Programming language RIGAL as a compiler writing tool

M. I. AUGUSTON
Institute of Mathematics and Computer Science of the Latvia University
226250, Riga, Rainis boulevard 29,
Latvia, USSR

Abstract. A new programming language for compiler writing is described briefly in this paper. The main data structures are atoms, lists and trees. The control structures are based on a advanced pattern matching. All phases of compilation,including parsing, optimization and code generation, can be programmed in this language in short and readable form.

1. Introduction

Programming language RIGAL is intended to be a tool for syntactic analysis, program optimization, code generation and for preprocessor and convertor writing. The main principles of RIGAL are the following:

a) formal grammars are used in the pattern matching mode;

b) operations, such as tables formation, code generation, message output are executed simultaneously with parsing, like in YACC [1], CDL-2 [2]. Therefore, the principle of dependence of control structures in the program upon data structures used for program's work is well supported;

c) attribute grammars can be modeled easily in RIGAL. The language has reference facility, which is a good solution of global attribute problem;

d) the language has rich spectrum of tree manipulation facilities, like in Vienna Definition Language [3], including tree grammars. Trees can be conveniently used as data structure for different table implementation;

e) the language supports multi pass compiler design, as well. Trees can be used as intermediate data;

f) the language provides the split of program into small modules (rules) and presents various means to arrange interaction of these modules;

g) RIGAL is a closed language, i.e. all the necessary computations and input-output could be executed by internal means and there is no need to use external semantic subroutines which are written in Pascal or C. Therefore, the portability of RIGAL programs to other computers is increased.

Detailed presentation of language, as well as more examples are given in [4].

2. Data Structures. Operations

The only data structures in RIGAL are atoms, lists and trees.

The atom is a string of characters or a number. Identifiers and numbers can be written just in the RIGAL program text, other atoms must be quoted. Special atom NULL represents an empty list, an empty tree and Boolean value "false".

Variable name begins with symbol $. Values can be assigned to variables.

For example, $E := ABC assigns atom ABC to variable $E.

The **list** is an ordered sequence of objects - atoms, another lists and trees.

The **list constructor** yields a list of objects. For example, $E := (. A B C .) assigns a list of atoms A, B and C to the variable $E . It is possible to get elements from a list by indexing, e.g. $E[2] is atom B, but $E[-1] is atom C. Operation L !. *e* appends element *e* to the end of list L. Two lists L1 and L2 can be concatenated by operation L1 !! L2 .

For tree creation the **tree constructor** is used, e.g.

<. A : B, C : D .>

Objects placed just before ':' are called **selectors**. A selector may be any atom, except NULL. Selectors of the same level must be different. The pair 'selector : object' in the tree is a **branch**. Branches in the tree are unordered.

Statement $E := <. A : X, B :(.3 7.), C :<. A : 2 .> .> assigns a tree with three branches to variable $E. It is possible to extract components from the tree by means of selection operation . For example, $E.A is atom X, $E.C.A is atom 2, and $E.B[2] is atom 7. But $E.D yields NULL, because there is no branch with selector D in this tree.

Operation T1 ++ T2 appends branches of tree T2 to tree T1. If T1 has a branch with the same selector, this branch is replaced by the branch from the T2.

The equality of objects can be checked using operations = and <>. The result is atom T ('true') or NULL ('false').

Common arithmetic and relational operations +, -, *, *div*, *mod*, <, >, <= and >= are defined for numerical atoms .

The arguments of logical operations *and*, *or* and *not* can be arbitrary objects. If the object differs from NULL, it represents the 'true' value.

It is possible to write $X !.= *e* instead of statement $X := $X !. *e* .The same short form is allowed for operations !!, ++ and + .

3. Rules. Simple Patterns

Rules are used basically to check if an object or sequence of objects complies with some grammar.

The grammar is described by the patterns. A rule call should be accomplished by some object or object sequence called **rule arguments** . They are matched with the patterns in the rule. Some statements (assignments, input-output operations) can be executed simultaneously with pattern matching .

Rule can compute and *return* some value to the calling point. Thus, the concept of rule is analogous to concept of procedure and function in conventional languages.

The rule execution ends with success or failure depending on the result of argument and pattern matching. That is why a rule can be used as a pattern in another rule.

Example of a rule definition: #L1 A B ##

Here the pattern contains atoms A and B. Rule #L1 can be called, for example, in such a way: #L1(A B). Atoms A and B are given as arguments. In this case argument matching with pattern is successful.

Call #L1(A C) fails, because the second argument fails to match the second pattern, but #L1(A B C) ends with success, as the third argument is not requested by any pattern.

Variable can be a pattern element. It matches successfull just one object from the argument sequence and, as a side effect, this object is assigned to the variable.

Statements can be placed before the pattern element and after it. The statement group is closed in symbols '/'. Statements in the group are separated by semicolons.

The *return* statement in the rule ends the rule execution and yields the returned value.

Example. #L2 $A $B / *return* (. $B $A.)/ ##

Variable $X gets value (.2 1.) after statement $X:= #L2(1 2) is executed .

If *return* statement is not described in the rule, then the NULL value is returned.

The rule pattern element can refer to some rule (recursively, as well).

Example.

#L3 B C ##

#L4 A #L3 D ##

Call #L4(A B C D) is successful, but #L4(A E F D) is unsuccessful.

Every pattern element, when successfully matched with corresponding rule argument, returns some value. Value of atom pattern coincides with this atom, value of variable pattern coincides with the value obtained by this variable as a result of matching with the argument. The value of rule is formed by the *return* statement. Value, returned by the pattern element can be assigned to a variable.

Example.

#L5 $R !.:= $A $RN !!:= #L6 / *return* $R/ ##

#L6 $M !.:= $B $M !.:= $C / *return* $M/ ##

When statement $X := #L5(1 2 3) is executed, variable $X gets value (. 1 2 3 .). All variables in the rule are initialized by NULL. Then the first application of pattern element $R !.:= $A in #L5 assigns to variable $R the value of expression NULL !. $A that equals to (. $A .) .

There are some built-in rules, such as #NUMBER or #IDENT. Patterns of the type $Id := #IDENT and $N := #NUMBER are used so often, that a default is introduced. If the name of the pattern variable begins with letter "I", then it matches atoms - identifiers, and, if it begins with letter "N", then it matches numerical atoms.

One rule can contain several groups of patterns. They are called **branches** of the rule. Arguments of the rule are matched with branches in succession, until one branch succeeds. Branches in the rule are delimited by symbol ';;'. In case of branch failure, some statements can be executed. Such statements can be written at the end of the branch after keyword *onfail*. In case of branch failure, control is transferred to them and statements,given in *onfail*- unit , are executed. So, in case of branch failure, causes of failure could be analyzed and message could be output.

4. Patterns

List pattern (. P1 P2 ... Pn .) matches only those objects, which are lists and elements of which match patterns P1, P2, ... and Pn respectively.

There are patterns for **alternative**

(P1 ! P2 ! ... ! Pn)

and for **option**

[P1 P2 ... Pn]

with usual semantics.

Iterative sequence pattern (* P1 P2 ... Pn *) denotes the repetition of pattern sequence zero or more times. It is possible to define rules with a variable number of arguments.

Example.

```
#Sum (* $S +:= $Num *) / return $S/ ##
```

Call #Sum(2 5 11) returns 18.

Example. Rule that determines length of the list can be defined as follows:

```
#Len (. (* $E / $L +:= 1 / *) .) / return $L/ ##
```

Call #Len((. A B C .)) returns 3.

Iterative sequence pattern (+ ... +) is similar to (* ... *), but the former denotes the repetition one or more times.

It is possible to describe iterative sequence patterns with delimiters in such form: (* P1 P2 ... Pn * delimiter) . An atom or a rule name can be used as delimiter .

Example. Analysis of a simple Algol-like declaration. A fragment of variable table coded in a tree form is returned as a result.

```
#Declaration
        $Type := ( integer ! real )
        (+ $Id / $Rez ++:= <. $Id : $Type .> / + ',')
        / return $Rez / ##
Call #Declaration ( real X ',' Y ) returns value <. X : real, Y : real .>
```

Example. Simple arithmetic expression parsing. When successful, an expression tree is returned, which can be regarded as an intermediate form for the next compilation phases.

```
#Expression
        $A1 := #Additive_el
        (* $Op := ( '+' ! '-' ) $A2 := #Additive_el
         / $A1 := <. op : $Op , arg1 : $A1 , arg2 : $A2 .> / *)
        / return $A1 / ##
#Additive_el
        $A1 := #Term
        (* $Op := ( '*' ! ' div' ) $A2 := #Term
         / $A1 := <. op : $Op, arg1 : $A1, arg2 : $A2 .> / *)
        / return $A1 / ##
#Term
        $A := ( $Id ! $Num ) / return $A / ;;

        '(' $A := #Expression ')' / return $A / ##
       Call #Expression( X '*' Y '+' 7 ) returns value
<. op: '+', arg1: <. op: '*', arg1: X, arg2: Y .>, arg2: 7 .>
```

5. Tree Patterns

Tree pattern can be written in the form

<. a1 : p1, a2 : p2, ... , an : pn .>

where ai are atoms and pi are patterns.

Tree pattern branches are applied to corresponding branches of the argument tree in the same order as they are written in the pattern.

Therefore,the order of tree traversing may be controlled. We can traverse some branches of the tree repeatedly, if the corresponding selector in the tree pattern is repeated, or we can omit some branches of the tree, if the corresponding selectors are omitted in the pattern.

Optional branches in tree pattern are enclosed in brackets '[' and ']'.

Example. Let us suppose expression tree to be formed like in the above mentioned example. The task is to traverse the tree and *return* a list that represents the Polish postfix form of this expression.

```
#Postfix_form
    <.   arg1: $Rez := #Postfix_form,
         arg2: $Rez !!:= #Postfix_form,
         op:  $Rez !.:= $Op .>  / return  $Rez / ;;

    $Rez := ( $Id ! $Num )  / return (. $Rez .) /   ##
```

Call #Postfix_form(<. op: '-', arg1: X, arg2: <. op: '*', arg1: Y, arg2: 5 .> .>) returns value (. X Y 5 '*' '-' .)

Iterative tree pattern has the form <* $V : P *> , where $V is a variable and P - pattern. This pattern defines a loop over the tree. All selectors of the argument tree are assigned to variable $V one by one. Pattern P is applied to each object, which corresponds in the argument tree to the current selector in variable $V.

Example. Traversing the variable table.

```
#Var_table
        <* $Id : $E := ( integer ! real )
        / $R !.:= (. $Id $E .) / *> / return $R /   ##
```

Call #Var_table(<. X : integer, Y : real, Z : real .>) returns value (. (. X integer .) (. Y real .) (. Z real .) .)

6. Pattern of Logical Condition Testing

Pattern S'(*expression*) is performed as follows. First of all the expression is evaluated, and iff its value is different from NULL then matching with the argument is successful. Special variable $$ in the expression denotes current argument, to which this pattern is applied. Using this kind of pattern many context-sensitive situations can be described.

For example, by such pattern we can recognize a special case of assignment statement "X := X + E"

```
$Id ':=' S'( $$ = $Id ) '+' #Expression
```

We can skip tokens, until semicolon using pattern (* S'($$ <> ';') *)

7. Statements

There is an analog of McCarthy's conditional statement in RIGAL. It has the following form :

```
if    cond -> stmts
elsif cond -> stmts
        ... ... ...
fi
```

If the value of conditional expression is not equal to NULL , then

corresponding statements are executed.

The *fail* statement ends the execution of the rule branch with failure.

Loop statement

forall $V *in* Expr *do* stmts *od*

iterates its body over a list or a tree described by expression Expr. All elements of the list or all selectors of the tree are assigned to variable $V in succession .

Loop statement of the type

loop stmts *end*

repeats statements of the loop body, until one of the statements -*break*, *return* or *fail* is not executed.

Objects designed by RIGAL program can be saved on disc and loaded back by statements *save* and *load*.

Text files can be opened by *open* statement for text output of messages, generated code, etc.

Sequence of atoms can be output in text file FFF by the statement of the form

FFF << expr1 expr2 ... ... exprN

A blank is output after each atom. Symbol @ cancels the additional blank symbol output.

Statement '<<' always begins output on a new line, but '<]' statement continues output on current line.

8. Program Structure

RIGAL program consists of main program and rules. Operations like initial object loading from external files, rule calls and created object saving on external files are concentrated in the main program .

When RIGAL is used for parsing, first the input text has to be processed by scanner - a program, which converts the text into a list of tokens. This list is an object of RIGAL, and can be loaded and processed by RIGAL program.

Intermediate results, e.g., abstract syntactic trees can be saved, and another RIGAL program can load and update them.

9. Attribute Grammars and RIGAL. Global References

There is a close analogy between attribute grammar and RIGAL program. Rules in RIGAL correspond to grammar nonterminals, and variables - to attributes.

In attribute grammars the greatest part of attributes are used as transit attributes. To avoid this, global attributes are introduced in the attribute grammar implementations.

There is a good conformity between data structures and control structures in RIGAL program. Hence, it is possible to use the control structures of RIGAL program, in order to refer to the data elements.

Rule variables can be accessed from some other rule using reference of the type :

last #L $X

This reference denotes the value of variable $X in the last (in time) and still active instance of rule #L. Such references can be used in left and right sides of assignment statement. Therefore, implementation of both synthesized and inherited attributes is possible as it is demonstrated in the following scheme.

```
#LA ... ... ...
 assigns value to attribute $A
 calls #LB
 ... ... ...
##
#LB ... ... ...
 $B1 := last  #LA $A
  -- uses inherited attribute $A from #LA
 calls #LC
  -- after this call the value of attribute $C from #LC is
  -- assigned to the synthesized attribute $B2
 ... ... ...
##
#LC ... ... ...
 assigns value to attribute $C
 last #LB $B2 := $C
  -- the value is assigned to the synthesized attribute
  -- $B2 of #LB
 ... ... ...
##
```

Multiple visits in attribute grammars could be modeled by explicit multple rule calls.

10. Simple Telegram Problem.

We shall consider simple telegram problem [5] as compiler model. A program is required to process a stream of telegrams. The stream is available as a sequence of words and spaces. The stream is terminated by an empty telegram.

Each telegram is delimited by symbol "****". Telegrams are to be processed to determine the number of words with more than twelve characters for each telegram. Telegrams together with statistics have to be stored on an output file eliminating all but one space between the words.

For demonstration purpose the program is divided into two parts - parsing phase and output code generation phase.

The input stream is represented as a list of tokens, where a token is an one-character atom. The first phase builds an intermediate result - abstract syntactic tree.

The structure of input, intermediate and output data can be described by RIGAL rules.

The input stream.

```
#telegram_stream
      (+ #telegram #end +) [ #blanks ] #end ##
#telegram
      [ #blanks ] (+ #word #blanks +)  ##
#word
      (+ #letter +)  ##
#blanks
      (+ ' ' +)  ##
#end
      '*' '*' '*'  ##
```

```
#letter
      ( A ! B ! C ! ... ! X ! Y ! Z ) ##
```

The intermediate data.

```
#S_telegram_stream
      (. (+ #S_telegram +) .) ##
#S_telegram
      <. text :  (. (+ #S_word +) .),
        long_word_n: $N .> ##
#S_word
      (. (+ #letter +) .)  ##
```

The output stream.

```
#output_telegram_stream
      (+  #telegram1 #end +)  #end ##
#telegram1
      (+ #word ' ' +) $long_word_num  ##
```

The main program has form:

```
#telegram_problem
       load $T 'Letters.lex';
      $S := #telegram_stream_analysis($T);
      open  Out 'LP:';
      #generate_output_stream($S)
##
```

The rules are:

```
#telegram_stream_analysis
      (. (+ $R !.:= #A_telegram #end +)  [ #blanks ]  #end .) / return  $R /  ##
#A_telegram
       $long_word_num := 0/  [ #blanks ]
      (+ $R !.:= #A_word #blanks +)
      / return  <. text: $R,  long_word_n: $long_word-num .> /  ##
#A_word
      (+ $R !.:= #A_letter +)
      / if  #Len($R) > 12 ->  last  #A_telegram $long_word_num + :=1
       -- rule #Len was defined in Unit 4.
        fi;  return $R /  ##
#A_letter
 $R := ( A ! B ! C ! ... ! X ! Y ! Z ) / return $R / ##

#generate_output_stream
      (. (+ #G_telegram +) .)   / Out << '***'/  ##
#G_telegram
      <. text: (. (+ #G_word / Out <] ' ' / +) .),    long_word_n: $N .>
      / Out <] $N '***' / ##
#G_word
      (. (+ $L / Out <] @ $L / +) .)  ##
```

These rules are obtained from rules, which describe data structures, by adding operations to the corresponding patterns. The whole program is written by the recursive descent method.

11. Implementation

RIGAL is implemented on PDP-11 in RSX-11,and then ported to VAX-11 in VMS and to IBM PC AT in MS DOS.

First the interpreter of RIGAL was written in Pascal, and afterwards the optimizing compiler RIGAL --> Pascal was written in RIGAL itself.

12. Conclusions and Future Work

As it was demonstrated above, RIGAL supports syntax-oriented style of compiler design. Programs written in RIGAL are well-structured and it is easy to read and debug them.

As it was proved by our experience [6], the optimizing RIGAL compiler in VAX/VMS environment makes it possible to implement production quality compilers for high level languages.

RIGAL can be considered as yet another language prototyping tool in the sense of [7], because it allows the designer to develop an experimental translator in a short period of time.

RIGAL support system besides interpreter for debugging purposes and optimizing compiler includes a cross-referencer, which helps to avoid misuse of global variables.

In order to improve static and dynamic type checking, variable type descriptions in the form of formal comments would be added to the language.

Taking in account that the control structures of RIGAL program are very close to input data structures, it seems promising to develop automatic and semiautomatic methods for test example generation for the given RIGAL program.

References

[1] **Johnson S.C.** YACC - yet another compiler compiler. -Bell laboratories, Murray Hill,N.J., 1978, a technical manual.

[2] **Koster C.H.A.** Using the CDL compiler compiler.- Lecture Notes in Computer Science, 1977, vol. 21.

[3] **Lucas P.** Formal definition of programming languages and systems. - IFIP congress, 1971.

[4] **Auguston M.I.** Programming language RIGAL. - Riga, LSU, 1987, (in Russian).

[5] **Jackson M.** Principles of Program design. - Academic Press , 1975.

[6] **Barzdin J.M., Kalnins A.A., Auguston M.I.** SDL tools for rapid prototyping and testing. - in SDL'89 : The Language at Work, ed. O.Faergemand and M.M.Marques, North-Holland, 1989, pp.127-133.

[7] **Herndon R., Berzins V.** The Realizable Benefits of a Language Prototyping Language. - IEEE Transactions on Software Engineering, vol.14, No 6, June 1988, pp. 803-809.