The Compiler Writing Language

RIGAL

Mikhail Auguston
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
email: mikau@cs.nmsu.edu

The documentation kit is available at
/usr/unsupported/solaris/rsys
start with README file.

RIGAL Home Page and distribution kit for MS DOS and UNIX
are maintained by Vadim Engelson at University of Linkoping,
Sweden:
RIGAL is a **simple** and **powerful** tool for

- syntactic analysis
- code optimization
- code generation
- static program analysis
- preprocessor and filter writing
- interpreter design
- language rapid prototyping
The Main Idea

“Write a **grammar** describing the structure of input data and attach actions within it”
Other principles of RIGAL are:

- the language has built-in means for **pattern matching** with formal **grammars**

- operations are executed **simultaneously** with pattern matching

- **attribute grammars** can be simulated easily

- RIGAL has a rich spectrum of **tree** manipulation means, for instance, RIGAL has **tree grammars**

- RIGAL supports **multi-pass** compiler design. Trees can be used as an intermediate data

- RIGAL encourages splitting of a program into small **modules** (rules) and presents various means to arrange interactions of these modules, e.g. a good solution for **global attribute** problem
Data Structures and Operations

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

Atoms

‘Hello’ ‘:=’ 257 T NULL
‘abc’ abc

Variables and Assignments

$E := ‘ABC’
$Count := $Count + 1 or
$Count +:= 1
$Cond := ( $A = 7) AND ( $B > 0)
Lists

A list is an ordered sequence of objects which may be atoms, other lists or trees.

The list constructor (. . . . ) yields a list of objects.

$E := (. A B C .)

It is possible to get elements from a list by indexing

$E[ 2]$ is atom B
$E[ -1]$ is atom C
$E[ 25]$ is atom NULL

Operations on lists

(. 1 2 3 .) !. 4 is (. 1 2 3 4 .)
$E !! (. a b .) !! (. c .) is

(. A B C a b c .)

but 2 !. 3 is NULL

$X := $X !! list or $X !! := list
$X := $X !. elt or $X !. := elt
Trees

For tree creation the tree constructor `< ... >` is used

\[ E := <. A : B, C : D .> \]

Objects placed just before ‘:’ are called selectors.
Selectors of the same level must be different.
The pair ‘selector : object’ is a branch of the tree.
The tree is an unordered set of branches.

\[ <. A: B, C: D .> \text{ equals } <. C: D, A: B .> \]
Operations on Trees

Let 

\[ E := \langle. A : X, \]
\[ B : (. \alpha \beta .), \]
\[ C : \langle. A: 2 .> .> \]

Then 

\[ E.A \] is atom X
\[ E.C.A \] is atom 2
\[ E.B[2] \] is atom \( \beta \)
\[ E.D \] yields NULL

Tree concatenation

\[ \langle. A: 1, B: 2 .> ++ \langle. B: 3, C: 5 .> \]
yields

\[ \langle. A: 1, B: 3, C: 5 .> \]

\[ X := X ++ t \] or \[ X ++:= t \]
Rules. Simple Patterns

#Add -- adds two numbers and returns the sum

$X \quad $Y \quad / \quad \text{RETURN} \quad $X + $Y / \quad ##

This rule can be called as a function

$R := \#\text{Add} ( 3 \quad 5 )$

Atoms and rule names also can be used as patterns.

#L1 $R!.:= A \quad $R!!:= \#L2

/RETURN $R / \quad ##

#L2 $S!.:= B \quad $S!..:= C

/RETURN $S / \quad ##

#L1 ( A B C ) is successful and returns (. A B C .)
#L1 ( A B C D ) is successful and returns (. A B C .)
#L1 ( A D C ) fails and returns NULL
#L1 ( A B ) fails and returns NULL
Patterns

**List pattern** (. P1 P2 ... Pn .)

**Iterative sequence pattern** (* P1 P2 ... Pn *)
denotes the repetition of pattern sequence zero or more times.

**Example.** Length of list.

```plaintext
#Len /$L := 0/(. (* $E /$L +:= 1/ *).)

/ RETURN $L / ##
```

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

**Example.** Sum of arbitrary number of numbers.

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

/ RETURN $S / ##
```

#Sum(2 5 11) returns 18.

Hence, rules can have a variable number of arguments.
Examples.

#head (. $H (* $E *).) / RETURN $H / ##
#tail (. $H (* $Res !.:= $E *) .)

/RETURN $Res/ ##

More Patterns

Alternatives pattern

( P1 ! P2 ! P3 ! P4 )

Iterative Patterns with a delimiter

(+ P + delimiter )

(* P * delimiter )

Some Built-in Rules

$Id := #IDENT

$Num := #NUMBER
Example. Analysis of a simple Algol-like declarations. A fragment of the variable table, coded in a tree form, is returned as a result.

#Declaration

$Type := ( integer ! real )
(+ $Id
/ $Rez ++:= <. $Id : $Type .> /
+ `,`)

/ RETURN $Rez %

#Declaration ( real X `,` Y `,` Z )

returns the value <. X : real, Y : real, Z: real .>
Grammar for a simple Arithmetic Expression

#Expression
   #Additive_el
   (* ( ‘+’ ! ‘-’ ) #Additive_el *)
#

#Additive_el
   #Term
      (* ( ‘*’ ! ‘div’ ) #Term *)
#

#Term
   ( #IDENT ! #NUMBER ) ;;
   ‘(‘ #Expression ‘)’
#

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

```
#Expression
$A1 := #Additive_el
(* $Op := ( ‘+’ ! ‘-’ ) $A2 := #Additive_el
/ RETURN $A1 / ##

#Additive_el
$A1 := #Term
(* $Op := ( ‘*’ ! ‘div’ ) $A2 := #Term
/ RETURN $A1 / ##

#Term
$A := ( #IDENT ! #NUMBER ) / RETURN $A / ;;
‘( ‘ $A := #Expression ’)’ / RETURN $A / ##
```
#Expression( X \textasciitilde{} '\ast{}' \textasciitilde{} Y \textasciitilde{} '\textasciitilde{} '+' \textasciitilde{} 7 ) \text{ returns a value}

\texttt{<. \text{op:} \texttt{ '+'}, \text{arg1:} <. \text{op:} \texttt{ '*'}, \text{arg1:} \texttt{ X,}}
\texttt{arg2: \texttt{ Y .}>,}
\texttt{arg2: 7 .}>
Tree Patterns

**Tree pattern** can be written as

\[
<. \ a_1 : p_1, \ a_2 : p_2, \ldots, \ a_n : p_n .>
\]

where \( a_i \) are atoms and \( p_i \) 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**.
Example. The task is to traverse the expression tree and return a list that represents the Polish postfix form of this expression.

```
#Postfix_form
  <. arg1: $Rez := #Postfix_form,
      arg2: $Rez !!: = #Postfix_form,
      / RETURN $Rez / ;;

$Rez := ( #IDENT ! #NUMBER )
      / RETURN (. $Rez .) /

##

#Postfix_form( <. op: ‘-’, arg1: X,
                  arg2: <. op: ‘*’, arg1: Y,
                  arg2: 5 .>
                  .>)

returns value (. X Y 5 ‘*’ ‘-’ .)
Patterns of Logical Condition Testing

\[ S' \ ( \text{expression} ) \]

If the value of expression differs from NULL, the pattern is successful, otherwise the pattern fails.

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

\textbf{Example}. To skip the token sequence until the nearest symbol ';'
may be described by the pattern:

\[ (* \ S' \ ( \ $$ \ < > \ ';') \ *) \]

\textbf{Example}. Assignment statement of the form

\[ X := X + E \]

could be described by a pattern:

\[ $Id \ :=\ S' \ ( \ $$ = $Id \ ) \ '+' \ #Expression \]
Attribute Grammars and RIGAL

Global References

**Rules** in RIGAL correspond to **nonterminals** in AG

**Variables** in RIGAL correspond to **attributes** in AG

#LA ... ... ... assigns value to attribute $A1

$A2 := #LB ( . . . ) . A2 .... ....

-- after this call the value is assigned to the

-- synthesized attribute $A2

#

#LB ... ... ...

$B1 := LAST #LA $A1 -- global reference

-- uses inherited attribute $A1 from #LA

assigns value to attribute $B2 ... ... ...


##
Conditional Statement

IF expression -> statements

One or more optional ELSIF branches may follow

ELSIF expression -> statements

FI

Example.

IF $X > 100 -> $X +:= -1

ELSIF $T -> $X := $Y

FI
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/

##
```
Loop Statements

FORALL $VAR IN expression
DO statements OD

FORALL SELECTORS $VAR
BRANCHES $VAR1
IN expression
DO statements OD

FORALL BRANCHES $VAR1
IN expression
DO statements OD

loops over a list or a tree.

LOOP statements END;

repeats statements of the loop body, until one of the statements - BREAK, RETURN or FAIL is not executed.
Input and Output

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

SAVE $Var file-specification
LOAD $Var file-specification

Debugging Print

PRINT expression
Text Output

OPEN  FFF  file-specification

FFF  <<  Expr1  Expr2  ...  ExprN

Example.

OPEN  FFF  ‘my_directory/a.txt’ ;

FFF  <<  A  B  12 ;

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

“A  B  12 “

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”

FFF  <<  ...

always begins output at the beginning of a new line.

FFF  [ ]  ...  continues output at the current line.
Built-in Rules

Predicates: \#ATOM(E), \#NUMBER(E), \#IDENT(E), \#LIST(E)

and

\#TREE(E).

\#LEN(E) returns the number of atom symbols or the number of list elements or the number of tree branches.

Examples. \#LEN( abc) yields 3
\#LEN( ( . a b c d .) ) yields 4
\#LEN( <. a: b, c: d .> ) yields 2
#EXPLODE (E)
returns one character atom list that represents the value E ‘decomposed’ in separate characters.

**Examples.**

#EXPLODE(X25) yields (. ‘X’ ‘2’ ‘5’ .).
#EXPLODE(-34) yields (. ‘-’ ‘3’ ‘4’ .).

#IMPLODE (E1 E2 ... EN)
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.

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

( Model of two-pass compiler )

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

The input stream.

```plaintext
#telegram_stream

( + #telegram #end + ) [ #blanks ]

#end

#telegram(+ #word #blanks +) ##

#word (+ #letter +) ##

#blanks (+ " " +) ##

#end "*" "*" "*

#letter

( A ! B ! C ! ... ! Z !

a ! b ! ... ! 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_tlgr_stream

(+ #telegram1 #end +) #end ##
#telegram1

(+ #word ` ` +) $long_word_num ##
The main program:

```plaintext
#telegram_problem

LOAD $T ‘Letters.lex’;

$S :=

    #telegram_stream_analysis($T);

OPEN Out ‘LP:’;

#generate_output_stream($S) ##
```
-- First Pass: Parsing

#telegram_stream_analysis

(. (+ $R !.:= #A_telegram #end +)
   [ #blanks ] #end .) / RETURN $R /

## #A_telegram

/ $lng_word_num := 0/

(+ $R !.:= #A_word #blanks +)

/ RETURN <. text: $R,

  long_word_n:

    $lng_word_num .> / ##

#A_word

(+ $R !.:= #A_letter +)

/IF #Len($R) > 12 ->

LAST #A_telegram $lng_word_num + :=1

FI;

RETURN $R / ##

#A_letter

$R := ( A ! B ! C ! ... ! Z ! a ! b ! ...

  ! z) / RETURN $R / ##
### Second Pass: Output

```
#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.