Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada
Compiladores

Definição da Linguagem

Hélio Bezerra Duarte Filho
Luísa Rocha de Azevedo Santos
Raul Silveira Silva
Silvino Gustavo A. de Medeiros
Thales Aguiar de Lima

2018

Manual

1 Introdução

A linguagem CPa é o trabalho em curso da disciplina de Compiladores. Ela foi concebida de acordo com os seguintes aspectos:

- **C-like**: A linguagem é C-like, ou seja, possui forte inspiração na sintaxe de C. Declarações de variáveis, arranjos e funções possuem sintaxes parecidas com C;
- **for de Pascal**: O laço `for` dessa linguagem (que aqui é chamado `para`) não é inspirada no for de C, e sim no `for` da linguagem Pascal;
- **Tipagem Forte**: a tipagem dessa linguagem é forte, isto é, não há conversão implícita entre tipos diferentes de dados;
- **Em português**: as estruturas, comandos, palavras-chave e outros termos importantes da linguagem são escritos em português.

2 Tipos de dados

2.1 Tipos de dados primitivos

- **int** (16 bits): inteiro;
- **real** (32 bits): número representado em ponto flutuante;
- **reald** (64 bits): representa o número real de precisão dupla;
- **char** (8 bits): caractere ou byte;
- **vazio**: se utilizado como tipo de retorno de uma função, o tipo **vazio** indica que essa função não retorna nenhum valor. Não é possível declarar uma variável do tipo **vazio**.

CPa representa numericamente valores booleanos verdadeiro e falso. Expressões lógicas, por exemplo, resultam em valores numéricos. Valores diferentes de 0 são avaliados como verdadeiro e o 0 é avaliado como falso, assim como em C.

2.1.1 Ponteiros

Uma variável do tipo ponteiro representa um endereço de memória. Um tipo ponteiro pode ser descrito através da sintaxe `*tipo`, onde `tipo` é o tipo do conteúdo armazenado nesse endereço.

O valor de um endereço pode ser obtido através do operador `&` usado numa variável, e o valor associado ao endereço contido no ponteiro pode ser obtido através do operador `*` usado num ponteiro.

```
// Ponteiros
int x;
int *ref_de_x;
int *pont_x = &x;
ref_de_x = pont_x;
int valor_de_x = *pont_x;
```

2.1.2 Vetores

Vetores são sequências de elementos de um determinado tipo. Vetores de `n` posições possuem elementos organizados em posições que vão de 0 a `n-1`. Um tipo vetor pode ser descrito por `tipo[n]`, onde `tipo` é o tipo de cada elemento do vetor e `n` é o tamanho do vetor.

O acesso aos elementos dessa sequência é feito assim como em C, ou seja, é realizado através do uso de colchetes. Por exemplo, dado um vetor `p`, então `p[2]` é o valor do terceiro elemento dessa sequência.

```
// Declaracao de vetores
int[10] vetor_de_int;
caractere[15] vetor_de_char;
int elm_tres = vetor_de_int[2];
```

2.2 Tipos definidos pelo usuário

2.2.1 Estruturas

Através de estruturas, o usuário poderá criar um novo tipo seguido a sintaxe:

```
// Declaracao de tipos definidos pelo usuario
estrutura nome {
    tipo campo1;
    tipo campo2;
     ⋮
    tipo campoN;
}
```

onde `nome` é o nome do novo tipo e `campo1-campoN` é uma lista de campos que compõem a estrutura, onde cada campo segue a mesma estrutura de uma declaração ou inicialização de variável. Para acessar um determinado campo, o usuário deve criar uma variável do tipo definido por ele, e então usar a seguinte sintaxe: `variável.campoN`.

Um exemplo de campo que o usuário pode definir é um novo tipo Ponto como a seguir:

```
// Declaracao de tipos definidos pelo usuario
estrutura Ponto {
    real x;
    real y;
}
```

Essa estrutura cria um novo tipo chamado `Ponto` que possui duas propriedades `x` e `y`. A sintaxe abaixo mostra como criar uma nova variável desse tipo, tal qual a maneira acessar os seus campos.

```
// Variavel do tipo Ponto
Ponto pt;
// Acesso aos campos da estrutura
real currX = pt.x;
real currY = pt.y;
```

2.2.2 Enumeração

Enumerações também estão inclusas na linguagem como instrumento para criação de novos tipos. A sintaxe para definir um novo tipo através de uma enumeração é:

```
enum id {id1, id2, id3, ..., idN}
```

Para criar um novo tipo, utiliza-se a palavra `enum` seguido do identificador do novo tipo a ser criado seguido de um bloco contendo uma lista de valores possíveis para aquele tipo. Por exemplo, para criar um novo tipo `DiasDaSemana`, faríamos:

```
enum DiasDaSemana {dom, seg, ter, qua, qui, sex, sab
};
```

2.3 Conversão de dados

A conversão de dados é feita de forma explícita através de função nativas para os tipos nativos ou para funções de conversão definidas pelo usuário para tipos não-nativos. As função nativas são:

- `parareal(int a)`: converte de `int` para `real`;
- `parareal(reald a)`: converte de `reald` para `real`;
- `parareald(int a)`: converte de `int` para `reald`;
- `parareald(real a)`: converte de `real` para `reald`;
- `paraint(real a)`: trunca um `real` para um `int`;

- `paraint(reald a)`: trunca um `reald` para um `int`;
- `paraint(caractere a)`: converte de `int` para `caractere`;
- `paracaractere(int a)`: converte de `caractere` para `int`.

3 Operadores

3.1 Operadores de Atribuição

- "=": o operador de atribuição atribui um valor ou resultado de uma expressão situada à direita do operador a uma variável especificada à esquerda;

3.2 Operadores Aritméticos

- `+`, `-`: soma e subtração, usado para tipos numéricos, ponteiros e caracteres;
- `*`, `/`: multiplicação e divisão, usados para tipos numéricos apenas;
- `-`: sinal negativo (operador unário), para tipos numéricos;
- `%`: módulo da divisão, para inteiros;
- `«`: deslocamento de bits para a esquerda, para tipos inteiros;
- `»`: deslocamento de bits para a direita, para tipos inteiros;
- `&`: endereço (unário), usado para qualquer variável, campo de estrutura ou elemento de vetor;
- `*`: conteúdo (unário), usado para ponteiros.

3.3 Operadores Lógicos

- `&`: operação lógica AND sem curto circuito;
- `&&`: operação lógica AND com curto circuito;
- `|`: operação lógica OR sem curto circuito;
- `||`: operação lógica OR com curto circuito;
- `!`: operação lógica NOT (operador unário).

3.4 Operadores Relacionais

Os operadores relacionais são utilizados em avaliações de expressões lógicas, resultando em *1* caso sejam verdadeiras ou *0* caso sejam falsas. São eles:

- `>`, `>=`: maior e maior-igual;
- `<`, `<=`: menor ou menor-igual;
- `==`: igual;
- `!=`: diferente.

3.5 Operador ternário

O operador ternário envolve um argumento do tipo numérico e dois outros argumentos de um tipo qualquer, tal que ambos os argumentos sejam do mesmo tipo.

```
1     expressao ? valor1 : valor2
```

Caso o valor de`expressao` seja diferente de `0`, então o `valor1` é o valor resultante, caso contrário, o `valor2` é o valor resultante.

4 Definições

4.1 Constantes

As constantes são definidas pelo comando `const`, na seguinte sintaxe:

```
1     // Declaracao de constante
2     const tipo nome expressao;
```

onde `tipo` é o tipo da constante, `nome` é o nome da constante, a `expressao` é o seu valor inicial, que deve usar valores apenas de outras constantes ou operações entre constantes.

4.2 Variáveis

As variáveis podem ser declaradas na seguinte sintaxe:

```
1     // Declaracao de variavel
2     tipo nome;
```

onde `tipo` é o tipo da variável e `nome` é o nome da variável. Nesse caso, o valor padrão do tipo é armazenado na variável. Opcionalmente, o programador pode informar um valor inicial, da forma:

```
1     // Inicializacao de variavel
2     tipo nome = expressao;
```

onde `expressao` é o valor inicial, calculado após a declaração da variável. Múltiplas variáveis podem ser declaradas com seus identificadores separados por vírgula como abaixo:

```
    // Inicializacao de multiplas variaveis
    int var1, var2, var3, ..., varN;
```

Outros exemplos de declarações e inicializações de variáveis podem ser vistos abaixo:

```
    // Exemplos de declaracao de variavel
    int x;
    int a, b;
    // Exemplos de inicializacao de variavel
    int k = 1;
    int p = 0, q = 1, r;
```

5 Estruturas de controle

5.1 Comandos

5.1.1 Condicionais

Existem dois tipos de condicionais, `se` e `escolha`. No `se`, uma expressão é analisada e caso o valor dela seja nulo, o conjunto de comandos definidos não é executado e pula para o próximo comando ou para o conjunto de comandos `cc`. A sintaxe do `se` é dada por:

```
    // Condicional "se"
    se (expressao)
        comando
```

onde a `expressão` é o valor calculado para definir a próxima sequência de comandos a ser executa, e `comando` é um único comando ou um bloco.

```
    // Condicional "se-cc"
    se (expressao)
        comando
    cc comando
```

Exemplos de condicionais:

```
    // Condicional - exemplo 1
    se (a > b)
        max = a;
    cc
        max = b;
    // Condicional - exemplo 2
    se (trocar == 1) {
```

```
        tmp = a;
        a = b;
        b = tmp;
    }
```

O comando `escolha` pode ser usado para iniciar a execução de uma sequência de comandos a partir de um ponto escolhido. Cada um desses pontos de chamado de `caso`, e quando um caso é satisfeito, todos os comandos seguintes a ele dentro da `escolha` são executados.

A sintaxe da `escolha` é dada por:

```
    escolha (expressao) {
        caso expressao1:
            comandos1
        caso expressao2:
            comandos2
        ...
        caso expressaoN:
            comandosN
        cc:
            comandosCC
    }
```

O valor de `expressao` vai ser comparado às expressões em cada caso. Os casos são analisados sequencial e, quando um caso é satisfeito, todos os seguintes são satisfeitos também. Se nenhum dos casos for satisfeito, um caso alternativo chamado `cc` pode ser opcionalmente definido.

Um exemplo de uso de uma `escolha` pode ser visto a seguir:

```
    // Escolha de uma operacao
    escolha (op) {
    caso '+':
        resultado = a + b;
        parar;
    caso '-':
        resultado = a - b;
        parar;
    caso '*':
        resultado = a * b;
        parar;
    caso '/':
        resultado = a / b;
        parar;
    cc:
        escrever("Operacao nao reconhecida.");
    }
```

5.1.2 Laços

A linguagem CPa apresenta as seguintes estruturas para execução de laços:

- **enquanto**;
- **fazer...enquanto**;
- **para** (for de Pascal);

O laço `enquanto` executa as instruções contidas no bloco enquanto a condição for satisfeita. Sua sintaxe é:

```
1    enquanto (expressao) {
2        comando
3    }
```

onde `expressao` é uma expressão verificada para que, caso seja `0`, o laço seja quebrado. Se a expressão for `1`, o comando ou o bloco de comandos definido por `comando` será executado.

Exemplos do uso do comando `enquanto`:

```
1    // Laco para calcular somatorio
2    enquanto (i < n) {
3        s = s + i;
4        i++;
5    }
```

O laço `fazer...enquanto` é similar ao laço `enquanto`, com a diferença que a primeira iteração é sempre executada, independente do valor da condição. Esse tipo de laço segue a estrutura:

```
1    fazer
2        comando
3    enquanto (expressao)
```

onde `comandos` pode ser um único comando ou um bloco que será executado repetidamente enquanto o valor de `expressao` for diferente de `0`.

```
1    fazer {
2        escrever(no.nome)
3        no = no.proximo
4    } enquanto (no != 0)
```

O laço `para` faz o incremento ou decremento da variável de iteração automaticamente. Esse laço não é parecido com o `for` de C, mas sim com o `for` de Pascal. Sua sintaxe é:

```
1    // Laco crescente
2    para nome de (expressao1) asc (expressao2)
3        comando
```

```
    // Laco decrescente
    para nome de (expressao1) desc (expressao2)
        comando
```

Nesse tipo de laço, antes de iniciar o laço, a variável de iteração chamada `nome` é inicializada com o valor `expressao1` e incrementada (comando `asc`) ou decrementada (comando `desc`) até passar do valor da `expressao2`, condição que é verificada ao final de cada iteração.

Exemplo de uso do laço `para`:

```
    // Laco para calcular somatorio
    int s = 0;
    para i de (0) asc (n - 1) {
        s = s + i;
    }
```

5.1.3 Outros

- `parar` - finaliza manualmente a estrutura de controle atual (apenas usados para laços e escolhas);
- `continue` - dentro de laços, pula para a próxima iteração;
- `retornar` - finaliza uma função;
- `rotulo:` - cria um rótulo com o nome `rotulo`;
- `irpara rotulo` - pula a execução para o rótulo definido por `rotulo`.

5.2 Blocos

Blocos são usados para criar um novo escopo de variáveis ou para agrupar comandos. Eles podem ser definidos entre chaves:

```
{
    comando1;
    comando2;
     ⋮
    comandoN;
}
```

onde cada `comando` é uma expressão ou uma nova estrutura de controle.

5.3 Funções

Funções são abstrações de expressões. Elas podem ser declaradas através da sintaxe:

```
tipo nome(parametros);
```

onde `tipo` é o tipo de retorno da função, `nome` é o nome da função e `parâmetros` é uma sequência de parâmetros formais separados por vírgula, onde cada parâmetro é dado por um tipo seguido de um nome.

Para definir o corpo de uma função, é usada a sintaxe:

```
tipo nome(parametros) {
    comandos
}
```

onde `comandos` é uma sequência de comandos, em que o único comando a ser executado deve ser um comando de retorno.

Um exemplo de definição de função pode ser dada por:

```
// Definicao de soma de inteiros
int soma(int a, int b) {
    retornar a + b;
}
```

5.3.1 Passagem de Parâmetros

Parâmetros podem ser passados de duas formas em CPa: por *valor* e por *referência*. Passagem de parâmetros por valor acarreta numa cópia local do parâmetro dentro da função, ou seja, o parâmetro passado àquela função não é modificado fora dela. Exemplo disso é a função `soma` acima.

Já numa passagem de parâmetros por referência, a função pode modificar o argumento passado a ela. Passagem por referência é indicado pelo símbolo "&" colocado antes do identificador do argumento na chamada da função. Abaixo, a função `swap` demonstra como seria uma passagem por referência em CPa:

```
vazio swap (int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
int x = 2, y = 3;
swap(&x, &y);
```

5.3.2 Entrada e saída

A saída padrão é uma função pré-definida que recebe como argumento um tipo numérico ou um vetor de caracteres. O nome dessa função é `escrever`. Já a entrada padrão é um conjunto de funções que retornam um `int`, um `real`, um `reald` ou um vetor de caracteres. Essas funções são:

- ler: lê uma cadeira de caracteres do tipo `caracte`;
- lerint: lê um número do tipo `int`;
- lerreal: lê um número do tipo `real`;
- lerreald: lê um número do tipo `reald`.

```
1    int x = lerint();
2    int y = lerint();
3    escrever("O resultado eh:");
4    escrever(x + y);
```

5.4 Funções pré-definidas

Além das funções de entrada e saída e de conversão de tipos, existem também as funções auxiliares:

- `tamanho`: recebe qualquer valor e retorna o seu tamanho ocupado em *bytes*;
- `alocar`: recebe um inteiro que representa a quantidade de *bytes* a serem alocados dinamicamente e retorna um ponteiro para o início da memória alocada;
- `deletar`: recebe um ponteiro e uma quantidade de *bytes* a ser desalocada da memória dinamicamente.

5.5 Programas

Programas em CPa podem ser definidos como uma sequência de declarações e definições onde uma das funções definidas delas deve ser uma função

```
int main(caractere* args, int n)
```

onde `args` é um ponteiro para o início do vetor de argumentos dados ao programa, e `n` é o tamanho desse vetor.

6 Bibliografia

OSÓRIO, Fernando S. Curso de: Linguagem C. São Leopoldo, 1992

A Gramática

A.1 Declarações

⟨*start*⟩ ::= ⟨*incs*⟩ ⟨*decs*⟩
| ⟨*decs*⟩

⟨*incs*⟩ ::= ⟨*inc*⟩ ⟨*incs*⟩
| ⟨*inc*⟩

⟨*inc*⟩ ::= '`importar`' STRING '`;`'

⟨*decs*⟩ ::= ⟨*dec*⟩ ⟨*decs*⟩
| ⟨*dec*⟩

⟨*dec*⟩ ::= ⟨*const-dec*⟩
| ⟨*var-dec*⟩
| ⟨*func-dec*⟩
| ⟨*struct-dec*⟩
| ⟨*enum-dec*⟩

⟨*const-dec*⟩ ::= '`const`' ⟨*type*⟩ ID ⟨*expr*⟩ '`;`'

⟨*var-dec*⟩ ::= ⟨*type*⟩ ⟨*id-decs*⟩ '`;`'

⟨*id-decs*⟩ ::= ⟨*id-dec*⟩ '`,`' ⟨*id-decs*⟩
| ⟨*id-dec*⟩

⟨*id-dec*⟩ ::= ID
| ID '`=`' ⟨*expr*⟩

⟨*struct-dec*⟩ ::= '`estrutura`' ID '`{`' ⟨*var-decs*⟩ '`}`'

⟨*enum-dec*⟩ ::= '`enum`' ID '`{`' ⟨*ids*⟩ '`}`'

⟨*ids*⟩ ::= ID
| ⟨*ids*⟩ ',' ID

⟨*var-decs*⟩ ::= ⟨*var-dec*⟩
| ⟨*var-decs*⟩ ⟨*var-dec*⟩

⟨*func-dec*⟩ ::= ⟨*func-sign*⟩ '`;`'
| ⟨*func-sign*⟩ ⟨*block*⟩

⟨*func-sign*⟩ ::= ⟨*type*⟩ ID '`(`' ⟨*params*⟩ '`)`'
| ⟨*type*⟩ ID '`(`' '`)`'

⟨*params*⟩ ::= ⟨*type*⟩ ⟨*id-dec*⟩
| ⟨*type*⟩ ⟨*id-dec*⟩ '`,`' ⟨*param-decs*⟩

⟨*block*⟩ ::= '`{`' ⟨*stmts*⟩ '`}`'

⟨*stmts*⟩ ::= ⟨*stmt*⟩
| ⟨*stmt*⟩ ⟨*stmts*⟩

⟨*type*⟩ ::= ID
| '`int`'
| '`caractere`'
| '`string`'
| '`real`'
| '`reald`'
| '`vazio`'
| ⟨*type*⟩ '`[`' ⟨*expr*⟩ '`]`'
| '`*`' ⟨*type*⟩

A.2 Comandos

⟨*stmt*⟩ ::= ⟨*dec*⟩
| ⟨*attribution*⟩
| ⟨*if-condition*⟩
| ⟨*switch-case*⟩
| ⟨*for*⟩
| ⟨*while*⟩
| ⟨*do-while*⟩
| ⟨*block*⟩
| ⟨*expr*⟩ '`;`'
| '`retornar`' '`;`'
| '`retornar`' ⟨*expr*⟩ '`;`'
| '`parar`' '`;`'
| '`continuar`' '`;`'
| '`irpara`' ID '`;`'
| ID '`:`'

⟨*attribution*⟩ ::= ⟨*expr-var*⟩ '`=`' ⟨*expr*⟩ '`;`'
| ⟨*expr-var*⟩ ⟨*bin-op*⟩ '`=`' ⟨*expr*⟩ '`;`'

⟨*if-condition*⟩ ::= ⟨*if*⟩
| ⟨*if*⟩ ⟨*else*⟩

⟨*if*⟩ ::= ‘`se`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*stmt*⟩

⟨*else*⟩ ::= ‘`cc`’ ⟨*stmt*⟩

⟨*while*⟩ ::= ‘`enquanto`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*stmt*⟩

⟨*do-while*⟩ ::= ‘`faz`’ ⟨*stmt*⟩ ‘`enquanto`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ‘;’

⟨*for*⟩ ::= ‘`para`’ ID ‘`de`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ‘`asc`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*stmt*⟩
| ‘`para`’ ID ‘`de`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ‘`desc`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ⟨*stmt*⟩

⟨*switch-case*⟩ := ‘`escolha`’ ‘`(`’ ⟨*expr*⟩ ‘`)`’ ‘`{`’ ⟨*cases*⟩ ‘`}`’

⟨*cases*⟩ ::= ⟨*case*⟩
| ⟨*case*⟩ ⟨*cases*⟩

⟨*case*⟩ ::= ‘`caso`’ ⟨*expr*⟩ ‘`:`’ ⟨*stmts*⟩
| ‘`cc`’ ‘`:`’ ⟨*stmts*⟩

A.3 Expressões

⟨*expr*⟩ ::= ⟨*expr-leaf*⟩
| ⟨*pre-op*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*pos-op*⟩
| ⟨*expr*⟩ ⟨*bin-op*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*relat-op*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ‘`?`’ ⟨*expr*⟩ ‘`:`’ ⟨*expr*⟩

⟨*bin-op*⟩ ::= ‘+’
| ‘−’
| ‘∗’
| ‘/’
| ‘`|`’
| ‘`||`’
| ‘`&`’
| ‘`&&`’
| ‘>>’
| ‘<<’

⟨*relat-op*⟩ ::= ‘>’
| ‘>=’
| ‘<’
| ‘<=’
| ‘==’
| ‘! =’

⟨*pre-op*⟩ ::= ‘−’
| ‘!’
| ‘∗’
| ‘&’
| ‘++’
| ‘−−’

⟨*pos-op*⟩ ::= ‘++’
| ‘−−’

⟨*expr-leaf*⟩ ::= ⟨*func-call*⟩
| ⟨*expr-lit*⟩
| ⟨*expr-var*⟩
| ‘(’ ⟨*expr*⟩ ‘)’

⟨*func-call*⟩ ::= ⟨*expr-var*⟩ ‘(’ ‘)’
| ⟨*expr-var*⟩ ‘(’ ⟨*args*⟩ ‘)’

⟨*args*⟩ ::= ⟨*expr*⟩
| ⟨*expr*⟩ ‘,’ ⟨*args*⟩

⟨*expr-lit*⟩ ::= INTEIRO
| REAL
| STRING
| CARACTERE
| ⟨*array-lit*⟩

⟨*array-lit*⟩ ::= ‘{’ ⟨*args*⟩ ‘}’

⟨*expr-var*⟩ ::= ID
| ⟨*field*⟩
| ⟨*array-el*⟩

⟨*field*⟩ ::= ID ‘.’ ID
| ⟨*expr-leaf*⟩ ‘.’ ID

⟨*array-el*⟩ ::= ID ‘[’ ⟨*expr*⟩ ‘]’
| ⟨*expr-leaf*⟩ ‘[’ ⟨*expr*⟩ ‘]’

B Tokens

B.1 Expressões regulares e Lista de tokens

Token	Regex
PLUS	'+'
PLUS2	'++'
MINUS	'−'
MINUS2	'−−'
AMPERSEND	'&'
AMPERSEND2	'&&'
BITOR	'|'
OR	'||'
STAR	'∗'
DIV	'/'
ATTR	'='
ATTRADD	'+ ='
ATTRSUB	'− ='
ATTRMUL	'∗ ='
ATTRDIV	'/ ='
ATTRSHIFTL	'>>='
ATTRSHIFTR	'<<='
ATTRBITOR	'| ='
ATTROR	'|| ='
ATTRAMPERSEND	'&='
ATTRAMPERSEND2	'&&='
EQQ	'=='
NEQ	'!='
LEQ	'<='
GEQ	'<='
NEG	'!'
LT	'<'
GT	'>'
SHIFTL	'<<'
SHIFTR	'>>'

Tabela 1: Palavras reservadas.

Token	Regex
INT	'int'
REAL	'real'
REALD	'reald'
CARACTERE	'caractere'
VOID	'vazio'

Tabela 2: Tipos primitivos.

Token	Regex
SEMI	';'
COLON	':'
DOT	'.'
COMMA	','
LPAREN	'('
RPAREN	')'
LBRACKET	'['
RBRACKET	']'
LBRACE	'{'
RBRACE	'}'
QUESTION	'?'

Tabela 3: Pontuação.

[1]commentblock retirado de: `https://stackoverflow.com/questions/13569827/complete-comments-regex-for-lex`

[2]STRING retirado de: `https://stackoverflow.com/questions/2039795/regular-expression-for-a-literal-in-flex-lex`

Token	Regex
IMPORTAR	‘importar’
CONSTANTE	‘const’
PARAR	‘parar’
CONTINUAR	‘continuar’
RETORNAR	‘retornar’
IRPARA	‘irpara’
SE	‘se’
CC	‘cc’
ESCOLHA	‘escolha’
CASO	‘caso’
PARA	‘para’
DE	‘de’
ASC	‘asc’
DESC	‘desc’
FAZER	‘fazer’
ESTRUTURA	‘estrutura’
ENUM	‘enum’

Tabela 4: Palavras chave

Token	Regex
LINECOMMENT	"//"([^\n])* "\n"
BLOCKCOMMENT [1]	"/*"([^*] | "*"+ [^*/])* "*"+ "/"
DIGIT	0-9
LETTER	a-zA-Z
HEX	-?0x(DIGIT|[A-Fa-f])+
FRAC	-?(DIGIT*"DIGIT+|DIGIT+"."DIGIT*)
SCINOT	"e" -? DIGIT+
INTEIRO	-? (DIGIT+ | HEX)
REAL	(FRAC|-?DIGIT+|-?DIGIT+SCINOT)f
REALD	FRAC|FRAC SCINOT|-?DIGIT+SCINOT
STRING [2]	\"([^"\]| \\.)*\"
CHAR	\’([^’\]| \\.)\’
ID	(_*LETTER | LETTER)(LETTER|DIGIT|"_")*

Tabela 5: Expressões regulares de outros tokens