

Lucas Peixoto de Almeida Cavalcante

## **Especificação da linguagem de programação LisC**

Especificação da linguagem de programação LisC, definida pelo aluno, para a disciplina de Compiladores, correspondente à parte da avaliação da AB1 do semestre de 2019.1, sob orientação do **Prof. Alcino Dall Igna Jr.**

Universidade Federal de Alagoas

Instituto de Computação

Brasil

Maceió/Al  
2019.1

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>3</b>
<b>2</b>	<b>ESTRUTURA DO PROGRAMA</b>	<b>4</b>
<b>3</b>	<b>IDENTIFICADORES</b>	<b>5</b>
<b>4</b>	<b>PALAVRAS-RESERVADAS</b>	<b>6</b>
<b>5</b>	<b>TIPOS DE DADOS</b>	<b>7</b>
<b>5.1</b>	<b>Tipos de dados primitivos</b>	<b>7</b>
5.1.1	Inteiro	7
5.1.2	Ponto flutuante	7
5.1.3	Booleano	8
5.1.4	Caractere	8
<b>5.2</b>	<b>Estrutura de dados</b>	<b>8</b>
5.2.1	Arranjos unidimensionais	8
5.2.2	Cadeia de caractere	9
<b>5.3</b>	<b>Equivalência de tipos</b>	<b>10</b>
5.3.1	Coerções admitidas	10
5.3.2	Verificação de tipo	10
5.3.3	Tipagem	10
<b>5.4</b>	<b>Constantes com nome</b>	<b>10</b>
<b>6</b>	<b>ATRIBUIÇÕES E EXPRESSÕES</b>	<b>11</b>
<b>6.1</b>	<b>Atribuições</b>	<b>11</b>
<b>6.2</b>	<b>Expressões</b>	<b>11</b>
6.2.1	Aritmética	12
6.2.2	Relacional	12
6.2.3	Lógica	13
6.2.4	Concatenação	13
6.2.5	Formatação	13
<b>6.3</b>	<b>Ordem de precedência e associatividade geral</b>	<b>14</b>
<b>7</b>	<b>INSTRUÇÕES</b>	<b>15</b>
<b>7.1</b>	<b>Entrada</b>	<b>15</b>
<b>7.2</b>	<b>Saída</b>	<b>15</b>
<b>7.3</b>	<b>Controles lógicos</b>	<b>16</b>

---

<b>7.4</b>	<b>Estruturas de repetição</b>	<b>16</b>
7.4.1	Desvios incondicionais	17
<b>8</b>	<b>FUNÇÕES</b>	<b>19</b>
8.1	Parâmetros das funções	19
8.2	Retorno das funções	20
8.3	Funções como parâmetros	20
<b>9</b>	<b>PROGRAMAS EXEMPLO</b>	<b>21</b>
9.1	Alô mundo	21
9.2	Fibonacci	21
9.3	Shell sort	22
<b>10</b>	<b>ESPECIFICAÇÃO DOS TOKENS</b>	<b>23</b>

# 1 Introdução

Esse documento tem como objetivo especificar as características da linguagem de programação LisC. Ao longo dos capítulos subsequentes entenderemos as particularidades dessa linguagem no que diz respeito ao escopo dela, aos nomes, tipos e estrutura de dados, atribuições, expressões e mais.

A especificação dessa linguagem servirá de base para a implementação dos analisadores léxico e sintático, que serão desenvolvidas posteriormente no decorrer da disciplina. Ambos os analisadores serão implementados na linguagem de programação C. A linguagem permitirá ser analisada em passo único, com o analisador léxico funcionando on-the-fly a medida que o analisador sintático requisita novos tokens. Ao fim deste documento também existirá uma seção reservada para a especificação dos tokens da linguagem.

## 2 Estrutura do programa

Esse capítulo tem como objetivo especificar como se dá a estrutura geral de um programa em LisC. O início de um programa em LisC ocorre com a definição de uma função chamada **start**, que retorna um *inteiro*. Sem essa função a compilação do programa dá erro. Um exemplo de código de uma função **start** pode ser visto no **código 2.1**. Em LisC temos que as funções só podem ser declaradas e definidas num escopo global, ou seja, não é permitido que uma função seja criada dentro de outra função. Note que nesse exemplo de código podemos observar que os blocos da linguagem são delimitados com a presença de chaves({}).

```
1  defun start () (int) {  
2      defint a 5;  
3      echo ("a = %d") (a);  
4      return 0;  
5  }
```

Código 2.1 – Código exemplo de uma função begin.

A presença de abre e fecha chaves define a presença de um novo escopo dentro da linguagem, variáveis que são criadas dentro desse escopo são consideradas variáveis locais, já as variáveis criadas fora de um escopo são chamadas de variáveis globais. As variáveis globais são visíveis dentro de qualquer escopo, já as locais só são vistas dentro do escopo ao qual ela foi criada, além disso a linguagem admite **mascaramento** de variáveis. Parâmetros de funções tem como escopo todo o bloco da função.

Também no código 2.1 podemos ver o primeiro exemplo da criação de uma variável. A variável criada foi uma variável do tipo inteiro. A declaração de uma variável se dá a partir da junção da palavra **def** com o tipo da variável em questão, como nesse caso tivemos uma variável inteira, a definição da mesma foi feita utilizando a palavra reservada **defint**. Note que a vinculação de tipos de dados às variáveis é estática, sendo assim o tipo é definido explicitamente pelo programador no momento da declaração da variável.

Toda instrução da linguagem LisC deve ser terminada com o caractere ";". E os comentários da linguagem são apenas comentários de linha e são feitos a partir dos caracteres "//". As instruções da linguagem devem estar dentro de escopos de funções. No escopo global só é permitido a declaração ou definição de funções e variáveis.

## 3 Identificadores

Os identificadores em LisC são declarados como na maioria das outras linguagens, iniciando com uma letra e seguido ou não de uma ou mais letras e dígitos. Em LisC os identificadores também são sensíveis à capitalização, além de não possuir limite de caracteres. A definição do conjunto de letra e dígito é dada como segue:

**Letra** :pertence ao conjunto de caracteres  $[A - Z] \cup [a - z]$

**Dígito** :pertence ao conjunto de caracteres  $[0 - 9]$

Dessa forma temos que os identificadores em LisC podem ser representados pelo autômato cuja expressão regular é dada por  $[A-Za-z]([A-Za-z][0-9])^*$ . Note que os identificadores não podem fazer parte do conjunto das palavras reservadas da linguagem. No **capítulo 4** vamos saber quais são as palavras reservadas da linguagem.

## 4 Palavras-reservadas

As palavras reservadas da linguagem podem ser vistas na tabela [1](#).

Tabela 1 – Tabela contendo todas as palavras reservadas da linguagem.

defun	defint	defchar	deffloat	defbool
if	else	int	for	while
echo	return	break	continue	nil
true	false	char	float	bool
string	defstring	start	read	

## 5 Tipos de dados

A linguagem LisC possui os seguintes tipos de dados: inteiro, ponto flutuante, caractere e booleano. E possui as seguintes estruturas de dados: cadeia de caracteres e arranjos unidimensionais. No capítulo 6 veremos com mais detalhes como atribuir valor as variáveis que serão vistas nesse capítulo, o que envolve os tipos de operação.

### 5.1 Tipos de dados primitivos

Nessa seção iremos falar a respeito de cada um dos quatro tipos de dados primitivos existentes.

#### 5.1.1 Inteiro

As variáveis do tipo inteiro representam os números inteiros. Podemos ver no **código 5.1** como é feita a declaração de uma variável desse tipo. A palavra reservada que representa esse tipo de variável é a palavra **int**.

```
1 defint <nome>;
```

Código 5.1 – Declaração de uma variável inteira

As constantes literais de uma variável inteira estão contidas no intervalo de  $\{-2.147.483.648, +2.147.483.647\}$ . Temos como exemplo os seguintes valores:  $[-2, -40, 0, 55, 12]$ .

#### 5.1.2 Ponto flutuante

As variáveis do tipo ponto flutuante representam os números reais. Podemos ver no **código 5.2** como é feita a declaração de uma variável desse tipo. A palavra reservada que representa esse tipo de variável é a palavra **float**.

```
1 deffloat <nome>;
```

Código 5.2 – Declaração de uma variável tipo ponto flutuante.

As constantes literais de uma variável float estão contidas no intervalo de  $\{-3.4e+38, 3.4e+38\}$ . Temos como exemplo os seguintes valores:  $[-2.2, -40.5, 0, 55.0, 12.1]$ .



### 5.1.3 Booleano

As variáveis do tipo booleano representam apenas dois valores: verdadeiro ou falso. Podemos ver no **código 5.3** como é feita a declaração de uma variável desse tipo. A palavra reservada que representa esse tipo de variável é a palavra **bool**.

```
1 defbool <nome>;
```

Código 5.3 – Declaração de uma variável booleana

As constantes literais booleanas podem ser somente: **true** ou **false**. Desse modo não existe um intervalo associado a esse tipo de variável. Em LisC qualquer valor diferente de zero é considerado verdadeiro e zero é considerado falso.

### 5.1.4 Caractere

As variáveis do tipo caractere representam os caracteres alfanuméricos da tabela **ASCII**. Podemos ver no **código 5.4** como é feita a declaração de uma variável desse tipo. A palavra reservada que representa esse tipo de variável é a palavra **char**.

```
1 defchar <nome>;
```

Código 5.4 – Declaração de uma variável caractere

As constantes literais do tipo de variável caractere estão contidas no intervalo de  $[0, 127]$ . Temos como exemplo os seguintes valores: "a", "b", "c".

## 5.2 Estrutura de dados

Essa seção irá tratar dos tipos de dados que representam estruturas de dados dentro da linguagem.

### 5.2.1 Arranjos unidimensionais

As variáveis do tipo arranjos unidimensionais podem ser de qualquer um dos tipos primitivos já apresentados na **seção 5.1**. Esse tipo de variável é muito utilizada para guardar uma quantidade de elementos de uma variável do mesmo tipo. Note que a declaração é basicamente a mesma da declaração das variáveis primitivas, a única diferença é a existência de colchetes para definir qual será a quantidade exata dos elementos a serem guardados. Perceba, portanto, que a quantidade de elementos é estática e definida em tempo de compilação, de modo que o tamanho não pode ser alterado após ter sido declarada e/ou definida. Temos no **código 5.5** um exemplo da declaração desse tipo de variável para cada um dos tipos primitivos.

```
1  defint  numeros [5];
2  defchar  letras [10];
3  defbool  binarios [15];
4  deffloat  taxas [4];
```

Código 5.5 – Declaração de uma variável do tipo arranjo unidimensional.

As constantes literais desse tipo de variável não possuem um intervalo de valores, visto que esse tipo de análise não faz muito sentido. Exemplos desse tipo de variável podem ser: {1,2,3}, {true,false,true,false}, {2.3, 1.6, -1.2}.

O limite inferior de um arranjo unidimensional é 0, e o limite superior é dado pelo tamanho - 1. Desse modo, o acesso a um elemento específico dentro do arranjo pode ser feito a partir do **código 5.6**.

```
1  defint  numeros [5]  {1,2,3,4,5}
2  echo ("acessando indice %d do arranjo numeros, valor: %d") (2, numeros
    [1])
```

Código 5.6 – Acessando elemento dentro de um arranjo unidimensional

### 5.2.2 Cadeia de caractere

As variáveis do tipo cadeia de caractere são basicamente um arranjo unidimensional de uma variável primitiva do tipo *char*. Esse tipo de estrutura também é conhecido como *string*. Temos no **código 5.7** um exemplo da declaração de uma variável do tipo cadeia de caractere com o tamanho definido e outra com o tamanho sendo calculado a partir do valor associado a variável no momento da definição.

```
1  defstring  palavra [10];
2  defstring  texto [] = "Saudosa maloca";
```

Código 5.7 – Declaração de variáveis do tipo cadeia de caractere.

Note que na variável *texto* o tamanho alocado para ela foi de 15 caracteres, sendo 14 para o texto em si (*Saudosa maloca*) e mais um para marcar o fim da string, que seria o caracter `"\0"`. Para strings, o caracter considerado nulo é o caracter `"\0"` que é utilizado em C, por exemplo, para marcar o fim de uma string.

As formas de acesso a caracteres específicos de uma cadeia de caracteres é feito da mesma forma que em arranjos unidimensionais.

## 5.3 Equivalência de tipos

### 5.3.1 Coerções admitidas

Os tipos de coerções admitidas na linguagem de programação LisC são relacionadas apenas as variáveis do tipo primitiva. As coerções possíveis podem ser visualizadas na **tabela 2**. A linguagem não tem nenhuma forma explícita de conversão de tipos.

Tabela 2 – Coerções admitidas em LisC

De	Para
Inteiro	String, Ponto flutuante, Character, Booleano
Ponto flutuante	String, Inteiro, Booleano
Booleano	String
Character	String, Inteiro

### 5.3.2 Verificação de tipo

Para garantir que os tipos em LisC sejam respeitados, é feita uma verificação em tempo de compilação. Essa verificação é feita tanto em operações entre variáveis, para garantir que os tipos são compatíveis, como na verificação dos tipos de parâmetros de funções.

### 5.3.3 Tipagem

A linguagem LisC é fortemente tipada.

## 5.4 Constantes com nome

A linguagem LisC não admite constantes com nomes.

## 6 Atribuições e expressões

Nesse capítulo iremos tratar a respeito das atribuições e das expressões presentes em LisC. Será visto como que as atribuições são feitas e entenderemos um pouco melhor os tipos de expressões presentes na linguagem, sua precedência, quais tipos de variáveis estão associados a operação e etc. Primeiro falaremos sobre as atribuições na [seção 6.1](#) e em seguida veremos a respeito das expressões na [seção 6.2](#).

### 6.1 Atribuições

As atribuições são responsáveis por associar algum valor a uma variável. No [código 6.1](#) podemos ver alguns exemplos de atribuições em LisC.

```
1  defint age 15;
2  set age ((age + 15) - 20);
3  deffloat lado, area_quadrado, area_triangulo;
4  set lado 5;
5  set area_quadrado (lado * lado);
6  set area_triangulo (area_quadrado - 1);
7  defbool x (area_quadrado > area_triangulo);
8  defbool res !x;
```

Código 6.1 – Exemplos de atribuições.

Note que nesses exemplos as atribuições foram feitas de várias formas. No momento da declaração de uma variável é possível definir de imediato um valor inicial. Após a definição da variável, qualquer atribuição é feita a partir da palavra reservada *set*.

Uma variável pode ter seu valor alterado a partir de uma constante literal, de uma expressão aritmética, relacional ou lógica a depender do tipo da variável, retorno de função ou até mesmo de uma concatenação de cadeia de caracteres, caso a variável seja do tipo cadeia de caracteres, caso contrário poderá existir uma coerção automática se for possível.

### 6.2 Expressões

Existem cinco tipos diferentes de expressões em LisC: aritméticas, relacionais, lógicas, concatenação e formatação. As seções a seguir vão especificar as características de cada uma das expressões disponíveis.

### 6.2.1 Aritmética

As expressões aritméticas estão relacionadas apenas as variáveis do tipo numéricas, ou seja, *int* e *float*. Em LisC temos somente os operadores abaixo, que estão listados de cima para baixo, do maior para o menor na ordem de precedência.

**Negativo** : símbolo “-“, é um operador unário.

**Divisão e multiplicação** : símbolo “/“ e “\*“, ambos são operadores binários.

**Soma e subtração** : símbolo “+“ e “-“, ambos são operadores binários.

Em expressões aritméticas temos que o operando mais a esquerda é chamado de operando alvo, ou seja, é ele quem vai ditar qual será o tipo final do resultado da operação. Sendo assim, caso o operando mais a esquerda seja do tipo *int*, mesmo que o segundo operando seja do tipo *float* o resultado da operação será um valor do tipo *int*.

### 6.2.2 Relacional

As expressões relacionais tem como resultado um valor sempre do tipo booleano. Podemos ver abaixo os tipos de operadores relacionais disponíveis.

**Igualdade** : símbolo é o “==“

**Desigualdade** : símbolo é o “!=“

**Menor que** : símbolo é o “<“

**Maior que** : símbolo é o “>“

**Menor ou igual que** : símbolo é o “<=“

**Maior ou igual que** : símbolo é o “>=“

Para tipos booleanos, como é um tipo de variável que só tem dois valores, só faz sentido pensar nos operadores de igualdade e desigualdade. Contudo, com relação aos outros tipos de variáveis podemos criar uma expressão relacional com qualquer um dos operadores disponíveis.

Nas operações aritméticas tínhamos que mesmo que os operandos envolvidos não fossem do mesmo tipo, existia uma coerção automática para concluir o resultado da operação. Contudo, para expressões relacionais esse tipo de coerção não existe e resulta em erros. O motivo é trivial de perceber, não faz sentido relacionar duas variáveis de tipos diferentes.

### 6.2.3 Lógica

As expressões lógicas são utilizadas apenas quando os seus operandos são booleanos. Podemos ver abaixo os tipos de operandos lógicos disponíveis.

**Negação** : símbolo “!”, unário

**Conjunção** : símbolo “&&”, binário

**Disjunção** : símbolo “||”, binário

Os operadores foram dispostos em ordem de precedência, do maior para o menor.

### 6.2.4 Concatenação

As operações de concatenação tem como resultado sempre um valor do tipo string. Quando a operação é feita com a presença de operandos que não são do tipo string, é necessário que seja feita uma coerção dos tipos para string. Vale lembrar da [seção 5.3.1](#) que todos os tipos de variáveis primitivas tem a possibilidade de coerção para string.

Temos no [código 6.2](#) exemplos de código envolvendo concatenação em LisC, onde o operador de concatenação é o “<<”.

```
1  defint idade 5;
2  deffloat tamanho 30.0;
3  defstring nome "Asnestro"
4  defstring frase ("Tenho um cachorro chamado " << nome << ", ele tem "
5  << idade << " anos e mede " << tamanho << "cm");
6  echo frase;
7  \\ a saida desse código e "Tenho um cachorro chamado Asnestro, ele
8  tem 5 anos e mede 30.0cm"
```

Código 6.2 – Exemplo de expressões de concatenação

### 6.2.5 Formatação

Os operadores de formatação são muito utilizados na apresentação do valor de variáveis. Esse operador é utilizado para variáveis do tipo *int* e *float*. Quando se trata de um inteiro fazemos a formatação do número de campos desse inteiro, já quando se trata de um ponto flutuante fazemos a formatação do número de casas decimais. Podemos ver no [código 6.3](#) exemplos do uso do operador de formatação.

```
1  // <variavel>%<numero de campos>
2  // <variavel>%<numero de casas decimais>
3  defint time 1551;
4  defint hora time\%2;
```

```

5  deffloat pi 3.14159265359;
6  deffloat pi-formatado pi%%2;

```

Código 6.3 – Exemplo do uso do operador de formatação

Note que a variável *hora* é uma variável inteira e a variável *pi-formatado* é uma variável de ponto flutuante. O resultado de uma operação de formatação tem sempre como resultado uma variável do tipo do elemento a esquerda que foi utilizado

### 6.3 Ordem de precedência e associatividade geral

Visto que todos os tipos de operadores já foram apresentados, se faz necessário definir qual a precedência entre operadores de tipos de expressões diferentes. A **tabela 3** nos mostra a ordem de precedência entre todos os operadores da linguagem.

Além disso, existe a possibilidade de alterar a ordem de precedência no meio da operação por meio dos parênteses *()*. Caso uma expressão seja colocada entre parênteses, esta deve ser executada primeiro do que as mais externas ao parênteses, ou seja, temos um acréscimo no nível de precedência da operação.

Ordem de precedência	Operadores	Símbolos
Do maior para o menor	Negativo, Negação	-,!
	Multiplicação, Divisão	*,/
	Soma, Subtração	+,-
	Concatenação	«
	Igualdade, Desigualdade, Maior que, Menor que, Maior ou igual que, Menor ou igual que	==, !=, >, <, >=, <=
	Conjunção, Disjunção	&&,

Tabela 3 – Ordem de precedência geral

Caso existam operadores adjacentes com o mesmo nível de precedência, a expressão será resolvida da esquerda para a direita. Contudo, caso os operadores em questão sejam os operadores negativo ou de negação, temos a associatividade da direita para a esquerda.

## 7 Instruções

Nesse capítulo vamos discutir um pouco mais sobre algumas instruções importantes em LisC, sendo elas as instruções de Entrada, Saída, controles lógicos e estruturas de repetição. Algumas dessas instruções já foram apresentadas ao longo deste documento, mas agora elas serão definidas de forma mais explícita.

### 7.1 Entrada

Esse tipo de instrução aparecerá agora pela primeira vez nesse documento. As instruções de leitura do *stdin* são feitas a partir da palavra reservada *read*. Podemos ver um exemplo desse tipo de instrução no **código 7.1**.

```
1 // read <variaveis que devem ser inseridas>
2 defint a, b, c;
3 read (a, b, c);
4 // a instrucao acima espera do stdin a entrada de 3 valores
5 // que serao associados as variaveis "a", "b" e "c".
```

Código 7.1 – Exemplo de código da instrução read.

A instrução *read* dá a possibilidade de receber mais de um valor de entrada por instrução.

### 7.2 Saída

A instrução de escrita no *stdout* já foi vista algumas vezes nesse documento, como no **código 6.2**, por exemplo. Podemos ver no **código 7.2** como a instrução pode ser utilizada.

```
1 // echo <texto formatado> <variaveis a serem substituidas no texto
2 formatado>
3 defint a 5;
4 defint b 6;
5
6 echo ("a = " << a << "\nb = %d") (b);
```

Código 7.2 – Exemplo de código da instrução echo

Também é possível utilizar constantes literais no texto formatado.



## 7.3 Controles lógicos

O controle lógico que a linguagem LisC fornece é o controle via as palavras reservadas *if* e *else*. O *else* só pode existir caso, previamente, uma instrução *if* tenha sido definida. Dessa forma, todo *else* está associado a um único *if*, e um *if* pode ou não estar associado a um *else*. Podemos ver no **código 7.3** um exemplo do uso desses tipos de instruções. Notem que todo *if* é seguido de uma expressão lógica entre parênteses, que será testada para ver se o código entrará no bloco do *if* ou do *else*.

```
1  defbool cond true;
2  defbool cond2 false;
3
4  if (cond) {
5      echo "Cond e verdadeira";
6  }
7  else {
8      echo "Cond e falsa";
9  }
10
11 if (cond2) {
12     echo "Cond2 e verdadeira";
13 }
14 else {
15     echo "Cond2 e falsa";
16 }
17
18 // a saída desse código seria "cond e verdadeira" no primeiro if
19 // "cond2 e falsa" no segundo if.
```

Código 7.3 – Exemplo do uso de instruções de controles lógicos

Dentro de uma estrutura de repetição podem ser usadas quaisquer instruções que poderiam ser utilizadas num escopo de função, ou seja, podem existir outras estruturas de decisão, chamadas de funções, definição de variáveis, estruturas de repetição e etc.

Uma estrutura lógica *if* não precisa necessariamente estar associada a um bloco de código criado a partir de um *else*.

## 7.4 Estruturas de repetição

Existem dois tipos de estrutura de repetição em LisC, a primeira é a estrutura de repetição baseada num controle lógico, já a segunda é a estrutura de repetição baseada num contador. A primeira estrutura será executada até quando a expressão lógica for verdadeira. A segunda estrutura será executada até quando a variável controlada não atingir o valor final dela.

Podemos ver nos **códigos 7.4** e **7.5** exemplos dessas duas estruturas de repetição, que são definidas através das palavras reservadas *while* e *for*, respectivamente.

```
1 // while (<expressao logica>) {<bloco de codigo>}
2 defint a 1;
3
4 while (a > 0) {
5     read a;
6 }
```

Código 7.4 – Exemplo da estrutura de repetição while

```
1 // for (<contador>, <valor inicial>, <valor final>, <passo>) {<bloco de
   codigo>}
2 defint count;
3
4 for (count, 0, 10, 1) {
5     echo count;
6 }
```

Código 7.5 – Exemplo da estrutura de repetição for

Note que a estrutura de repetição *for* permite controle de passo. Caso o controle de passo não seja passado, o valor padrão é 1. Caso seja passado, o controle de passo passa a ser o valor especificado.

### 7.4.1 Desvios incondicionais

LisC fornece a possibilidade de efetuar desvios incondicionais dentro de estruturas de repetição. Desse modo, caso o programador queira a qualquer momento finalizar um laço de repetição, basta ele utilizar a instrução *break*;. De outro modo, caso o programador queira pular a execução de um laço de repetição, fazendo com que se inicie imediatamente a execução do laço novamente, se utiliza a instrução *continue*;. No **código 7.6** podemos visualizar essas instruções atuando.

```
1 defint count;
2
3 for (count, 0, 20) {
4     if (count < 10) {
5         continue;
6     }
7
8     if (count == 17) {
9         break;
10    }
11
12    echo count << "\n";
```

```
13     }  
14  
15     // esse codigo tras como saida  
16     // 11  
17     // 12  
18     // 13  
19     // 14  
20     // 15  
21     // 16
```

Código 7.6 – Exemplo de código de desvios incondicionais

## 8 Funções

Já vimos no **código 2.1** um exemplo da definição de uma função. Segue no **código 8.1** um exemplo mais detalhado.

```

1 // Definicao de uma funcao:
2 // defun <nome> (<parametros de entrada>)(<tipo de retorno>) {<bloco
3   de codigo>}
4 // Chamada de uma funcao:
5 // <nome>(<parametros de entrada>)
6
7 defint num 10;
8
9 defun plus5(int a)(int) {
10   return a + 5;
11 }
12
13 defun start ()(int) {
14   echo plus5(num);
15
16   return 0;
17 }
18
19 // saida desse codigo e:
20 // 15

```

Código 8.1 – Exemplo de definição de uma função

A linguagem não permite a definição de funções aninhadas. Note que o retorno de uma função é determinado a partir da palavra reservada *return*

### 8.1 Parâmetros das funções

Com relação aos tipos que as funções podem ter em seus parâmetros e em seu retorno, temos que as funções em LisC podem ter tanto as entradas como as saídas de qualquer um dos tipos primitivos e dos não primitivos (arranjos unidimensionais e cadeia de caracteres) da linguagem já aprenados na **seção 5**. A passagem de parâmetros para funções é feita somente por valor.

## 8.2 Retorno das funções

Caso a função não tenha retorno, em sua definição os segundos parênteses devem estar vazios, e nesse caso não é necessário o uso da palavra reservada *return* no bloco da função. Caso o programador deseje explicitar o momento de retorno da função, se faz necessário que o *return* seja usado sem nenhum valor associado a ele, ou seja, *return;*.

## 8.3 Funções como parâmetros

Funções não podem ser passadas como parâmetros, não podem ser sobrecarregadas e não podem ser genéricos.

## 9 Programas exemplo

### 9.1 Alô mundo

```
1  defun start()(int) {
2      echo "Alo mundo";
3      return 0;
4  }
```

Código 9.1 – Programa Alô mundo

### 9.2 Fibonacci

```
1  defun start()(int) {
2      defint n;
3      defint first 0;
4      defint second 1;
5      defint next;
6      defint c;
7
8      echo "Digite o numero de termos da sequencia";
9      read (n);
10
11     echo "Os primeiros %d termos de fibonacci sao:" (n);
12
13     while (c < n) {
14         if (c <= 1) {
15             set next c;
16         }
17         else {
18             set next (first + second);
19             set first second;
20             set second next;
21         }
22
23         if (c == n-1) {
24             echo next;
25         }
26         else {
27             echo next << ", ";
28         }
29
30         set c c+1;
```

```
31     }
32     return 0;
33 }
```

Código 9.2 – Programa Fibonacci

### 9.3 Shell sort

```
1  defun shell_sort(int ar[], int size)(void) {
2      defint i, j, value;
3      defint gap 1;
4
5      while (gap < size) {
6          set gap 3*gap+1;
7      }
8
9      while (gap > 0) {
10         for(i, gap, size) {
11             set value ar[i];
12             set j i;
13             while (j > gap-1 && value <= ar[j - gap]) {
14                 set ar[j] ar[j-gap];
15                 set j j-gap;
16             }
17             set ar[j] value;
18         }
19         set gap gap/3;
20     }
21 }
22
23 defun start()(int) {
24     defint ar[] {12, 34, 54, 2, 3};
25     defint i;
26     defint n 5;
27
28     shell_sort(ar, n);
29
30     for(i, 0, n) {
31         echo ar[i];
32     }
33 }
```

Código 9.3 – Programa do shell sort

## 10 Especificação dos tokens

Nesse capítulo iremos falar sobre a especificação dos tokens da linguagem LisC. Na **seção 1** já foi apresentado que a linguagem de programação escolhida para implementar o analisador léxico e sintático será em C.

```
1 /**
2  * Enum for token category
3  *
4  */
5 enum cat {
6     INT_TYPE = 0,
7     CHAR_TYPE, // 1
8     STRING_TYPE, // 2
9     BOOL_TYPE, // 3
10    FLOAT_TYPE, // 4
11    FORMAT_FIELD, // 5
12    FORMAT_DECIMAL, // 6
13    DEFUN, // 7
14    DEFINT, // 8
15    DEFCHAR, // 9
16    DEFFLOAT, // 10
17    DEFBOOL, // 11
18    DEFSTRING, // 12
19    IF, // 13
20    ELSE, // 14
21    FOR, // 15
22    WHILE, // 16
23    ECHO, // 17
24    RETURN, // 18
25    BREAK, // 19
26    CONTINUE, // 20
27    NIL, // 21
28    SET, // 22
29    READ, // 23
30    START, // 24
31    CONST_FLOAT, // 25
32    CONST_INT, // 26
33    CONST_STRING, // 27
34    CONST_CHAR, // 28
35    CONST_BOOL, // 29
36    OP_MINUS, // 30
37    OP_PLUS, // 31
38    OP_DIVIDE, // 32
39    OP_MULT, // 33
```



```
40 OP_EQUAL, // 34
41 OP_DIFF, // 35
42 OP_LEQ, // 36
43 OP_LESS, // 37
44 OP_GEQ, // 38
45 OP_GREATER, // 39
46 OP_OR, // 40
47 OP_AND, // 41
48 OP_NOT, // 42
49 OP_CONCAT, // 43
50 OPEN_PAREN, // 44
51 CLOSE_PAREN, // 45
52 OPEN_BRACKETS, // 46
53 CLOSE_BRACKETS, // 47
54 OPEN_KEYS, // 48
55 CLOSE_KEYS, // 49
56 SEMICOLON, // 50
57 COMMA, // 51
58 VOID, // 52
59 ID, // 53
60 CAT_EOF, // 54
61 };
62 typedef enum cat cat_t;
```

Código 10.1 – Enumeração das categorias dos tokens

As **tabelas 4** e **5** contêm as descrições das categorias dos tokens mostrados no **código 10.1**.

Categoria simbólica	Descrição
INT_TYPE	Tipo inteiro
CHAR_TYPE	Tipo char
STRING_TYPE	Tipo string
BOOL_TYPE	Tipo booleano
FLOAT_TYPE	Tipo ponto flutuante
FORMAT_FIELD	Operador limitador de quantidade de campos
FORMAT_DECIMAL	Operador limitador de casas decimais
DEFUN	Marcador de declaração de função
DEFINT	Marcador de declaração de inteiro
DEFCHAR	Marcador de declaração de char
DEFFLOAT	Marcador de declaração de ponto flutuante
DEFBOOL	Marcador de declaração de booleano
DEFSTRING	Marcador de declaração de string
IF	Condicional se
ELSE	Condicional else
FOR	Iterador for
WHILE	Iterador while
ECHO	Saída
RETURN	Retorno de função
BREAK	Desvio incondicional break
CONTINUE	Desvio incondicional continue
NIL	Nulo
SET	Função de atribuição
READ	Entrada
START	Função principal
CONST_FLOAT	Constante literal float
CONST_INT	Constante literal inteira
CONST_STRING	Constante literal string
CONST_CHAR	Constante literal char
CONST_BOOL	Constante literal booleana

Tabela 4 – Categorias e descrições

Categoria simbólica	Descrição
OP_MINUS	Operador subtração ou unário negativo
OP_PLUS	Operador adição
OP_DIVIDE	Operador divisão
OP_MULT	Operador multiplicação
OP_EQUAL	Operador igualdade
OP_DIFF	Operador diferença
OP_LEQ	Operador menor igual que
OP_LESS	Operador menor que
OP_GEQ	Operador maior igual que
OP_GREATER	Operador maior que
OP_OR	Operador ou
OP_AND	Operador and
OP_NOT	Operador negação
OP_CONCAT	Operador concatenação
OPEN_PAREN	Abertura de parênteses
CLOSE_PAREN	Fechamento de parênteses
OPEN_BRACKETS	Abertura de colchetes
CLOSE_BRACKETS	Fechamento de colchetes
OPEN_KEYS	Abertura de chaves
CLOSE_KEYS	Fechamento de chaves
SEMICOLON	Ponto e vírgula
COMMA	Vírgula
VOID	Tipo void
ID	Identificador
CAT_EOF	Fim de arquivo

Tabela 5 – Categorias e descrições

Na **tabela 6** temos as expressões regulares auxiliares.

Nome	Expressão regular
letra	[A-Za-z]
dígito	[0-9]
símbolo	'!' '#' '\$' '%' '&' '\' '\"' '(') ' *' '+' ',' '-' ' '/' ' ':' ';' '<' '>' '=' ' '?' '@' ' ' ' ' '_ ' '{' ' '}'

Tabela 6 – Expressões regulares auxiliares

FALTANDO CATEGORIA E SUAS EXPRESSÕES REGULARES