University of Latvia
Institute of Mathematics and Computer Science

RIGAL PROGRAMMING SYSTEM FOR SUN / UNIX

PROGRAMMER'S GUIDE

by VADIM ENGELSON

TABLE OF CONTENTS

1.1 Introduction

RIGAL is a high level programming language designed for syntax analysis, program optimization, code generation, converter writing and rapid design of language prototypes. RIGAL is designed in the Institute of Mathematics and Computer Science of the University of Latvia in 1987 - 1990. It was implemented in RSX-11M, VAX/VMS, MS-DOS, MS-Windows, UNIX/SUN operating systems.

See the manual pages on how to install RIGAL in your system.

1.2 How to create a simple program

After the installation try the example.

Create file 'a.rig' with the following text:

```
-- THIS IS PROGRAM
#START
  $A:=(. ALPHA BETA GAMMA DELTA .);
  PRINT $A;
  $B:=#TAIL($A);
  PRINT $B;
##
#TAIL
  (. $E (* $R!. := $K *) .)
  / RETURN $R /
##
```

There are two rules in this program - #START and #TAIL. The first rule is the main one, it consists of four statements. There are two variables - $A and $B. The first statement assigns a list of four atoms to the variable $A, the third statement assigns to $B the result of rule #TAIL which works on the argument $A. The second and the fourth statements print out the values of the variable (to the standard output by default).

The second rule #TAIL consists of the list pattern (. .) , the iteration pattern (* *) and the variable patterns $E and $K. Values of $K are appended to the list in the variable $R. The RETURN statement returns the result of this rule to the calling

rule.

Execute command to check the file:

```
rc a
```

If the program has no errors then continue, else correct the errors reported and check the program again.

Now file 'a.rsc' appears in your directory.

Execute command to interpret the program

```
ic a
```

You will see the result on the sceen:

```
(. ALPHA BETA GAMMA DELTA .)
(. BETA GAMMA DELTA .)
```

Execute command to compile the program

```
rc a -c
```

Now file 'a' appears in your directory.

Execute command to startup the compiled program

```
./a
```

You see the same result:

```
(. ALPHA BETA GAMMA DELTA .)
(. BETA GAMMA DELTA .)
```

Now you can try other examples or write them yourself.

2. Additional features of the language

Some language limitations, extensions and special cases depending of the implementation are discussed.

2.1 RIGAL lexics

Tokens in RIGAL programs are the following :

-- rule names , e.g. #ABC
-- variable names , e.g. $VAR
-- RIGAL key words, e.g. FORALL, ELSIF
-- RIGAL delimiters and special tokens, e.g. ## ! +
-- atoms (RIGAL data)

The tokens differ in different cases (upper-case and lower-case). Type the key words in the upper-case only.

In the following description LETTERS are A..Z,a..z,

LETTERS_&_DIGITS are A..Z,a..z,0..9,_ (underscore).

The restrictions to RIGAL token syntax follow.

RULE NAMES begin with # followed by LETTER and zero or more LETTERS_&_DIGITS.

VARIABLE names begin with $ followed by LETTER and zero or more LETTERS_&_DIGITS. If the first letter is N or I (upper-case only), such pattern variable matches the number or identifier respectively.

Key words are : AND BREAK CLOSE DIV DO ELSIF END FAIL FI FORALL IF IN LAST LOAD LOOP MOD NOT NULL OD ONFAIL OPEN OR PRINT RETURN SAVE

Note that all they are in uppercase.

Delimiters and special tokens are :

/ ;; (* *) ! (+ +) <* *> : , ; := ++ !. !! + - * ( ) <> =< = >= < > :: <. .> (. .) [ ] . -> << <] @ S' V'

Numerical atoms in RIGAL are decimal, from 0 to 2**31-1. The numbers may be written in octal mode, followed by the letter "B".

Non-numerical atoms are quoted by apostrophes. An apostrophe inside the atom must be written twice. Atom maximal length is 80 characters.

If the non-numerical atom is LETTER followed by zero or more LETTERS_&_DIGITS, and this atom is not a RIGAL key-word, then it is allowed to write it without apostrophes.

2.2 Including other files

Use the %INCLUDE <filename>.rig directive to include other files into your program. The suffix (.rig) is necessary, path also may be specified. The file name is followed by spaces. Use %INCLUDE only between or after the rules. Don't use it between the statements or the patterns.

2.3 Atom types

The atoms (excluding NULL) are implemented as Pascal records which consist of the type of the atom and the string (or numerical) value.

There are the following atom types in RIGAL:

1. IDENTIFIER. Such atom value is LETTER followed by zero or more LETTERS_&_DIGITS.

Such atoms match the built-in rule #IDENT and I-variables in the patterns. They may be used as the selectors in the trees.

2. NON-NUMERIC atoms which are not identifiers.

3. NUMBER. They match built-in rule #NUMBER and N-variables in the patterns.

4. PASCAL STRING CONSTANT. They are created by the PASCAL scanner and match built-in rule #TATOM.

5. PASCAL REAL CONSTANT. They are created by PASCAL scanner and match built-in rule #FATOM predicate. Arithmetical operations with them are allowed through rule #CALL_PAS(80) only.

Other atom types (KEYWORD, SPECIAL, VARIABLE, RULENAME) are created by the RIGAL scanner and intended for internal use.

2.4 Lazy boolean evaluation

AND and OR operations differ in the interpreter and in the compiled program. The interpreter evaluates all the arguments of boolean expressions, but the compiled program evaluates the necessary arguments only.

2.5 Opening text files

The statement

```
OPEN <internal-file-name> <expression>
```

opens text file for output.

The expression must have a value, which represents valid UNIX file name. If it is not an atom - ERROR 11 message is issued. If the file with such internal name is already open, then ERROR 16 is issued, and the new file is not being open. Only 5 files can be open simultaneously, ERROR 13 appears if you try to open the 6-th file. ERROR 14 appears if you try to output to the closed file. ERROR 19 appears if the file name is not valid.

File name which consists of one space symbol corresponds to the standard output.

Examples :

```
OPEN  LST  '~john/dd.rep';
OPEN SCREEN ' ';
OPEN   PASCODE   #IMPLODE   ($Name '.pas');
```

2.6 Text output

The <] and << statements write atoms to text file. NULL is written as empty string. The #TATOM atoms are written in apostrophes and inner apostrophes are duplicated. The list is written as sequence of the atoms in the list. The label of the list and the inner structure are not printed. The trees are not allowed to be written: the '<*!<*TREE*>!*>' mark will

appear in the file.

2.7 Length of the text file line

Default maximal length of the line is 80 bytes.

Before every atom has written its length and length of the current line are added, and the sum is checked. If the atom does not fit in 80 bytes, then if insert-space mode is ON, (@ switch) it causes switch to a new line, but, if OFF, then the atom is written to the current line (and length of this line will be more than the default maximal value).

2.8 New line format

The << statement writes EOL byte and then writes the required atoms. If the << statement is executed immediately after OPEN, the first line in the file remains empty.

2.9 LOAD and SAVE statements

RIGAL data loading and saving require the following statements:

```
LOAD $Variable <expression>;

SAVE $Variable <expression>;
```

The expression value must be an atom, its value must be a valid file name. If it is not an atom, ERROR 11 appears. Limitations and defaults for file name are the same as in OPEN statement. If the file name is invalid, LOAD statement causes ERROR 18, but SAVE causes ERROR 15. Piping and standard input and output are not allowed here.

Screen file (' ') is not allowed. If the file does not exist, LOAD statement assigns NULL to the variable. If the variable value is NULL, SAVE statement does not create any file.

2.10 PRINT statement

The PRINT statement writes the argument value to the special file, its name is received as the command line option. The statement writes to standard output by default.

Atoms, lists and trees are written in structured form with the indentation. Identifiers, numbers and Pascal REAL CONSTANTS are written without the apostrophes, but Pascal STRING

CONSTANTS and non-numerical atoms in apostrophes. The sequence of the tree selectors is not determined. The maximum depth of nesting can be changed by #CALL_PAS(78 level). The default level is 15.

The trace information is written to the same file in the same format.

2.11. Built-in rules

There are predicates such as #ATOM(E), #NUMBER(E), #IDENT(E), #LIST(E) and #TREE(E). They return their argument and end successfully or return NULL and end unsuccessfully.

The built-in rule #LEN(E) returns the number of characters in the atom, the number of list elements or tree branches.

#EXPLODE(E) returns E decomposed in separate characters.

#IMPLODE(E1 E2 ... EN) concatenates atoms or lists E1, E2, ..., EN

#CHR(N). The rule returns an atom, which consists of just one ASCII character with the code N (0 <= N <= 255).

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

#PARM(T) . Returns a list of the command line parameters.

#DEBUG(E). If E equals the atom 'RULES', then debugging trace is turned ON , if NORULES, it is turned OFF. The list of debugging trace messages is given in Appendix C.

#CALL_PAS(N ...) - special Pascal subroutines.

3. Scanners and special subroutines (#CALL_PAS)

The scanners and special subroutines are called in the form

#CALL_PAS (<number> <parameters ...>)

where <number> defines the option to be executed. The options are intended for interface between Rigal and Pascal data types, and they implement procedures which cannot be written in Rigal.

Notions mentioned here are :

- parameters of #CALL_PAS (after number of option) are Rigal objects, and the value returned from this rule is Rigal

object.

- non-numerical atoms, i.e. strings (including #FATOM , #TATOM and #IDENT), their values are used as values of Pascal type STRING (except #FATOM in #CALL_PAS(80))

- numerical atoms (#NUMBER), their values are used as values of Pascal integer type (32 bits).

3.1. ACCESS TO STANDARD INPUT AND OUTPUT

1. If the first parameter is given, the option writes it on the screen. The parameter must be non-numerical. The option reads one line from the standard input. The option returns the non-numerical atom containing the string. Zero length string is represented by NULL.

```
LOOP
  $E:=#CALL_PAS(1 'Enter Y or N ');
  IF $E ->
     IF ($E=Y)OR($E=y)-> $REZ:=T;BREAK;FI;
     IF ($E=N)OR($E=n)-> $REZ:=NULL;BREAK;FI;
  FI;
END;
```

This program expects Y or N from the user.

31. The first parameter is numerical or non-numerical. The second parameter is numerical. The option converses the first parameter value to the string, and spaces are added to it (or some last characters are cut). The length of the string becomes equal to the second parameter value. The option calls Pascal WRITE procedure and writes the string on the screen.

3.2. LEXICAL ANALYZERS

35. The first parameter must be file name. The atoms are read from file. The optional second parameter is string of scanner options. See 4.2 for the detailed definition.

36. The first parameter must be list of atoms(strings). These atoms are parsed to smaller atoms. The optional second parameter is string of scanner options. See 4.3 for the detailed definition.

38. No parameters. Returns list of errors which was produced by last lexical analysis. See 4.11 for details.

14 and 15. Parameter must be existing file name, and the lexical scanner of Rigal is applied to this file. The option returns the list of tokens according to Rigal lexical rules. If the file does not exist, NULL is returned. Option 14 forces to include other files marked with %INCLUDE directive.

16. Parameter must be existing file name. All the lines of the file are read and appended to the list of atoms (one atom corresponds to one line).

3.3. MANIPULATIONS WITH THE COORDINATES

4. Parameter is any object, except list or tree. When the lexical scanner forms the output list, every numerical and non-numerical atom contains the coordinate of the corresponding token in the source file. See 4.8 on detailed definition of the coordinate. The option returns the value of the coordinate, or 0 if the object has no coordinate.

```
#MAIN
 OPEN LST 'A.LST';
 $L:=#CALL_PAS(35 'A.PRG');
 #TEST($L)
##

#TEST
 ( 'PROGRAM' !
   $E
   / LST << #CALL_PAS(4 $E) ' is coordinate of error ' /
 )
##
```

This program prints the error message with the coordinate of the first token in file A.LST.

44. The first parameter is an atom. The second parameter (any number in 0..65535) is assigned to the coordinate of the atom (the first parameter) and modified first parameter is returned. This option is opposite to no. 4.

3.4. STRING CONVERSION

21. Parameter must be non-numerical. The option converts the string to number. If the conversion is impossible, NULL is returned.

85. Converts string to uppercase.

86. Converts string to lowercase.

87. Evaluates substring. First argument is string, second and optional third are numbers. The option takes number of characters specified by the third argument, starting from character specified by second argument. If third argument is absent, it takes substring until the end of string.

```
#CALL_PAS(87 'AFRIKA' 3 2) returns 'RI'
```

88. Both arguments are strings. The option returns position of the first string in the second one. If position is not found 0 is returned.

```
#CALL_PAS(88 'RI'  'AFRIKA') returns 3
```

3.5. REAL NUMBERS

80.You can access some operations with real numbers by #CALL_PAS(80) utility.

Real numbers are represented as atoms (non identifiers) with length equal to the size of internal representation of 'real' type in your system.

Following operations can be preformed; the second parameter of

#CALL_PAS rule used as an operation sign.

Operation "S" converts atom (or Fatom) to real atom:
$REAL:=#CALL_PAS(80 S '444.22');
$REAL:=#CALL_PAS(80 S '-44.22E05');
#CALL_PAS(80 S '-4+') (invalid string) returns NULL
Operation "I" converts integer to real atom:
$REAL:=#CALL_PAS(80 I 55);
#CALL_PAS(80 I '55') or
#CALL_PAS(80 I $REAL_ATOM) returns NULL
Operation "T" converts real atom to integer;
The value is truncated (but not rounded).
$NUM:=#CALL_PAS(80 T $REAL);
#CALL_PAS(80 T $NOT_REAL) returns NULL
Is possible overflow (run time error).
Operation "Z" converts real atom to string in
corresponding formats.
The format is integer
Format=100*Total_digits+Digits_after_point.
The format value should present in parameters:

If value of $REAL is 13.6254,
#CALL_PAS(80 Z $REAL 702) returns ’ 13.63’
If argument is not real or format is not integer,
NULL is returned.
Operations ’+’,’-’,’*’,’/’ return the result:
$REAL1 is 2.2
$REAL2 is 1.5
#CALL_PAS(80 ’+’ $REAL1 $REAL2) returns 3.7
When ’*’ or ’/’ calculates run time error (overflow)
can occur;
If you divide to 0.0 , NULL is returned.
If one of arguments is not real, NULL is returned.
Operations ’<’,’>’,’<=’,’>=’,’=’,’<>’
return the first (real) argument if succeed and NULL
if fail:
$REAL1 is 2.2
$REAL2 is 1.5
#CALL_PAS(80 ’>=’ $REAL1 $REAL2) returns 2.2
#CALL_PAS(80 ’<=’ $REAL1 $REAL2) returns NULL
If one of arguments is not real, NULL is returned.
If wrong operation is given, NULL is returned.

3.6. OTHER SERVICES

78. The first parameter is a number. The option sets maximal depth of nested elements, printed by statement PRINT. The default value is 15.

89. Returns list of two strings - system time and date:
(.’ 16:30:47 ’ ’18 Apr 92 ’ .)

90. Closes all the files and halts the program.

3.7. LOW LEVEL MEMORY CONTROL

NOTE: These control options are intended for advanced users only.

40. Parameter of any type. The option returns a number. Its value is equal to the S-address of the parameter (converted to number).

41. No parameters. The option returns a number. Its value is equal to the S-address of the returned atom (converted to num-

ber).

42. No parameters. The option returns a number. Its value is equal to the page number where the returned numerical atom was just allocated. This number is in -128..127.

43. No parameters. If the virtual memory manager uses the current (physical) disk drive as secondary disk, then the option returns the page number of the returned numerical atom (otherwise 0 is returned).

46. The first parameter is any Rigal object. The option tests the current environment. If the interpreter executes the program, the option returns the same value as the parameter. If the compiled program is working, the option returns NULL.

99. Parameter is an atom which is inserted into the generated Pascal code when the Rigal program is compiled.

4 The lexical analyzer

The analyzer reads unstructured data, e.g. strings of text or files and returns list of tokens or other structures. You can call the analyzer using #CALL_PAS built-in rule.

The analyzer

- supports Pascal, C and Modula-2 lexics;
- produces labelled trees and lists;
- takes input from Rigal atom or from disk file;
- produces error message list;
- supports different coordinate computation modes;

4.1 The simplest lexical analyzer

This rule requires a file name and it returns the list of tokens.

If you write

```
$A:=#CALL_PAS(35   'A.PAS');
```

and file A.PAS contains

```
ABC+752 'TEXT'
```

then value of variable $A becomes a list of 4 atoms:

```
(. ABC '+' 752 'TEXT' .)
```

The coordinates of these atoms are 1, 4, 5 and 9 respectively. Use built-in rule #CALL_PAS(4 ..) to get the coordinates.

Following questions are discussed here:

How to read from text file ?
How to analyze the atoms like 'ABC+D' ?
How to set options for the analyzer ?
How to change lexical grammar ?
How to control computation of the coordinates ?
How to get tree and list structures ?
How to get error messages ?

4.2 Reading from file

SYNTAX: $A:=#CALL_PAS (35 filename options)
or
$A:=#CALL_PAS (35 filename)

FUNCTION: Reads tokens from the file

REMARKS: Rigal tries to find the given file name and read from this file. If the file is absent, NULL is returned. File is set of zero or more lines, separated by the newline character. Each line is set of zero or more characters. The newline character isn't included into the line. Lines longer than 127 characters are truncated. Lexical grammar and the coordinates depend of the options specified.

4.3 Reading from list of atoms

SYNTAX: $A:=#CALL_PAS (36 list_of_atoms options)
or
$A:=#CALL_PAS (36 list_of_atoms)

FUNCTION: Forms tokens from the list of atoms

REMARKS: Rigal reads data from the given list. It should be a Rigal expression; its value should be list of non-numerical atoms. This list is interpreted as a set of zero or more lines, each line is a set of characters. Numeric atoms, lists and trees are ignored (they are interpreted as empty lines). Lexical grammar and coordinates depend of the options given.
EXAMPLE:

```
#CALL_PAS(36 (. ' ABC+D' (. to ignore .) 'EF' .))
```

returns value

```
(. 'ABC' '+' 'D' 'EF' .)
```

4.5 Options

Options are presented as string of characters followed by + (on) or - (off) immediately after them. Default value of options is 'D-C-P+p+m-U+S+O+s-t-L-A+R+Y-B-N-'. Options supplied by the user overwrite default ones. Note that some options require other ones, some of them are incompatible. All such cases are noted below in NOTE sections.

4.6 Common components of lexical grammars

SPACES. One or more spaces, tabs, newline characters are ignored during lexical analysis.

COMMENTS. They are ignored during lexical analysis.

The options allow to choose among different comment start and comment end symbols.

If a comment is not ended at the end of file, error code 12 is produced.

STRING CONSTANTS. They are converted into TATOM atoms.

The options allow to choose among different string constant start and end symbols and conversion modes. If the end of line appears within string constant, error code 11 is produced and string con- stant is truncated.

NUMBERS. They are converted to NUMBER or FATOM atoms.

Integer numbers with less than 10 digits are converted to NUMBER atoms. Integer numbers with 10 or more digits, floating point numbers and fixed point numbers are converted to FATOM atoms. The following grammar rule defines all the acceptable numbers:

Number ::= Digits [. [Digits]] [(e | E) [(+ | -)] [Digits]]

Digits ::= Digit*

Digit ::= (0|1|...|9)

IDENTIFIERS. They are converted to IDATOM atoms.

The following grammar rule defines all the acceptable identifiers:

Identifier ::= Letter Letter_or_digit*

Letter_or_digit ::= (Letter | Digit | _)

The options allow to change the set of letters and to convert all the letters to the uppercase.

SPECIAL SYMBOLS. They are converted to ATOMs.

These symbols consist of 2 or more characters. Set of special symbols should be specified by options.

SINGLE CHARACTERS. They are converted to ATOMs.

NOTE: All the atoms have length up to 80 bytes. All the atoms longer are truncated and error code 10 is produced.

4.7 Using options

The specific properties of different lexical grammars are set by the following option groups:

Special symbols:
A - Pascal-oriented symbols
L - C-oriented symbols

Identifier conversion: U - convert to uppercase

Comment modes:
P - Pascal-like comments ((* *) { })
C - C-like comments (/* */)

String constants:
p - Pascal-like ('..')
m - Modula-2-like ('..' and "..")

For Pascal input you can use the default options. For C input use use Pascal or C options and switch some of them on(off) if necessary. Option 'C+P-m+p-U-L+A-' is recommended for C-like lexics.

For other languages you should modify some assignments in the source file SCAN.PAS available in Rigal sources library.

SYNTAX: A+ A- (default A+)
FUNCTION: Sets Pascal-oriented special symbols and letter set
REMARKS: This option sets the following special symbols:

:= <= >= ** .. <>

#Digits_and_letters $Digits_and_letters %Digits_and_letters

{$Any_characters}

SYNTAX: L+ L- (default L-)

FUNCTION: Sets C-oriented special symbols and letter set
REMARKS: This option sets the following special symbols:

```
->   ++   --   >>   <<   ==   +=
*=   -=   /=   %=   &=   ^=   |=   !=
0Digits_and_letters
```

NOTE: L+A+ is not allowed

SYNTAX: P+ P- (default P+)
FUNCTION: Sets Pascal-like comments
REMARKS: This option sets comments start symbol { and (* , comment end symbol } and *) . Nested comments are not allowed. {$...} is not comment. NOTE: Requires A+.

SYNTAX: C+ C- (default C-)
FUNCTION: Sets C-like comments
REMARKS: This option sets comments start symbol /* , comment end symbol */ . Nested comments are not allowed.

SYNTAX: p+ p- (default p+)
FUNCTION: Sets Pascal-oriented string constants
REMARKS: String constants start and end with one apostrophe (’). An apostrophe within string constant should be written twice, but only one apostrophe is included into the atom value.
NOTE:Requires P+.

SYNTAX: m+ m- (default m-)
FUNCTION: Sets C- or Modula-oriented string constants
REMARKS: String constants start and end with one apostrophe (’) and is converted to TATOM , or double quotes (") and is converted to KEYWORD. If back slash appears within string constant, this and the next character are included into the atom value. atom value.
NOTE: Requires C+.

4.8 Coordinate calculation

The lexical analyzer assigns a number (coordinate) to every atom produced. This number should be in 0..32767. You can access this number using built-in rule #CALL_PAS(4 ..). Following options give different modes to calculate this number.

NOTE: Coordinate function does not check for coordinate value overflow. The function works correctly within given bounds only.

SYNTAX: R+ (default R+)
FUNCTION: Sets coordinate mode to Line*80+Column
REMARKS: This option sets the coordinate function to line_number*80+column_number, where line_number in 1..410, column_number in 1..127.

SYNTAX: s+R- (default R+)
FUNCTION: Sets coordinate mode to Line
REMARKS: This option sets the coordinate function to the line number (in 1..32767).

SYNTAX: t+R- (default R+)
FUNCTION: Sets coordinate mode to token number
REMARKS: This option sets the coordinate function to the token number (in 1..32767).

SYNTAX: Y+R- (default R+)
FUNCTION: Sets coordinate mode to the byte number
REMARKS: This option sets the coordinate function to byte number of the first character of token in source (in 1..32767). Newline and carriage-return characters are not included in this number. Coordinate marks (see below) and structure control characters are included.

SYNTAX: B+R- (default R+)
FUNCTION: Sets coordinate mode to mark_value+bytes_number
REMARKS: This option sets the coordinate function to the last coordinate mark value plus byte number of the first character of token starting from the mark (in 1..32767). Newline and carriage-return characters are not included in this number. Coordinate marks (see below) and structure control characters are included.

SYNTAX: N+R- (default R+)
FUNCTION: Sets coordinate mode to mark_value
REMARKS: This option sets the coordinate function to the last coordinate mark value (in 1..32767).

4.9 Coordinate marks

You can insert a coordinate mark Chr(17) in source and any number immediately after the mark. The coordinate mark is used for calculation of coordinates of all the atoms produced between this mark and next one. The coordinate marks are not converted to atoms.

EXAMPLE:

```
#CALL_PAS(36 #IMPLODE ( #CHR(17) 100 ’ a’) ’N+R-’)
```

returns (.A.) and the coordinate of atom A is 100.

4.10 Structure control characters

You can insert the structure control characters in source for lexical analyzer. It allows to have an arbitrary complex structure of trees and lists rather than simple list of atoms.

Chr(21) - start list
Chr(22) - end list
Chr(23) - start tree
Chr(24) - end tree
Chr(25) - assign name to object

The entire source for lexical analyzer should be accepted by the following grammar. If the source does not matches grammar, error codes are produced.

Entire_source ::= (* Item *)

Item ::= (start_list (* Item *) end_list |
start_tree (* Selector Item *) end_tree |
name_character Item1 Item2 |
token)
Item1 ::= Item
Item2 ::= Item
Selector ::= Non-numeric_token

The corresponding Rigal structure is

#Entire_result (. #Object .) ##

```
#Object     ( (. (* #Object *) .) !
              <* #Selector : #Object *> !
              #Object1 :: #Object2 !
              #Atom ) ##
#Object1 #Object ##
#Object2 #Object ##
```

EXAMPLE:

```
#CALL_PAS(36 #IMPLODE(#CHR(21) ' A B ' #CHR(22) ' C ' ))
```

returns

```
(. (. A B .) C .)
```

4.11 Lexical error messages

The following options allow to control error messages:

SYNTAX: S+ S- (default S+)
FUNCTION: Send error messages to the screen
REMARKS: This option sends the error code number, line and column number to the screen. This option is intended for debugging purposes.

SYNTAX: O+ O- (default S+)
FUNCTION: Append error messages to the list
REMARKS: This option collects the error code numbers in the special list of messages. You can have this list using

$LIST:=#CALL_PAS(38)

$LIST is Rigal list of numeric atoms. Each number is error code number, but the coordinate of the numeric atom is erroneous place coordinate. $LIST is NULL if there was no errors during last lexical analysis. This option is intended for smart diagnostic purposes. Text of the lexical error messages (corresponding to the given message code number and the coordinates) should be constructed by the programmer.

4.12 Error message code numbers

1 - unexpected end of file before end of tree

2 - unexpected end of file within tree branch
3 - unexpected end of file within tree
4 - unexpected end of file before end of list
5 - unexpected end of tree
6 - unexpected end of list or another control character
7 - unexpected end of file in name
8 - unexpected end of file in named object
9 - unexpected control character
10 - too long atom (more than 80 bytes)
11 - end of string constant not found in the current line
12 - end of file before end of comment
13 - control character within comment
14 - control character within string constant

APPENDIX A. Messages of the syntax checker

2- symbol '##' not found after the main rule
3- rule name '#...' not found in the beginning of the rule
5- symbol '##' not found after the rule
21- rule name '#...' not found in the beginning of the main rule
22- symbol '##' not found after the main rule
23- main rule name is built-in rule name
30- ending '/' not found after list of statements
31- symbol '.)' matching '(.' was not found
37- unexpected element in pattern
38- unexpected keyword or symbol in pattern
40- unexpected branch found in tree pattern after variable $a:...
41- variables not allowed in tree pattern <. $a : ...
42- variables not allowed in tree pattern <. ... [$a : ..]
43- only variables and atoms allowed as branch name in tree pattern
44- symbol ':' not found in tree pattern
45- only one pattern as branch value in tree pattern allowed
46- symbol ']' matching ']' not found in tree pattern
47- symbol '.>', '*>' or ',' must be after branch pattern
48- pattern <* ... .> not allowed
49- pattern <. ... *> not allowed
50- a variable not find in the last branch of <* ... *>
51- no more than 5 branches in <* ... *> allowed
52- symbol '::' not allowed before rule name
53- symbol '::' not allowed before '('
54- must be at least one pattern or action within (* *)
55- must be at least one pattern or action within (+ +)

56- must be at least one pattern or action within [ ]
57- empty tree pattern is not allowed
58- symbol '::' is not allowed in this position
59- empty alternative in (..!..) or empty () are not allowed
61- symbol ';;' or '##' expected
63- rule name #... not found in the beginning of the rule
64- symbol ';;' or '##' expected
65- this rule name is built-in rule name
66- symbol ';;' or '##' expected
71- wrong delimiter in (*...* ..) pattern
72- symbol ')' expected in (*...* ..) pattern
73- symbol '*)' matching '(*' not found
74- wrong delimiter in (+...+ ..)
75- symbol ')' expucted in (+...+ ..) pattern
76- symbol '+)' matching '(+' not found
77- symbol ']' matching '[' not found
81- only one element might be in every part of (..!..!..!..)
82- only one element might be in the last part of (..!..!..!..)
83- unexpected symbol in pattern (..!..!..!..)
84- unexpected '!' found in pattern (..!..!..!..)
86- symbol '(' expected after s' or v'
87- symbol ')' expected after s'(... or v'(...
91- assignment symbol ':=' expected after '!!', '!.', '++' or
101- - internal error :mainadr<>listmain
102- - internal error : push for trees
201- -ins- symbol 'fi','od', '/', '##' or ';' expected
202- symbol '->' after 'if' expected
203- symbol 'fi' after 'if...->' expected
205- symbol '->' after 'elsif' expected
206- symbol 'fi' after 'elsif...->' expected
207- statement expected after 'elsif..->'
208- symbol 'end' matching 'loop' no found
209- statement expected after 'if..->';
210- variable name after 'forall' expected
211- symbol 'in' after 'forall' expected
212- symbol 'do' after 'forall...in' expected
213- symbol 'od' or '' expected after forall..do...
215- symbol ':=' expected after this object
216- assignment symbol ':=' expected after '!!','!.','+' or
217- variable expected after "selectors"
218- variable expected after "branches"
220- variable expected after 'load'
221- variable expected after 'save'
222- file identifier expected after 'open'
223- file identifier expected after 'close'

224- wrong beginning of the statement
225- unexpected symbol after rule call #..(..)
301- symbol ')' expected
302- rule name #... expected after 'last'
303- vaiable name $... expected after 'last #...'
304- built-in rule not allowed in 'last'
307- symbol '$$' allowed only inside s' pattern
313- symbol '(' expected after 'copy'
314- symbol ')' expected after 'copy (...'
321- symbol ']' matching '[' not found in expression
323- unexpected symbol (end of '<<'-statement not found)
324- null or a'... not allowed in the left side of the assignment
325- symbol '(' expected after rule call in expression
327- wrong object or symbol in expression
329- symbol ':' expected in <. ... .> constructor
330- symbol ',' or '.>' expected in <. ... .> constructor
331- symbol '','' is unexpected
405- rule was defined two times (this is the second)
406- rule was not defined in program
407- call of the main rule not allowed
408- variable $ not allowed in the main rule
409- variable $$ not allowed in the main rule
501- internal error : 1st parm - not rule name
503- internal error : 2nd parm - not variable
504- more than 255 variables in rule
505- more than 400 rules in program
521- internal error : no rulename in v-list
522- internal error : no listmain in r-list variable list
523- internal error : this rule not found
524- internal error : no num in v-list 1-st parm
525- internal error : no num in v-list 2-nd parm
526- rule #### was not defined in program
527- rule #### was defined two times
528- internal error : wrong type in bltin rule table
699- - unexpected end of program

APPENDIX B. Run time error messages

1 - Interpreter stack size overflow (stack size = ####);
(Usually looping in the sequence of calls)
2 - Assignment left side is not list or tree
3 - List index is not number
4 - Using [..] not for list
5 - Index value exceeds list bounds

6 - Not list or tree after "::"
7 - Not atomic name before "::"
8 - NULL in left side of assignment
9 - Not numerical value in left side of "+:=" statement
10 - Not numerical value in right side of "+:=" statement
11 - File specification is not atom
12 - Too long file specification
13 - Too much open text files
14 - File not open for output
15 - Unable to open file in SAVE
16 - File was not closed before new opening
17 - Atom length exceeds file record length
18 - Unable to open file in LOAD
19 - Unable to open file in OPEN
21 - Selector after "." is not identifier
22 - Selector in tree constructor is not identifier
23 - Not tree before "." operation
24 - Not tree or list as base of FORALL-IN statement
25 - Atom length more than 80 bytes in #IMPLODE
26 - "BRANCHES" option cannot be used for lists

APPENDIX C. Debugging messages

=>>>CALLS RULE #... IN STATEMENT
1-ST ARGUMENT($):

=>>>CALLS RULE #... IN PATTERN
1-ST ARGUMENT($):

>> (BRANCH NO 2)

=>>>CALLS BUILT-IN RULE #... IN STATEMENT: SUCCESS
RESULT:

=>>>CALLS BUILT-IN RULE #... IN PATTERN: SUCCESS
RESULT:

<<<=EXITS FROM RULE #... : SUCCESS
RESULT:

APPENDIX D. Syntax grammar of Rigal

Singature:
NNNN - nonterminal
'TTTT' - terminal (inner quotes are duplicated)
S1.S2 - concatenation without the delimiters
[S] - optional

S1 ! S2 - alternative
(* S *) - iteration 0 or more times
(+ S +) - iteration 1 or more times
(* S *) - delimiter A
(+ S +) - delimiter A
(   ) - metaparentheses

Lexical rules:

Letter ::= A..Z,..z
Digit ::= 1!2!3!4!5!6!7!8!9!0
Number ::= (+ digit +)
Symbol ::= visible character
String ::= (+ Symbol +)

Base rules of the grammar

1) RIGAL_program ::= main_program (* rule *)
2) main_program ::= '#'.name statements '##'
3) rule ::= '#'.name (* simple_rule * ';;') '##'
4) simple_rule ::= (* (sequence_of_patterns !
                        ('/' statements '/')) *)
                    ['ONFAIL' statements]

Patterns

5) sequence_of_patterns ::=
(* element_of_sequence_of_patterns *)
6) element_of_sequence_of_patterns ::= pattern !
   ('(*' sequence_of_patterns ('*)' !
                        ('*' atom ')') !
                        ('*' '#'.name ')')
)!
            ('(+' sequence_of_patterns ('+)' !
                        ('+' atom ')') !
                        ('+' '#'.name   ')')
)!
            ('[' sequence_of_patterns ']') !
            ('(' sequence_of_patterns ')') !
            ('('
              (+ element_of_sequence_of_patterns +
                  '!') ')')
7) pattern ::= ([variable assignment_symbol]
                    (atom !
                      variable !

```
                        ′#′.name !
                        pattern_of_list !
                        pattern_of_tree !
                        ′S′′′ ′(′ expression ′)′ !
                        ′V′′′ ′(′ expression ′)′ !
                        ( ′(′ (* pattern * ′!′ ) ′)′ )
                    )
                ) !
        ( [ ′/′ statements ′/′]
            pattern  [′/′ statements ′/′] )
8) variable ::= ′$′.name
9) assignment_symbol ::= [ ′+′ ! ′++′ ! ′!!′ ! ′!.′  ] ′:=′
10) atom ::= name !  number !  number.′B′ !  ′NULL′ !
              ( ′′′.    String       .′′′ )
11) name ::= letter [. (* ( letter ! digit ! ′_′ ) *) ]
12) pattern_of_list ::=
       [ ( atom ! variable ) ′::′ ]
     ′(.′  sequence_of_patterns ′.)′
13) pattern_of_tree ::=
         [ ( atom ! variable ) ′::′ ]
         (( ′<.′ (* element_of_pattern_of_tree * ′,′) ′.>′ ) !
          ( ′<*′ (* element_of_pattern_of_tree  ′,′ *)
                  variable ′:′ pattern ′*>′ )       )
14) element_of_pattern _of_tree ::=
       ( atom ′:′ pattern )  !  ( ′[′ atom ′:′ pattern ′]′ )

                            Statements

15) statements ::= (* statement * ′;′ )   [ ′;′ ]
16) statement ::=  assignment !  condition!  input_output !
             return ! fail ! loop ! call ! break
17) assignment ::= object  assignment_symbol  expression
18) condition ::=
   ′IF′ expression  ′->′ statements
   (*  ′ELSIF′  expression  ′->′  statements  ′)
   ′FI′
19) input_output ::=
             (′LOAD′  variable  filename   )!
             (′SAVE′  variable  filename   )!
             (′OPEN′ name  filename  )!
             ( name ′<<′ (* ( expression ! ′@′ ) *)  )!
             ( name ′<]′ (* ( expression ! ′@′ ) *)  )!
             ( ′CLOSE′ name )    !
             ( ′PRINT′ expression )
20) filename ::=  expression
```

```
  21) return ::= 'RETURN' expression
  22) fail ::= 'FAIL'
  23) loop ::= ( 'FORALL'
                  [ ['SELECTORS']  variable ]
                  [ 'BRANCHES' variable ]
                  'IN' expression
                       'DO' statements 'OD'   ) !
               ( 'LOOP'  statements  'END' )
  24) break ::=  'BREAK'

                      Expressions

  25) object ::=
          [ prefix ] ( variable ! '$' )   (* index ! selector *)
  26) expression ::=   atom !
              ( [ prefix ] ( variable ! '$' ! '$$' ) ) !
              constructor !
              ( un_operation expression ) !
              ( expression bin_operation expression ) !
              call !
              ( expression (+ index ! selector +) ) !
              ( '(' expression ')' )
  27) prefix ::= 'LAST' '#'.name
  28) index ::= '[' expression ']'
  29) selector ::= '.' expression
  30) constructor ::= constructor_of_list !  constructor_of_tree
  31) constructor_of_list ::= '(.' (* expression *) '.)'
  32) constructor_of_tree ::=
                     '<.' (*  expression  ':'  expression  *  ',')
  33) bin_operation ::= '!!' ! '!.' ! '++' ! '='  ! '<>' !
               'AND' ! 'OR' ! '+'  ! '-'  ! '*'  ! 'DIV'! 'MOD'!
               '>'  ! '<'  ! '>=' ! '<=' ! '::'
  34) un_operation ::= 'NOT' ! '-'
  35) call ::= '#'.name '(' (* expression *)  ')'
```

APPENDIX E. FAST REFERENCE TABLE ON #CALL_PAS PARAMETERS

ATOM_OR_NULL:=#CALL_PAS(1 ATOM) WRITE and READLN from screen
NUM:=#CALL_PAS(4 ATOM_OR_NUM) Takes coordinate
LIST_OR_NULL:=#CALL_PAS(14 FILENAME) Rigal scanner with %INCLUDE
LIST_OR_NULL:=#CALL_PAS(15 FILENAME) Rigal scanner without %INCLUDE
LIST_OR_NULL:=#CALL_PAS(16 FILENAME) Reads file by lines

NUM_OR_NULL:=#CALL_PAS(21 ATOM) Numerical value from string.
#CALL_PAS(30 NUM_OR_ATOM) WRITE
#CALL_PAS(31 NUM_OR_ATOM SIZE) WRITE(X:SIZE)
$LIST:=#CALL_PAS(35 FILE [OPTIONS]) LEXICAL ANALYSER OF FILE
$LIST:=#CALL_PAS(36 LIST_OF_STRINGS [OPTIONS])
LEXICAL ANALYSER OF STRINGS
$LIST_OF_NUM:=#CALL_PAS(38) LEXICAL ERROR LIST
NUM:=#CALL_PAS(40 ANY) S-ADDRESS
NUM:=#CALL_PAS(41) S-ADDRESS of itself
NUM:=#CALL_PAS(42) Page number
NUM:=#CALL_PAS(43) Page number on disk (or 0)
NUM_OR_ATOM:=#CALL_PAS(44 NUM_OR_ATOM NUM) Assign new coordinate to atom.
ANY_OR_NULL:=#CALL_PAS(46 ANY) NULL if this program is compiled
#CALL_PAS(78 DEPTH) maximal depth for PRINTs

#CALL_PAS(80 OP ???) Float processor:
#FATOM or NULL:=#CALL_PAS(80 S String) -- convertion
#FATOM:=#CALL_PAS(80 I #NUMBER) -- ---"---
#NUMBER:=#CALL_PAS(80 T #FATOM) -- Truncation
String:=#CALL_PAS(80 Z #FATOM X*100+Y) -- Formatter
(X digits, Y after point)
#FATOM OR NULL:=#CALL_PAS(80 op #FATOM #FATOM) "op" is +
- * / < > = <= >= <>

TEXT:=#CALL_PAS(85 TEXT) UPPERCASE
TEXT:=#CALL_PAS(86 TEXT) LOWERCASE
TEXT:=#CALL_PAS(87 TEXT N1 [HOW_MANY]) SUBSTR(TEXT,N1,HOW_MANY) (default - until till the end)
NUM:=#CALL_PAS(88 TEXT1 TEXT2) POSITION OF TEXT1 IN TEXT2,
(0 if not found)
LIST:=#CALL_PAS(89) Returns data and time:
#CALL_PAS(90 [errorlevel]) Halts program, closes files.
#CALL_PAS(99 'a Pascal statement') insert to the compiled text.