

# Sumário

<b>Estrutura geral de um programa</b>	<b>2</b>
<b>Nomes</b>	<b>2</b>
<b>Tipos e estruturas de dados</b>	<b>2</b>
Forma de declaração	2
Operadores	3
Constantes literais	3
Declarações	3
Cadeias de caracteres	3
Arranjos	3
Equivalência de tipos e coerção	4
<b>Atribuição e expressões</b>	<b>4</b>
Precedência	4
Associatividade	4
<b>Sintaxe e exemplo de estruturas de controle</b>	<b>4</b>
Comando de seleção	4
Comandos de iteração	5
Controle lógico	5
Controle por contador	5
Desvio incondicional	5
Instruções de entrada e saída	5
<b>Subprogramas</b>	<b>6</b>
Métodos de passagem de parâmetros	6
<b>Exemplos</b>	<b>7</b>
Alô mundo	7
Série de Fibonacci	7
Shell sort	8
<b>Especificação dos tokens</b>	<b>9</b>
Enumeração com as categorias dos tokens	9
Categorias simbólicas e expressões regulares	9
Expressões regulares auxiliares	11

## Estrutura geral de um programa

A linguagem admite escopo global e as variáveis são declaradas ao especificar o tipo e o nome em qualquer parte do programa. Funções são declaradas especificando o tipo de retorno e a lista de parâmetros.

A execução é iniciada a partir da função *init()* e as funções são definidas em qualquer parte especificando o tipo de retorno, o nome e a lista de parâmetros.

## Nomes

Os nomes são sensíveis à caixa e tem tamanho máximo de 100 caracteres alfanuméricos iniciando com uma letra.

Os nome são reconhecidos pela expressão regular `[_a-zA-Z][_a-zA-Z0-9]*`, são permitidos caracteres alfanuméricos e underscore exceto o primeiro que não pode ser um dígito.

## Tipos e estruturas de dados

### Forma de declaração

Os tipos de dados predefinidos e suas operações suportadas são:

Tipo	Declaração	Operações
Inteiro	<code>int nome;</code>	Aritméticas e relacionais
Ponto flutuante	<code>float nome;</code>	Aritméticas e relacionais
Caractere	<code>char nome;</code>	Relacionais
Booleano	<code>bool nome;</code>	Relacionais e lógicas
Cadeias de caracteres	<code>string nome;</code>	Relacionais e concatenação
Arranjos	<code>tipo nome[tamanho];</code>	Concatenação

## Operadores

Aritméticos	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> (módulo), <code>-</code> (unário)
Relacionais	<code>==</code> , <code>!=</code> , <code>&lt;=</code> , <code>&gt;=</code> , <code>&lt;</code> , <code>&gt;</code>
Lógicos	<code>!</code> , <code>and</code> , <code>or</code>
Concatenação	<code>++</code>

## Constantes literais

```
int          -?(\d+)
float        -?(\d+)\.(\d+)
char         \'\'
bool         (true)|(false)
string       "(.*)"
Arranjos     { el1, el2,... }
```

## Declarações

Os tipos inteiro, ponto flutuante, caractere, booleano e string são declarados seguindo o formato `tipo nome`;

As constantes nomeadas são precedidas pela palavra reservada `const` e em seguida deve ser declarada a variável.

## Cadeias de caracteres

Para as cadeias de caracteres deve ser utilizadas aspas ( `"` ) para atribuição:

```
string nome = "valor";
```

## Arranjos

Os arranjos são declarados na forma `tipo nome[tamanho]`;

Os elementos dos arranjos são referenciados por índices listados entre colchetes e são armazenados em posições contíguas na memória.

## Equivalência de tipos e coerção

A equivalência de tipos é por nome e é permitido conversões de tipo explícitas (*cast*). A coerção admitida é de *int* para *float*.

## Atribuição e expressões

O operador de atribuição é o símbolo '=' onde a expressão do lado direito é atribuída à variável alvo do lado esquerdo.

Os operadores *and* e *or* são avaliados em curto-circuito e os tipos das operações são definidos de acordo com a variável alvo.

## Precedência

Precedência da mais alta para a mais baixa

- *unário*
- \*, /, %*
- + e - binários*
- <, >, <=, >=*
- ==, !=*
- and, or*
- = atribuição*

## Associatividade

*Esquerda: \*, /, %, + e - binários*  
*<, >, <=, >=, ==, !=*  
*and, or*  
*Direita: - unário, = atribuição*

## Sintaxe e exemplo de estruturas de controle

Todos os blocos devem possuir chaves (*{}*)

### Comando de seleção

O comando de seleção é o *if* que seleciona um caminho baseado em cada condição. Para mais de uma condição é usado o *else if* e o bloco opcional *else* que é selecionado caso nenhuma das condições sejam atendidas.

```
if(condição) {
```

```

        ...
    } else if(condição) {
        ...
    } else {
        ...
    }
}

```

## Comandos de iteração

### Controle lógico

Enquanto a condição não for verdadeira o bloco de código é executado.

```

while (condição){
    ...
}

```

### Controle por contador

A inicialização de variáveis é feita em *var\_int* e a *condição* é testada antes de cada iteração, o *incremento* é uma constante inteira que será utilizada como incremento da *var\_int* feito no final da iteração.

```

for (var_int; condição; incremento){
    ...
}

```

## Desvio incondicional

*break*

Permite encerrar um loop

```

while (condição){
    ...
    break;
    ...
}

```

## Instruções de entrada e saída

A instrução de leitura da entrada padrão é realizada especificando as variáveis em que serão atribuídos os valores lidos.

```

read var1, var2,...;

```

Na instrução de saída padrão deve ser especificado o texto formatado e as variáveis ou constantes que vão ser substituídas

```
print "%2d %.2f", var1, cons1;
```

## Subprogramas

As funções são declaradas no corpo do programa e seguem a forma:

```
tipo nome(parâmetros) {  
    ...  
}
```

Os procedimentos são declarados como funções. A palavra reservada *proc* é usada para declarar os procedimentos.

```
proc nome(parâmetros) {  
    ...  
}
```

## Métodos de passagem de parâmetros

As funções implementam o modo de entrada e saída, os parâmetros são passados por valor-resultado. Funções não podem ser passadas como parâmetro.

## Exemplos

### Alô mundo

```
int init(){
    print "Alô mundo";
    return 0;
}
```

### Série de Fibonacci

```
int init(){
    int a, b, i, aux, limite;
    a = 0;
    b = 1;

    print "Digite um número limite: ";
    input limite;

    i = 0;

    while(i < limite){
        aux = a + b;
        a = b;
        b = aux;
        print "%d, ", auxiliar;
        i = i + 1;
    }

    return 0;
}
```

## Shell sort

```
proc shellSort(int vet[], int size){

    int i , j , value;
    int gap = 1;

    while(gap < size) {
        gap = 3*gap+1;
    }

    while (gap > 0) {
        for(i = gap; i < size; 1) {
            value = vet[i];
            j = i;
            while (j > gap-1 and value <= vet[j - gap]) {
                vet[j] = vet [j - gap];
                j = j - gap;
            }
            vet[j] = value;
        }
        gap = gap/3;
    }
}

int init(){

    int i, tam;

    input tam;
    int vet[tam];

    for(i = 0; i < tam; 1){
        input vet[i];
    }

    shellSort(vet, tam);

    for(i = 0; i < tam; 1){
        print "%d ", vet[i];
    }
}
```



## Especificação dos tokens

A linguagem em que os analisadores léxico e sintático serão implementados será C++

### Enumeração com as categorias dos tokens

```
enum class Category {  
    Init=1, Integer, Float, Char, String, Boolean,  
    PtVg, Pt2, Vg, AbPar, FePar, AbCol, FeCol, AbChav, FeChav,  
    Procedure, Return, Break,  
    Input, Print,  
    For, While, If, ElseIf, Else,  
    OpEq, OpMaior, OpMenor, OpMaiorEq, OpMenorEq, OpDifer,  
    OpMais, OpMenos, OpMult, OpDiv, OpMod,  
    OpAtr, OpConcat,  
    Id, CteInt, CteFloat, CteChar, CteBool, CteStr, Eof  
};
```

### Categorias simbólicas e expressões regulares

Init	<i>init</i>
Integer	<i>int</i>
Float	<i>float</i>
Char	<i>char</i>
String	<i>string</i>
Boolean	<i>bool</i>
PtVg	<i>;</i>
Pt2	<i>:</i>
Vg	<i>,</i>
AbPar	<i>(</i>

FePar	)
AbCol	[
FeCol	]
AbChav	{
FeChav	}
Procedure	<i>proc</i>
Return	<i>return</i>
Break	<i>break</i>
Input	<i>input</i>
Print	<i>print</i>
For	<i>for</i>
While	<i>while</i>
If	<i>if</i>
Elseif	<i>else if</i>
Else	<i>else</i>
OpEq	==
OpMaior	>
OpMenor	<
OpMaiorEq	>=
OpMenorEq	<=
OpDifer	!=
OpMais	+
OpMenos	-
OpMult	*

OpDiv	/
OpMod	%
OpAtr	=
OpConcat	++
Id	<code>[_a-zA-Z][_a-zA-Z0-9]*</code>
CteInt	<code>-?(\\d+)</code>
CteFloat	<code>-?(\\d+)\\. (\\d+)</code>
CteChar	<code>\\'\\.\\'</code>
CteBool	<code>(true)   (false)</code>
CteStr	<code>\\ "(\\.*) \"</code>
Eof	<code>\\0</code>

## Expressões regulares auxiliares

```

letras = '[a - zA - Z]'
dígitos = '[0-9]'
símbolos = '[ . , ; ! + - * \\ / % < > = ( ) [ ] { } " ' ]'
```