ANALISADOR SINTÁTICO – TRABALHO FINAL DA DISCIPLINA DE COMPILADORES – PARTE 2: RELATÓRIO

Rodrigo Matheus Rodrigues de Oliveira; Daniel Winter Santos Rocha; Mírian Monique Silva Lacerda; Carolina Teciliana da Silva Araújo.

Estudantes de Engenharia da Computação, IFMG – *campus* Bambuí.

1 INTRODUÇÃO

Este documento contém um relatório detalhando o desenvolvimento e testes de um analisador sintático, segunda parte do trabalho final da disciplina de Compiladores, do oitavo período do curso Bacharelado em Engenharia da Computação.

Bevale (2017) afirma que a análise sintática é a segunda etapa do processo de compilação (a primeira é a análise léxica) e na maioria dos casos utiliza gramática livre de contexto para especificar a sintaxe de uma linguagem de programação. A principal tarefa do analisador sintático, conhecido como *parsing*, é determinar se o programa de entrada representado pelo fluxo de tokens possui as sentenças válidas para a linguagem de programação. No caso afirmativo, queremos adicionalmente descobrir a maneira (ou uma das maneiras) pela qual a cadeia pode ser derivada seguindo as regras da gramática.

Foi solicitado pelo professor que se implementasse um analisador sintático para a linguagem “Javazim”, cuja descrição se encontra no Anexo I. O analisador deveria ser implementado em linguagem Java, com o auxílio da ferramenta Java Cup, além da ferramenta JFlex. O programa implementado deverá retornar um objeto da classe *Token* sempre que chamado, podendo haver herança da classe *yytoken* do JFlex ou utilizar uma classe própria. A saída do programa deverá exibir o *token* em questão, juntamente de seu lexema. Além de reconhecer os *tokens* da linguagem, é necessário que reconheça possíveis erros e retorne ao usuário a linha e coluna de ocorrência. Além disso, linhas em branco, espaços e tabulações não devem ser reconhecidas como *tokens*, portanto deverão ser descartados.

2 METODOLOGIA

Foi utilizado para desenvolvimento deste trabalho o *software* NetBeans IDE 8.2[1] juntamente do gerador de analisador léxico JFlex[2], para reconhecimento dos *tokens* e o Java Cup para reportar possíveis erros ocorridos no programa-fonte, informando qual o erro encontrado e sua localização no arquivo-fonte. No Anexo II é possível encontrar o conteúdo do documento gerado a partir da linguagem Javazim, ou seja, os *tokens* gerados para a linguagem. Estes *tokens* foram feitos em um arquivo de extensão “flex” e a biblioteca utilizada os traduziu para um arquivo de extensão “java”. Desta forma, o programa desenvolvido poderá reconhecer os *tokens* a partir de uma entrada. No Anexo III encontra-se o arquivo .cup desenvolvido para o analisador sintático.

A Figura 1 exibe uma interface que foi desenvolvida para interação com o usuário. Buscando ser simples e intuitiva, esta interface possui três campos de texto, um onde será digitado o código fonte em questão, o segundo onde serão exibidos os resultados da análise léxica e o terceiro onde será exibido o resultado da análise sintática. Além de botões para início das análises, limpar os campos de texto e importar um arquivo com um programa previamente feito.

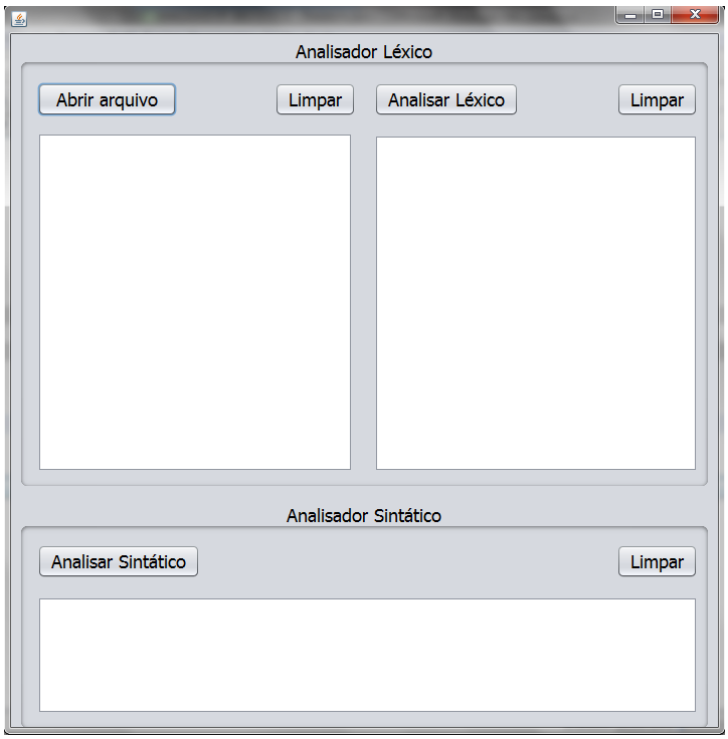

Figura 1 – Interface do analisador sintático.
Fonte: Os autores (2019).

[1] https://netbeans.org/downloads/8.2/
[2] https://jflex.de/download.html

Detalhes a respeito do desenvolvimento do analisador léxico serão omitidas deste artigo, uma vez que já foram apresentadas mais detalhadamente na primeira parte do trabalho.

Após o desenvolvimento dos analisadores solicitados pelo professor, deu-se início à fase de testes. Seis algoritmos foram implementados na linguagem Javazim, sendo três sem erros sintáticos e três contendo erros sintáticos, a fim de validar a eficiência dos analisadores desenvolvidos. O conteúdo, tais quais os resultados dos testes em questão, se encontram na sessão seguinte deste documento.

3 RESULTADOS E DISCUSSÃO

Este trabalho gerou uma série de arquivos contendo o código fonte de sua implementação, porém serão omitidos neste relatório. Em anexo, juntamente com o arquivo deste documento, estarão os arquivos contendo todo o código fonte gerado, além do projeto desenvolvido no NetBeans IDE 8.2 e outros arquivos considerados pertinentes por parte dos autores.

Após o desenvolvimento do algoritmo do analisador, foram efetuados seis testes, divididos em três com entrada correta e três com entrada contendo erros sintáticos.

A Figura 3 contém as saídas dos testes de entrada correta, já os testes com entradas com erros léxicos se encontram na Figura 4. O Anexo IV contém todas as entradas de forma completa, além de estar disponível em um arquivo nomeado “testes.txt” juntamente dos demais arquivos deste trabalho.

A caixa de texto inferior apresenta uma mensagem na cor verde dizendo que não há erros sintáticos, ou uma mensagem de com a cor vermelha dizendo que há erro sintático, e apontando sua localização na forma de linha e coluna dentro do código. Caso haja mais de um erro, o programa apontará somente o primeiro encontrado, seguindo a ordem da primeira para a última linha, da esquerda para a direita. O segundo erro apenas será identificado no momento em que o primeiro for corrigido e o botão de análise sintática for acionado novamente.

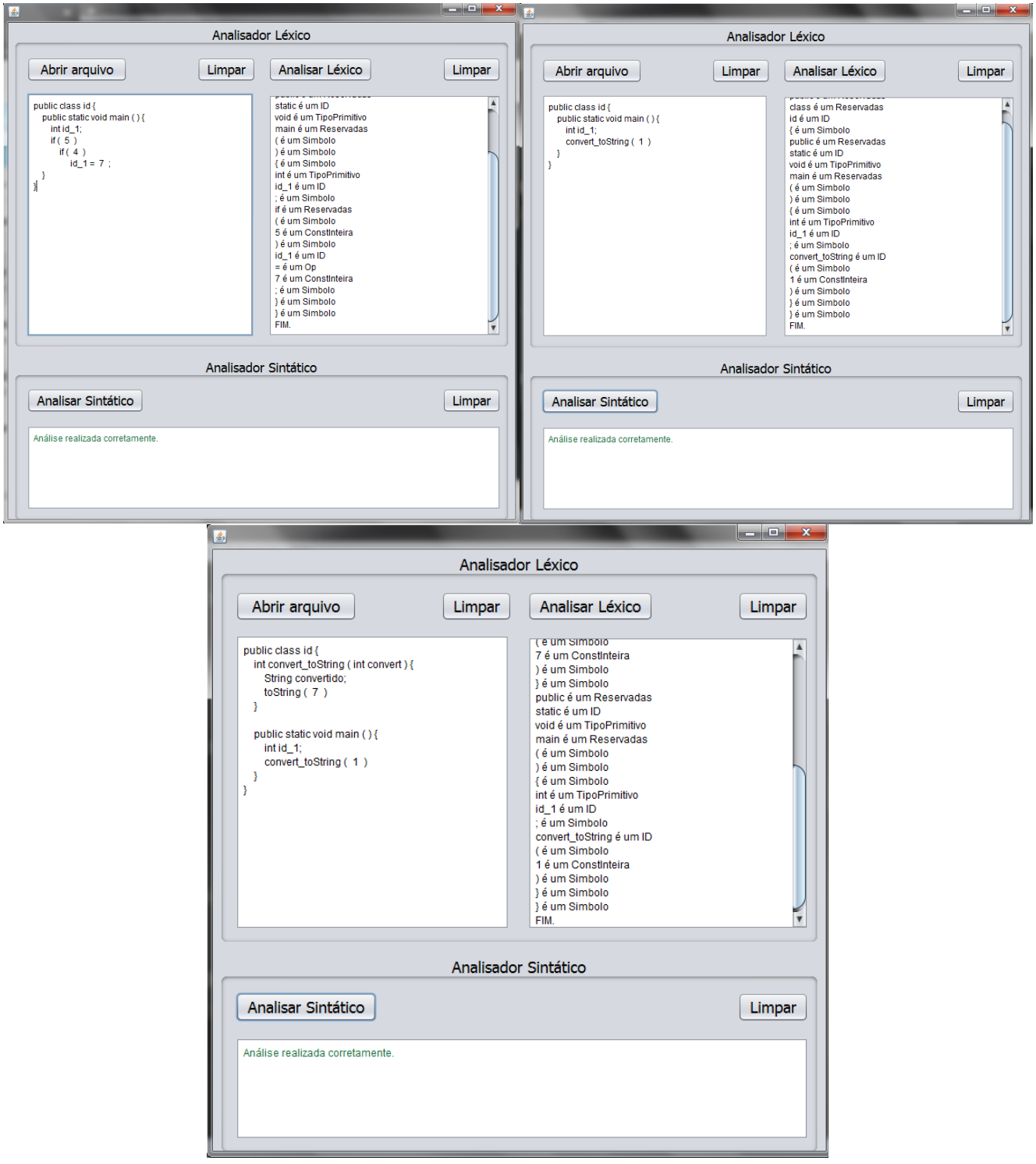

Figura 3 – Testes sem erros léxicos.

Fonte: Os autores (2019).

Como nos três testes acima não haviam erros sintáticos, foi emitida uma mensagem em verde dizendo que a análise foi realizada corretamente. A seguir se encontram três testes contendo erros sintáticos.

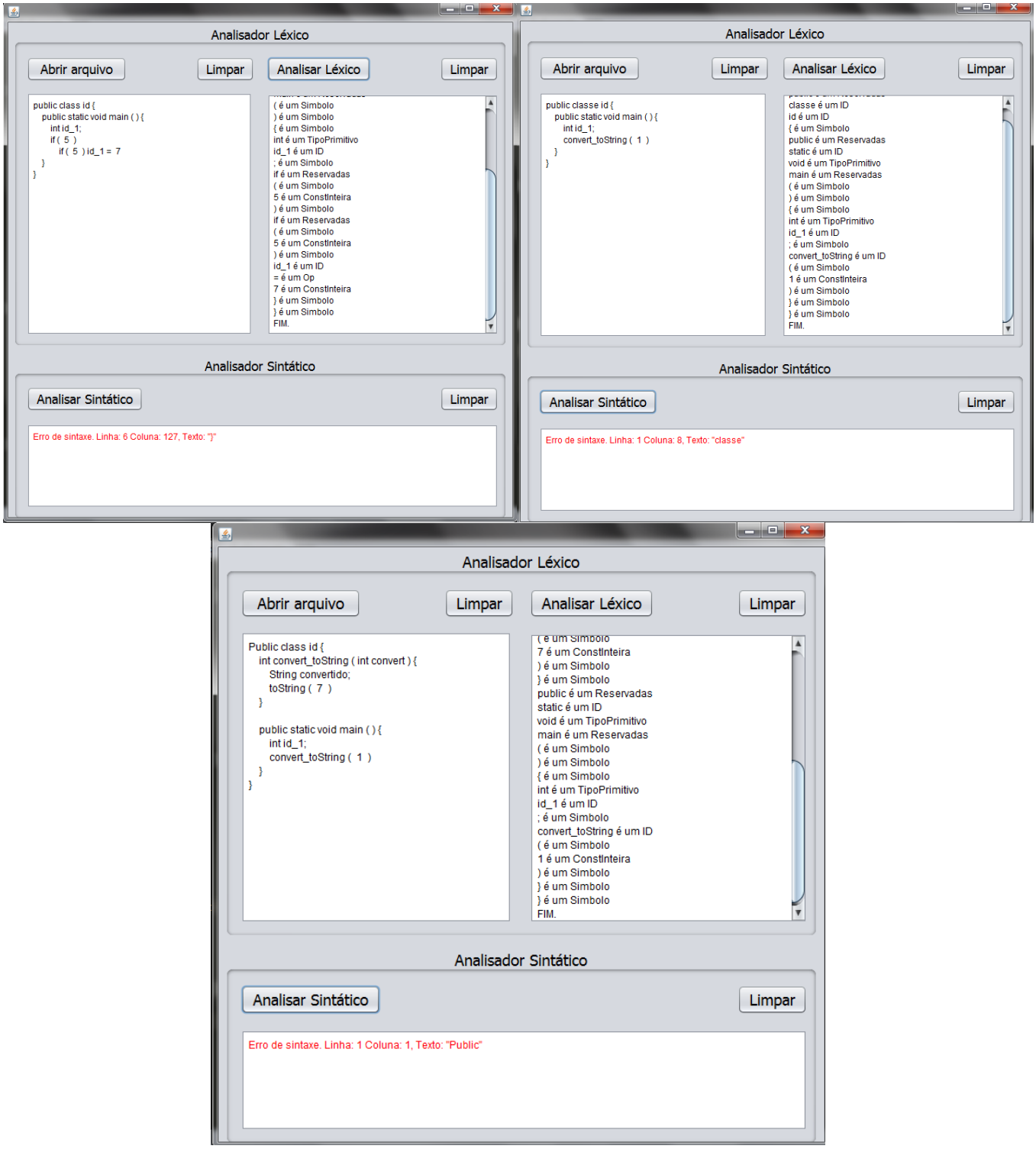

Figura 4 – Testes contendo erros léxicos.

Fonte: Os autores (2019).

Durante a geração dos arquivos de extensão “java” pelo JFlex e Java Cup, foi possível identificar informações importantes a respeito da análise sintática, tal qual número de terminais e não terminais, produções, conflitos. A Figura 5 trás a mensagem gerada pelo NetBens IDE 8.2 com tais informações.

```
------- CUP v0.11a beta 20060608 Parser Generation Summary -------
  0 errors and 3 warnings
  33 terminals, 20 non-terminals, and 41 productions declared,
  producing 178 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "Sintax.java", and "sym.java".
---------------------------------------------------- (v0.11a beta 20060608)
CONSTRUÍDO COM SUCESSO (tempo total: 1 segundo)
```

Figura 5 – Informações durante a execução do analisador.
Fonte: Os autores (2019).

É importante frisar que, caso haja testes do analisador léxico e sintático por partes de terceiros, o código fonte deve ser alterado na função “main”, modificando os caminhos (*paths*) conforme a máquina que está sendo utilizada.

4 CONCLUSÃO

Este trabalho se mostrou uma importante forma de colocar em prática todo o conteúdo teórico visto em sala de aula, tanto da disciplina de Compiladores quanto da anteriormente cursada disciplina de Linguagens Formais e Autômatos. Não só mostrou a importância dos analisadores léxico e sintático dentro do compilador, como também se fez um grande auxiliador na prática com desenvolvimento destes analisadores e utilização das ferramentas anteriormente citadas.

Todo o conteúdo aqui desenvolvido e conhecimento absorvido pelos autores servirão de base para trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

Bevale, Tais**.** O que é e como funciona a análise sintática ascendente e descendente?. Disponível em: <https://pt.stackoverflow.com/questions/181635/o-que-%c3%a9-e-como-funciona-a-an%c3%a1lise-sint%c3%a1tica-ascendente-e-descendente>. Acesso em: 11 dez. 2019.

Santos, Adriano Lages dos. Material disponibilizado pelo professor. Disponível em: <https://sites.google.com/ifmg.edu.br/adriano/inicial/compiladores>. Acesso em: 11 dez. 2019.

Anexo I – A linguagem Javazim.

A linguagem *Javazim*

Programa ⟶ Classe < EOF >
Classe ⟶ "public" "class" < ID > "{" ListaMetodo Main "}"
DeclaracaoVar ⟶ T ipo < ID > ";"
ListaMetodo ⟶ ListaMetodo Metodo | λ
Metodo ⟶ Tipo < ID > "(" ListaParam")" "{" (DeclaracaoVar)∗ ListaCmd Retorno }
ListaParam ⟶ P aram ”, ” ListaParam | Param
Param ⟶ Tipo < ID >
Retorno ⟶ "return" Expressao";" | λ
Main ⟶ "public" "static" "void" "main" "(" ")" "{" (DeclaracaoVar)∗ ListaCmd "}"
Tipo ⟶ TipoPrimitivo"[" "]" | TipoPrimitivo
TipoPrimitivo ⟶ "boolean" | "int" | "String" | "float" | "void"

ListaCmd ⟶ ListaCmd Cmd | λ
Cmd ⟶ "{" ListaCmd "}"
| CmdIF | CmdWhile | CmdAtrib | CmdFunc | CmdPrint | CmdPrintln
CmdIF ⟶ "if" "(" Expressao ")" Cmd
| "if" "(" Expressao ")" Cmd "else" Cmd
CmdWhile ⟶ "while" "(" Expressao ")" Cmd
CmdPrint ⟶ "print" "(" Expressao ")" ";"
CmdPrintln ⟶ "println" "(" Expressao ")" ";"
CmdAtrib ⟶ < ID > "=" Expressao ";"
| < ID > "[" Expressao "]" "=" Expressao";"
CmdFunc ⟶ < ID > "(" (Expressao ("," Expressao)∗)? ")"
Expressao ⟶ Expressao Op Expressao
| < ID > "[" Expressao"]" | < ID >
| < ID > "(" (Expressao ("," Expressao)∗)? ")"
| < ConstInteira > | < ConstReal > | < ConstString > | "true" | "false"
| "new" TipoPrimitivo"[" Expressao "]"
| OpUnario Expressao | "(" Expressao")"
Op ⟶ "||" | "&&" | "<" | "<=" | ">" | ">=" | "==" | "!=" | "/" | "*" | "-" | "+"
OpUnario ⟶ "-" | "!"

Anexo II – Conteúdo gerado para a ferramenta JFlex.

```
package codigo;
import java_cup.runtime.Symbol;
%%
%class LexerCup
%type java_cup.runtime.Symbol
%cup
%full
%line
%char

L=[a-zA-Z_]+ // LETRAS.
D=[0-9]+ // DÍGITOS.
S=["("|")"|"["|"]"|"{"|"}"|(')|;]+ // SÍMBOLOS DA LINGUAGEM.
E=[@,#,$,¨] // CARACTERES ESPECIAIS.
Enter=[\n,\r]+ // QUEBRA DE LINHA.
Espaco=[" ", ,\t,\n,\r]* // ESPAÇO.
ID={L}({L}|{D}|_)* // IDENTIFICADORES.
ConstInteira=("(-"{D}+")")|{D}+ // CONSTANTE INTEIRA.
ConstReal=({D})*(.)({D})* // CONTANTE REAL.
ConstString=(')({L}|{D}|{E})*(') // CONSTANTE STRING.

TipoPrimitivo=(boolean)|
              (int)|
              (String)|
              (float)
Tipo={TipoPrimitivo}("[")("]")
DeclaracaoVar={TipoPrimitivo}([])([])(;) | {Tipo}([])([])(;)

ListaCmd=(Cmd)*
Cmd=("{"){ListaCmd}("}")|
   CmdIF|
   CmdWhile|
   CmdAtrib|
   CmdFunc|
   CmdPrint|
   CmdPrintln
CmdIF=(if)("("){Expressao}(")"){Cmd}|
     (if)("("){Expressao}(")"){Cmd}(else){Cmd}
CmdWhile=(while)("("){Expressao}(")")
CmdPrint=(print)("("){Expressao}(")")(;)
CmdPrintln=(println)("("){Expressao}(")")(;)
CmdAtrib={ID}(=){Expressao}(;)|
        {ID}("["){Expressao}("]")(=){Expressao}(;)
CmdFunc={ID}({Expressao}(","{Expressao})*)
Expressao={D}{Op}{D}|
          {ID}([])([])| // CONSERTAR.
          {ID}|
          {ID}("(") (")")|
          {ConstInteira}|
          {ConstReal}|
          {ConstString}|
          (true)|
          (false)|
          (new){TipoPrimitivo}([])([])|
          {OpUnario}{D}
Op=[||, &&, <, >, +, -, *, /]
OpUnario=[_,!]
Public=(public)
```

```
%{
   private Symbol symbol(int type, Object value){
      return new Symbol(type, yyline, yycolumn, value);
   }
   private Symbol symbol(int type){
      return new Symbol(type, yyline, yycolumn);
   }
%}
%%

{Espaco} {return new Symbol(sym.E, yychar, yyline, yytext());}
"//".* {/*Ignore*/} // DEFINE O QUE É COMENTÁRIO.
{Op} {return new Symbol(sym.Op, yychar, yyline, yytext());}
{OpUnario} {return new Symbol(sym.OpUnario, yychar, yyline, yytext());}
{D} {return new Symbol(sym.Digito, yychar, yyline, yytext());}
{TipoPrimitivo} {return new Symbol(sym.TipoPrimitivo, yychar, yyline, yytext());}
(if) {return new Symbol(sym.If, yychar, yyline, yytext());}
(else) {return new Symbol(sym.Else, yychar, yyline, yytext());}
(while) {return new Symbol(sym.While, yychar, yyline, yytext());}
(print) {return new Symbol(sym.Print, yychar, yyline, yytext());}
(println) {return new Symbol(sym.Print, yychar, yyline, yytext());}
(public) {return new Symbol(sym.Public, yychar, yyline, yytext());}
(static) {return new Symbol(sym.Static, yychar, yyline, yytext());}
(void) {return new Symbol(sym.Void, yychar, yyline, yytext());}
(main) {return new Symbol(sym.Main, yychar, yyline, yytext());}
(class) {return new Symbol(sym.Class, yychar, yyline, yytext());}
(return) {return new Symbol(sym.Retorno, yychar, yyline, yytext());}
(new) {return new Symbol(sym.New, yychar, yyline, yytext());}
(True) {return new Symbol(sym.True, yychar, yyline, yytext());}
(False) {return new Symbol(sym.False, yychar, yyline, yytext());}
"=" {return new Symbol(sym.Igual, yychar, yyline, yytext());}
//"+" {return new Symbol(sym.Soma, yychar, yyline, yytext());}
//"-" {return new Symbol(sym.Subtracao, yychar, yyline, yytext());}
//"*" {return new Symbol(sym.Multiplicacao, yychar, yyline, yytext());}
//"/" {return new Symbol(sym.Divisao, yychar, yyline, yytext());}
"(" {return new Symbol(sym.AbreParentese, yychar, yyline, yytext());}
")" {return new Symbol(sym.FechaParentese, yychar, yyline, yytext());}
"[" {return new Symbol(sym.AbreCochete, yychar, yyline, yytext());}
"]" {return new Symbol(sym.FechaCochete, yychar, yyline, yytext());}
"{" {return new Symbol(sym.AbreChave, yychar, yyline, yytext());}
"}" {return new Symbol(sym.FechaChave, yychar, yyline, yytext());}
";" {return new Symbol(sym.PontoVirgula, yychar, yyline, yytext());}
"," {return new Symbol(sym.Virgula, yychar, yyline, yytext());}
{ConstInteira} {return new Symbol(sym.ConstInteira, yychar, yyline, yytext());}
{ConstReal} {return new Symbol(sym.ConstReal, yychar, yyline, yytext());}
{ConstString} {return new Symbol(sym.ConstString, yychar, yyline, yytext());}
{L}({L}|{D}|_)* {return new Symbol(sym.ID, yychar, yyline, yytext());}
. {return new Symbol(sym.ERROR, yychar, yyline, yytext());}
```

Anexo III – Conteúdo gerado para a ferramenta Java Cup.

```
package codigo;

import java_cup.runtime.Symbol;

parser code
{:
   private Symbol s;

   public void syntax_error(Symbol s){
      this.s = s;
   }
```

```
    public Symbol getS(){
        return this.s;
}
:};

terminal
    Public, Static, Void, Main, Class, New,
    If, Else, While, Print, Println, PontoVirgula, Virgula,
    Igual, Soma, Subtracao, Multiplicacao, Divisao,
    AbreParentese, FechaParentese, AbreCochete, FechaCochete, AbreChave, FechaChave,
    Op, OpUnario, TipoPrimitivo, Digito,
    ConstInteira, ConstReal, ConstString, True, False,
    ID, Retorno,
    DeclaracaoVar,
    E, ERROR
;

non terminal
    PROGRAMA, CLASSE, DECLARACAOVAR, LISTAMETODO, METODO, LISTAPARAM,
PARAM,
    RETORNO, MAIN, SENTENCA, TIPO, LISTACMD, CMD, CMDIF, CMDWHILE,
    CMDPRINT, CMDPRINTLN, CMDATRIB, CMDFUNC, EXPRESSAO
;

start with PROGRAMA;

PROGRAMA::= CLASSE;
CLASSE::= Public E Class E ID E AbreChave E METODO E MAIN E FechaChave |
        Public E Class E ID E AbreChave E MAIN E FechaChave;
DECLARACAOVAR::= TIPO E ID PontoVirgula;
LISTAMETODO::= METODO;
METODO::= TIPO E ID E AbreParentese E LISTAPARAM E FechaParentese E AbreChave E
DECLARACAOVAR E CMD E FechaChave;
LISTAPARAM::= PARAM;
PARAM::= TIPO E ID;
RETORNO::= Retorno E ID E PontoVirgula;

MAIN::= Public E Static E Void E Main E AbreParentese E FechaParentese E AbreChave E
FechaChave |
       Public E Static E Void E Main E AbreParentese E FechaParentese E AbreChave E
DECLARACAOVAR E CMD E FechaChave;
SENTENCA::= DECLARACAOVAR E LISTACMD; // descobrir porque está bugando aqui.
TIPO::= TipoPrimitivo AbreCochete E FechaCochete | TipoPrimitivo; // NÃO FUNCIOU E AQUI.
CMD::= CMDIF | CMDWHILE | CMDPRINT | CMDPRINTLN | CMDATRIB | CMDFUNC;
CMDIF::= If E AbreParentese E EXPRESSAO E FechaParentese E CMD |
       If AbreParentese E EXPRESSAO E FechaParentese E CMD E Else E CMD;
CMDWHILE::= While E AbreParentese E EXPRESSAO E FechaParentese ;
CMDPRINT::= Print E AbreParentese E EXPRESSAO E FechaParentese E PontoVirgula;
CMDPRINTLN::= PrintIn E AbreParentese E EXPRESSAO E FechaParentese E PontoVirgula;
CMDATRIB::= ID E Igual E EXPRESSAO E PontoVirgula |
         ID E AbreCochete E EXPRESSAO E FechaCochete E Igual E EXPRESSAO;
CMDFUNC::= ID E AbreParentese E EXPRESSAO E FechaParentese;
EXPRESSAO::= Digito E Op E Digito |
          ID E |
          ID E AbreCochete E FechaCochete |
          ID E AbreParentese E EXPRESSAO E FechaParentese |
          ConstInteira |
          ConstReal |
          ConstString |
          True |
          False |
```

```
New E TipoPrimitivo E AbreCochete E EXPRESSAO E FechaCochete |
OpUnario E EXPRESSAO |
AbreParentese E EXPRESSAO E FechaParentese;
```

Anexo IV – Entradas dos testes das análises sintáticas.

Testes sem erro sintático;

TESTE 1.

```
public class id {
   public static void main ( ) {
      int id_1;
      if ( 5 )
         if ( 5 ) id_1 = 7 ;
   }
}
```

TESTE 2.

```
public class id {
   public static void main ( ) {
      int id_1;
      convert_toString ( 1 )
   }
}
```

TESTE 3.

```
public class id {
   int convert_toString ( int convert ) {
      String convertido;
      toString ( 7 )
   }

   public static void main ( ) {
      int id_1;
      convert_toString ( 1 )
   }
}
```

Testes com erro sintático:

TESTE 4.

```
public class id {
   public static void main ( ) {
      int id_1;
      if ( 5 )
         if ( 5 ) id_1 = 7
   }
}
```

TESTE 5.

```
public classe id {
   public static void main ( ) {
      int id_1;
      convert_toString ( 1 )
   }
}
```

TESTE 6.

```
Public class id {
   int convert_toString  ( int convert  ) {
      String convertido;
      toString (  7  )
   }

   public static void  main  ( ) {
      int id_1;
      convert_toString  (  1  )
   }
}
```