

Universidade Federal de Alagoas - UFAL  
Instituto de Computação - IC  
Graduação em Ciência da Computação  
Compiladores 2018.1

Antonio Manoel de Lima Neto

# **Especificação da Linguagem**

## Blaise

13 de Julho de 2018

# Sumário

<b>1.</b>	<b>Especificação Mínima da Linguagem</b>	<b>2</b>
1.1.	<b>Estrutura geral do programa</b>	<b>2</b>
1.1.1.	Formas de nome	3
1.1.2.	Variáveis	4
1.1.3.	Escopo	4
1.1.4.	Constantes nomeadas	4
1.2.	<b>Tipos de Dados</b>	<b>4</b>
1.3.	<b>Conjunto mínimo de operadores</b>	<b>5</b>
1.3.1.	Operadores e associatividade	5
1.3.2.	Precedência de operadores	5
1.4.	<b>Instruções</b>	<b>6</b>
1.5.	<b>Atribuição</b>	<b>7</b>
1.6.	<b>Subprogramas</b>	<b>8</b>
1.6.1.	Funções	8
1.6.2.	Procedimentos	8
1.7.	<b>Exemplos</b>	<b>8</b>
<b>2.</b>	<b>Especificação dos tokens</b>	<b>11</b>
2.1.	Enumeração da categoria dos tokens	11
2.2.	Expressões regulares dos terminais	11
2.3.	Tokens	11
2.4.	Palavras reservadas	12
2.5.	Delimitadores	13
2.6.	Operadores	13

# 1. Especificação Mínima da Linguagem

## 1.1. Estrutura geral do programa

Um programa consiste basicamente das seguintes partes:

```
program {nome do programa}
uses {nomes de bibliotecas separadas por ponto-e-vírgula (opcional)}
const {bloco de declaração de constantes globais}
var {bloco de declaração de variáveis globais}
(* comments *)
function {declaração de funções}
{variáveis locais}
begin
...
end;

procedure {declaração de procedimentos}
{variáveis locais}
begin
...
end;

begin {início do bloco do programa principal}
...
end. {fim do bloco do programa principal}
```

A segunda linha do programa **uses {lib}** é um comando de pré-processador que diz ao compilador para inserir **{lib}** antes de execução a compilação.

Todo bloco de declaração e comandos é cercado por um comando **begin** e um comando **end**. Contudo o comando **end** que encerra um bloco comum é seguido de um ponto-e-vírgula (;), enquanto que o comando **end** que indica o fim do programa é seguido de um ponto-final (.).

As linhas entre **(\*...\*)** representam comentários para leitura humana e serão ignoradas pelo compilador.

### 1.1.1. Formas de nome

Deve ter comprimento maior ou igual a 1, iniciado com caractere do alfabeto (letras de A à Z) e seguido por caracteres (A-Z), dígitos (0-9) ou o caractere de conexão *underscore* (**\_**). A grafia dos identificadores deve ser composta ignorando sensibilidade.

- Os identificadores podem possuir qualquer comprimento.
- Nenhum identificador deve ter a mesma grafia de qualquer outra palavra-símbolo ou keyword.
- As palavras-chave (*keywords*) são reservadas, e estão listadas a seguir:

<b>program</b>	<b>var</b>	<b>if</b>	<b>for</b>
<b>procedure</b>	<b>begin</b>	<b>then</b>	<b>to</b>
<b>function</b>	<b>end</b>	<b>do</b>	<b>while</b>
<b>integer</b>	<b>string</b>	<b>else</b>	<b>read</b>
<b>real</b>	<b>const</b>	<b>step</b>	<b>readln</b>
<b>char</b>	<b>boolean</b>	<b>exit</b>	<b>write</b>
<b>and</b>	<b>or</b>	<b>not</b>	<b>writeln</b>
<b>div</b>	<b>uses</b>	<b>type</b>	<b>of</b>

#### 1.1.2. Variáveis

A declaração e conversão devem ser explícitas. As variáveis são dinâmicas na pilha, e seguem as regras de compatibilidade por nome.

<b>Coerções suportadas</b>
Real para integer
Integer para real
Char para integer

#### 1.1.3. Escopo

O escopo é do tipo estático e os blocos são delimitados pelos termos **begin** e **end**. Identificadores podem ser locais ou globais e existem regras para acessar identificadores que não estão declarados localmente.

Além disso, também há devidas regras para lidar com procedimentos aninhados dentro de outros procedimentos. Qualquer coisa declarada em um bloco que contenha procedimentos aninhados é **não-local** aquele procedimento aninhado (e.g. Identificadores globais são não-locais com respeito a todos os outros blocos, exceto ao programa principal.) O processo de acessar um identificador declarado fora de seu bloco é considerado um **acesso não-local**.

As regras são como a seguir:

1. O escopo de um identificador inclui todas as declarações dentro do bloco que contém sua definição. Isto inclui blocos aninhados, exceto pelo que é denotado na regra 2.
2. O escopo de um identificador não se estende a nenhum bloco aninhado que contenha um identificador definido localmente com a mesma grafia.
3. O escopo de um parâmetro é idêntico ao escopo de uma variável local no mesmo bloco.

```
program scope_example;
var
    A1: Integer; (* variável global *)
procedure bloco1(A1: Integer); (* Previne acesso a variável global
A1 *)
    var B1, B2 : Integer; (* variável local ao bloco 1 *)
    begin;
    ..
    end;
procedure bloco2;
    var
        A1, (* Previne acesso a variável global A1 *)
        B2 : (* variável local ao bloco 2, não conflita com B2
do bloco 1 *)
            Integer;
    begin;
    ..
    end;
```

#### 1.1.4. Constantes nomeadas

São variáveis declaradas com a palavra reservada **const** onde o valor, uma vez definido, não pode ser alterado.

**const** <identifier> := <valor>;

## 1.2. Tipos de Dados

Tipo numérico: Há números inteiros e de ponto flutuante, variáveis desse tipo devem ser declaradas com as palavras reservadas *integer* e *real*, respectivamente. As operações suportadas são: adição (+), subtração (-), multiplicação(\*) e divisão (/).

Já os caracteres utilizam a codificação ASCII e devem ser declaradas com a palavra reservada *char*. Cadeias de caracteres são definidas como na linguagem Pascal, utilizando arranjo unidimensional.

### 1.3. Conjunto mínimo de operadores

#### 1.3.1. Operadores e associatividade

##### Operadores aritméticos

Ordem	Operação	Operador	Associatividade	Operandos	Resultado
1º	Parênteses	( )	Não associativo	-	-
2º	Multiplicação	*	Esquerda	integer ou real	integer ou real
2º	Divisão	/	Esquerda	integer ou real	real
3º	Adição	+	Esquerda	integer ou real	integer ou real
3º	Subtração	-	Esquerda	integer ou real	integer ou real

##### Operadores relacionais

Operação	Operador	Associatividade	Operandos	Resultado
Menor que	<	Não associativo	integer, real, array, char, string	booleano
Maior que	>	Não associativo	integer, real, array, char, string	booleano
Menor ou igual que	<=	Não associativo	integer, real, char, array, string	booleano
Maior ou igual que	>=	Não associativo	integer, real, array, char, string	booleano
Igual	=	Não associativo	integer, real, booleano, char, array, string	booleano
Diferente	<>	Não associativo	integer, real, booleano, char, array, string	booleano

## Operadores lógicos

Operação	Operador	Associatividade	Resultado
Conjunção	and	Esquerda	booleano
Disjunção	or	Esquerda	booleano
Negação	not	Direita	booleano

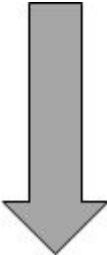
## Operadores binários

Operação	Operador	Associatividade	Operandos	Resultado
bitwise not	~	Não associativo	integer	integer
bitwise and	&	Não associativo	integer	integer
bitwise or		Não associativo	integer	integer

## Concatenação de Cadeias de Caracteres

Operação	Operador	Associatividade	Operandos	Resultado
concatenação	+	Esquerda	strings	string

### 1.3.2. Precedência de operadores

Precedência	Operador
Mais Alta	~, not
	*, /, div
	- (unário)
	+, - (binário)
	<, >, <=, >=, =, <>
	&, and, or
Mais baixa	~, + (operador de concatenação)

## 1.4. Instruções

Os arranjos unidimensionais devem ser declarados de acordo com a seguinte sintaxe:

*<variável>: array [<valor-inicial>..*valor-final*>] of <tipo>;*

A estrutura condicional mais simples é a condicional de uma via, onde a condição é um booleano ou condição relacional, e tem a sintaxe a seguir:

*if condição then smt;*

Já a estrutura condicional de duas é da seguinte forma:

*if condição then smt1 else smt2;*

A estrutura iterativa com controle lógico permite a repetição de computações até a condição ser satisfeita. A validação é realizada ao início de cada iteração, e a sintaxe é da seguinte forma:

*while(condição) do smt;*

Na estrutura de laço controlada por contador, o tipo da variável controladora do laço é um inteiro e a verificação da condição é feita uma vez a cada início da iteração. O comando *step* define o salto do contador a cada iteração, é um parâmetro opcional e implicitamente definido como 1 caso seja omitido.

*for <variável> := <valor-inicial> to <valor-final> step <valor> do*

O comando de entrada é responsável por receber dados inseridos pelo usuário e armazená-los em variáveis. Esse comando é representado pela palavra-chave **Read** ou **Readln**. Ambos desempenham a mesma função, a única diferença é que após a entrada de dados com **Read** o cursor fica na mesma linha, e no caso do **Readln** o cursor vai para a próxima linha. A mesma diferença é válida para **write** e **writeln**.

Função	Sintaxe
Entrada	<b>read</b> (<nome-da-variável>, <nome-da-variável>, ...);
Saída	<b>write</b> (<nome-da-variável> : largura: precisão);

Os tipos, por sua vez, serão representados da seguinte maneira:

## 1.5. Atribuição

O comando (**:=**) associativo à *direita* será utilizado para atribuição.



```
x := 200 + 100;  
y := num * x;
```

## 1.6. Subprogramas

Um subprograma é uma unidade/módulo que realiza uma tarefa em particular e pode ser invocado por outro subprograma ou o programa principal. Há dois tipos de subprogramas na linguagem:

### 1.6.1. Funções

A declaração de funções é dada através da palavra reservada **function** seguida do nome da função, parâmetros e tipo de retorno. A passagem de valores é feita através de parâmetros, e estes, por sua vez, são especificados com seus tipos dentro de parênteses separados por ponto-e-vírgula (;). Para indicar o tipo do retorno da função deve-se adicionar um último parâmetro após os parênteses com uma palavra reservada de tipo de dados. Um exemplo da sintaxe é dada a seguir;

```
function    nome-da-função(param1:    tipo;    param-n:    tipo):  
tipo-de-retorno;
```

### 1.6.2. Procedimentos

A única diferença de procedimentos para funções é que funções sempre terão um valor de retorno no fim, enquanto que procedimentos não permitem retorno de algum valor para o programa ou subprograma chamante.

```
procedure nome-do-procedimento(param1: tipo; param-n: tipo);
```

## 1.7. Exemplos

Alô Mundo

```
program Hello;  
begin  
    writeln ('Hello World.');
```

end.

## Série de Fibonacci

```
program Fibonacci;
procedure fib(n: Integer);
var
    u, v, w: Integer;
begin
    if n <= 1 then
        write('0');
        exit(0);
    if n = 2 then
        write('0, 1, 1');
        exit(1);
    u := 1;
    v := 1;

    write('0, 1, 1')
    while w+u < n do
    begin
        w := u + v;
        u := v;
        v := w;
        write(', ');
        write(w);
    end;
end;
(* Main program *)
begin
    var n: Integer;
    read(n);
    fib(n);
end.
```

## Shell Sort

```
program shellsort
function shellsort (vector: array of Integer, size: Integer) :
Integer
    begin
        var
            j, value : Integer;
            var gap := 1;

        while gap < size do
            begin
                gap = 3 * gap + 1;
            end;

        while gap > 1 do
            begin
                gap = gap / 3;
                for i := gap to size do
                    begin
                        value = vector[i];
                        j = i - gap;
                        while j >= 0 and value < vector[j] do
                            begin
                                vector[j + gap] = vector[j];
                                j = j - gap;
                            end;
                        vector[j + gap] = value;
                    end;
                end;
                shellsort := vector;
            end;

        procedure echoArray (vector : array of Integer, size: Integer)
        begin
            var i : Integer;
            for i := 0 to size do
                begin
                    write(vector[i])
                end;
            end;

        begin
            var i, n : Integer;
            read(n);
            type int_vector = array[0..n] of Integer;
            var vector: int_vector;
            for i := 0 to 10 do
                begin
                    read(vector[i]);
                end;
            end;
```

```

    echoArray(vector, n);
    array = shellsort(vector, n);
    echoArray(vector, n);
end.

```

## 2. Especificação dos tokens

### 2.1. Linguagem da implementação dos analisadores

Os analisadores léxico e sintático serão implementados na linguagem Python.

### 2.2. Enumeração dos tokens

```

from enum import Enum

class Tokens(Enum):
    IDENTIFIER = 1
    CONST_INT = 2
    CONST_REAL = 3
    CONST_BOOL = 4
    CONST_CHAR = 5
    CONST_STR = 6
    INTEGER = 7
    REAL = 8
    CHAR = 9
    STRING = 10
    BOOL = 11
    USES = 12
    FUNCTION = 13
    PROCEDURE = 14
    BEGIN = 15
    END = 16
    IF = 17
    THEN = 18
    ELSE = 19
    FOR = 20
    TO = 21
    STEP = 22
    WHILE = 23
    DO = 24

    READ = 25
    READLN = 26
    WRITE = 27
    WRITELN = 28
    AND = 29
    OR = 30
    NOT = 31
    DIV = 32
    OPEN_PAR = 33
    CLOSE_PAR = 34
    OPEN_BRA = 35
    CLOSE_BRA = 36
    SEMICOLON = 37
    COLON = 38
    QUOTE = 39
    OPEN_COM = 40
    CLOSE_COM = 41
    OP_ART_MU = 42
    OP_ART_AD = 43
    OP_RELAT = 44
    ATRIB = 45
    UNARY = 46
    BIT_NOT = 47
    BIT_AND = 48
    BIT_NOR = 49

```

### 2.3. Tokens

ID	Token	Expressão Regular
1	IDENTIFIER	<code>^([A-Za-z_][[:alpha:]]*)\$</code>
2	CONST_INT	<code>^[[:digit:]]*\$</code>
3	CONST_FLOAT	<code>^[[:digit:]]*\.\?[[:digit:]]+([eE][-+]?[[:digit:]]*)?\$</code>

4	CONST_BOOL	^('true' "false")\$
5	CONST_CHAR	^("[[:alpha:]]{1}")\$
6	CONST_STRING	^("[[:alpha:]]*")\$

## 2.4. Palavras Reservadas

ID	Token	Expressão Regular
7	INTEGER	^'integer'\$
8	REAL	^'real'\$
9	BOOL	^'bool'\$
10	CHAR	^'char'\$
11	STRING	^'string'\$
12	USES	^'uses'\$
13	FUNCTION	^'function'\$
14	PROCEDURE	^'procedure'\$
15	BEGIN	^'begin'\$
16	END	^'end'\$
17	IF	^'if'\$
18	THEN	^'then'\$
19	ELSE	'else'
20	FOR	'for'
21	TO	'to'
22	STEP	'step'
23	WHILE	'while'
24	DO	'do'

25	READ	'read'
26	READLN	'readln'
27	WRITE	'write'
28	WRITELN	'writeln'
29	AND	'and'
30	OR	'or'
31	NOT	'not'
32	DIV	'div'

## 2.5. Delimitadores

ID	Token	Expressão Regular
33	OPEN_PAR	'\('
34	CLOSE_PAR	'\).'
35	OPEN_BRA	'\[
36	CLOSE_BRA	'\]
37	SEMICOLON	'\;'
38	COLON	'\:'
38	QUOTE	'\"'
40	OPEN_COM	'\('
41	CLOSE_COM	'\*)'

## 2.6. Operadores

ID	Token	Expressão Regular
42	OP_ARIT_MU	'\* \/'
43	OP_ARIT_AD	'\+ \-'
44	OP_RELAT	'>=' <=' > < ' <>'

45	ATRI	' $\vdash$ '
46	UNARY	' $\neg$ '
47	BIT_NOT	' $\sim$ '
48	BIT_AND	' $\&$ '
49	BIT_OR	' $\vee$ '