

L.S.Baraz, E.V.Borovikov, N.G.Blagoleva,
P.A.Zemtsov, E.V.Nalimov, V.A.Tsikoz

RAPIRA PROGRAMMING LANGUAGE

A) ■ GLIMPSE THROUGH EXAMPLES (24.09.87)

The Rapira programming language[1, p.199-206] was designed by G.A.Zvenigorodsky as a bridge from the educational language Robic[1] to the standard programming languages. That version of the language was implemented as an interpreter within the programming system "Shkolnitsa" ("Schoolgirl") on the "Agat" computer[2].

The experience in its usage has enabled to begin a new version. A programming system for it is being implemented on "Yamaha" and "Korvet" computers.

This article provides an overview of the new version. The examples of small programs and their work show the main features of Rapira. (The mark '♦' indicates program output). In fact, the examples illustrate mainly object processing. The statements of Rapira are those typical for the procedure-oriented languages, therefore they are introduced almost without comments. (A comment in Rapira begins with '\ ' symbol and spreads upto to the end of the line. The comments below, explaining the algorithm, but not the language features are marked with '■').

Dialog with Rapira system

In fact, there are no Rapira programs when working with the system. There are definitions of procedures and functions and there are single statements being executed immediately in current environment. The current environment is initialized when system starts and may be saved at any moment. In particular, it includes all the global variables with their values.

Lexica

Rapira has Russian lexica. But two more variants were developed: English and Moldavian. Here the English variant for new Rapira version is described with the Russian equivalents for keywords and standard names.

Objects and operations

Objects: logical (yes, no), integer and real numbers, texts, sequences, the empty object (empty).

Procedures, functions, modules and devices are also objects.

For all objects operations = and /= are defined.

Logical: yes, no.

Logical operations: and, or, not.

Numbers: 125 \ integer
330.84 \ real
1.9e-8 \ real

Numeric operations: + and - (unary: and binary), %, /, ** (exponentiation), // and /% (integer division and remainder - for integer numbers), \, <, >=, <=.

Example 1.

```

fun PRIME(N)                                \ Is N a prime number ?
  if N<2 then                                \ return a function value
    return no
  fi
  for M from 2 to sqrt(N)+0.5 do
    if M /% N = 0 then
      return no
    fi
  od
  return yes
end
output: PRIME(2003)
• yes

```

Texts:

```

"Could we have some butter for The Royal slice of bread?"
""                                           \ empty text
" 2 * 2 = 11 "
Operations for texts: # (length), + (concatenation),
* (multiplication by non-negative integer),
[] (select a symbol in the text), [:] (subtext
slice).

```

Example 2.

```

proc FRAME (N)                               \ output a frame: N x N
  output: "@" * N                             \ @@@@
  repeat (N-2) do
    output: "@" + " " * (N-2) + "@"         \ @ @
  od
  output: "@" * N                             \ @@@@
end
call FRAME (5)
• @@@@
• @ @
• @ @
• @ @
• @@@@

```

Example 3.

```

proc VICE_VERSA ()                            \ output reversed text
  outputs: "Enter a text "                   \ prompt for input
  input text: PHRASE                          \ PHRASE will be = "case"
  REVERSED:= ""                               \ empty text
  for K to #PHRASE do                          \ from 1
    REVERSED:= PHRASE[K] + REVERSED          \ #"case" = 4
  od                                           \ "case"[2] = "a"
  output: REVERSED
end
call VICE_VERSA()
• Enter a text
  case
• esac

```

Example 4.

```

proc TEXT_BY_WORDS (PHRASE)           \|| output text by words
  PHRASE:= PHRASE + " "              \|| - that is for
                                      \|| last word processing

  while PHRASE /= "" do
    K:= index(" ", PHRASE)           \  first entry number
                                      \  " " in PHRASE (or 0)

    if K /= 1 then
      output: PHRASE[:K-1]           \  "ab cd e"[:2] = "ab"
    fi
    PHRASE:= PHRASE[K+1:]            \  "ab cd e"[4:] = "cd e"
  od
end                                     \  "ab cd e"[4:5] = "cd"
TEXT_BY_WORDS
("...I'll go and tell The cow Now Before she goes to bed.")
♦ ...I'll
♦ go
♦ and
♦ tell
♦ The
♦ cow
♦ Now
♦ Before
♦ she
♦ goes
♦ to
♦ bed.

```

Sequences: ♦ 2, 3, 5, 7, 11 ▶
 ♦ "a", "e", "i", "o" ▶
 ♦ ♦ 1, 4, 6, 7, 9 ▶ ▶, 10, 4 44, 70, 99 ▶ ▶

Sequence is an ordered set of objects. Any Rapira object may be an element of a sequence (including the other sequence).

The sequence construct operation ♦ || builds a sequence.

Sequential operations: # (length), + (concatenation), * (multiplication by non-negative integer), [] (element select in the sequence), [:] (subsequence slice).

Example 5.

```

fun TEXT_TO_WORDS (PHRASE)           \|| divide text to words
  PHRASE:= PHRASE + " "
  WORDS:= ♦                             \ empty sequence

  while PHRASE /= "" do
    K:= index(" ", PHRASE)
    if K /= 1 then
      WORDS:= WORDS + ♦ PHRASE[:K-1] #
                                      \ ♦ "ab", "cd" ▶ + ♦ "e" ▶ =
                                      \                               ♦ "ab", "cd", "e" ▶
    fi
    PHRASE:= PHRASE[K+1:]
  od
  return WORDS
end

```


output: SIEVE(5)

• < 2, 3, 5 >

output: SIEVE(200)

• < 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53
 • , 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 11
 • 3, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 18
 • 1, 191, 193, 197, 199 >

Variables

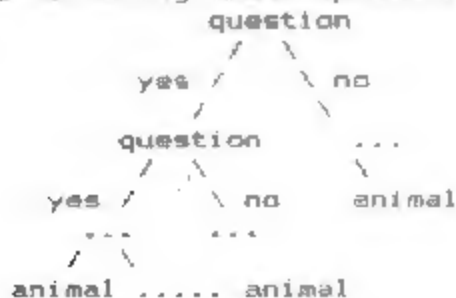
Any object may act as the value of a variable.

Example 8.

Game: a person thinks of an animal; computer tries to guess it asking questions, for 'yes' and 'no' answers only. If the animal is unknown, computer asks the help thus enriching its knowledge.

Algorithm.

The following tree represents knowledge:



Computer asks a question and chooses the left or the right subtree depending on the answer; and it does so until it reach a leaf with an animal name:

```

/
animal
  
```

Then computer asks, if this animal is thought of. If not, computer gives up, asks for that animal and a question to distinguish them both. As a result, the leaf is replaced by the subtrees:



Realization.

The tree is represented by the sequence KNOWLEDGE:

< question, subtree for 'yes', subtree for 'no' >

Each subtree is a similar sequence of a text (animal's name).

When subtree is chasing, the ATTEMPT variable value is a sequence or a text, depending upon the descend depth.

```

proc ANIMALS ( <=KNOWLEDGE )           \|| game: computer guesses
                                       \|| an animal
output: KNOWLEDGE[1], "(yes,no)"      \|| ask a question
input: ANSWER                          \|| input of an object
if ANSWER then
  N:= 2
  else
  N:= 3
fi
ATTEMPT:= KNOWLEDGE[N]                 \|| reduction variants
if is_seq(ATTEMPT) then                 \|| there is a choice still
  ANIMALS(<=ATTEMPT)
  else
  output: "Is it a ", ATTEMPT, "? (yes,no)" \|| one animal left
  input: ANSWER
  if ANSWER then                       \|| animal is guessed
    output: "I've guessed!"
  else
    \|| the animal is unknown
    \|| computer enriches its knowledge
    output: "I give up. What animal are you thinking of?"
    input text: BEAST
    output: "Type in a question to disitinguish"
    output: "a ", BEAST, " and a ", ATTEMPT, ":"
    input text: QUESTION
    output: "What will be an answer for a ", BEAST, "?"
    input: ANSWER
    if ANSWER then
      ATTEMPT:= < QUESTION, BEAST, ATTEMPT >
    else
      ATTEMPT:= < QUESTION, ATTEMPT, BEAST >
    fi
  fi
fi
KNOWLEDGE[N]:= ATTEMPT
end

ALL_ANIMALS:= < "Does it live in water?", "fish", "ostrich" >
do
  ANIMALS(<=ALL_ANIMALS)
od

♦ Does it live in water?(yes,no)
no
♦ is it a ostrich?(yes,no)
no
♦ I give up. What animal are you thinking of?
turtle
♦ Type in a question to disitinguish
♦ a turtle and a ostrich:
Has it wings?
♦ what will be an answer for a turtle?
no
♦ Does it live in water?(yes,no)
no
♦ Has it wings?(yes,no)
yes

```

- ◆ Is it a ostrich?(yes,no)
no
 - ◆ I give up. What animal are you thinking of?
parrot
 - ◆ Type in a question to disitinguish
 - ◆ a parrot and a ostrich:
Can it fly?
 - ◆ What will be an answer for a parrot?
yes
 - ◆ Does it live in water?(yes,no)
yes
 - ◆ Is it a fish?(yes,no)
yes
 - ◆ I've guessed!
- ...

Writing a variable you may use slices and selections:
KNOWLEDGE[N]:= ATTEMPT \ see Example 8

Example 9.

```

proc REPLACE ( <=WHERE, =>WHAT, =>BY_WHAT )
    \ replacement in a text
    \ <=WHERE - in-out-parameter,
    \ =>WHAT, =>BY_WHAT - in-parameters

N:= 1
do
    K:= index (WHAT, WHERE [N:]) \ endless loop
    \ search for next pattern
    if K = 0 then
        exit \ loop exit
    fi
    K:= N+K-1
    WHERE [K:K+#WHAT-1]:= BY_WHAT \ replace a pattern
    \ by the new text
    N:= K + #BY_WHAT
od
end

```

```

FAIRYTAIL:= "Somebody said," + lf + """Bother!"" + lf +
    "And then he said," + lf + """Oh,deary me!"" + lf +
    "Somebody sobbed, ""Oh,deary me!"" + lf +
    "And went back to bed."
REPLACE (<=FAIRYTAIL, =>"Somebody", =>"The King")
    \ <=FAIRYTAIL - actual in-out-parameter
    \ =>"Somebody", =>"The King" -
    \ actual in-parameters

```

output: FAIRYTAIL

- ◆ The King said,
- ◆ "Bother!"
- ◆ And then he said,
- ◆ "Oh,deary me!"
- ◆ The King sobbed, "Oh,deary me!"
- ◆ And went back to bed.

Statements

Statements: assignment, procedure call, conditional, cases, loops, output, input, loop exit, return from procedure or function.

Almost all statements were used in the examples above.
One more example:

Example 10.

```

proc GAME ()
extern: GAME_START, YOUR_TURN, MY_TURN, WRONG
intern: RIDDLE, MIN, MAX, YOU_GUESSED, I_GUESSED
output: "Who'll guess first?"
GAME_START()
do
  output: "Your turn:"; YOUR_TURN()
  output: "My turn:"; MY_TURN()
  case
    when YOU_GUESSED and I_GUESSED: \ select a true condition
      output: "Nobody won!"
    when YOU_GUESSED:
      output: "You won!"
    when I_GUESSED:
      output: "I won!"
  esac
  if YOU_GUESSED or I_GUESSED then
    return \ procedure return
  fi
until WRONG() \ end of loop if
\ WRONG() = yes
output: "You've missed somewhere!"
output: "I don't want to play anymore."
end

proc GAME_START ()
extern: RIDDLE, MIN, MAX, YOU_GUESSED, I_GUESSED
output: "Think of a number from 1 to 1000"
output: "and try to guess mine"
RIDDLE := int_rand(1000)
MIN := 1
MAX := 1001
YOU_GUESSED := no
I_GUESSED := no
end

proc YOUR_TURN ()
extern: RIDDLE, YOU_GUESSED
intern: ATTEMPT
input: ATTEMPT
case
  when ATTEMPT > RIDDLE:
    output: "My number is less!"
  when ATTEMPT < RIDDLE:
    output: "My number is greater!"
esac
YOU_GUESSED := (ATTEMPT = RIDDLE)
end

```

```

proc MY_TURN ()
extern: MIN, MAX, I_GUESSED
intern: ATTEMPT, ANSWER, ANSWER_CORRECT
  ATTEMPT:= (MIN+MAX) // 2
  output: "Is it ", ATTEMPT, "?"
  do
    output: "Answers: =, > (riddle> ", ATTEMPT,
           " ), < (riddle< ", ATTEMPT, " )"
    input text: ANSWER
    ANSWER_CORRECT:= yes
    case ANSWER
      when ">": MIN := ATTEMPT
      when "<": MAX := ATTEMPT
      when "=": I_GUESSED:= yes
      else ANSWER_CORRECT:= no
            output: "You did a mistake"
    esac
  until ANSWER_CORRECT
end

fun WRONG ()
extern: MAX, MIN
  return MIN+1=MAX
end

```

GAME()

- ◆ Who'll guess first?
- ◆ Think of a number from 1 to 1000
- ◆ and try to guess mine
- ◆ Your turn:
 - 512
- ◆ My number is less!
- ◆ My turn.
- ◆ Is it 500?
- ◆ Answers: =, > (riddle> 500), < (riddle< 500)
- <
- ◆ Your turn:
 - 256
- ◆ My number is greater!
- ◆ My turn.
- ◆ Is it 250?
- ◆ Answers: =, > (riddle> 250), < (riddle< 250)
- <
- ◆ Your turn:
 - 384
- ◆ My turn.
- ◆ Is it 125?
- ◆ Answers: =, > (riddle> 125), < (riddle< 125)
- >
- ◆ You won!

Procedures and functions

All the unlocal names of procedure (function) should be mentioned in the "extern" list. (Local names may be mentioned optionally in the "intern" list). Names that are used in a

procedure are known also in all procedures called from it (if they are marked as "extern" there), because the search of external names is being held dynamically through the chain of calls (see Example 10).

Procedure parameters:

in-:	=>WHAT	\ see Example 3
	PHRASE	\ see Example 4
	(arrow => is optional)	
in-out-:	<=WHERE	\ see Example 9
	<=KNOWLEDGE	\ see Example 3

Functions may have in-parameters only.

Modules and devices

Means of modules and devices planned:

- connecting a module enlarges the list of standard names (including standard procedures and functions names).
- connecting a module enlarges the set of simple statements.
- a device is a special form of module. A device may be communicated with by means of input and outputs:

```
output >>printer:
    "This text - to printer"
```

Input/output statements may contain mode specifications:

```
input text: QUESTION \ see Example 3
```

"text" is a keyword for the KEYBOARD device.

References

1. Звенигородский Г.А. Первые уроки программирования/ Под ред. А.П.Ершова. - М.:Наука,1985. - 298 с. - (Б-чка "Квант". Вып. 41).
2. Программная система "Школьница" и ее реализация на персональной ЭВМ/Г.А.Звенигородский и др.// Микропроцессорные средства в системах. - 1984. - № 1. - с.50-55.

B) RAPIRA. FORMAL LANGUAGE SPECIFICATION
(modules, devices and
exception handling excluded)

SYNTAX (24.09.87)

CHARACTER SET:

- letters;
- digits;
- special characters:
 , ; : ; () [] < > =
 + - * / % # _ " \ ^ >
- extra characters.

SEPARATOR ::= (space ::= ' ') ;
 (carriage_return ::= '␣') ;
 (comment ::= '\{ character } '␣')

LEXEME ::= specsymbol ;
 keyword ;
 name ;
 unsigned_int ;
 unsigned_real ;
 text

specsymbol ::= ; , : ; () [] < >
 := => < > <= >= = /#
 + - * / // /% **

keyword ::= \ In the Russian variants:

and	\	и
call	\	вызов
case	\	выбор
do	\	цикл
else	\	иначе
end	\	концы
esac	\	все
exit	\	выход
extern	\	чужие
fi	\	все
for	\	для
from	\	от
fun	\	функ
if	\	если
input	\	ввод
intern	\	свой
nlf	\	впс
not	\	не
od	\	кц
or	\	или
output	\	вывод
proc	\	проц
repeat	\	повтор
return	\	возврат
step	\	шаг
text	\	текста

~ except '␣'

```

then          \  то
to            \  до
until        \  км по
when         \  при
while        \  пока

name ::=      letter ( letter | '_' | digit )

int ::=      [ '+' | '-' ] unsigned_int

unsigned_int ::= digit ( digit )

real ::=     [ '+' | '-' ] unsigned_real

unsigned_real ::=      digit ( digit )
                      ( '.' digit | digit ) [ 'e' int ] ;
                      'e' int

text ::=     ''' { character } '''

CONDITION ::= EXPRESSION ::=
  expr_a { [ 'and' | 'or' ] expr_a }

expr_a ::=
  [ 'not' ]
  expr_b
  ( ( '>' | '<' | '>=' | '<=' | '=' | '/=' )
    expr_b )

expr_b ::=
  [ '+' | '-' ]
  expr_c
  ( ( '*' | '/' | '/' | '/' | '/' | '+' | '-' )
    expr_c )

expr_c ::=
  [ '*' ]

( name ;
  text ;
  unsigned_int ;
  unsigned_real ;
  procedure ;
  function ;
  ( seq_construct ::= '<' [ 1_expr ] '>' ;
    '( expression )' )

  ( [ index_expr ::= '[ 1_expr ]' ;
    '[ [ expression ] ':'
      [ expression ] ]' ] )
  actual_fun_param )

1_expr ::=      expression < ',' expression >

```

''' except ''' and '<'

```

actual_func_param ::=
    '(' [ [ '='>' ] expression
        ( ',' [ '='>' ] expression ) ] ')'

PROCEDURE ::=
    'proc' [ name ]
    '(' [ [ '='>' ; '=' ] name
        ( ',' [ '='>' ; '<=' ] name ) ] ')' ;;
    [ declarations ]
    stmts
    'end'

;; ::=
    ( ';' ; '*' )

declarations ::= ( 'extern' ':' name ( ',' name ) ;;
    [ 'intern' ':' name ( ',' name ) ;; ] ) ;
    ( 'intern' ':' name ( ',' name ) ;;
    [ 'extern' ':' name ( ',' name ) ;; ] )

FUNCTION ::=
    'fun' [ name ]
    '(' [ [ '='>' ] name
        ( ',' [ '='>' ] name ) ] ')' ;;
    [ declarations ]
    stmts
    'end'

DIALOG_UNIT ::= ( statement ; routine_definition ) ;;

stmts ::= ( statement ;; )

routine_definition ::= ( procedure ; function )

STATEMENT ::=
    ( ( skip ::= '' ) ;
    assignment ;
    call ;
    if ;
    case ;
    loop ;
    output ;
    input ;
    ( loop_exit ::= 'exit' ) ;
    ( proc_return ::= 'return' ) ;
    ( fun_return ::= 'return' expression ) )

assignment ::=
    ( variable ::= name ( index_expr ) ) '=' expression

call ::= ( 'call' expression ; name ) actual_proc_param

actual_proc_param ::=
    '(' [ ( '<=' variable ;
        [ '='>' ] expression )
        ( ',' ( '<=' variable ;
            [ '='>' ] expression ) ) ] ')'

```

```

if ::= 'if' condition
      'then' stmts
      [ 'else' stmts ]
      'fi'

case ::= (
  'case' expression
  ( 'when' expression ( ',' expression ) ':' stmts )
  {
  'case'
  ( 'when' condition ':' stmts )
  }
  [ 'else' stmts ]
  'esac'

loop ::= [ ( 'for' name [ 'from' expression ]
            [ 'to' expression ]
            [ 'step' expression ] ;
          'repeat' expression ) ]
        [ 'while' condition ]
        'do'
          stmts
        ( 'od' | 'until' condition )

output ::= 'output' [ 'nif' ]
          [ ':' expression ( ',' expression ) ]

input ::= 'input' [ 'text' ] ':' variable ( ',' variable )

```

Context restrictions:

- "loop_exit" statement may appear in loops only;
- "proc_return" statement may appear in procedures only;
- "fun_return" statement may appear in functions only;
- procedures and functions: the procedure (function) name and all the names in "extern", "intern" and parameter lists must be different;
- the routine name in "routine_definition" is obligatory (see "DIALOG_UNIT").

SEMANTICS (24.09.87)

NAMES AND VARIABLES.

Any language object may act as a variable value.
 There exist standard names (see below).
 See procedures for names scoping rule.
 If a name is not standard in the given scope and has not
 been assigned to, then it has an empty value (the 'empty'
 object as a value).

OBJECTS.

object:	object_denotation:=	
empty	'empty'	
logical	'yes' 'no'	
procedure	procedure	/* see syntax */
function	function	/* see syntax */
integer	int	
	unsigned_int	/* see syntax */
real	real	
	unsigned_real	/* see syntax */
text	text	/* see syntax */
sequence	'[' [object_denotation	
	{ ',' object_denotation }] ']'	

Text.

"" represents "" in the text denotation.

Sequence.

Sequence is an ordered set of arbitrary objects.

Procedures and functions.

Procedure (function) name is local inside the procedure (function).

Parameters : in- : ['>'] name
 and in-out-: '<' name .

Functions have in-parameters only.

Names declaration: undeclared non standard name is considered to be local ("intern").

Names scoping rule: in the dynamic order of calls.

Functions may be simulated by procedures (not regarded here).

OPERATION PRIORITIES (descending):

- ◀ ▶ () [] [:] ()-function call
- #
- **
- * / // %
- + - (unary and binary)
- > < >= <=
- = /=
- not
- and
- or

STATEMENTS AND OPERATIONS.

LEGEND:

$a, b, c, z, b_j, f, f_j, w, k, h, p, u, x$	expressions
i	- name
v	- variable
S, S_j	- statements

BASIC SUBSET:

Operations:

$p(\dots)$	- p is function	- function call.
$\langle \dots \rangle$		- sequence constructor.
$\#k$	- k is text or sequence	- length.
$k [a]$	- k is text or sequence a is integer $1 \leq a \leq \#k$.	- selection. result: a-th sequence element (a-th character in text).
$a \times \times b$	- a, b are numbers b is real; b is integer;	- exponentiation. $a \times \times b = \exp(\ln(a) \times b)$ see partial simulation.
$a * b$	- a, b are numbers; a is text or sequence, b is integer; b is text or sequence, a is integer	- multiplication. multiply by integer see partial simulation.
a / b	- a, b are numbers a, b are integers; otherwise:	- division. result: integer or real, real.
$- a$	- a is number	- negated value.
$a + b$	- a, b are numbers; a, b are texts; a, b are sequences	- numbers addition, sequence and text concatenation.
$a > b$	- a, b are numbers	- greater ? (yes/no),
$a = b$	- a, b are any objects	- equal ? if a, b are sequences - see partial simulation.

Statements:

$i := x$	- x is any object	- assign the value of x to i .
call $p(\dots)$	- p is procedure	- procedure call.
if f then S fi	- f is logical!	- conditional statement.
output: ...		- output of objects: object denotations are displayed at the screen (except texts, which are displayed unquoted).
output n!): ...		- output with no line feed (temporary here: due to device).


```

repeat M do
  RES:= RES+B
od
if A>=0 then return RES
else return -RES \ for text and
fi \ sequences: error
end
(a, b)
3) exponentiation to integer powers:
a ** b =
fun EXP_TO_INT (A, B)
  if A=0 and B=0 then
    error()
  fi
  if B>0 then N:= B
  else N:= -B
  fi
  P:= 1
  while N>0 do
    if N/2 /= 0 then
      P:= P*A
    fi
    A:= A*A
    N:= N/2
  od
  if B>0 then return P
  else return 1/P
  fi
end
(a, b)

```

SIMULATED:

Statements:

1) Procedure call:

```
i ( ... ) = call i ( ... ) \ i is procedure
```

2) Conditional:

```
if f then S1
else S2
fi =
```

```
call
```

```
proc IF_THEN_ELSE (F)
  extern: ... \ all the names from S1,S2,f
  if F then
    S1
  return
  fi
  S2
end
```

```
(f)
```

```
\ f is logical
```

3) Case:

```
case
  when f1: S1
  when f2: S2
  ...
  when fn: Sn
  else S0
esac =
```

```

if      f1 then S1
else if f2 then S2
else ...
      if fn then Sn
else          S0
fi fi ... fi

```

```

Case a
  when b1: S1
  when b2: S2
  ...
  when bn: Sn
  else    S0
esac

≡

call
  proc CASE_EXPR (A)
  extern: ... \ all the names from S1,b1,a
  if      A = b1 then S1
  else if A = b2 then S2
  else ...
        if A = bn then Sn
  else          S0
  fi fi ... fi
end
(a) \ a is any object

```

```

Case a
  when b1, b2: S1
  ...
esac

≡

Case a
  when b1: S1
  when b2: S1
  ...
esac

```

Operations:
4) not f

```

≡
fun LOG_NOT (F)
  if F then return yes
  else    return no
fi
end
(f)

```

5) f and g

```

≡
fun LOG_AND ( )
  extern: ... \ all the names from f,g
  if not f then
    return no
  else
    if g then return yes
    else    return no
  fi
fi
end
(f,g) \ f,g are logical

```

```

6) f or g      ≡      fun LOG_OR ( f
                    externs ...           \ all the names from f,g
                    if f then
                                return yes
                    else
                        if g then return yes
                        else      return no
                    fi
                    fi
                    end

7) +a          ≡      ( )           \ f,g are logical
8) a < b      ≡      @+a
9) a < b      ≡      a + (-b)

fun LATER (A, B)
    return not(A>B or A=B)
end

10) a >= b    ≡      (a, b)
11) a <= b    ≡      not(a<b)
12) a /= b    ≡      not(a=b)

Statements:
13) Loop, loop exit:
    do
        S
    od
    ≡
    call
    proc ENDLESS ( )
        externs ...           \ all the names from S
        S
        ENDLESS()
    end
    ( )

    exit      ≡
    return    \ it loops as above

while f do
    S
until g      ≡
do
    if not(f) then exit fi
    S
    if g then exit fi
od

while f do
    S
od
≡
while f do
    S
until no

```

```

do
  S
until f
≡
while yes do
  S
until f

repeat n
  while f do
    S
  until g
≡
call
  proc REPEAT_LOOP (N)
  externs: ... \ all the names from S, f, g
  if not(is_int(N) and N>=0) then
    error()
  fi
  while N>0 and f do
    S
    N:= N-1
  until g
end
(a) \ n is integer

repeat n do
  S
od
≡
repeat n
  while yes do
    S
  until no

for i from a step c
  while f do
    S
  until g
≡
call
  proc FOR_FROM_STEP_LOOP (A, C)
  externs: ... \ i, all the names from S, f, g
  if not(is_int(A) or is_real(A)) then
    error()
  fi
  if not(is_int(C) or is_real(C)) then
    error()
  fi
  if C=0 then error() fi
  i:= a
  protect i
  while f do
    S
    unprotect i
    i:= i+C
    protect i
  until g
  unprotect i
end
(a, c) \ a, c are numbers

```

```

for i from a step c do
  S
od
  =
  for i from a step c
    while yes do
      S
    until no

for i step c do
  S
od
  =
  for i from 1 step c do
    S
  od

for i from a do
  S
od
  =
  for i from a step 1 do
    S
  od

for i from a to b step c
  while f do
    S
until S
  =
  call
  proc FOR_FROM_TO_STEP_LOOP (A, B, C)
    extern: ... \ i, all the names from S,f,g
    if not(is_int(B) or is_real(B)) then
      error()
    fi
    for i from A step C
      while ((B-i)*C >= 0) and f do
        S
      until g
    end
  end
  (a, b, c) \ a, b, c are numbers

for i from a to b step c do
  S
od
  =
  for i from a to S step c
    while yes do
      S
    until no

```

Operations:

14) $k [a, b, \dots, z]$ $=$ $k[a][b] \dots [z]$

15) $k [a : b]$

$k [a : b]$ \equiv

```

fun SLICE (K, A, B)
  if not is_int(B) then
    error()
  fi
  if (B-A+1 < 0) then
    error()
  fi

```

```

if is_text(K) then
  H:= ""
  for IND from 1 to B do
    H:= H+K[IND]
  od
else
  H:= 4
  for IND from A to B do
    H:= H + 4 K[IND]
  od
fi
return H
end
      (k, a, b)      \ k is text or sequence,
                        \ a, b are integers

k [ 1 : b ] =      k[1:b]
k [ : ] =      k
k [ a : ] =
fun Tail (k, A)
  return K[A:#K]
end
      (k, a)      \ k is text or sequence,
                        \ a is integer

16) a // b
=
fun INT_DIVISION (A, B)
  if not(is_int(A) and is_int(B)) then
    error()
  fi
  if B<=0 then
    error()
  fi
  RES:= 0
  if A>=0 then
    while A>=B do
      A:= A-B
      RES:= RES+1
    od
  else
    while (-A) >= B do
      A:= A+B
      RES:= RES-1
    od
  if A/=0 then RES:= RES-1 fi
  fi
  return RES
end
      (a, b)      \ a, b are integers

17) a %% b
=
fun REMAINDER (A, B)
  return A - A//B*B
end
      (a, b)      \ a, b are integers

```

Statements:

```

18) v [ a : b ] := h      ≡
    call
      proc SLICE_ASSIGN (<=K, A, B, H)
        if (B-A+1 < 0) then
          error()
        fi
        K:= K[A-1] + H + K[B+1:]
      end
    (v, a, b, h) \ (v,h are sequences;
                  \ v,h are texts ),
                  \ a, b are integers

```

```

19) v [ a ] := x        ≡
    call
      proc SEL_ASSIGN (<= K, A, X)
        if is_text(K) then
          if #X /= 1 then
            error()
          fi
          K[A:A]:= X
        else
          K[A:A]:= ◀ X ▶
        fi
      end
    (v, a, x)
    \ (v is sequence: any x ;
    \ v is text: x - text of length 1),
    \ a is integer

```

Definitions:

```

20) procedure definition ( see "DIALOG_UNIT" ):
    proc i
      ...
    end
    ≡
    i:= proc i
      ...
    end

```

```

21) function definition ( see "DIALOG_UNIT" ):
    fun i
      ...
    end
    ≡
    i:= fun i
      ...
    end

```

STANDARD NAMES (24.09.87)

\ In the Russian variant:

constants:		
empty		\ пусто
yes		\ да
no		\ нет
lf	(line feed, more precisely: newline, lf = "\n")	\ лс
pi	(pi = 3.14...)	\ пи, pi
functions:		
abs(x)	- absolute value of number x	\ abs
sign(x)	- number sign of x (-1, 0, +1)	\ sign
sqrt(x)	- square root of number x	\ корень, sqrt
entier(x)	- the largest integer, that is less than x (entier (5.9) = 5 ; entier (-5.9) = -6 ; entier (5.4) = 5 ; entier (-5.4) = -6)	\ целч
round(x)	- the nearest integer to x (round (5.9) = 6 ; round (-5.9) = -6 ; round (5.5) = 6 ; round (-5.5) = -6 ; round (5.4) = 5 ; round (-5.4) = -5)	\ окрч
rand(x)	- generator of random numbers in the interval: [0 - x] (x >= 0)	\ дсч
int_rand(x)	- generator of integer random numbers in the interval: [1 - x] (x >= 0)	\ цсч
index(E, S)	- index of the element E in the sequence S of the subtext E in the text K. result: index of the first occurrence or 0, if absent	\ индекс
is_empty(x),		\ тип_пуст
is_log(x),		\ тип_лог
is_int(x),		\ тип_цел
is_real(x),		\ тип_вещ
is_text(x),		\ тип_текст
is_seq(x),		\ тип_корт
is_proc(x),		\ тип_проц
is_fun(x)	- the result is 'yes' or 'no' according to the type of object x	\ тип_функ
graphic procedures (the values of arguments x1,y1,r are numbers):		
dot(x,y)	- dot,	\ тчк
line(x1,y1,x2,y2)	- line,	\ лин
cfar(x,y,r)	- circumference,	\ окр
color(cir)	- color setting (cir - integer number or 1-char text)	\ цвет

```

region(x,y,clr)      - painting of monocolored \ use
connected region by
the other color
((x,y) - coordinates of
a dot within
the region)

rect(x1,y1,x2,y2,clr)- painted rectangle with \ npxm
the diagonal
((x1,y1):(x2,y2))

triangle(x1,y1,x2,y2, \ tpr
x3,y3,clr)- painted triangle

circle(x,y,r,clr)   - circle \ xpyr

functions:
sin (x),             \ sin
cos (x),             \ cos
tg (x),              \ tg
arcsin(x),          \ arcsin
arctg (x),          \ arctg
exp (x),             \ exp
ln (x),              \ ln
lg (x),              \ lg

```

RAPIRA

Differences in implementation on "Korvet" and "Yamaha" in comparison with the language specification (24.09.87) (24.09.87)

At present moment only the Russian variant of Rapira on "Korvet" and "Yamaha" is available.

SYNTAX.

Character set.

The following Russian and Latin letters are equal (except in text constants):

K E H X B A P O C M T
e x a o c

Capital and small letters are equal (except in text constants).

Lexemes.

Keywords are reserved.

name ::= (letter | '_') (letter | '_' | digit)

text ::= "" (^ letter |
^ \ digit [digit [digit]] |
character^ |
" " ; ""

For specysymbols '<' and '>' there are equal '<X' and 'X>'.

Procedures and functions.

Denotations for procedure object and function object are excluded; there are procedure and function definition at the level of "DIALOG_UNIT" only.

SEMANTICS.

Objects.

In text:
^ letter denotes a control character;
^ \ number denotes a character with the given code.

Statements.

Input - not only for object_denotations, but for any expressions by commas, either.

"input text" and "output nif" are not implemented.

STANDARD NAMES.

No "tg" and "arcsin" functions;

GRAPHICS:

No "region" procedure.

Graphic window's boundary coordinates:

x-axis : 0 - 255;
y-axis : 0 - 191;

Colors are determined only by integers:

0-14 (14 - for reverse color)

" except "" and "<"