PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 12/72

SDM - A SYNTAX DIRECTED MACRO - PROCESSOR

by

S. R. P. Teixeira
and
D. J. Nunes

Computer Science Department - Rio Datacenter

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

SDM - A SYNTAX DIRECTED MACRO-PROCESSOR

S. R. P. Teixeira
Associate Professor
Computer Science Department
PUC/RJ

and

D. J. Nunes
Assistant Professor
Universidade Federal do Rio Grande do Sul

This paper was published in the Sixth Asilomar Conference on Circuits and Systems - San Francisco - California - November 15 - 17 /1972

Series Editor: Prof. A. L. Furtado

December /1972

ABSTRACT

SDM is a Syntax-Directed Macro-Processor which allows one to extend the syntax and semantics of a high level language. It is a string processor which uses R-expressions to define the syntax of new statements (macros) and a language, SL, to define the semantics of these statements.

1. INTRODUCTION

The basic idea in a macro-processor is the direct replacement of certain symbols with their associated pieces of text. A good description of the general idea can be found in[1].

The macro concept has thus far been mainly associated with assembly languages[2][3], although it has proved useful in connection with high level languages[1][4][7]. In the former case, due to the simple syntax of assembly languages, macro-processors usually do not allow flexibility of format to macro-calls. This is undesirable when one attempts to generalize the macro concept to high level languages. Also, it is often the case that no syntax checking of the arguments is done when the macro is called and error messages are produced in a stage of compilation after macro-processing, making it difficult to relate an error message with the macro-call that produced it.

In this paper we describe a syntax - directed macro-processor called SDM, which is based on the idea of syntax macros[5][6]. SDM allows one to extend the syntax and semantics of a high level language, it is compiler independent[7] and is applicable to any language. It is a string processor which uses R-expressions to define the syntax of new statements (macros) and a language, SL, to define the semantics of these statements. SDM provides flexibility of syntax to macro-calls and good error detection.

2. R-EXPRESSION

An R-expression has strings and metavariables as its basic elements. It is similar to a regular expression in the sense of finite automata[8]. It uses the following operations: or, concatenation, optional occurrence and repetition. These may occur recursively to any level.

The notation used in an R-expression is as follows:

| | vertical stroke separates alternatives.

? denotes optional occurrence of the immediately preceding syntactical unit.

# denotes the occurrence of the immediately preceding syntactical unit one or more times.

In the case of

```
'let' '(' (<variable>'='<expression>';') #
  ?<variable>'='<expression>')'                    (1)
```

we have that: <variable>'='<expression>';' may occur zero or more times.

A macro-call according to the syntax above could be:

```
let (a = b+c*d; b = a+d)
```

Another example of an R-expression is:

```
'if'<expression><oper><expression>'then'<all>
('else'<all>|<null>)                                    (2)
```

Each R-expression begins with a reserved string. There is one such string for each defined statement (macro).

The following grammar in Backus-Naur Form[9] defines the syntax of R-expressions:

```
<R-expression> ::= <*string><body>|<*string>
<body> ::= <element>|<element> <body>
<element> ::= <*string>|<*metavariable>|<list_of_alternatives>|
              <option>|<repetition>
<list_of_alternatives> ::= (<body>|<more>)
<more> ::= <body>|<body>'|'<more>
<option> ::= (<body>)?|<element>?
<repetition> ::= (<body>) #  |<element> #
```

The star in <*string> and <*metavariable> is to denote that these syntactic classes are not being defined in the grammar above.

Each metavariable which appears as an element of an R-expression represents a formal parameter. They represent syntactic classes which are defined by the user in a nonrestricted Backus-Naur Form at the beginning of his program. As a macro-call is scanned, each actual parameter is checked by a recognizer to determine whether it belongs to the syntactic class denoted by the corresponding metavariable. In case it does not, an error message is printed and the processor

proceeds to the end of the macro call . Nevertheless, other macro-calls that may occur in the actual parameters of this one are fully analysed by the processor.

So, for the R-expression (1), the user could define <expression> by writing:

```
<expression> : <term>|<expression>'+'<term>;
<term> : <primary>|<term>'*'<primary>;
<primary> :  <variable>|<integer>; end
```

at the beginning of his program.

Some of the syntactic classes which are already defined in the system are: <identifier>,<variable>,<integer>,<macro-call> and <all>. The class <macro_call> represents a call of any of the macros which were defined by the user. The class <all> represents any sequence of characters terminated by the reserved word es. It is useful when an actual parameter may be any sequence of statements of a sophisticated base language. In this case, the user may find cumbersome to define the syntax of the whole base language and may use <all>. Nevertheless, any macro call in an actual parameter described by <all> will be analysed and expanded by the processor. For the syntactic recognition of actual parameters, the processor uses a version of[11] , which is the most efficient general context-free recognizer know. Furthemore, the recognizer will work correctly even if the user specifies an ambiguous syntax. On the other hand, recognition of the structure of a macro-call, as specified by the operators of an R-expression, is done by a standard top-down technique[10] .

In order to make error recovery efficient, each macro call ends with a reserved word. In the presently implemented version of SDM, this word (em) is the same for every macro, and does not appear at the end of R-expressions. In a new version of SDM currently being implemented, the reserved word that ends a macro call may be different for different macros. Then, it has to be specified as the last element of the R-expression that describes the syntax of the macro.

In order to be able to refer to parameters and operators of an R-expression, the processor numbers them from left to right. A new number is produced for each formal parameter, ? , #, | and each) that ends a list of alternatives. So for the examples (1) and (2) above we have:

```
'let' '('(<variable>'='<expression>';') # ? <variable>'=' <expression>')'
          1          2          3 4    5          6

'if'<expression><oper><expression>'then'<all>('else'<all>|<null>)
     1        2        3          4          5 6 7   8
```

A number which refers to a | or a) that ends a list of alternatives, has status 'true' if the preceding alternative has been matched sucessfully to the macro call being processed. Otherwise it has status 'false'.

Similarly, a number associated to a ? , has status true if and only if the preceding syntactic unit has been found on the macro call being analysed.

3. SL LANGUAGE

The semantics of a macro is defined by a semantic routine written in the SL language. This routine is associated to the corresponding R-expression. SL has output - statements, if-statements and repeat - statements.

An output-statement has one of the following forms: line, c<integer>,<string>, $<integer> and ℓ<integer>. The statement line starts a new line of 80 characters (card image) on the output. c<integer> causes any subsequent output to be positioned such that its first nonblank symbol starts at a specified column. <string> causes the sequence of characters between quotes to be output. $ <integer>, where <integer> refers to a formal parameter in the R-expression, outputs the corresponding actual parameter. ℓ<integer> is used in the generation of labels. When ℓ i is executed for the first time , a three digit integer is generated and written on the output. Another occurrence, of ℓ i will output the same integer only if both occurrences are in the same semantic routine and are used in connection with the same macro-call, otherwise a different integer will be output and in this respect this statement will never repeat itself. If $i_1 \neq i_2$, ℓi_2 and ℓi_2 output different integers in any case.

In SL, statements may be grouped into a compound[9] by the use of begin and end. An if-statement has the form: if boolean<expression> then <statement>, where the basic elements of a <boolean expression> are integers which have status 'true' or 'false' with respect to the R-expression.

The repeat - statements semantically define the repeat operations in the R-expression, they have the form:
repeat <integer><statement-list> end, where the <integer> is associated to a repeat operator. A repeat-statement r_1 is said to be controlled by another repeat-statement r_2 if and only if r_1 is in the <statement-list> of r_2 in the semantic routine, and the syntactic unit u_2, immediately preceding the repeat operator $\#_2$ (referred to by r_2) contains the operator $\#_1$ (referred to by r_1). In this case the<statement-list> of r_2 is executed as many times as u_2 was found in the macro-call. For each of these occurrences of u_2 , the <statement-list> of r_1 is executed as many times as u_1 was found. If r_1 is not being controlled by another repeat-statement then its <statement-list> is executed as many times as u_1 was found in the macro-call. If we have the R-expression: 'C'('B''A' $\#_1$) $\#_2$, and the macro-call:

CBAAAABAAABAABA, then the routine:

repeat 2 'E' repeat 1 'H' end end

in the definition, will cause the output:

EHHHHEHHHEHHEH

While the routine:

repeat 1 'E' repeat 2 'H' end end

produces the output:

EHHHHEHHHH ..... (10 times)

The syntax of the SL language is given in the appendix.

4. MACRO-DEFINITION

In order to define a certain macro, the user has to write:

macro<R-expression>define<semantic-routine>endmacro

where the <R-expression> defines the syntax of the macro and the <semantic-routine> is a program in the SL language which defines de semantics.

If we assume that <expression> and <oper> have been defined respectively as the class of the aritmetic expressions, and the class of comparison operators in FORTRAN, then the macro-definition:

```
macro 'IFL'<express><oper><express>'THEN'<all>('ELSE'<all>)?

define c7 'IF (' $1 $2 $3') GO TO' ℓ1 if 6

then begin c7 $5 end c7 'GO TO' ℓ2

c2 ℓ1 c7 $4 c2 ℓ2 c7 'CONTINUE' c1 endmacro
```

will cause the expansion of*:

```
IFL X.LT.Y THEN
     IFL  X.GT.Z THEN
     R = (Z-Y)/2.
     Y = R-X**2 EM
     A = A+B ES
           ELSE
           Y=0  EM
```

* Note that: ES EM may be substituted by EM only.

into:

```
      IF(X.LT.Y) GO TO 997
      Y = 0
      GO TO 996
997   IF(X.GT.Z) GO TO 999
      GO TO 998
999   R = (Z-Y)/2.
      Y = R-X**2
998   CONTINUE
      A = A+B
996   CONTINUE
```

5. CONCLUSION

SDM has been implemented as a preprocessor and it is compiler independent. Its main advantages are: independence of the base language, good error detection and the flexible syntax of macro calls.

APPENDIX

Syntax of the SL language:

<semantic-routine> ::= <statement-list>

<statement-list> ::= <statement>|<statement><statement-list>

<statement> ::= <*string>|$<*integer>|c *integer>|ℓ<*ingerger>|
line|<if-statement>|<repeat-statement>

<repeat-statement> ::= repeat<*integer><statement-list>end

<if-statement> ::= if<boolean-expression>then<compound>

<compound> ::= <statement>|begin<statement-list>end

<boolean-expression> ::= <term>|<boolean-expression> v <term>

<term> ::= <factor>|<term> ∧ <factor>

<factor> ::= ¬ <primary>|<primary>

<primary> ::= <*integer>|(<boolean-expression>)

REFERENCES

1. Strachey, C., A General Purpose Macrogenerator, Computer J. 8 (Oct. 1965), 225-241.

2. Ferguson, D.E., The evolution of the meta-assembly program, Comm. ACM 9 (March 1966), 190-193.

3. McIlroy, M.D., Macro Extensions of Compiler Languages, Comm. ACM 3 (Apr. 1960), 214-220.

4. Day, A.C., A Macro-Processor, for FORTRAN, Technical Report Nº 2, Computer Centre, University College of London.

5. Leavenworth, B.M., Syntax Macros and Extended Translation, Comm. ACM 9 (Nov. 1966), 790-793.

6. Cheatham, T.E., The Introduction of Definitional Facilities into Higher Level Programming Languages, Proc. AFIPS 1966 Fall Joint Computer Conference, vol. 29, pp. 623-637.

7. Brown, P.J., A Survey of Macro-Processors. Annual Review of Automatic Programming, vol. 6, part 2, Pergamon Press, London, 1969, pp.37-88.

8. Ginzburg, A., Algebraic Theory of Automata, Academic Press, New York, 1968.

9. Revised Report on the Algorithmic Language ALGOL-60-Comm. ACM 6 (Jan. 1963), 1-17.

10. Cocke, J., and Schwartz, J.T., Programming Languages and Their Compilers, Preliminary Notes, New York University, New York 1970.

11. Earley, J., An Efficient Context-Free Parsing Algorithm. Comm. ACM 13 (Feb. 1970), 94-102.