# Especificação da Linguagem [nome?]

(Versão 0.4)

## Características da linguagem

- Imperativa;
- Fortemente tipada;
- Declaração explícita de variáveis;
- Vinculação estática de tipos;
- Sistema de escopo estático (léxico);
- Sensível à caixa (case-sensitive);

## Sistema de Tipos

A linguagem possui um sistema de tipo com duas classes: tipos primitivos e tipos agregados.

### Tipos primitivos

Os tipos primitivos são números inteiros, valores lógicos e strings, representados respectivamente pelos tipos `int`, `bool` e `string`.

### Tipos Agregados (arranjo)

O tipo agregado é um arranjo de algum tipo primitivo. Dessa forma, podemos ter as variantes: arranjo de inteiros, arranjo de lógicos e arranjo de strings.

### O tipo string

Dados do tipo string são constantes e não são indexados como na maioria das linguagens. O objetivo da existência do tipo string na linguagem é apenas fornecer uma maneira de apresentar (escrever) mensagens na tela.

## Especificação Léxica

### Identificadores

Chamamos de identificador qualquer nome criado pelo usuário da linguagem. Os identificadores seguem a mesma regra de formação da linguagem C:

- Devem iniciar com uma letra (minúscula ou maiúscula) ou um subtraço seguido de letras, subtraços ou dígitos entre 0 e 9.

Um identificador será expresso pelo símbolo `id` nas especificações sintáticas.

### Literais

Daremos o nome de literal a todo valor fixado no código. A linguagem possui representação de literais para seus três tipos primitivos.

#### Números

Os literais numéricos devem ser representados na base decimal e podem conter qualquer combinação de dígitos entre 0 e 9. Os números negativos não serão processados na fase léxica, mas sim na sintática e semântica. Dessa forma, o número `-42`, por exemplo, consiste de dois lexemas: `'-'` e `'42'`, e serão tratados como uma operação aritmética nas análises sintática e semântica.

#### Strings

Os literais string possuem a mesma regra de formação definida pela linguagem C.

Exemplo: `"isso é uma string!\n"`

#### Lógicos

Os literais lógicos verdadeiro e falso são representados pelos lexemas `true` e `false` respectivamente.

### Comentários

A linguagem possui apenas comentários de linha:

- Começam com `//` e seguem até o final da linha.

De forma geral, os comentários podem conter qualquer tipo de símbolo, inclusive os não permitidos pela linguagem. Os comentários devem ser processados corretamente pelo analisador léxico e em seguida descartados.

### Palavras reservadas e símbolos

As palavras reservadas e símbolos da linguagem são:

```
bool else false for if int read return skip stop string true var while write
( ) [ ] { } , ; + - * / % == != > >= < <= || && ! = += -= *= /= %= ? : "
```

## Especificação Sintática

### Programa

Um programa consiste de uma sequência não vazia de declarações de variáveis e subprogramas.

```
programa ::= dec {dec}
```

### Variáveis

Existem dois tipos de variáveis: as simples e as agregadas. Variáveis simples suportam apenas um único valor de um determinado tipo primitivo em um determinado momento.

Variáveis agregadas são de tipos agregados (arranjos), suportando mais de um valor de um mesmo tipo em um determinado momento.

**Exemplo:**

```
var a, b = 3, c = 2 + b: int;
var str1, str2 = "String 2": string;
var i, j = true: bool;
var x, v[10], z[3] = {1, 5, 8}: int;
```

**Declaração de Variáveis**

```
decVar ::= 'var' listaSpecVars ':' tipo ';'
listaSpecVars ::= specVar {',' specVar}
specVar ::= specVarSimples | specVarSimplesIni |
            specVarArranjo | specVarArranjoIni
```

Observe que a declaração de variáveis é indicada pela palavra reservada `var`. Observe também que múltiplas variáveis podem ser declaradas de uma vez e que elas podem ser inicializadas durante a declaração.

Na declaração de arranjos, o tamanho do mesmo deve ser especificado como um literal numérico.

## Subprogramas (procedimentos e funções)

A definição de procedimentos e funções possui uma sintaxe comum, exceto pela ausência do tipo de retorno para procedimentos. Diferentemente da linguagem C, por exemplo, não há separação entre declaração e definição de subprogramas, isto é, o subprograma é declarado durante sua própria definição.

**Exemplo:**

**Declaração de Procedimento**

```
def proc(y: int) {
    if (y < 0) {
        return;
    }
    x = 2 * y; // x é global!
}
```

**Declaração de Função**

```
def func(x[], y: int; z: bool): int {
    a = x[y-1]: int;
    return a + 1;
}
```

**Declaração de Subprogramas**

```
decSub ::= decProc | decFunc
```

**Declaração de procedimento**

```
decProc ::= 'def' id '(' [listaParâmetros] ')' bloco
```

**Declaração de função**

```
decFunc ::= 'def' id '(' [listaParâmetros] ')' ':' tipo bloco
```

**Lista de Parâmetros**

```
listaParâmetros ::= specParams {';' specParams}
```

```
specParams ::= param {',' param} ':' tipo
param ::= id | id '[]'
```

Parâmetros de tipo inteiro ou lógico são passados naturalmente por cópia e parâmetros de tipo arranjo ou string são passados naturalmente por referência.

## Comandos

Um comando pode ser um comando simples ou bloco de comandos.

```
comando ::= cmdSimples | bloco
```

A seguir são relacionados os comandos simples da linguagem:

- **Atribuição:**

  ```
  cmdAtrib ::= atrib ';'
  atrib ::= variável ('='|'+='|'-='|'*='|'/='|'%=') expressão
  ```

  O comando de atribuição avalia o valor da `expressão` e o armazena na `variável`. Uma atribuição somente pode ocorrer se a `variável` foi previamente declarada e se o tipo do resultado da `expressão` é o mesmo indicado na declaração da `variável`.

  As atribuições compostas devem ser traduzidas da seguinte maneira:

  ```
  var X= expressão -> var = var X expressão
  ```

- **Condicional If:**

  ```
  cmdIf ::= 'if' '(' expressão ')' comando ['else' comando]
  ```

  A estrutura condicional `if` é executada verificando o resultado da expressão de teste. Se ela resultar no valor `true`, apenas o primeiro comando será executado. Se a expressão resultar no valor `false`, caso a estrutura `else` esteja presente, apenas o segundo comando será executado.

- **Laço While:**

  ```
  cmdWhile ::= 'while' '(' expressão ')' comando
  ```

  O laço `while` inicia verificando o resultado da expressão de teste. Caso o valor seja `true`, o comando do seu corpo é executado e o laço volta a testar o valor da expressão de teste para a próxima iteração. Caso o valor seja `false`, a execução do laço é interrompida.

- **Laço For:**

  ```
  cmdFor ::= 'for' '(' atrib-ini ';' expressão ';' atrib-passo ')' comando
  ```

  O laço `for` inicia executando a atribuição de inicialização. A partir daí, antes de cada iteração, o resultado da expressão de teste é verificado. Se ele for `true`, o comando

corpo é executado e a atribuição de passo é executada em seguida, reiniciando o processo. Se antes de qualquer iteração o valor resultado pela expressão de teste for `false`, a execução do laço é interrompida.

- **Interrupção do laço:**

  ```
  cmdStop ::= 'stop' ';'
  ```

  O comando `stop` interrompe o laço mais próximo que o cerca. Ele só pode aparecer dentro do corpo de comandos de repetição `while` e `for`.

- **Salto de iteração do laço:**

  ```
  cmdSkip ::= 'skip' ';'
  ```

  O comando `skip` salta para a próxima iteração do laço mais próximo que o cerca, ignorando a execução dos comandos que o seguem dentro deste laço. Ele só pode aparecer dentro do corpo de comandos de repetição `while` e `for`.

- **Retorno de subprograma:**

  ```
  cmdReturn ::= 'return' [expressão] ';'
  ```

  O comando `return` encerra a execução do subprograma que o cerca retornando o valor resultado pela `expressão`. A expressão de retorno de uma função deve resultar em um valor do mesmo tipo para o qual a função foi definida. Funções devem obrigatoriamente conter pelo menos um comando `return`. Já procedimentos podem ou não conter comandos `return`. Caso o tenham, eles devem retornar nada: `return;` Como o programa principal é definido por meio de uma função, ele deve conter pelo menos um comando `return` e o valor retornado deve ser um número inteiro.

- **Chamada de procedimento:**

  ```
  cmdChamadaProc ::= id '(' [expressão {',' expressão}] ')' ';'
  ```

  Como a chamada de procedimentos não resulta em um valor, é necessário um comando para sua execução. A chamada de funções possui sintaxe semelhante, exceto por não ser um comando, e sim uma expressão.

- **Entrada Read:**

  ```
  cmdRead ::= 'read' variável ';'
  ```

- **Saída Write:**

  ```
  cmdWrite ::= 'write' expressão {',' expressão} ';'
  ```

## Bloco

Um bloco é uma sequência de (nenhuma ou várias) **declarações de subprogramas e variáveis** seguida de uma sequência de (nenhum ou vários) **comandos**. Um bloco é circundado por chaves **{ }**.

```
bloco ::= '{' {dec} {comando} '}'
```

## Expressão

Uma expressão pode conter valores dos três tipos definidos (inteiros, lógicos e strings), uso de variáveis, chamadas de função e outras expressões. Uma expressão pode estar cercada por parênteses e se relacionar a outras expressões por meio dos seguintes operadores:

| Precedência | Operador | Descrição | Associatividade |
|---|---|---|---|
| **1** | `-` | Negativo Unário | À direita |
| | `!` | Não lógico | |
| **2** | `* / %` | Multiplicação, divisão e resto | À esquerda |
| **3** | `+ -` | Adição e subtração | |
| **4** | `< <=` | Operadores relacionais < e ≤ respectivamente | |
| | `> >=` | Operadores relacionais > e ≥ respectivamente | |
| **5** | `== !=` | Operadores relacionais = e ≠ respectivamente | |
| **6** | `&&` | E lógico | |
| **7** | `\|\|` | OU lógico | |
| **8** | `? :` | Condicional ternário | À direita |

- O operador condicional ternário é formado da seguinte maneira:

  ```
  opTern ::= expressão-teste '?' expressão-então ':' expressão-senão
  ```

  A expressão `teste` é avaliada. Se o resultado for `true` a expressão `então` é resultada, caso contrário, a expressão `senão` é resultada. Dessa forma, o resultado desse operador é sempre uma expressão. O operador pode ser utilizado assim: `x = a > 0 ? a * 2 : a + 1;`

### Uso de variável

Como o uso de uma variável resulta no valor armazenado pela variável, todo uso de variável é uma expressão. Variáveis simples são usadas por meio do identificador (nome) associado a ela e variáveis compostas (arranjo) são usadas por meio do identificador e a posição numérica do elemento acessado.

```
variável ::= id | id '[' expressão ']'
```

Observe que a sintaxe do uso de variável não impede que uma variável declarada como simples seja utilizada como arranjo. Essa associação deve ser verificada na etapa de análise semântica.

# Especificação Semântica

## Programa

A última declaração deve ser obrigatoriamente a da rotina principal, pela qual se dará o início da execução do programa. Todas as declarações realizadas no programa (fora de qualquer subprograma) estão dentro do escopo global.

## Declaração

- Declaração de variáveis, funções e procedimentos são responsáveis por adicionar os símbolos envolvidos e suas vinculações na tabela de símbolos.
- Caso a declaração de uma variável contenha a inicialização da mesma, o tipo da expressão de inicialização deve ser o mesmo da variável.
- A última declaração global deve ser de uma função chamada "`main`" do tipo `int`.

## Comandos:

**If**

- A expressão condicional do comando `if` deve resultar em um valor do tipo lógico.

**While**

- A expressão condicional do comando `while` deve resultar em um valor do tipo lógico.

**For**

- As atribuições da inicialização e do passo devem ser analisadas como um comando de atribuição normal
- A expressão condicional deve resultar em um valor do tipo lógico.

**Stop**

- O comando `stop` só pode aparecer dentro de um comando de repetição (`while` ou `for`).

**Skip**

- O comando `skip` só pode aparecer dentro de um comando de repetição (`while` ou `for`).

**Return**

- Caso apareça dentro de uma função, o tipo da expressão de retorno deve ser o mesmo do retorno declarado da função. Caso apareça dentro de um procedimento, o comando `return` não pode ter expressão.

**Read**

- A variável utilizada no comando `read` deve estar declarada e visível no escopo atual.

**Write**

- Não há análise especial para o comando `write`.

**Chamada de procedimento**

- O procedimento chamado deve estar declarado e visível no escopo atual.
- O número de argumentos fornecidos deve ser o mesmo da declaração do procedimento.
- Os argumentos fornecidos devem ter a mesma ordem de tipo utilizada na declaração do procedimento.

**Atribuição**

- O lado esquerdo da atribuição deve ser uma variável declarada e visível no escopo atual (simples ou acesso de array)
- O lado direito deve ser uma expressão com tipo igual ao da variável do lado esquerdo da atribuição.

**Bloco**

- Define um novo escopo estático. O escopo é criado no início do bloco e finalizado no término do bloco.

## Expressões

**Aritmética (`+ - * / % neg`)**

- O(s) operando(s) devem ser do tipo inteiro. O tipo resultante é inteiro.

**Relacional (`> >= < <=`)**

- Os operandos devem ser do tipo inteiro. O tipo resultante é lógico.

**Igualdade (`== !=`)**

- Os operandos devem ser do mesmo tipo. O tipo resultante é lógico.

**Lógica (`&& || !`)**

- O(s) operando(s) devem ser do tipo lógico. O tipo resultante é lógico.

**Ternária**

- A expressão condicional deve resultar um valor do tipo lógico.
- As expressões consequente e alternativa devem possuir o mesmo tipo.

- O tipo resultante é o mesmo tipo da expressão consequente.

**Uso de variável**

- A variável deve estar declarada e visível no escopo atual. O tipo resultante é o tipo declarado da variável.

**Chamada de função:**

- Análise análoga à chamada de procedimento.
- O tipo resultante é igual ao tipo declarado da função.

## Exemplo de programa:

**Bubble-Sort**

```
var v[10]: int;

// Procedimento de ordenação por troca
// Observe como um parâmetro de arranjo é declarado

def bubblesort(v[]: int; n: int) {
    var i=0, j: int;
    var trocou = true: bool;
    while (i < n-1 && trocou) {
        trocou = false;
        for (j=0; j<(n-i-1); j++) {
            if (v[j] > v[j+1]) {
                var aux = v[j]: int;
                v[j] = v[j+1];
                v[j+1] = aux;
                trocou = true;
            }
        }
        i += 1;
    }
}

def main(): int {
    var i: int;
    write "Digite os valores do arranjo:\n";
    for (i=0; i<10; i++) {
        write "A[", i, "] = ";
        read v[i];
    }
    bubblesort(v, 10);

    write "Arranjo ordenado:\nA = ";
    for (i=0; i<10; i++) {
        write v[i], " ";
    }
}
```