COMPILADORES

Universidade Federal de Alagoas
Instituto de Computação

GimmeDaddy

Leandro Martins de Freitas Douglas Henrique Maximo da Silva

2017.2

Sumário

1 Introdução

Documento de especificação da linguagem de programação GimmeDaddy, linguagem esta que terá seus analisadores - léxico e sintático - desenvolvidos em Java a partir das especificações no presente documento.

2 Especificação da linguagem

2.1 Estrutura de um programa

Variáveis podem ser declaradas em qualquer parte do código, enquanto que instruções podem ser declaradas apenas dentro de funções. Instruções são explanadas na seção 2.6. A declaração e funcionamento de funções é detalhada na seção 2.7.

A declaração de variáveis deve ser explícita, isto é, o tipo deve ser indicado. Os blocos são delimitados por abre chaves({) e fecha chaves(}). O fim dos comandos é delimitado por um ponto e vírgula(**;**).

Admitimos apenas comentários de linha. Todo o texto após **//** numa linha é um comentário, e portanto não será executado.

A seguir temos alguns exemplos estruturais da linguagem:

```
1  //declaração de variável global
2  <tipoDaVariavelG> <nomeDaVariavelG>;
3  //declaração de função
4  <tipoDeRetornoDaFuncao> <nomeDaFuncao>(<listaDeParametros>*){
5     <tipoDaVariavel1> <nomeDaVariavel1>;
6      <tipoDaVariavel2> <nomeDaVariavel2>;
7      if(<nomeDaVariavel1> == <nomeDaVariavel2>) {
8        return nomeDaVariavel1 - nomeDaVariavel2;
9      } else {
10       return nomeDaVariavel2 - nomeDaVariavel1;
11     }
12 }
13 //função principal
14 void main() {
15    <nomeDaFuncao>(<listaDeParametros>*);
16 }
```

A execução sempre começa da função padrão, declarada como uma função sem retorno (vide seção 2.7.1), seu nome é a palavra reservada **main**. Sem esta função, a execução do código gera um erro. Daí, temos que um arquivo vazio não é um programa válido na linguagem GimmeDaddy.

Ao decorrer da leitura, todas essas estruturas e palavras chaves serão explanadas. Essa seção tem como objetivo apenas ilustrar como as construções da linguagem devem ser feitas.

2.2 Nomes, verificação de tipos e escopos

2.2.1 Nomes

Questões de projeto:

- Os nomes são sensíveis a capitalização? -Sim.
- As palavras especiais da linguagem são palavras reservadas ou palavras-chave? - Palavras reservadas.

2.2.2 Verficação de tipos

A linguagem é tipada estaticamente, o que significa que cada expressão e cada variável tem seu tipo conhecido em tempo de compilação.

2.2.3 Escopo

Utilizaremos um escopo estático. A seguir algumas regras são apresentadas.

1. O escopo de uma variável que é declarada fora de uma função é global.
2. O escopo de uma variável declarada em um bloco é o restante do bloco em que a declaração apareceu.
3. O escopo de um parâmetro formal de uma função é todo o seu corpo.

2.3 Tipos de dados e Constantes literais

2.3.1 Tipo int

Esse tipo de dado assume valores inteiros positivos ou negativos que podem ser representados em 4 bytes (32-bits) com um bit para sinal. Esse tipo é representado pela palavra reservada **int**.

2.3.2 Tipo float

Esse tipo de dado assume valores reais positivos ou negativos que podem ser representados em 4 bytes (32-bits). 1 bit é utilizado para sinal, 8 bits para o expoente e 23 bits para a fração. A parte inteira é separada da fracionária por um ponto, seguindo o padrão americano. Esse tipo é representado pela palavra reservada **float**.

2.3.3 Tipo caractere

O tipo caractere será tratado como uma categoria que assume valores alfanuméricos da tabela ASCII. Esse tipo é representado pela palavra reservada **char**.

2.3.4 Tipo boolean

Tipo de dados simples, assume apenas dois valores: verdadeiro e falso. Esse tipo é representado pela palavra reservada **boolean**.

2.3.5 Tipo arranjo unidimensional

Questões de projeto:

- Que tipos são permitidos para índices? -Inteiros
- As expressões de índices em referências a elementos são verificadas em relação à faixa? -Sim.
- Quando as faixas de índices são vinculadas? -Na declaração é definido o limite superior. O limite inferior, por padrão, é 0.
- Quando ocorre a alocação do arranjo unidimensional? -Em relação a esse aspecto, a linguagem funcionará como as matrizes dinâmicas da pilha fixa. Que são vinculadas estaticamente mas a alocação ocorre no momento de declaração.

- As matrizes podem ser inicializadas quando elas têm seu armazenamento alocado? -Não.
- Que tipos de fatias são permitidas? -Não são permitidas fatias.

A declaração de um arranjo unidimensional é feita da seguinte forma:

```
<tipoDosElementos> <nomeDoArranjo>[<tamanhoDoArranjo>];
```

A faixa de indexação dos elementos de um arranjo é de 0 a $n-1$, onde n é o tamanho do arranjo. Para acessar um elemento do arranjo $teste$ na posição i, fazemos:

```
teste[i];
```

Ao tentar acessar um índice negativo ou maior que o limite superior declarado, um erro será gerado. A atribuição utilizando arranjos será explanada nos dois últimos exemplos da seção 2.6.5.

2.3.6 Tipo cadeia de caracteres

Questões de projeto:

- As cadeias devem ser apenas um tipo especial de vetor de caracteres ou um tipo primitivo? -As cadeiras são um tipo especial de Vetor de caracteres.
- As cadeias devem ter tamanho estático ou dinâmico? -Estático.

Este é um tipo no qual os valores consistem em arranjos unidimensionais cujos elementos são do tipo caractere. A este arranjo damos o nome de cadeia de caracteres.

2.3.7 Constantes literais

As constantes literais para cada tipo suportado na lingaguem são definidas como na tabela a seguir.

Tabela 1: Literais

Tipo	**Intervalo**	**Exemplos**
Booleano	[true, false]	true, false
Inteiro	[-32768, +32767]	-2, -1, 0, 1, 2, 3
Ponto flutuante	[-3.4e+38, -1.4e−45] ∪ [1.4e−45, 3.4e+38]	-2.5, -1.5, 1.5, .37, .2, 1., 2.51
Char	[32, 126] (valor ascii)	'a', 'b', 'c', '1', '@', ')'
Arranjos	-	[0, 3, 1], [1.5, 3.5, 2.7], ['a', 'b', '1', '@']
Cadeias de caracteres	-	"ab1@", ['a', 'b', '1', '@']

2.4 Operadores

Cada operador realiza alguma operação específica entre uma ou duas sentenças, que nesse caso, como elementos de uma operação, chamamos de operando. É importante salientar que operações entre operandos de tipos diferentes geram erros, exceto em operações de concatenação, como será visto na seção 2.4.4.

Os operadores são organizados em categorias, nas seguintes seções essas categorias são descritas.

2.4.1 Operadores aritméticos

Operações aritméticas são definidas apenas para números inteiros ou de ponto flutuante. Temos os seguintes operadores:

Tabela 2: Operadores aritméticos em ordem decrescente de precedência

Operador	**Símbolo**	**Aridade**
Negativo	-	Unário
Multiplicação, Divisão	*, /	Binário
Soma, Subtração	+, -	Binário

2.4.2 Operadores relacionais

Operadores relacionais necessitam de dois operandos. Realizam comparação entre os operandos e retornam **true** caso ela seja verificada verdadeira e **false** caso contrário.

2.4.2.1 Para tipos booleanos

Para tipos booleanos temos os seguintes operadores:

Tabela 3: Operadores relacionais para operandos booleanos

Operador	**Símbolo**
Igualdade	==
Desigualdade	!=

2.4.2.2 Para os demais tipos

Para tipos exceto os booleanos temos os seguintes operadores:

Tabela 4: Operadores relacionais para demais tipos

Operador	Símbolo
Igualdade	==
Desigualdade	!=
Menor que	<
Maior que	>
Menor ou igual que	<=
Maior ou igual que	>=

No caso de comparação de caracteres, a comparação é feita entre os códigos ASCII dos operandos. Já para arranjos e, consequentemente, cadeias de caracteres a comparação é feita entre os tamanhos dos arranjos.

2.4.3 Operadores lógicos

Operadores lógicos realizam operações apenas entre sentenças booleanas. Temos os seguintes operadores:

Tabela 5: Operadores lógicos, em ordem decrescente de precedência

Operador	Símbolo	Aridade
Negação	!	Unário
Conjunção	&	Binário
Disjunção	|	Binário

É importante notar que os operadores de conjunção e disjunção retornam **true** caso a expressão seja avaliada como verdadeira e **false** caso contrário. A negação inverte o valor lógico da expressão, retornando **true** quando falso e **false** quando verdadeiro.

2.4.4 Operador de concatenação

Representaremos o operador de concatenação pelo síbolo **++**. Sendo A e B dois operandos, A ++ B retorna uma cadeia preenchida com o(s) elemento(s) de A seguido(s) pelo(s) elemento(s) de B. Note que:

$$A \text{ ++ } B \neq B \text{ ++ } A.$$

Se algum dos operandos for do tipo numérico ou booleano, será transformado para cadeia de caracteres para que a operação possa ser realizada.

Para fim de exemplos considere $tam(X)$ o tamanho do operando X e que A e B foram convertidos para cadeias de caracteres onde se fez necessário. A seguir temos alguns casos de concatenação:

- Entre dois caracteres A e B: o resultado é uma cadeia N, tal que $tam(N) = 2$.
- Entre um caractere A e uma cadeia de caracteres B: o resultado é uma cadeia N, tal que $tam(N) = tam(A) + tam(B) = 1 + tam(B)$.
- Entre duas cadeias de caracteres A e B: o resultado é uma cadeia N, tal que $tam(N) = tam(A) + tam(B)$.

2.5 Ordem de precedência e Associatividade

Dentre os operadores apresentados, alguns têm maior prioridade de avaliação. A seguir temos a ordem de precedência dessas categorias:

Tabela 6: Operadores em ordem decrescente de precedência

Operadores
Negação (!)
Aritméticos
De concatenação
Relacionais
Disjunção e conjunção (| e &)

Parênteses são utilizados para aumentar a precedência de uma operação, assim, operações entre parênteses têm precedência máxima. No caso de operações aninhadas com parênteses, as operações mais internas têm maior precedência.

Quando temos sentenças com operadores adjacentes de mesma precedência devemos utilizar associatividade para esolher qual operação será avaliada primeiro. A associatividade de todos os operadores é à esquerda, com exceção do - unário, que é à direita.

Para entedermos melhor a ordem de precedência e a associatividade, tome o seguinte exemplo:

```
1  !(1 + 3 * 2 - 2 ++ 'c' == "3c")
2
3  Avaliamos assim:
4  !(1 + 3 * 2 - 2 ++ 'c' == "3c" | true)
5  !(1 + 6 - 2 ++ 'c' == "3c" | true)
6  !(7 - 2 ++ 'c' == "3c" | true)
7  !(5 ++ 'c' == "3c" | true)
8  !("5c" == "3c" | true)
9  !(false | true)
10 !(true)
11 false
```

2.6 Instruções

2.6.1 Estruturas de decisão

2.6.1.1 Estrutura condicional de uma via

Uma estrutura condicional de uma via é escrita da forma como se segue:

```
1 if (<condicaoLogica>){
2    <corpoDoBloco>
3 }
```

Onde, <condicaoLogica> pode ser uma relação booleana ou um literal boleano (true ou false). Se essa condição (<condicaoLogica>) for avaliada como verdadeira, o programa executa o que está dentro do bloco (<corpoDoBloco>), caso contrário, o programa deve continuar sua execução a partir da proxima linha depois do fim do bloco.

2.6.1.2 Estrutura condicional de duas vias

Uma estrutura condicional de duas vias é especificada como se segue:

```
1 if (<condicaoLogica>){
2    <corpoDoBloco>
3 } else {
4    <execucaoAlternativa>
5 }
```

Semelhante a estrutura condicional de uma via, na de duas vias, <condicaoLogica> também pode ser uma relação booleana ou um literal boleano (true ou false). Se a condição for avaliada como verdadeira, o programa executa o que está dentro do primeiro bloco (<corpoDoBloco>), caso contrário, o programa deve pular para o segundo bloco (else) e executar o que está dentro dele (<execucaoAlternativa>).

2.6.2 Estruturas de repetição

2.6.2.1 Estrutura iterativa com controle lógico

Uma estrutura iterativa com controle lógico é associada a palavra reservada **while** e escrita da forma como se segue:

```
1 while (<condicaoLogica>){
2    <corpoDoBloco>
3 }
```

Onde, <condicaoLogica> pode ser uma relação booleana ou um literal boleano (true ou false). Se essa condição (<condicaoLogica>) for avaliada como verdadeira, o programa repetirá a execução do que está dentro do bloco (<corpoDoBloco>) até que a condição se torne falsa. Nesse caso, se, por exemplo, em uma quinta iteração a condição se tornar falsa dentro do bloco interno ao while, na sexta iteração a condição será avaliada como falsa e o programa continua sua execução a partir do que vier exatamente depois do bloco while. Se, no momento em que a execução do programa chega no bloco while (pela primeira vez) a condição não for verdadeira, o programa continua sua execução a partir do que vier exatamente depois do fim do bloco while.

2.6.2.2 Estrutura iterativa com controle por contador

Uma estrutura iterativa com controle por contador é associada a palavra reservada **for** e escrita da forma como se segue:

```
1 for (<variavelControle> = <valorInicial>; <condicaoLogica>){
2    <corpoDoBloco>
3 }
```

Onde, <variavelControle> é uma variável do tipo inteiro declarada anteriormente, a qual será atribuído um valor inicial (<valorInicial>) e que será usada para fazer o controle do laço. A variável de controle é incrementada em 1 a cada iteração.

Se em uma iteração a condição se tornar falsa, na próxima iteração a condição será avaliada como falsa e o programa não mais executa o que está dentro do bloco (<corpoDoBloco>) e pula para o que vier exatamente depois do bloco. Se, no momento em que a execução do programa chega no bloco **for** (pela primeira vez) a condição não for verdadeira, o programa continua sua execução a partir do que vier exatamente depois do fim do bloco for.

2.6.3 Instrução de entrada

A instrução de entrada da linguagem está relacionada a palavra reservada **read** e é responsável por salvar o(s) valor(res) de entrada digitados pelo usuário em variáveis alvo. Se a entrada conter um número n de itens, deve haver, também, um número n de variáveis para atribuir esses valores. É importante notar que a variável alvo deve ser declarada antes do seu uso na instrução **read**. Os usos de **read** são como se segue:

```
1 read(<variavel>); //Formato utilizado para salvar um item de entrada do usuário.
```

```
2 read(<variavel1>, <variavel2>); //Formato utilizado para salvar dois itens de
    entrada do usuário.
3 read(<variavel1>, <variavel2>, ..., <variavelN>); //Formato utilizado para
    salvar N itens de entrada do usuário.
```

2.6.4 Instrução de saída

A instrução de saída da linguagem é responsável por fazer uma formatação de informações e mostrá-las ao usuário. Está relacionada às palavras reservadas **print** e **println**, a primeira imprime sem pular linha, enquanto que a segunda imprime e pula uma linha. A saída contém um campo de texto que pode ser escrito pelo programador, que caso não queira preencher, basta deixar duas aspas no lugar. Além do campo de texto, a instrução também permite mostrar valores salvos em variáveis.

2.6.4.1 Formatação

Os usos de **print** são como se segue:

```
1 print(""); //Formato utilizado para mostrar apenas uma linha vazia.
2 print("<texto>"); //Formato utilizado para mostrar apenas um texto.
3 print("%<inicialDoTipo>", <variavel>); //Formato utilizado para mostrar apenas
    o valor de uma variável sem formatação especificada.
4 print("%.<tamanhoDoCampo><inicialDoTipo>", <variavel>); //Formato utilizado
    para mostrar apenas o valor de uma variável formatado.
5 print("<texto> %.<tamanhoDoCampo><inicialDoTipo>", <variavel>); //Formato
    utilizado para mostrar um texto e o valor de uma variável.
6 print("<texto> %.<tamanhoDoCampo1><inicialDoTipo1>
    %.<tamanhoDoCampo2><inicialDoTipo2>", variavel1, variavel2); //Formato
    utilizado para mostrar um texto e o valor de duas variáveis separados por
    expaço.
7 println("<texto> %.<tamanhoDoCampo1><inicialDoTipo1>
    %.<tamanhoDoCampo2><inicialDoTipo2> ...
    %.<tamanhoDoCampoN><inicialDoTipoN>", <variavel1>, <variave2>, ...,
    <variavelN>); //Formato utilizado para mostrar um texto e os valores de N
    variáveis separados por espaço.
```

- Linha 1: Construção mais simples da instrução, quando o programador não preenche texto algum nem coloca variáveis a serem mostradas.

- Linha 2: Construção simples. Utilizada para mostrar ao usuário apenas um texto (<texto>) preenchido pelo programador.
- Linha 3: Construção simples, Utilizada para mostrar ao usuário apenas o conteúdo de uma variável (<variavel>). Para identificar o tipo da variável, utiliza-se, após um símbolo de porcentagem, a inicial do nome do tipo (<inicialDoTipo>), que pode ser **i** para inteiros, **c** para char, **f** para float, **b** para booleando, **a** para arranjos e **s** para cadeias de caracteres.
- Linha 4: Construção simples. Utilizada quando o programador deseja mostrar o usuário o valor de uma variável formatado com o tamanho (<tamanhoDoCampo>) que ele definir após o símbolo de porcentagem seguido de um ponto. Essa formatação tem o mesmo comportamento para todos os tipos, exceto para valores de ponto flutuante. Para os tipos: inteiro, cadeia de caracteres, arranjos e tipos boleanos, essa formatação faz com que apenas os primeiros valores atomicos de cada tipo sejam mostrados, de forma a ficar do tamanho determinado pelo programador em <tamanhoDoCampo>.
- Linha 5: Construção composta. Utilizada quando o programador deseja imprimir um texto e o valor formatado de uma variável.
- Linha 6: Construção composta. Utilizada quando o programador deseja imprimir um texto e o valor formatado de duas variáveis.
- Linha 7: Construção composta. Utilizada quando o programador deseja imprimir um texto e o valor formatado de N variáveis, separados por espaço. Após a impressão uma linha é pulada.

2.6.5 Atribuição

A atribuição será tratada como uma instrução que guarda o valor de uma sentença, variálvel ou literal em uma variável alvo. A construção de uma atribuição é semelhante a outras lingagens imperativas. A seguir, são mostrados alguns exemplos do uso.

```
1 <variavelA> = <variavelB>; //Atribuição simples. Copia o valor de B para A
2 <variavelB> = <literalInteiro>; //Atribuição simples. Guarda em B um valor
    literal do tipo inteiro.
3 <variavelD> = <variavelB> + <variavelC>; //Atribuição composta. Salva o
    resultado de uma adição em D.
```

```
4 <variavelE> = <variavelB> * <variavelC>; //Atribuição composta. Salva o
      resultado de uma multiplicação em E.
5 <arranjoP>[<indice>] = <variavelB>; //Atribuição complexa. Salva o valor de uma
      variável na posição <indice> de um arranjo P.
6 <variavelG> = <variavelG> + <variavelB>; //Incrementa o valor de G com o valor
      de B.
7 <variavelH> = <variavelH> + <literalFloat>; //Incrementa o valor de H com o
      valor de um literal do tipo Float.
8 <variavelT> = <arranjoK>[<indice>]; //O valor da posição <indice> do arranjo K é
      armazenado na variável T.
```

É importante lembrar que a coerção de tipos só é admitida para operações de concatenação de caracteres ou de cadeias de carcadeias com outros tipos.

2.7 Funções

Questões de projeto:

- As variáveis locais são alocadas estaticamente ou dinamicamente? - Estaticamente.
- As definições de subprogramas podem aparecer em outras definições de subprogramas? - Não.
- Que método ou métodos de passagem de parâmetros são usados? - Por valor para tipos básicos, por referência para tipos arranjo.
- Os tipos dos parâmetros reais são verificados em relação aos tipos dos parâmetros formais? - Sim.
- Subprogramas podem ser passados como parâmetros? - Não.
- Os subprogamas podem ser sobrecarregados? - Não.
- Os subprogramas podem ser genéricos? - Não.

2.7.1 Declaração

Funções podem ser declaradas em qualquer local do arquivo de código fonte, exceto dentro de uma outra função, portanto não são admitidas funções aninhadas. O tipo de retorno de funções deve ser explícito; no caso de procedimentos, utilizamos a palavra reservada **void** para indicar retorno vazio. Funções que retornam arranjos têm seu retorno definido como $< tipoDoRetorno >$ [].

A declaração de uma função é feita da seguinte forma:

```
1 //função com retorno básico
2 <tipoDeRetorno> <nomeDaFuncao>(<listaDeParametros>*){
3    <corpoDaFuncao>
4 }
5
6 //função com retorno de arranjo
7 <tipoDeRetorno>[] <nomeDaFuncao>(<listaDeParametros>*){
8    <corpoDaFuncao>
9 }
```

Podemos perceber que a lista de parâmetros está envolta em <> $*$. Essa notação indica que tal elemento é opcional. Portanto uma função pode ou não receber parâmetros. Seja **p** um elemento da forma $< tipoDoParametro > < nomeDoParametro >< [] > *$, a lista de parâmetros de uma função é composta por nenhum ou vários elementos do tipo **p** separados por vírgulas.

2.7.2 Retorno

O retorno de funções é sinalizado pela palavra reservada **return** seguido do elemento que deve ser retornado; atente que o tipo do elemento retornado deve ser o mesmo do definido na declaração da função. Ao chegar à linha de retorno, o elemento é retornado e a execução da função é interrompida.

2.7.3 Chamada

A chamada de uma função é feita da seguinte maneira:

```
1   <nomeDaFuncao>(<listaDeParametros>*);
```

2.8 Programas exemplos

2.8.1 Hello World

```
1   void main() {
2     print("Hello World!")
3    }
```

2.8.2 Fibonacci

```
1  void fib(int limite, int seq[]) {
2    int aux;
3    aux = 0;
4
5    while(aux < limite) {
6      if(aux == 0 | aux == 1) {
7        seq[aux] = aux;
8      } else {
9        seq[aux] = seq[aux - 1] + seq[aux - 2];
10     }
11     aux = aux + 1;
12   }
13 }
14
15 void main() {
16   int limite;
17   int aux;
18   aux = 0;
19
20   read("%i", limite);
21   int sequencia[limite + 1];
22
23   fib(limite, sequencia);
24
25   while(aux < limite - 1) {
26     print("%i, ", sequencia[aux]);
27     aux = aux + 1;
```

```
  }
  println("%i", sequencia[limite - 1]);
}
```

2.8.3 Shell Sort

```
void shellSort(int vet[], int size) {
    int i , j , value;
    int gap;
    gap = 1;

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

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

void main() {
  int size;
  int i;

  read("%i", size);

  int vet[size];

```

```
32    for(i = 0; i < size) {
33      read("%i", vet[i]);
34    }
35
36    for(i = 0; i < size - 1) {
37      print("%i, ", vet[i]);
38    }
39    println("%i", vet[size-1]);
40
41    shellSort(vet, size);
42
43    for(i = 0; i < size - 1) {
44      print("%i, ", vet[i]);
45    }
46    println("%i", vet[size-1]);
47  }
```

3 Especificação dos Tokens

Como mencionado na introdução, utilizaremos Java para implementar os analisadores.

3.1 Enumeração de categorias

```
1  public enum Categorias {
2     MAIN, PONT_VIRG, VIRGULA,
3      AB_PARENTE, FE_PARENTE, AB_CHAVE, FE_CHAVE, AB_COLCHET, FE_COLCHET,
4      ID,
5      ATRIBUICAO,
6      CTE_INT, CTE_FLOAT, CTE_BOOL, CTE_CHAR,
7      CTE_CAD_IN, CTE_CAD_FL, CTE_CAD_BO, CTE_CAD_CH
8      TIPO_INT, TIPO_FLOAT, TIPO_BOOL, TIPO_CHAR, TIPO_VOID,
9      OPA_AD, OPA_SUB, OPA_MULT, OPA_DIV,
10     OPR_IGUAL, OPR_DIF, OPR_MEN, OPR_MEN_IG, OPR_MAI, OPR_MAI_IG,
11     OPL_OU, OPL_E, OPL_NAO,
12     OP_CONC,
13     RETORNO,
14     SE, SENAO,
15     FOR, WHILE,
16     ENTRADA, SAIDA, SAIDA_LN,
17     COMENTARIO
18 }
```

3.2 Expressões regulares e Lexemas

Tabela 7: Expressões regulares auxiliares

Nome	Expressão auxiliar
letra	['a'-'z'] | ['A'-'Z']
digito	['0'-'9']
simbolo	(' '|';'|'.'|','|':'|'!'|'@'|'#'|'$'|'%'|'&'|'|'|'('|')'|'['|'+'|']'|'{'|'?'|'}'|'+'|'-'|'*'|'/'|'='|'<'|'>'|'_')

Tabela 8: Descrição das categorias

Categoria simbólica	**Descrição**
MAIN	Função principal
PONT_VIRG	Ponto e vírgula
VIRGULA	Vírgula
AB_PARENTE	Abertura de parêntese
FE_PARENTE	Fechamento de parêntese
AB_CHAVE	Abertura de chave
FE_CHAVE	Fechamento de chave
AB_COLCHET	Abertura de colchetes
FE_COLCHET	Fechamento de colchetes
ID	Identificador
ATRIBUICAO	Símbolo de atribuição
CTE_INT	Constante literal inteira
CTE_FLOAT	Constante literal float
CTE_BOOL	Constante literal booleana
CTE_CHAR	Constante literal caractere
CTE_CAD_IN	Arranjo de inteiros
CTE_CAD_FL	Arranjo de floats
CTE_CAD_BO	Arranjo de booleanos
CTE_CAD_CH	Cadeia de caracteres
TIPO_INT	Declaração de tipo inteiro
TIPO_FLOAT	Declaração de tipo float
TIPO_BOOL	Declaração de tipo booleano
TIPO_CHAR	Declaração de tipo caractere
TIPO_VOID	Declaração de retorno vazio
OPA_AD	Operador de adição
OPA_SUB	Operador de subtração
OPA_MULT	Operador de multiplicação
OPA_DIV	Operador de divisão
OPR_IGUAL	Operador relacional para igualdade
OPR_DIF	Operador relacional para diferença
OPR_MEN	Operador relacional para menor que
OPR_MEN_IG	Operador relacional para menor ou igual
OPR_MAI	Operador relacional para maior que
OPR_MAI_IG	Operador relacional para maior ou igual
OPL_OU	Operador lógico de disjunção
OPL_E	Operador lógico de conjunção
OPL_NAO	Operador lógico de negação
OP_CONC	Operador de concatenação
RETORNO	Declaração de retorno
SE	Condicional se
SENAO	Condicional senão
FOR	Instrução de iteração por contador
WHILE	Instrução de iteração por controle lógico
ENTRADA	Instrução de entrada
SAIDA	Instrução de saída
SAIDA_LN	Instrução de saída com quebra de linha
COMENTARIO	Comentário

Tabela 9: Categorias e Expressões regulares

Categoria simbólica	**Expressão regular**
MAIN	'main'
PONT_VIRG	';'
VIRGULA	','
AB_PARENTE	'('
FE_PARENTE	')'
AB_CHAVE	'{'
FE_CHAVE	'}'
AB_COLCHET	'['
FE_COLCHET	']'
ID	letra(letra|digito|'_'(letra|digito))*
ATRIBUICAO	'='
CTE_INT	(('+'|'-')?digito+)
CTE_FLOAT	(('+'|'-')?)(digito*'.'digito+|digito+'.'digito*)('e"+'digito+|'e"-'digito+)?
CTE_BOOL	('true'|'false')
CTE_CHAR	"'(letra|digito|simbolo)"'
CTE_CAD_IN	'['CTE_INT(','CTE_INT)*']'
CTE_CAD_FL	'['CTE_FLOAT(','CTE_FLOAT)*']'
CTE_CAD_BO	'['CTE_BOOL(','CTE_BOOL)*']'
CTE_CAD_CH	'"'(letra|digito|simbolo)*'"' | '['CTE_CHAR(','CTE_CHAR)*']'
TIPO_INT	'int'
TIPO_FLOAT	'float'
TIPO_BOOL	'boolean'
TIPO_CHAR	'char'
TIPO_VOID	'void'
OPA_AD	'+'
OPA_SUB	'-'
OPA_MULT	'*'
OPA_DIV	''
OPR_IGUAL	'=='
OPR_DIF	'!='
OPR_MEN	'<'
OPR_MEN_IG	'<='
OPR_MAI	'>'
OPR_MAI_IG	'>='
OPL_OU	'|'
OPL_E	'&'
OPL_NAO	'!'
OP_CONC	'++'
RETORNO	'return'
SE	'if'
SENAO	'else'
FOR	'for'
WHILE	'while'
ENTRADA	'read'
SAIDA	'print'
SAIDA_LN	'println'
COMENTARIO	'//'(letra|digito|simbolo)*

Referências

[1] R. W. SEBESTA. *Conceitos de Linguagens de Programação - 9ª Edição.* 2011.