RIGAL -
a Programming Language for Compiler Writing

Mikhail AUGUSTON
Institute of Mathematics and Computer Science
The University of Latvia
Rainis boulevard 29, Riga, Latvia, SU - 226250

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

1. Introduction

Programming language RIGAL is intended to be a tool for parsing(context checking, diagnosing and neutralization of errors included), for code optimization, code generation, static analysis of programs, as well as for the programming of preprocessors and convertors.

Almost all the systems envisaged to solve compiler construction problems contain means to describe context-free grammar of the source language. Earlier systems, like the Floyd - Evans language, [1] present tools to work with stack, which is used for parsing. Parsing methods for limited grammar classes are implemented in the languages and systems of later generations (usually LL(1) or LR(1)).

Such systems as YACC (Johnson [2]), CDL-2 (Koster [3]), SHAG (Agamirzyan [4]) and many others make use of synchronous implementation of parsing and different computations, e.g., formation of tables, context checking, etc. Usually these actions are performed by call of semantic subroutines, written in some universal programming language (e.g., in Pascal or C).

Attribute grammars advanced by Knuth [5] have greatly influenced development of systems for compiler construction. Systems, like, SUPER (Serebryakov [6]), ELMA (Vooglaid, Lepp, Lijb [7]), MUG2 (Wilhelm [9]) are based on the use of attribute grammars not only for parsing, but for code generation as well.

Pattern matching is a convenient tool for programming of parsing, optimization and code generation. The REFAL programming language [10], acknowledged for translator writing, may serve as a good example.

Vienna method for defining semantics of programming languages [11] suggests the usage of labelled trees in order to present the abstract syntax of programs. Representation of compilation intermediate results in the tree form has become usual (see [12]).

Dependence of control structures in the program from data structures used for program's work is one of the basic principles in programming. The recursive descent method could be considered to be the application of dependence principle.

The above mentioned ideas and methods were taken into account when creating RIGAL language.

The language possesses few basic notions. Data structures contain atoms, lists and trees. Advanced mechanism of pattern matching lies at the basis of control structures.

The fact that RIGAL is a closed language makes RIGAL distinctive. That means that almost all the necessary computations and input-output could be executed by internal means and there is no need to use external semantic subroutines. Therefore the portability of RIGAL programs to other computers is increased.

Means for work with trees, different patterns including, enable both programming of parsing algorithms and optimization phases and code generation as well. The language supports design of multipass translators. Trees are used as intermediate data.

The language allows to split the program into small modules (rules) and presents various means to arrange interaction of these modules. Pattern matching is used for parameter passing.

RIGAL supports attribute translation scheme and easy

implementation of synthesized and inherited attributes is possible. The problem of global attributes is solved by usage of special references.

Lexical analysis is a separate task and requires special language facilities for description as it is, for example, in LEX/YACC [2] system. In the current implementation of RIGAL two scanners are included that accept lexics of Pascal and RIGAL.

2. Implementation

RIGAL was designed and implemented in the Computing Center of Latvia University in years 1987-1988. The first implementation was for PDP-11 in RSX-11.

At the present stage RIGAL interpreter has been developed and optimizing compiler RIGAL -> Pascal has been implemented by means of RIGAL itself. The interpreter and the compiler have been ported to VAX/VMS and IBM PC AT /MS DOS environments.

3. Lexical Rules

The text of RIGAL program is a sequence of tokens - atoms (e.g., identifiers and integers), keywords (e.g., *if, return*), special symbols (e.g., +, ##), names of variables and rules (e.g., $A, #L). Tokens may be surrounded by any number of blanks. A comment is any string of symbols that begins with two consecutive symbols '-' (minus). The end of the comment is the end of the line. For example,

```
#Sum     -- rule for addition of two numbers
      $N1    -- the first number
      $N2    -- the second number
    / return  $N1 + $N2 /    -- return of the result
##
```

4. Data

4.1 Atoms

An atom is a string of symbols. If the atom is an identifier (the first symbol is a letter followed by letters or digits or underscore symbols), in the text of RIGAL program it could be written directly: AABC total_number x25

Numerical atoms are integers, for instance, 2, 187, 0, -25

In other cases the atom is quoted: '+' ':=' '1st'

Some identifiers are reserved as keywords in RIGAL. If they are used as RIGAL atoms, they should be quoted. For example, 'if', 'return'. Besides, any atom, which is an identifier, also can be quoted - ABC and 'ABC' represent one and the same atom.

It should be noted that 25 and '25' are different atoms, the latter is just a string of symbols '2' and '5'.

Two special atoms are distinguished in the language.

NULL - this atom is frequently yielded as a result of different operations, if something was incorrect in the process of computations. This atom also represents an empty list, an empty tree and Boolean value "false".

T - usually this atom is yielded by logical operations and represents value "true".

4.2 Variables

The name of a variable must begin with the symbol $, followed by an identifier. Value can be assigned to a variable, for example, by the help of assignment statement: $E := A

In this case the atom A becomes value of the variable $E.

In RIGAL variables have no types, the same variable may have an atom, a list or a tree as a value in different time moments.

4.3 Lists

Ordered sequences, i.e., *lists* can be composed from atoms and

from other lists and trees, as well. A special function - *list constructor* serves for list formation. For instance, (. A B C .) forms a list of three atoms A, B and C.

Arguments of the list constructor may be expressions. The sample $E := (. (. 8 14 7 .) (. A B .) .)
could be rewritten as follows:

$A := (. 8 14 7 .); $B := (. A B .); $E := (. $A $B .);

Separate elements of the list can be selected by indexing. Hence, $B [1] is atom A, $A [2] is atom 14, $E [2] is list (. A B .), but $E [10] is atom NULL

If the value of the index is a negative number, for instance -N, then the N-th element, beginning from the end of the list, is selected. For example, $A [-1] is atom 7.

The necessity to add one more element to the list is quite common. Operation !. is envisaged for this purpose.

Example. (. A B .) !. C yields the list (. A B C .)

To link two lists in a new list the operation !! is applied (list concatenation). For instance, (. A B .) !! (. C D .) yields (. A B C D .).

4.4 Trees

Tree constructor is used to create a tree. For example,

<. A : B, C : D .>

One can imagine tree as a graph, the nodes and arches of which are marked by some objects.

Objects before ':' in the tree constructor are named *selectors.* In the given implementation solely atoms, which are identifiers (except NULL), may serve as selectors. In the graphical representation selectors correspond to arches of the graph. All selectors of one and the same level in the tree must be different.

Any object - atom, list or tree, except atom NULL, may correspond to terminal nodes of the graph ("leaves" of the tree). Hence, multilayer trees can be built. For instance,

```
<. A : B,  C : <. D : E, F : G .> .>
```

Pair "selector : object" in the tree is named *branch* of the tree. Branches are unordered in the tree.

Likewise for the list constructor, the tree constructor may be described by expressions (in both selector and object places), for instance,

```
$X := D;    $B :=  (. 2  8 .);
$C :=  <. A : <. M : K .>, $X : '+' , E :  $B   .>;
```

Select operation serves to extract the tree component. It is in the following form: *tree* . *sel* , where *tree* is the expression, whose value must be some tree, but *sel* is the expression, whose value must be an atom-identifier.

Consequently, $C . A is the tree <. M : K .> ,
$C . D is the atom '+' , $C . E is the list (. 2 8 .) ,
$C . E[2] is the atom 8 , $C . A . M is the atom K

If there is no branch with a given selector in the tree, then the result is NULL: $C . W is atom NULL.

Operation of the tree "addition" is performed as well: T1 ++ T2 , where T1 and T2 are trees. Tree T2 branches are added to the tree T1 one by one. If in the tree T1 there already exists a branch with the same selector, the branch is substituted by a new one. Therefore, the operation "++" is not commutative.

It should be pointed out that the tree constructor is computed from left to right, i.e., <. s1 : a1, s2 : a2, s3 : a3 .> gives the same result as the expression

```
(( NULL ++ <. s1 : a1 .> ) ++ <. s2 : a2 .>) ++ <. s3 : a3 .>
```

5. Expressions

Operations = and <> serve for the comparison of objects. The result of the comparison is either T ("true") or NULL ("false").

Atoms are matched directly, for instance, a = b gives NULL, 25 = 25 gives T, 17 <> 25 gives T.

Lists are considered equal iff they contain equal number of components and if these components are equal respectively.

Trees are considered equal iff they contain equal number of branches and if one of the trees contains the branch "S : OB", then the other tree also contains the branch "S : OB1" and OB = OB1.

Arithmetical operations +, -, *, *div, mod* are assigned for numerical atoms. The essence of these operations is similar to those in Pascal. The result of an arithmetical operation is a numerical atom. Atom NULL is also admitted as the argument of arithmetical operation, in this case integer 0 is supposed to be its value. Under matching these atoms are considered different, i.e., NULL = 0 gives NULL.

Besides the operations = and <> numerical values could be compared by the help of >, <, >= and <=.

Logical operations *and, or* and *not* usually are applied under conditions in conditional statements. Their arguments may be arbitrary objects. If the object differs from NULL, it is supposed to have the value "true" in a logical operation. Atom NULL represents the "false" value. The result of a logical operation always is either T or NULL.

In order to make complex hierarchic objects (trees and lists of complex structure) more visual and to improve the work of pattern matching, trees and lists may be labelled. A name is an atom opposed to the root node of the tree or the whole list. Labelling operation is written the following way:

A :: OB

where A is an atom , OB is an object (a tree or a list). For example, Add_op :: <. arg1 : 5, arg2 : 4 .>

The execution order of operations in the expression is controlled by parentheses "(" and ")". Priority is assigned to every operation, and, if the execution order is not defined by parentheses, operations are executed according to their priorities - beginning with the higher and proceeding to the lower.

Operations are listed in decreasing order of priorities (some operations will be discussed later).

1) Rule call, list constructor, tree constructor, *last*

2) Selector ".", index "[]", ::

3) *not*, unary -

4) *, *div*, *mod*

5) !. , !! , ++ , +, binary -

6) = , <> , > , < , >= , <=

7) *and*

8) *or*

Binary operations of the same priority are executed from left to right, while unary operations from right to left.

5.1 Accumulative Assignment Statements

It is quite often that working with a list or a tree elements are added step by step, thus the growing object is retained in the same variable. Therefore short form of assignment statement has been introduced. For the operations !. , !! , ++ and + statements of the form $X := $X *op* Expr can be written as $X *op* := Expr .

5.2 Semantics of Variables

In the implementation of RIGAL every object - atom, list or tree has a descriptor. It is a special data structure that contains some information about this object: the value of atom, the number of elements and pointers to elements for lists and trees. Variables have pointers to object descriptors as values.

Statement $X := OB assigns to the variable $X a pointer to the descriptor of object OB. After the execution of the statement $Y := $X both variables $X and $Y contain pointers to the same object OB.

Operations !.:= , !!:= , +:= and ++:= change the descriptor of their first argument, i.e., have a side effect. Sometimes it can be undesirable. For example,

```
$A := (. 3 17 .); $B := $A;  $B  !.:= 25;
```

The value of $B becomes list (. 3 17 25 .), but operation

!.:= has added element 25 immediately to the list (. 3 17 .), and the descriptor of this list is changed. As the value of the variable $A was a pointer to this descriptor, then after the execution of the statement $B !.:= 25 the value of the variable $A is changed, too.

To prevent this, we must have a copy of the object, to which $A refers to before assigning it to $B. Built-in rule #COPY(OB) is used for this purpose. It makes a copy of the descriptor of the object OB.

Now we can write a "safe" call of the operation !.:= in such a way: $A := (. 3 17 .); $B := #COPY($A); $B !.:= 25;

As a result $B takes the value (. 3 17 25 .), but $A remains equal to (. 3 17 .). The same effect can be obtained after execution of statements $A := (. 3 17 .); $B := $A !. 25;

6. Rules

The concept of rule is analogous to concepts of procedure and function in conventional languages, such as Pascal or C.

First of all, by the help of a rule we can check, whether an object or a sequence of objects complies with some grammar. For this purpose rule has a pattern. Objects, assigned under rule call (*arguments of the rule*), are matched with the pattern. If there is a necessity, some operations could be executed, for instance, computations and input-output operations, simultaneously with rule arguments and pattern matching.

Rule that is called can compute and return back some value (object),i.e., it can be used as function.

Depending on the result of rule arguments and pattern matching, the rule call ends with success or failure. Thus the rule call could be used in another rule patterns.

6.1. Simple Patterns

Definition of the rule begins with the indication of the name

of the rule in the form of #LLL , where LLL is an atom-identifier. In the most common case the pattern is indicated after the name of the rule. Rule definition ends with symbol '##'. For instance,

#L1 A B ##

In this case the pattern consists of atoms A and B. Call of the rule #L1 takes place, for instance, the following way :

#L1 (A B)

The sequence of objects - atoms A and B, is assigned as rule arguments.

After rule call the argument and pattern matching begins. The first argument - atom A is matched with the first pattern, which is also an atom. These atoms are equal, so their matching is successful. After that the next object from the sequence of arguments - atom B and the next pattern, also atom B, are matched. Their matching is also successful, therefore, the call of the rule #L1 is successful, too.

#L1(A C) call fails, because the second argument - atom C was not successfully matched with the pattern - atom B.

#L1(A) call fails, as there is no object for the second pattern with which it could be matched successfully.

But #L1(A B C) call is successful, as the third rule argument - atom C was not demanded by any pattern and matching of the first two rule arguments was successful.

Arbitrary atoms, both non-numerical and numerical, may be described as patterns.

Such operations as assignment statements, input - output and others may be indicated before the pattern, after it and between patterns. These statements are quoted in the pair of symbols '/' . If there is a necessity to write several statements within the pair '/' , these statements are separated by the symbol ';' .

A group of statements is executed in the rule pattern, if matching of the previous pattern with the corresponding rule arguments was successful.

The value returned by the rule is worked out by statement *return* . It has the following form: *return expression.*

Simultaneously this statement completes the execution of the rule.

Example.

```
#L2  'begin'  'end'  / return  'The pair begin-end' /  ##
```

Rule call is illustrated in the following example.

```
 $A  :=  #L2  (  'begin'    'end'  );
```

As a result the atom 'The pair begin-end' is assigned to the variable $A.

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

If the rule call ends in failure, then usually value NULL is returned, although in case of failure, it is possible to work out the returned value, which is not NULL; for this purpose statement *return* must be used in *onfail*-operations (see sect.6.3.).

Variable could be used as pattern. It is matched with one object (atom, list or tree) from the sequence of rule arguments. Matching always ends in success, and as a side effect the variable obtains this object as value. For example,

```
#L3  $A  $B  / return  (. $B  $A .)/  ##
```

After the call $X := #L3 (1 2) the variable $X obtains as value (. 2 1 .)

The rule pattern, in its turn, may refer to some rule (recursively, as well). Then the subsequence of the calling rule arguments become arguments of rule - pattern. If the call of the rule - pattern is successful, then matching of further patterns of the calling rule with the remaining arguments proceeds. If the call of the rule - pattern fails, then the pattern matching of the calling rule fails, too. For example,

```
#L4   A  #L5  D  ##
#L5   B  C       ##
```

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

There is a number of built-in rules in the language (see section 11.). For instance, #NUMBER is successfully matched with a numerical atom and returns it as a value, other arguments fail and return NULL. #IDENT is successfully matched with atom - identifier and returns it as value.

6.2. Assignment in Patterns

Every pattern, when successfully matched with the corresponding rule argument, returns some value. The value of atom pattern coincides with this atom, the value of variable pattern coincides with the value obtained by this variable as a result of matching with the arguments. The value of rule pattern is defined by statement *return* in this rule.

If the matching ends in failure, the pattern usually returns value NULL.

These values, returned by patterns, can be assigned at once to some variable. It is enough to write the name of the variable and the assignment symbol ':=' before the pattern element.
Example.

```
#L6   $A  $R := #L7  / return  (. $A .) !! $R /  ##
#L7   $B  $C   / return  (.  $B  $C .) /   ##
```

After execution of the statement $X := #L6 (1 2 3) the value of $X will be (. 1 2 3 .)

Symbols of accumulative assignment '!.:=' , '!!:=' , '++:=' and '+:=' can be used instead of ':=' in patterns .

Therefore, we can rewrite the previous example the following way:

```
#L6_1  $R !.:= $A    $R !!:= #L7_1  / return  $R /  ##
#L7_1  $M !.:=  $B   $M !.:=  $C    / return  $M /  ##
```

It should be noted that all variables in the rule are initialized by value NULL, so the value of the expression NULL !. $A that equals to (. $A .) is assigned to the variable $R by the first application of the pattern $R !.:= $A in #L6_1.

Patterns of the type $N := #NUMBER or $ID := #IDENT are used very often, therefore, following defaults are introduced in the language. If the first letter of the variable name is N, then this variable, having been used as pattern element, will have a successful matching only with a numerical atom, in other cases matching ends in failure, and variable obtains value NULL. If the first letter of the name of the variable is I, this variable is

matched successfully only with an atom-identifier.

6.3 Rule Branches. Onfail Operations

Several groups of patterns may be united in one rule.

The first group of patterns is applied to the rule arguments first. If matching of this group of patterns with the arguments is successful, the rule call is successful. But if matching has failed, transition to the next group of patterns takes place, and it is matched with the same arguments. It goes on until some group of rule patterns is matched with the arguments successfully. If not a single pattern group matches successfully with rule arguments, the rule call ends in a failure.

Such alternative pattern groups are called *rule branches*, and, when writing the rule, they are separated by the symbol ';;'.

If the branch fails, the execution of its patterns and statements is abandoned at the place, where branch pattern failed, and control is transferred to the next branch (if there is one) or the whole rule fails (if the current branch is the last one in the rule). Still there is a possibility to execute some operations before exit from the branch.

Onfail operation is a sequence of statements, written at the end of the branch and delimited from patterns and branch statements by keyword *onfail*.

If *onfail*-statements are described in the branch, then 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 can be analyzed and message can be output. Statement *return* can be executed in *onfail* operations, as well. Then exit from the rule takes place (with failure), and some other value than NULL can be returned.

6.4 Special Variables

Special variable without name $ denotes the first rule

argument matched by a rule pattern.

The value of special variable $$ equals to the value of current rule argument, to which current rule pattern is applied.

7. Compound Patterns

Lists, sequences of elements in lists and trees can be analyzed by patterns. Nesting of patterns practically is unlimited. It is allowed to insert statements before, after and within any pattern.

7.1 List Pattern

List pattern is written in the rule the following way:

(. S1 S2 ... SN .)

where S1, S2, ..., SN are patterns . For instance,

#L8 (. $E1 $E2 .) ##

Pattern of the rule #L8 is matched successfully with any list, containing precisely two elements. Such call is successful:

#L8((. (. 1 2 .) <. A : B .> .))

But the following calls end in failure.

#L8(A B) - because pattern will be applied to the first argument, i.e., to atom A.

#L8((. 13 .)) - because the argument is one element list.

In case of success the list pattern yields value that can be assigned to some variable. This value coincides with the whole list, to which the pattern was applied.

7.2 Iterative Pattern of Sequence

In RIGAL the following pattern is defined for sequence recognition: (* S1 S2 ... SN*) , where S1 , S2, ... SN are some patterns. This pattern describes the repetition of enclosed sequence of patterns zero or several times.

Rules with a variable number of arguments can be defined by

iterative pattern of sequence. For example,

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

This rule is used for summing up any amount of numbers.

```
#Sum( 2  5  11) = 18  and  #Sum( 3 ) = 3
```

Iterative pattern is very often used within list pattern. For instance, the following rule counts the number of list elements.

```
#Length (. / $L := 0 / (*  $E / $L +:= 1 / *) .) / return $L / ##
```

Samples of rule call.

```
#Length( (. A  B  C .) ) = 3  and  #Length( NULL ) = 0
```

Iterative pattern (+ S1 S2 ... SN +) is analogous to the pattern (* ... *), but assigns the repetition of enclosed pattern sequence one or several times.

In the iterative pattern of sequence the element delimiter is indicated in the form of

(* S1 S2 ... SN * *Delimiter*) or (+ S1 S2 ... SN + *Delimiter* +)

Atom or name of the rule may serve 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 .>
```

It should be noted that the pattern (* $E * ',') differs from the pattern (* $E ',' *) in the point that the presence of atom ',' is obligatory in the second pattern at the end of sequence.

7.3 Patterns for Alternative and Option

The choice of several possible patterns is written the following way: (S1 ! S2 ! ... ! SN)

Patterns S1 , S2 , ... , SN are applied one by one from left to right, until one of them succeeds.

In case of success alternative pattern yields value. It coincides with the value of the successful pattern within the alternative and may be assigned by some variable.

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

The call #Expression(X '-' Y '+' 7) returns the value

<. op: '+', arg1: <. op: '-', arg1: X, arg2: Y .>, arg2: 7 .>

In RIGAL we may write a rule that matches successfully with an empty sequence of arguments: #empty ##

Now the pattern for option can be written down: (S ! #empty)

In short form this issue may be written down in RIGAL the following way: [S]

where S is some pattern or pattern sequence. Pattern [S] always ends in success.

7.4 Tree Pattern

Tree pattern checks, whether the object is a tree with fixed structure. By means of this pattern access to the components of the tree is obtained. The tree pattern is described the following way:

<. Sel1 : Pat1, Sel2 : Pat2, ... , SelN : PatN .>

where Sel1, Sel2, ... SelN are atoms-identifiers, but Pat1, Pat2, ... PatN are patterns.

If the object, to which the tree pattern is applied, is not a tree, then the application of the pattern fails at once. If there is the selector Sel1 in the tree, then the pattern Pat1 is applied to the corresponding object. If there is no selector Sel1 in the

tree or the application of Pat1 has failed, the whole pattern also fails.

If matching the first branch was successful, branch matching of the pattern 'Sel2 : Pat2' , etc. begins.

Hence, pattern branches are applied to the tree in the same order as they are written in the pattern. Therefore, the order of tree traversing may be controlled. It is possible to have reiterative visit of branches (if selectors are repeatedly described in the tree pattern) or omission of branches (if corresponding selectors are not given in the pattern).

In case of success the tree pattern returns the value, which coincides with the whole object - a tree, to which the pattern was applied, irrespective of presence of all tree selectors in the pattern or absence of some.

Example. Let us suppose expression tree to be formed like in the above 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 .) /  ##
```

The call #Postfix_form(<. op: '-', arg1: X, arg2: <. op: '+', arg1: Y, arg2: 5 .> .>) returns the value (. X Y 5 '+' '-' .)

Some branches in the tree pattern may be described as optional, in this case they are enclosed in brackets '[' and ']' . If there is no selector of optional branch in the argument tree, its pattern is omitted and transition to next pattern branch takes place. If there is a selector of the type in the argument tree, the pattern branch is developed as usual.

7.5 Iterative Tree Pattern

The simplest form of iterative tree pattern is the following:

```
<*  $Var : P  *>
```

where $Var is some variable, and P is a pattern.

A loop over the tree is performed by the help of this pattern. All selectors of the argument tree are assigned to the variable $Var one by one. The pattern P is applied to each object, which corresponds in the argument tree to the current selector in the variable $Var. If even one application of the pattern P fails, the whole iterative tree pattern fails. For example,

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

Call example.

```
#Variable_table( <. X : integer, Y : real, Z : real .> ) =
     (. (. X  integer .) (. Y  real .) (. Z  real .)  .)
```

Sometimes, performing a loop over the tree, some branches should be updated in a special way. For this purpose iterative tree pattern with distinguished branches is used.

<* Sel1 : Pat1, Sel2 : Pat2, ..., SelN : PatN, $Var : P *>

where Sel1, Sel2, ... SelN are atoms-identifiers; Pat1, Pat2, ... PatN are patterns, i.e., like elements in simple tree pattern; $Var is a variable, but P is a pattern, as in simple case of iterative tree pattern.

The pattern is applied to the argument tree the following way. First of all distinguished pattern branches 'Sel : Pat' are developed. Their matching with branches of the argument tree happens exactly the same way as with simple tree pattern. Then the element '$Var : P' is applied to other branches of the argument tree the same way as in simple iterative tree pattern.

Some distinguished branches can be optional, for this purpose they are enclosed in brackets '[' and ']'. Semantics is the same as in the case of simple tree pattern.

Example. Let it be a tree of arbitrary size. In some of its subtrees there is the selector LABEL, to which numerical atom is attached. All these numbers over the whole tree must be collected in a list, and the list must be returned as result.

```
#Label_list
 <*  [ LABEL : $Result !.:= $N ],
        $S    : $Result !!:= #Label_list  *> / return  $Result / ;;
```

```
$E              ##
```

The rule has two branches. Traversing of the tree and its subtrees is described in the first branch. The resulting list is formed in the variable $Result. The traversing of subtrees is carried out by the help of recursive call of the rule #Label_list. The second branch of the rule consisting of just one pattern $E is applied at the leaves of the tree. The pattern matches successfully with any object and the whole branch returns value NULL, which is accepted as empty list at the previous level of recursion.

Call sample. #Label_list (<. A: <. LABEL: 5, B: abc .>, LABEL: 17, D: 25 .>) = (. 17 5 .)

7.6 Names of Lists and Trees in Patterns

The assignment of names was discussed in Section 5.

In list and tree patterns we can have matching of list and tree names with the described values or simply get these names in the described variable. For this purpose the name may be indicated before list or tree pattern: atom :: pattern

If the atom described in the pattern coincides with the name of the list (or tree), to which the pattern is applied, the application of the pattern to the argument begins. If it does not, other pattern elements are not applied and the pattern fails.

To obtain access to the name of the argument, instead of atom we must indicate the variable, in which the name is assigned as value.

7.7 Patterns of Logical Condition Testing

Pattern of the type: S' (*expression*)
works the following way. First of all the expression is evaluated. If its value differs from NULL, the pattern is successful. The value of the matched argument is returned as the value of the pattern, if matching was successful. If the value of the

expression equals to NULL, the pattern fails and returns NULL.

The value of special variable $$ in the expression of S-pattern equals to the value of the argument, to which S-pattern is applied.

The skip of token sequence until the nearest symbol ';' is described by the pattern: (* S' ($$ <> ';') *)

Let under parsing a case is accentuated when the assignment statement is in the form of X := X + E , where X is a variable, and E is an expression. This case could be described by pattern of the type: $Id ':=' S' ($$ = $Id) '+' #Expression

Pattern of the type: V' (*expression*)
works similar to S-pattern, yet in case of success no advancing along the sequence of rule arguments takes place, so the next pattern element is applied to the same argument.

This pattern is useful for context condition check.
Example. The pattern S' (#NUMBER($$) *and* ($$ > 7)) may be substituted by a sequence of patterns $Num V'($Num > 7)

8. Statements

8.1 Assignment Statement

In the left side of assignment statement a variable may be indicated, which is followed by an arbitrary number of list indexes and/or tree selectors. For example,
$X := (. A B C .); $Y := <. D : E, F : G .>;

After assignment $X[2] := T the value of $X is (. A T C .) .

After assignment $Y.D :=17 the value of $Y is <.D :17, F:G .>

The execution of the statement $Y.A := T yields the run time error message. The necessary result is obtained the following way: $Y ++:= <. A : T .> . The branch is deleted by assigning an empty object to the corresponding selector: $Y.D := NULL;

8.2 Conditional Statement

Conditional statement has the following form:

if expression -> statements

Then branches may follow (it is not compulsory)

elsif expression -> statements

Conditional statement ends with keyword *fi*.

In conditional statement branches expressions are computed one by one, until a value different from NULL is obtained. Then the statements described in this branch are executed.

8.3 Fail Statement

Fail statement finishes the execution of the rule branch with failure.

Example. In order to repair errors in parsing process, the sequence of tokens should be skipped quite frequently, for instance, until semicolon symbol. It is done the following way.

```
#statement    ...              ;; -- branches for statement analysis
    (*  #Not_semicolon   *)   ';'   -- no statement is recognised
##
#Not_semicolon   $E / if  $E = ';' -> fail  fi/   ##
```

8.4 Loop Statements

Statement of the type

forall $VAR *in* expression *do* statements *od*

loops over a list or a tree.

The value of the expression must be either a list or a tree. Value of the current list element (if the loop is over the list) or value of the current selector (if the loop is over the tree) is assigned to the loop variable $VAR one by one. Statements, describing body of the loop, may use the current value of the variable $VAR.

Loop statement of the type

loop statements *end*;

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

8.5 Rule Call

If a rule is called just to execute statements described in it, and value returned by the rule is not necessary, the rule call is written down as statement. It is analogous to procedure call in traditional programming languages. Success/failure of the rule and value returned by it is disregarded in such a call.

9. Input and Output

9.1 Save and Load Statements

Objects created by RIGAL program (atoms, lists, trees) can be saved in the file and loaded back to the memory.

Statement *save* $Var file-specification
unloads the object, which is the value of the variable $Var to the file with the given specification. File, formed by *save* statement, contains precisely one object (atom, list or tree).

We can load the object from the file in the memory having executed statement: *load* $Var file-specification

9.2 Text Output

To output texts (messages, generated object codes, etc.) several text files can be opened in the RIGAL program. The text file FFF is opened by statement: *open* FFF file-specification
File-specification may be an expression. It presents the name of the file on the device.

Statement of the type

FFF << Expr1 Expr2 ... ExprN

outputs a sequence of atoms to the file FFF. Values of expressions Expr1, Expr2, ... are either atoms or lists consisting of objects, different from trees.

This statement outputs atoms as sequences of symbols to the text file, inserting a blank after every atom. Atoms in the list are output in the same order as they are in the list.

Example. FFF << A B 12 ;

A string of characters is output in the text file FFF the following way: "A B 12 "

Symbol @ in the sequence of expressions of output statement switches over to other output mode of blanks separating atoms. By default at the beginning of execution of text output statement the output mode with blanks is on.

Example. FFF << A B @ C D 25 @ E F 57 ;

The following string of characters is output to the text file

"A B CD25E F 57"

Statement of the type FFF << ... always begins output with the beginning of a new record. Output statement of the type FFF <] ... continues output in the current record.

By the help of the statement of the type

Print expression

the value of atom, list or tree can be output in a readable form on the display, disc or on the printer. It is useful when debugging RIGAL programs.

10. Program Structure

Program written in the RIGAL language consists of the main program and rules. The main program text must be at the beginning of the RIGAL program text, but the text of the rules is written afterwards. Main program, as well as rule, begins by the name indication in the form of #main_program_name, then statements are described, separated by symbols ';' . The end of the main program is marked by the symbol '##'.

Usually operations that deal with the initial object loading, rule call and unloading of created objects in the files are concentrated in the main program. Therefore, main program has no pattern elements or arguments of its own.

When RIGAL is used for parsing, text file with the input information first is updated by scanner, which transforms it into a list of tokens. Thus rules describing required parsing can be

applied to this list. Intermediate results of the RIGAL program, for instance, abstract syntax trees obtained by parsing can be unloaded in the file by the help of *save* statement. They are unloaded so, that other RIGAL programs (for instance, those, which implement phase of code generation in compilers) can load them as input data. Sample of main program is given in Section 12.4.

Rules can be written down after main program in any order.

10.1 Local Variables

There are no special variable declarations in RIGAL. The fact that a variable is used in some statement, pattern or expression implies that the variable is defined as local variable of the rule or the main program.

For recursive rule calls local rule variables are pushed in a stack, so, that every active rule instance has its own set of local variables.

All variables are initialized by the NULL value, when the corresponding rule or the main program is called and when the execution of rule branch starts.

10.2 References to Variables of Other Rules

The construction of RIGAL makes it possible to obtain access from the rule to local variables of another rule. It has the following form: *last* #L $X

This reference denotes the value of the variable $X in the last (in time) and still active instance of the rule #L. Such references can be used in left and right sides of assignment statement.

If at the moment of evaluation of the expression *last* #L $X there is no active instance of the rule #L, the value of the expression *last* #L $X equals NULL. If such *last* #L $X is in the left side of assignment statement, the statement is disregarded (and run time error message is output). By the help of *last* we

may refer to both rule and main program variables.

10.3 Attribute Grammars and RIGAL Global Attributes

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. The usage of *last* references solves this problem in RIGAL.

Implementation of both synthesized and inherited attributes is possible as it is demonstrated in the following scheme.

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

11. Built-in Rules

There is a number of built-in rules in the language. These rules implement functions, the implementation of which is impossible or ineffective by other language means.

Call of built-in rules is written down the same way as call of rules defined by the user itself. Along with the value a

built-in rule yields success or failure hence, built-in rules are used as patterns.

There are predicates such as #ATOM(E), #NUMBER(E), #IDENT(E), #LIST(E) and #TREE(E).

The built-in rule #LEN(E) returns numerical atom as value. If E is an atom, then for non-numerical atoms it returns the number of atom symbols. For numerical atoms this rule returns the number of significant digits plus 1, if the atom is a negative number. #LEN(NULL) equals 0, #LEN('ABC') equals 3, #LEN(-185) equals 4.

If E is a list, then #LEN(E) returns the number of list elements, but, if E is a tree, then it returns the number of tree branches.

#EXPLODE(E). If E is an atom, then it succeeds and returns one character atom list that represents the value E 'decomposed' in separate characters. If E is a numerical atom, only significant digits are present.

Examples. #EXPLODE(X25) yields (. 'X' '2' '5' .).
#EXPLODE(-34) yields (. '-' '3' '4' .).

#IMPLODE(E1 E2 ... EN). This rule yields the concatenation of atoms or lists E1, E2, ..., EN in a new, non-numerical atom.

Examples. #IMPLODE(A B 34) equals 'AB34'.
#IMPLODE(25 (. A -3 .)) equals '25A-3'.

#CHR(N). The rule returns an atom, which consists of just one ASCII character with the code N (0 <= N <= 127).

#ORD(A). Returns an integer, which is an internal code of the first character of the nonnumerical atom A.

For instance, #ORD(A) = 65, #ORD(ABC) = 65.

#PARM(T) . Returns list of parameters which was assigned when the whole program called for execution.

#DEBUG(E). If E equals the atom 'RULES', then, as soon as the rule is called, information concerning calls of the rules (both user defined and built-in rules) and their execution results will be output. The call #DEBUG(NORULES) stops the debugging of the rules.

12. Sample Compiler

Compiler for the TOYLAN language, which is a very simple programming language, is discussed in the following units. Compiler works in two passes. The first phase is parsing and construction of the program's intermediate form as abstract syntax tree. The second phase is code generation.

Description of input and intermediate language grammars by means of RIGAL is presented. Thus formalized compiler documentation (admitting checking on a computer) is obtained.

12.1 TOYLAN Language

The description of TOYLAN syntax can be regarded as description of acceptable sequences of tokens. Atoms represent lexical elements of TOYLAN program: keywords, identifiers, constants, operation signs, delimiters. The context free grammar of TOYLAN can be described in the form of RIGAL program.

```
#PROGRAM       'PROGRAM' $Id
               (* #DECLARATION ';' *)  (+ #STATEMENT + ';' )  ##
#DECLARATION   ('INTEGER'!'BOOLEAN')    (+ $Id  + ',')        ##
#STATEMENT     ( #ASSIGNMENT      !  #INPUT     !
                 #OUTPUT          !  #CONDITIONAL  )          ##
#ASSIGNMENT    $Id ':='  #EXPRESSION                          ##
#INPUT         'GET' '(' (+ $Id  + ',') ')'                   ##
#OUTPUT        'PUT' '(' (+ #EXPRESSION + ',') ')'            ##
#CONDITIONAL   'IF' #EXPRESSION    'THEN' (+ #STATEMENT + ';')
               ['ELSE' (+ #STATEMENT + ';' )]    'FI'         ##
#EXPRESSION    #SUM      [  '='  #SUM  ]                      ##
#SUM           #FACTOR   (* '+'  #FACTOR   *)                 ##
#FACTOR        #TERM     (* '*' #TERM      *)                 ##
#TERM          $N ;; -- numeric constant
               ('TRUE' ! 'FALSE' ) ;; -- Boolean constants
               $Id;; -- variable
               '(' #EXPRESSION ')'                            ##
```

Context conditions are the following:

1) all variables used in statements and expressions must be declared,

2) one and the same variable name should not be declared twice,

3) left and right parts of assignment statement must be of the same type,

4) operands of input-output statements must be of the type INTEGER.

All variables of the type INTEGER have initial value 0, and all variables of the type BOOLEAN have initial value FALSE.

12.2 Intermediate Form of Program. Abstract Syntax Tree

Special languages to represent intermediate results of compilation are used in compiler building practice. For instance, P-code for PASCAL compilers and language DIANA [8] for ADA.

The result of the first phase of the TOYLAN compiler is a tree. Let's call it abstract syntax tree. Along program components it contains some semantic attributes, for instance, types of expressions. One of the most significant attributes is table of variables, obtained as a result of parsing of the TOYLAN program declarations.

The structure of abstract syntax tree of the TOYLAN program is described by the following rules.

```
#S_PROGRAM
     'PROGRAM'::<. NAME : $Id,
                   DECLARATIONS : #S_DECLARATIONS ,
                   STATEMENTS   : (.(* #S_STATEMENT *).) .>     ##
#S_DECLARATIONS      -- variables table
     <* $Id : ( INTEGER ! BOOLEAN )  *>                          ##
#S_STATEMENT
     ASSIGNMENT :: <. LEFT  : $Id,
                      RIGHT : #S_EXPRESSION    .>    ;;
```

```
INPUT :: (. (* $Id *) .)    ;;
OUTPUT :: (. (* #S_EXPRESSION *) .)      ;;
CONDITIONAL :: <. COND : #S_EXPRESSION,
                  THEN : (.(* #S_STATEMENT *).),
                  [ ELSE : (.(* #S_STATEMENT *).)] .>   ##
#S_EXPRESSION
COMPARE :: <. ARG1 : #S_EXPRESSION, ARG2 : #S_EXPRESSION,
              TYPE : BOOLEAN    .> ;;
ADD :: <. ARG1 : #S_EXPRESSION,  ARG2 : #S_EXPRESSION,
          TYPE : INTEGER       .> ;;
MULT :: <. ARG1 : #S_EXPRESSION,  ARG2 : #S_EXPRESSION,
           TYPE : INTEGER       .> ;;
<. VARIABLE : $Id ,  TYPE     : ( INTEGER ! BOOLEAN ) .>   ;;
<. CONSTANT : $N ,   TYPE     : INTEGER .>   ;;
<. CONSTANT : ( 0 ! 1) , TYPE : BOOLEAN .>             ##
```

12.3 Target Language BAL

The goal of the TOYLAN compiler is to obtain program text in a low level language BAL. This language is a simplified model of assembler languages. The memory of BAL-machine is divided into separate words. Every word may contain an integer, besides, there are work registers R0, R1, R2, ... of the word size each. Let us suppose the number of registers to be unlimited, in order to eliminate the problem of optimal register usage during generation phase.

Command of BAL ABC: DEFWORD N
reserves a word in the memory and imbeds integer N in it. We can refer to this word in other commands by name ABC.

Commands of BAL have two operands which are described by the name of memory word, by the name of register or by the literal of the type =NNN, where NNN is an integer. Commands can be marked by labels.

1) MOV A1,A2 This command moves memory word A1 to memory word A2.
2) LOAD RI,A Loading of word A into register RI.

3) SAVE RI,A Unloading of the contents of register RI into memory word A.

4) ADD RI,A or ADD RI,RJ The sum of operands is imbedded in RI.

5) MULT RI,A or MULT RI,RJ Multiplication of operands is imbedded in RI.

6) COMPARE RI,A or COMPARE RI,RJ If operand values are equal, it is 1 that is imbedded in RI, if they are not equal, 0 is imbedded.

7) BRANCH RI,M If the value of RI is equal to 0, then transfer to the command marked by label M takes place, otherwise, to the next command.

8) JUMP M Unconditional transfer to the command marked by label M.

9) EOJ Completes the execution of the BAL program.

10) READ A Reads the integer from standard input device and imbeds it in word A.

11) WRITE A or WRITE RI Outputs the integer from memory word or from register to standard output device.

12) NOP An empty statement.

12.4 Main Module of Compiler

The main program of the TOYLAN compiler contains calls of the first and second compilation phases and file opening statements.

```
#TOYLAN_COMPILER
    open REP 'TI:';  --message file is connected with the screen
    load $LEXEMS 'A.S'; -- a list of tokens is loaded from the
                        -- file A.S, where it was imbedded by scanner
    $S_TREE := #A_PROGRAM($LEXEMS);
    -- 1st  phase; result of parsing - abstract syntax tree - is
    -- imbedded in the variable $S_TREE; during parsing messages
    -- about discovered errors in file REP can be output.
    if $S_TREE  ->  open GEN 'A.BAL'; -- if the tree is created,
         -- then file is opened to output the generated BAL text
              #G_PROGRAM($S_TREE) -- 2nd phase - code generation
    elsif  T  ->  REP << errors are discovered   fi;
```

```
REP << end of compilation                    ##
```

The compilation listing that contains source text and error messages is not envisaged in the TOYLAN compiler. Formation of the listing can be a separate phase.

12.5 Parsing Phase

The rule #A_PROGRAM carries out parsing of tokens list, checks context conditions, generates error messages and builds abstract syntax tree of TOYLAN program.

Patterns of the rule #A_PROGRAM and of the rules subordinate to it, actually, coincide with patterns of the rule #PROGRAM and with the associated rules that describe context free grammar of the TOYLAN language. Just operations to check context conditions, to output error messages and to construct abstract syntax tree are added.

In our parser diagnostics is based on the following principles. First of all, "panic" reaction to an error should be avoided and several messages concerning one and the same error should not be output (though, we can't manage it always), secondly, error neutralization is transition to the analysis of the next statement, i.e., skip of tokens until the nearest symbol ';'.

```
#A_PROGRAM     -- the rule is applied to the list of tokens
     (. PROGRAM  $Id
          (*  $DECL++:= #A_DECLARATION  ';'  *)
                  --formation of variables table
          (+  $STATEMENTS !.:= #A_STATEMENT   + ';' )
                  --formation of statements list
     .)  / return 'PROGRAM' :: <. NAME : $Id,
                                  DECLARATIONS : $DECL ,
                                  STATEMENTS : $STATEMENTS .>/ ##
#A_DECLARATION      $TYPE := ( INTEGER ! BOOLEAN )
   (+ $Id /if last #A_PROGRAM $DECL.$Id or $REZ.$Id ->
              REP << VARIABLE $Id DOUBLE DEFINED  fi;
```

```
          $REZ++:= <.$Id : $TYPE .>/  + ',' ) / return $REZ / ##
#A_STATEMENT   $REZ := ( #A_ASSIGNMENT ! #A_INPUT !
                #A_OUTPUT ! #A_CONDITIONAL )   / return $REZ / ;;
     (* $A!.:=S'($$ <> ';' ) *)     -- skip until nearest ';'
              / REP << UNRECOGNIZED STATEMENT $A /          ##
#A_ASSIGNMENT  $Id  ':=' / $LPType := last #A_PROGRAM $DECL .$Id;
        if not $LPType  -> REP << VARIABLE $Id IS NOT DEFINED fi /
    $E:= #A_EXPRESSION
         /if $LPType <> $E . TYPE ->
              REP<< LEFT AND RIGHT SIDE TYPES ARE DIFFERENT
                   IN ASSIGNMENT STATEMENT fi;
          return ASSIGNMENT::<. LEFT: $Id, RIGHT: $E .> /
 onfail if $LPType  -> REP<< WRONG EXPRESSION IN ASSIGNMENT  fi  ##
#A_INPUT       GET  '('
     (+ $E !.:= $Id  /if  last #A_PROGRAM $DECL.$Id <> INTEGER ->
           REP << $Id IN STATEMENT GET IS NOT OF THE TYPE INTEGER
                  fi / + ',' ) ')' / return INPUT :: $E /   ##
#A_OUTPUT   PUT  '(' (+  $C := #A_EXPRESSION  / $E !.:= $C;
              /if  $C . TYPE  <> INTEGER ->
        REP << OPERAND OF PUT STATEMENT IS NOT OF THE TYPE INTEGER
               fi /   + ',' ) ')'/ return OUTPUT :: $E /       ##
#A_CONDITIONAL      'IF' $BE := #A_EXPRESSION
     /if $BE . TYPE <> BOOLEAN ->
              REP<< CONDITION IS NOT OF BOOLEAN TYPE fi /
                   'THEN' (+ $P1 !.:= #A_STATEMENT + ';' )
          [ 'ELSE' (+ $P2 !.:= #A_STATEMENT + ';' ) ]       'FI'
    / return CONDITIONAL :: <. COND : $BE , THEN : $P1 ,
                              ELSE : $P2  .> /             ##
#A_EXPRESSION       $A := #A_SUM   [  '=' $B := #A_SUM
    / $A := COMPARE::<. ARG1 : $A, ARG2 : $B, TYPE : BOOLEAN.>/ ]
     / return $A /            ##
#A_SUM   $A := #A_FACTOR    (* '+' $B := #A_FACTOR
          / $A := ADD::<. ARG1: $A, ARG2: $B, TYPE: INTEGER .>/ *)
     / return $A  /           ##
#A_FACTOR  $A := #A_TERM     (* '*' $B := #A_TERM
```

```
        /$A := MULT::<. ARG1: $A, ARG2: $B, TYPE: INTEGER .>/ *)
    / return $A /                ##
#A_TERM
    $N   / return <. CONSTANT : $N , TYPE : INTEGER .>;;
  ( ( TRUE / $K :=1/ ) ! ( FALSE / $K :=0 / ) )
                /return <. CONSTANT: $K, TYPE: BOOLEAN .>/  ;;
    $Id  / $X:= last #A_PROGRAM $DECL.$Id;
        if not $X  ->  REP << VARIABLE $Id IS NOT DECLARED
        elsif T -> return <. VARIABLE: $Id, TYPE: $X .> fi / ;;
    '(' $E := #A_EXPRESSION ')' / return $E /          ##
```

12.6 Code Generation Phase

Code generation is performed when traversing abstract syntax tree.

To avoid possible conflicts between variable names in the TOYLAN program and register names (of the type RNNN) and labels (of the type LNNN) in the object program, variable names are substituted by standard names of the type VARNNN.

```
#G_PROGRAM     / $LABEL := 0 /   --global variable $LABEL serves
                                 --to generate unique labels.
  PROGRAM::<.DECLARATIONS: $TAB := #TABLE_OF_NUMBERS,
              --creation of the table of unique variable numbers
            STATEMENTS: (.(* #G_STATEMENT *).) / GEN << 'EOJ' /,
            DECLARATIONS : #G_DECLARATIONS       .>        ##
#TABLE_OF_NUMBERS <* $Id: $TYPE /$N :=$N+1; $T++:=<. $Id: $N.>/ *>
                  /return $T/     ##
#G_STATEMENT ( #G_ASSIGNMENT ! #G_INPUT !
               #G_OUTPUT     ! #G_CONDITIONAL )       ##
#G_ASSIGNMENT    ASSIGNMENT::<. LEFT: $Id := $NAME,
  RIGHT :( ( <. VARIABLE: $Id1:=#NAME .>
              /GEN << MOV @ $Id1 ',' $Id / ) !
          ( <. CONSTANT : $N .> /GEN << MOV @ '=' $N ',' $Id /) !
          ( $NREG := #G_EXPRESSION
              /GEN << 'SAVE' @ 'R' $NREG ',' $Id / ) )  .>    ##
```

```
#G_INPUT  INPUT::(. (* $Id := #NAME /GEN << READ $Id / *) .)   ##
#G_OUTPUT      OUTPUT :: (. (*
     ( ( <. VARIABLE : $Id := #NAME .> /GEN << WRITE $Id / ) !
       ( <. CONSTANT : $N .> /GEN << WRITE @ '=' $N / )  !
       ( $NREG := #G_EXPRESSION /GEN << WRITE @ 'R' $NREG /) )
                          *) .)                                ##
#G_CONDITIONAL      CONDITIONAL ::
      <. COND : $NREG := #G_EXPRESSION
            / $LABEL1 :=#NEW_LABEL(); $LABEL2 :=#NEW_LABEL() /,
         THEN   : / GEN << BRANCH @ 'R' $NREG ',L' $LABEL1 ) /
                    (. (* #G_STATEMENT *) .)
                / if $.ELSE -> GEN << JUMP @ 'L' $LABEL2 fi;
                               GEN << @ 'L' $LABEL1 ': NOP' / ,
       [ ELSE : (. (* #G_STATEMENT *) .)
                / GEN << @ 'L' $LABEL2 ': NOP' / ]      .>   ##
#G_EXPRESSION  --returns the number of the register containing
                --result of the evaluation of expression
     / $NREG := 0 / -- number of the first accessible register
     $REZ := #G_EXPR / return $REZ /    ##
#G_EXPR      ( <. VARIABLE: $ID :=#NAME .>  !
               <. CONSTANT: $N / $ID := #IMPLODE('=' $N)/ .>)
             / $REG := #COPY( last #G_EXPRESSION $NREG ) ;
               GEN << 'LOAD' @ 'R' $REG ',' $ID  ;
               last #G_EXPRESSION $NREG + := 1; return $REG /  ;;
          $OP::<. ARG1 : $R1 := #G_EXPR, ARG2 : $R2 := #G_EXPR .>
           / GEN << $OP @ 'R' $R1 ',R' $R2 ; return $R1 /    ##
#G_DECLARATIONS
<* $ID: $TYPE /$ID1 := #NAME($ID); GEN<< $ID1 ':' DEFWORD 0 /*> ##
#NEW_LABEL      --auxiliary rule
     /last #G_PROGRAM  $LABEL+:=1;
      return #COPY (last #G_PROGRAM $LABEL )/                 ##
#NAME      $ID  --returns standard name of the variable $ID in $TAB
     / return #IMPLODE( VAR  last #G_PROGRAM $TAB.$ID)/      ##
```

13. 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.

Our experience [13] proves that 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 [14], because it allows the designer to develop an experimental translator in short period of time.

Besides interpreter for debugging purposes and optimizing compiler RIGAL support system 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 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] A.Aho, J.Ullman. The theory of parsing, translation and compiling// Prentice-Hall, Inc. Englewood Cliffs,N.J. 1972. - vol.1,2.

[2] S.C.Johnson. YACC - Yet Another Compiler Compiler // Bell Laboratories, Murray Hill,N.J., 1978, A technical manual.

[3] C.H.Koster. Using the CDL Compiler Compiler// Lecture Notes in Computer Science, Vol.21, Springer-Verlag, Berlin, 1977.

[4] I.R.Agamirzyan. Compiler Design Technological Support System SHAG. // Space mechanics algorithms, Leningrad, vol. 79, 1985, pp.1-53., (in Russian).

[5] D.E.Knuth. Semantics of context-free languages// Mathematical

Systems Theory, 2, 2, 1968, pp.127-146.

[6] V.A.Serebryakov. Methods of Attribute Translation.// In: Programming Languages, Moscow, "Nauka", 1985, pp.47-79, (in Russian).

[7] A.O.Vooglaid, M.V.Lepp, D.B.Lijb. Input Languages of the ELMA System.// Proceedings of the Tallinn Polytechnical Institute, #524, 1982, pp.79-96, (in Russian).

[8] The intermediate language DIANA : Design and Implementation// Lecture Notes in Computer Science, Vol.180, Springer-Verlag Berlin, 1984.

[9] R.Vilhelm. Presentation of the compiler generation system MUG2 : Examples, global flow analysis and optimization// Le point sur la compilation, INRIA, 1978, pp.307-336.

[10] Basic REFAL and its implementation on computers.// CNIPIASS, Moscow, 1977, (in Russian).

[11] P.Lucas. Formal definition of programming languages and systems // IFIP Congress , 1971.

[12] M.Ganapatti, C.N.Fisher, J.L.Hennessy. Retargetable compiler code generation// ACM Computing Survays, 14(4), 1982.

[13] J.Barzdin, A.Kalnins, M.Auguston, 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.

[14] R.Herndon, V.Berzins, The realizable benefits of a language prototyping language.// IEEE Transactions on Software Engineering , vol.14, No 6, June 1988, pp.803-809.