

Universidade Federal de Alagoas - UFAL
Instituto de Computação - IC
Graduação em Ciência da Computação
Compiladores 2018.2

Antonio Manoel de Lima Neto

Especificação da Linguagem

Blaise

27 de Março de 2019

Sumário

1.	Especificação Mínima da Linguagem	3
1.1.	Estrutura geral do programa	3
1.1.1.	Escopo	3
1.2.	Tipos de Dados	4
1.2.1.	Constantes literais dos tipos	4
1.2.2.	Arranjos unidimensionais	5
1.3.	Conjunto mínimo de operadores	6
1.3.1.	Operadores e associatividade	6
1.3.2.	Precedência de operadores	6
1.4.	Instruções	8
1.4.1.	Condicional de uma via	8
1.4.2.	Condicional de duas vias	8
1.4.3.	Estrutura iterativa por controle lógico	9
1.4.4.	Estrutura iterativa por contador	9
1.4.5.	Entrada e saída	10
1.5.	Atribuição	10
1.6.	Subprogramas	10
1.6.1.	Funções	10
1.6.2.	Procedimentos	10
1.7.	Exemplos	11
2.	Especificação dos tokens	11
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	14
3.	Definição da gramática	14

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}
(* multi-line

comment *)
function {declaração de funções}
{variáveis locais}
begin //single-line comment
...
end;

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

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

Todo bloco de comandos é cercado pelas palavras-chave (*keywords*) **begin** e **end**, contudo o **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 (.)**.

1.1.1. 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;

```

1.1.2. Formas de nome

Os nomes podem possuir qualquer comprimento, não são case-sensitive e não podem assumir os nomes de palavras reservadas listadas a seguir:

program	var	if	for
procedure	begin	then	to
function	end	do	while
integer	string	else	read
real	const	step	of

char	boolean	exit	write
and	or	not	writeln
div	uses	type	

1.2. Tipos de Dados

- **Numérico:** Números podem ser inteiros e de ponto flutuante, variáveis desse tipo devem ser declaradas com as palavras reservadas *integer* ou *real*, respectivamente. As operações suportadas são: adição (+), subtração (- binário), multiplicação(*), divisão (/) e negação (- unário).
- **Booleano:** Assume valores lógicos representados pelas keywords *TRUE* e *FALSE*.
- **Char/String:** Os caracteres utilizam a codificação ASCII e devem ser declaradas com a palavra reservada *char*. Cadeias de caracteres são declaradas como um arranjo de caracteres e possuem dimensão fixa.
- **String:** Outra sintaxe para criar um arranjo de caracteres onde a dimensão da cadeia é dinâmica.
- **Array:** Os arranjos unidimensionais devem ser declarados especificando-se um intervalo e o tipo do arranjo.
- **Type:** A definição de um tipo pode ser para definir o tipo de arranjos ou servir como aliases para outro tipo.

Tipo	Forma	Exemplo de declaração
<i>Inteiro</i>	<i><variável>, ..., <variável>: integer;</i>	x, y, z: integer
<i>Ponto flutuante</i>	<i><variável>, ..., <variável>: real;</i>	x, y, z: real
<i>Caractere</i>	<i><variável>, ..., <variável>: char;</i>	c: char
<i>Booleano</i>	<i><variável>, ..., <variável>: boolean;</i>	choice: boolean

Arranjos	<code><variável>, ..., <variável>: array [<i><valor-inicial> .. <valor-final></i>] of <tipo>;</code>	<code>my_list: array[0..10] of string</code>
Cadeia de caractere	<code><variável>: array[0..<i>n</i>] of char;</code>	<code>c: array[0..10] of char</code>
String	<code><variável>, ..., <variável>: string;</code>	<code>str1, str2: string</code>

1.2.1. Constantes literais dos tipos

São variáveis declaradas com a palavra reservada **const** onde o valor, uma vez definido, não pode ser alterado. Somente constantes do tipo *integer*, *real*, *char* e *string* podem ser declaradas.

const <identifier> = <valor>;

Atribuições não são permitidas, logo, após a declaração a seguinte atribuição causará um erro de compilação.

<identifier> := <algum-valor>;

- **Literais do tipo Integer**

- Base decimal
- São compostos apenas por dígitos [0-9]
- Não podem começar com 0, exceto caso o valor seja o próprio 0.
- Podem ser precedidos do operador unário negativo (-)

`a = 2; // Constante do tipo integer`

- **Literais do tipo Real**

- Não podem começar com 0, exceto caso o valor seja o próprio 0.
- Podem conter *ponto* (.) para separar o valor inteiro da parte decimal.
- Podem conter o (*e*) para valores exponenciais, contudo não deve ser usado antes de um ponto.
- O exponencial deve ser seguido de (+) ou (-) para definir o sinal do expoente.
- Podem ser precedidos de valor unário.

`e = 2.7182818; // Constante do tipo real`

- **Literais do tipo Char**

- Deve estar entre aspas simples ou aspas duplas.
- Pode conter qualquer caractere comum ou especial, exceto aspas simples ou duplas.
- Deve ter comprimento de apenas um caractere.

c = '4'; // Constante do tipo char

- **Literais do tipo String**

- Deve estar entre aspas simples ou as duplas
- Pode conter qualquer caractere comum ou especial, exceto aspas simples ou duplas.
- Deve se estender apenas ao comprimento de uma única linha.

s = 'Hello world'; // Constante do tipo string

1.2.2. Arranjos unidimensionais (Array)

- Elementos de um arranjo devem ser do mesmo tipo.
- A dimensão do arranjo deve ser especificada na sua declaração.
- O índice começa e termina nos valores especificados na sua declaração.

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	integer ou 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	\geq	Não associativo	integer, real, char, array, string	booleano
Igual	$=$	Não associativo	integer, real, booleano, char, array, string	booleano
Diferente	\neq	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

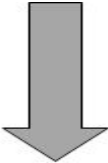
Concatenação de Cadeias de Caracteres

Operação	Operador	Associatividade	Operandos	Resultado
concatenação	+	Esquerda	strings, char/string e int, char/string e bool, char e string, char e char	string

Coerções suportadas
Real para integer
Integer para real
Char para integer
Integer para char
Char para string

1.3.2. Precedência de operadores

Precedência	Operador
Mais Alta	not, ~(unário)

	<code>*, /, and</code>
	<code>+, - (binário), or</code>
	<code><, >, <=, >=, =, <></code>
Mais baixa	<code>+</code> (operador de concatenação)

1.4. Instruções

- **Estrutura condicional de uma via:**

A estrutura condicional mais simples é a condicional de uma via, onde a expressão deve resultar em um booleano. Se a expressão for avaliada como verdadeira, então as instruções após a keyword ***then*** são executadas, caso contrário todo o bloco condicional é ignorado.

if expressão then instruções;

- **Estrutura condicional de duas vias:**

Na estrutura de duas vias, caso a expressão seja avaliada como falsa, o bloco de instruções após a keyword ***else*** serão executadas. Além disso, qualquer comando no bloco de instruções ***if*** e anterior ao ***else*** não devem conter ponto-e-vírgula.

if expressão then instruções-1 else instruções-2;

Em caso de ***if-then-else*** aninhados, a keyword ***else*** será pareada com o ***if*** acima mais próximo e que já não esteja pareado a um ***else***.

- **Estrutura iterativa com controle lógico:**

A estrutura iterativa com controle lógico permite a repetição de computações enquanto a expressão for avaliada como verdadeira. A validação é realizada ao início de cada iteração, portanto, é possível que as instruções nunca sejam executadas.

while(expressão) do instruções;

```
while(a < b) do
begin
write(a);
if a = c then:
break; //interrompe laço
end;
```

Além disso, as instruções podem ser interrompidas prematuramente através das palavras reservadas **break** ou **continue**, que podem ser utilizadas para pular para o fim da estrutura **while** ou pular a instrução seguinte a estrutura respectivamente.

- **Estrutura iterativa controlada por contador:**

Na estrutura de laço controlada por contador, um laço de repetição executa instruções um determinado número de vezes. O tipo da variável controladora do laço deve ser do tipo inteiro e a verificação da condição é feita uma vez a cada início de 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 instruções
```

```
for i := 0 to 6 do //step omitido
  begin
    write(i);
  end
```

As instruções são como as definidas na estrutura condicional, única ou bloco de instruções. O valor inicial é comparado ao valor final (caso o valor inicial seja maior, as instruções não serão executadas), após esta checagem as instruções após o **do** serão executadas.

- **Entrada e saída**

O comando de entrada (**read**) é responsável por receber dados inseridos pelo usuário e armazená-los em variáveis. Esta instrução espera ao menos um parâmetro de entrada, podendo também receber uma lista de parâmetros.

O comando de saída tem duas opções **write** e **writeln**. O **writeln** funciona da mesma maneira que o **write**, porém ele colocará um caractere de nova linha após o texto impresso. Ambos suportam múltiplas saídas separadas por vírgula, e com largura e precisão formatados através do operador dois-pontos (:).

Função	Sintaxe
Entrada	read (<nome-da-variável>, ... , <nome-da-variável>);
	read (a, b, c);
Saída	write (<nome-da-variável>:largura:precisão, ...);
	write ('a=', a, ' and b=', b); write ('a=', a:4:2);

1.5. Atribuição

O comando (\coloneqq) é utilizado para atribuição de valor ou resultado de expressão à uma variável. A expressão é avaliada e o resultado computado é associado a variável em questão. No caso de operações com tipos diferentes mas compatíveis, o tipo do resultado será definido pelo tipo da variável alvo.

```
<variável> := <expressão>;  
x := 200;  
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.

1.6.1. Funções

A declaração de funções é dada através da palavra reservada **function** seguida do nome da função, lista de parâmetros seguidos por tipo e separados por ponto-e-vírgula, e por fim, o tipo da variável retornada. A passagem de valores é feita através de uma lista de parâmetros, e estes, por sua vez, são especificados com seus tipos dentro do escopo do parênteses, e caso sejam do mesmo tipo podem ser agrupados com vírgula. O tipo do retorno da função deve-se adicionado como último parâmetro, após os parênteses, com uma palavra reservada de tipo de dados.

```
function nome-da-função(param-1, ... ,param-n: tipo; ... ; param-n: tipo):  
tipo-do-retorno;
```

Exemplo:

```
function soma(a, b: integer): real;  
begin  
    soma:= a + b;  
end
```

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(param-1, ... , param-m: tipo; ... ;param-n:  
tipo);
```

Exemplo:

```
procedure sayHello(name: string);  
begin  
    write("Hello ", name);  
end
```

1.7. Exemplos

Alô Mundo

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

```
end.
```

Série de Fibonacci

```
procedure fib(n: Integer);
var
  a, b, aux: Integer;

begin
  a := 0;
  b := 1;
  if n = 0 then
    writeln(a)
  else
    begin
      write(a);
      write(', ');
      writeln(b);
      write(', ');
      while a + b <= n do
        begin
          writeln(a + b);
          aux := a;
          a := b;
          b := b + aux;
        end;
    end;
end;

var n: Integer;
begin
  read(n);
  fib(n);
end.
```

Shell Sort

```
program shellsort;
const
  MaxN = 100;
type
  TArray = array [0..MaxN] of Integer;
var
  k, n : Integer;
  v, inv_v: TArray;

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

procedure ShellSort ( var A : TArray; N : Integer );
var
  i, j, step, tmp : Integer;
begin
  step:=N div 2;
  while step>0 do
    begin
      for i:=step to N do
        begin
          tmp:=A[i];
          j:=i;
          while (j >= step) and (A[j-step] > tmp) do
            begin
              A[j]:=A[j-step];
              j := j - step;
            end;
          A[j]:=tmp;
        end;
      step:=step div 2;
    end;
    echoArray(A, n);
  end;

begin
  read(n);
  for k := 0 to n do
    begin
      read(v[k]);
    end;
  write('desordenados: ');
```

```

    echoArray(v, n);
    writeln('');
    write('ordenados: ');
    shellsort(v, 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 Category(Enum):
    IDENTIFIER = 1
    CONST_INT = 2
    CONST_REAL = 3
    CONST_BOOL = 4
    CONST_CHAR = 5
    CONST_STR = 6
    INTEGER = 7
    REAL = 8
    BOOL = 9
    CHAR = 10
    STRING = 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
    WRITE = 26
    TYPE = 27
    OF = 28
    ARRAY = 29
    EXIT = 30
    PROGRAM = 31

    VAR = 32
    OPEN_PAR = 33
    CLOSE_PAR = 34
    OPEN_BRA = 35
    CLOSE_BRA = 36
    SEMICOLON = 37
    DECLR = 38
    QUOTE = 39
    OPEN_COM = 40
    CLOSE_COM = 41
    COMMA = 42
    THROUGH = 43
    MINUS_UNAR = 44
    DOT = 45
    NOT = 46
    MULT = 47
    DIV = 48
    MINUS = 49
    PLUS = 50
    OP_RELAT = 51
    ASSIGN = 52
    INVALID = 53
    EOF = 54
    WRITELN = 55
    CONST = 56
    AND = 57
    OR = 58
    BREAK = 59
    CONTINUE = 60

```


2.3. Tokens

Categoria	Expressão Regular
IDENTIFIER	([A-Za-z_][[:alnum:]]*)
CONST_INT	[[:digit:]]+
CONST_REAL	[[:digit:]]+\.?([[:digit:]]+([eE][+-]?[[:digit:]]*+)?)
CONST_BOOL	(TRUE FALSE)
CONST_CHAR	("[[:alpha:]]{1}")
CONST_STR	("[[:alpha:]][[:alpha:]]*")

2.4. Palavras Reservadas

Categoria	Lexema
INTEGER	'integer'\$
REAL	'real'\$
BOOL	'bool'\$
CHAR	'char'\$
STRING	'string'\$
USES	^'uses'
FUNCTION	^'function'
PROCEDURE	^'procedure'
BEGIN	^'begin'\$
END	^'end\$
IF	^'if'
THEN	'then'\$
ELSE	^'else'

FOR	^'for'
TO	'to'
STEP	'step'
WHILE	^'while'
DO	'do'\$
READ	^'read'
WRITE	^'write'
TYPE	^'type'
OF	'of'
ARRAY	'array'
EXIT	^'exit'
PROGRAM	^'program'
VAR	^'var'

2.5. Delimitadores

Categoria	Lexema
OPEN_PAR	'('
CLOSE_PAR)'
OPEN_BRA	'['
CLOSE_BRA	']'
SEMICOLON	','
DECLR	':'
QUOTE	'''
OPEN_COM	'(*'
CLOSE_COM	'*)'
COMMA	','

THROUGH	‘..’
DOT	‘.’

2.6. Operadores

Categoria	Lexema
NOT	'not'
MULT	'*'
DIV	'/'
PLUS	'+'
MINUS	'-'
OP_RELAT	'> = < > < <>'
ASSIGN	':='
MINUS_UNAR	'~'