));
      DCL X CHAR(*) VAR;
      DCL I BIN FIXED;
      DO I = 1 TO NT;
            IF TAB(I) = X
            THEN RETURN('1' B);
      END;
            RETURN('0'B);
      END INCL;
      # UNANCHOR;
            GET LIST(NT,(TAB(I) DO I = 1 TO NT);
            GET LIST(NC);
            DO I = 1 TO NC;
                  GET SKIP EDIT(CARD)(A(80));
                  # CARD @ = 1;
                  DO WHILE(#MAT#(CARD,# // <SAS>));
                  PUT SKIP LIST('/* ASSIGNMENT TO',
                       V, '*/');
                  END;
                  PUT SKIP LIST(CARD);
            END;
      END EX;
```

3. PATTERN-MATCHING AND STRUCTURED PROGRAMMING

Since one goal in the present implementation was to conform to the principles of structured programming [1] special care was devoted to the choice of control structures to be used in connection with pattern - matching.

The starting point is the realization that a pattern is a definition of a set of strings. In the simplest case a finite set is defined by ennumerating its members (extensional definition); if string variables are involved the set may still be finite but will be time - varying; in the more complex cases possibly infinite sets are specified algorithmically (intensional definition).

Now if x is a string and P a pattern, the pattern - matching process consists of checking whether $x \varepsilon P$, which in PL/I terms would be an operation returning a bit string value.

It has been shown [10] that the two constructs

a. IF p THEN f1 ELSE f2;

b. DO WHILE(p); f; END;

together with the simple sequential execution of statements are sufficient to describe any flow-chartable program. In both constructs pattern - matching expressions could occupy the place of p or be part of it (the variable ## S could also appear in the rame context).

This should be contrasted to the success/failure exits used for example in SNOBOL, which are equivalent to GO TO's.

Side effects are allowed to occur if assignments within a pattern are specified or if a replacement action takes place. The latter is equivalent to:

```
IF  p  THEN  perform replacement;
ELSE
```

and the assignment can be described similarly. Note also that a text of the form:

```
DO WHILE (#MAT# (σ//ℓ1 → r1 //σ)|
          #MAT# (σ//ℓ2 → r2 //σ)|
           ......................|
          #MAT# (σ//ℓn → rn //σ));
END;
```

represents a straighforward implementation of Markov algoritms [11]. If the pattern-matching succeds for any of the n alternatives the corresponding replacement is executed and the first alternative is tried, unless the failing alternative is the n^{th} one in which case the DO group is exited (warning: this worksonly with the optimiying compiler).

The desirability of including Markov algorithms in programming languages is indicated in [12]. An even simpler scheme may be implemented by:

```
# S = '1'B;
DO WHILE(# S);
     #MAT(σ//ℓ1 → r1//σ);
     #MAT(σ//ℓ2 → r2//σ);

     ....................
     #MAT(σ//ℓn → rn//σ);

END;
```

Here the next sequential rule will be tried regardless of the success or failure at the current rule, and after the n^{th} rule has been tried the set of rules will be tried again starting from the first one if any of them has succeded.

4. FURTHER WORK

We expect to profit from the comments of a number of users for extending the module in useful ways. Attempts will also be made to increase its efficiency.

As aspect to deserve special consideration is the inclusion of left.recursive patterns. The supervisor shall aldo be modified in order to male the P/PL/I module and the FORMAC [13] preprocessor compatible, with a view to applications in symbolic mathematics.

REFERENCES

1. Dahl, Q., Dijkstra, E., and Hoare, C. - "Structured Programming" Academic Press - 1972.

2. Cocke, J. and Schwartz, J. - "Programming Languages and their Compilers" - Courant Institute of Mathematical Sciences - 1970.

3. Fateman, R. - "The user level semantic matching capability in MACSYMA" - Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation - 1971.

4. Naur, P. (ed.) - "Revised Report on the Algorithmic Language ALGOL 60", in Programming Systems and Languages" - Rosen, S. (ed.) McGraw-Hill, 1967.

5. Griswold, R. et al - "The SNOBOL 4 Programming Language" - Prentice Hall - 1971.

6. Griswold, R. - "The macro implementation of SNOBOL 4: a case study of machine-independent software development" - W.H. Freeman - 1972.

7. Gimpel, J. - "A theory of discrete patterns and their implementation in SNOBOL 4" - CACM vol. 16 - Feb. 1973.

8. Wegner, P. - "The structure of SNOBOL" - Cornell University - T.R. 68-9.

9. Santos, C. and Furtado, A. - "G/PL/I- extending PL/I for graph processing" Proceedings of the Computer and Information Sciences Symposium - 1972.

10. Mills, H. - "Mathematical Foundations for Structured Programming" IBM Corporation - 1972.

11. Galler, B. and Perlis, A. - "A view of Programming Languages" - Addison-Wesley - 1969.

12. Cheatham, T. - "The recent evolution of programming Languages" Proceedings of the IFIP Congress 1971 - vol. 1 - North-Holland 1972

13. Tobey, R. - "PL/I FORMAC interpreter - user's reference manual" 360D - 03.3.004 - IBM Corporation - 1967.