Escola Politécnica
da Universidade de
São Paulo

Segunda Prova

PCS-2056 : Linguagens e Compiladores

Autores:
Eduardo Russo
Helio Kazuo Nagamachi

Professor:
Ricardo Rocha

10 de dezembro de 2010

Sumário

1 Objetivo

Este documento tem como objetivo descrever a construção do compilador de Lua – uma linguagem de programação de extensão – para um ambiente de execução escrito em Java para a segunda prova da disciplina de **Linguagens e Compiladores**.

2 Gramática

A gramática da linguagem Lua apresentada para a prova em Wirth – código 1 – foi reduzida de forma a necessitar de apenas três Autômatos de Pilha Estruturado (APEs).

Código 1: *Gramática.*

```
1  trecho = {comando [";"]} [ultimocomando [";"]].
2  bloco = trecho.
3  comando = listavar "=" listaexp |
4              chamadadefuncao |
5              "do" bloco "end" |
6              "while" exp "do" bloco "end" |
7              "repeat" bloco "until" exp |
8              "if" exp "then" bloco {"elseif" exp "then" bloco} ["
                  else" bloco] "end" |
9              "for" "Nome" "=" exp "," exp ["," exp] "do" bloco "
                  end" |
10             "for" listadenomes "in" listaexp "do" bloco "end" |
11             "function" nomedafuncao corpodafuncao |
12             "local" "function" "Nome" corpodafuncao |
13             "local" listadenomes ["=" listaexp].
14 ultimocomando = "return" [listaexp] | "break".
15 nomedafuncao = "Nome" {"." "Nome"} [":" "Nome"].
16 listavar = var {"," var}.
17 var = "Nome" | expprefixo "[" exp "]" | expprefixo "." "Nome".
18 listadenomes = "Nome" {"," "Nome" } .
19 listaexp = {exp ","} exp.
20 exp = "nil" | "false" | "true" | "Numero" | "Cadeia" |  "..." |
      funcao |
21             expprefixo | construtortabela | exp opbin exp |
                  opunaria exp.
22 expprefixo = var | chamadadefuncao | "(" exp ")".
23 chamadadefuncao = expprefixo args | expprefixo ":" "Nome" args.
24 args = "(" [listaexp] ")" | construtortabela | "Cadeia".
25 funcao = "function" corpodafuncao.
26 corpodafuncao = "(" [listapar] ")" bloco "end".
27 listapar = listadenomes ["," "..."] | "...".
28 construtortabela = "{" [listadecampos] "}".
29 listadecampos = campo { separadordecampos campo} [
      separadordecampos ].
30 campo = "[" exp "]" "=" exp | "Nome" "=" exp | exp.
31 separadordecampos = "," | ";".
```

```
opbin = "+" | "-" | "*" | "/" | "^" | "%" | ".." |
    "<" | "<=" | ">" | ">=" | "==" | "~=" |
    "and" | "or" .
opunaria = "-" | "not" | "#".
```

Para a redução, foi utilizado um aplicativo desenvolvido em Python para facilitar o trabalho. Esta redução é apresentada no código 2

Código 2: *Gramática reduzida.*

```
comando=(("Nome"|(("Nome"|(("Nome"|"("exp")"){"["exp"]"|"."Nome
   "}[":"Nome"]("("[({exp","}exp)]")"|("{"[(("["exp"]""="exp|"
   Nome""="exp|exp){(","|";")("["exp"]""="exp|"Nome""="exp|exp)
   }[(","|";")])]"}")|"Cadeia"){[":"Nome"]("("[({exp","}exp)]")
   "|("{"[(("["exp"]""="exp|"Nome""="exp|exp){(","|";")("["exp
   "]""="exp|"Nome""="exp|exp)}[(","|";")])]"}")|"Cadeia")})|"("
   exp")"){"["exp"]"|"."Nome"})("["exp"]"|"."Nome")){","("Nome
   "|(("Nome"|(("Nome"|"("exp")"){"["exp"]"|"."Nome"}[":"Nome
   "]("("[({exp","}exp)]")"|("{"[(("["exp"]""="exp|"Nome""="exp|
   exp){(","|";")("["exp"]""="exp|"Nome""="exp|exp)}[(","|";")])
   ]"}")|"Cadeia"){[":"Nome"]("("[({exp","}exp)]")"|("{"[(("["
   exp"]""="exp|"Nome""="exp|exp){(","|";")("["exp"]""="exp|"Nome
   ""="exp|exp)}[(","|";")])]"}")|"Cadeia")})|"("exp")"){"["exp
   "]"|"."Nome"})("["exp"]"|"."Nome"))})"="({exp","}exp)|(("
   Nome"|"("exp")"){"["exp"]"|"."Nome"}[":"Nome"]("("[({exp","}
   exp)]")"|("{"[(("["exp"]""="exp|"Nome""="exp|exp){(","|";")
   ("["exp"]""="exp|"Nome""="exp|exp)}[(","|";")])]"}")|"Cadeia")
   {[":"Nome"]("("[({exp","}exp)]")"|("{"[(("["exp"]""="exp|"
   Nome""="exp|exp){(","|";")("["exp"]""="exp|"Nome""="exp|exp)
   }[(","|";")])]"}")|"Cadeia")})|"do"(({comando[";"]}[("return
   "[({exp","}exp)]|"break")[";"]]))"end"|"while"exp"do"(({
   comando[";"]}[("return"[({exp","}exp)]|"break")[";"]]))"end"|"
   repeat"(({comando[";"]}[("return"[({exp","}exp)]|"break")
   [";"]]))"until"exp|"if"exp"then"(({comando[";"]}[("return"[({
   exp","}exp)]|"break")[";"]])){"elseif"exp"then"(({comando
   [";"]}[("return"[({exp","}exp)]|"break")[";"]]))}["else"(({
   comando[";"]}[("return"[({exp","}exp)]|"break")[";"]]))]"end
   "|"for""Nome""="exp","exp["exp]"do"(({comando[";"]}[("return
   "[({exp","}exp)]|"break")[";"]]))"end"|"for"("Nome"{","""Nome
   "})"in"({exp","}exp)"do"(({comando[";"]}[("return"[({exp","}
   exp)]|"break")[";"]]))"end"|"function"("Nome"{"."""Nome"}[":""
   Nome"])("("[(("Nome"{","""Nome"})[","""..."]|"...")]")"(({
   comando[";"]}[("return"[({exp","}exp)]|"break")[";"]]))"end")
   |"local""function""Nome"("("[(("Nome"{","""Nome"})
   [","""..."]|"...")]")"(({comando[";"]}[("return"[({exp","}exp)
   ]|"break")[";"]]))"end")|"local"("Nome"{","""Nome"})["="({exp
   ","}exp)].

funcao="function"("("[(("Nome"{","""Nome"})[","""..."]|"...")]")
   "(({comando[";"]}[("return"[({exp","}exp)]|"break")[";"]]))"
```

```
    end").
4
5 exp=("nil"|"false"|"true"|"Numero"|"Cadeia"|"..."|funcao|(("Nome
    "|(("Nome"|"("exp")"){"["exp"]"|"."Nome"}[":"Nome"]("("[({
    exp","}exp)]")"|("{"[(("["exp"]""="exp|"Nome""="exp|exp)
    {(","|";")("["exp"]""="exp|"Nome""="exp|exp)}[(","|";")])]"}")
    |"Cadeia"){["":"Nome"]("("[({exp","}exp)]")"|("{"[(("["exp
    "]""="exp|"Nome""="exp|exp){(","|";")("["exp"]""="exp|"Nome
    ""="exp|exp)}[(","|";")])]"}")|"Cadeia")})|"("exp")"){"["exp
    "]"|"."Nome"})|("{"[(("["exp"]""="exp|"Nome""="exp|exp)
    {(","|";")("["exp"]""="exp|"Nome""="exp|exp)}[(","|";")])]"}")
    |("-"|"not"|"#")exp)
    {("+"|"-"|"*"|"/"|"^"|"%"|".."|"<"|"<="|">"|">="|"=="|"~="|"
    and"|"or")("nil"|"false"|"true"|"Numero"|"Cadeia"|"..."|funcao
    |(("Nome"|(("Nome"|"("exp")"){"["exp"]"|"."Nome"}[":"Nome
    "]("("[({exp","}exp)]")"|("{"[(("["exp"]""="exp|"Nome""="exp|
    exp){(","|";")("["exp"]""="exp|"Nome""="exp|exp)}[(","|";")])
    ]"}")|"Cadeia"){["":"Nome"]("("[({exp","}exp)]")"|("{"[(("["
    exp"]""="exp|"Nome""="exp|exp){(","|";")("["exp"]""="exp|"Nome
    ""="exp|exp)}[(","|";")])]"}")|"Cadeia")})|"("exp")"){"["exp
    "]"|"."Nome"})|("{"[(("["exp"]""="exp|"Nome""="exp|exp)
    {(","|";")("["exp"]""="exp|"Nome""="exp|exp)}[(","|";")])]"}")
    |("-"|"not"|"#")exp)}.
```

3 Análise léxica

Para a análise léxica, foram identificadas as palavras reservadas na linguagem apresentadas a seguir:

#	-	+	*	/	∧
<	<=	>	>=	==	∼=
:	[	]	=	{	}
false	true	return	not	break	local
if	then	elseif	else	while	repeat
%	.	..	...	and	or
&	;	(	)	end	nil
function	for	in	do	until	

Considerando essas palavras reservadas, mas as possíveis sequências de palavras, foram

criados autômatos para reconhecer e retornar os *tokens*. Seus tipos são apresentados no código 3.

Código 3: *Tipos de tokens definidos: palavra reservada, identificador, número, final de arquivo e cadeia de palavras.*

```
1  package lex;
2
3  /**
4   * Os tipos de token
5   */
6  public enum TokenType {
7      RESERVED_WORD , IDENTIFIER , NUMBER , EOF , CADEIA
8  }
```

A construção do léxico utilizou os códigos apresentados na figura 1 e sua classe principal – que chama as funções dos outros códigos – encontra-se no código 4.

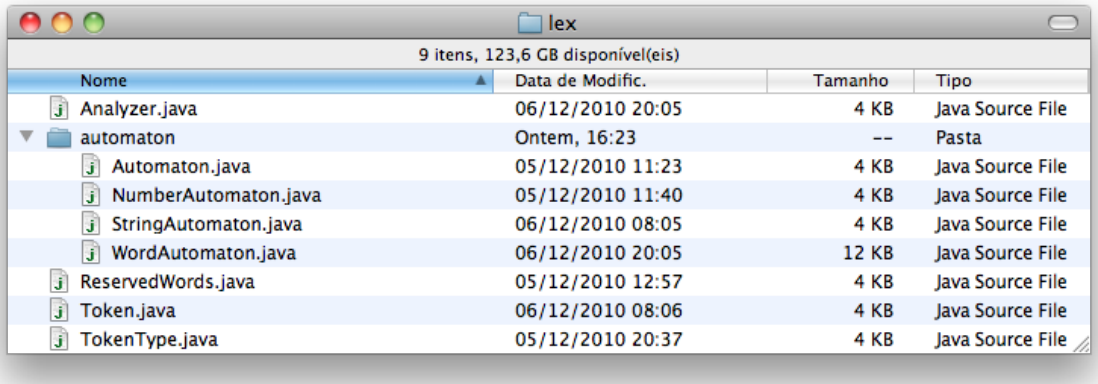

Figura 1: *Códigos utilizados para construir o analisador léxico.*

Código 4: *Classe "Analyzer.java" do pacote "lex", controla o processo de análise léxica.*

```
1  package lex;
2  import java.io.File;
3  import java.io.FileNotFoundException;
4  import java.io.FileReader;
5  import java.io.IOException;
6  import java.util.ArrayList;
7  import lex.automaton.Automaton;
8  import lex.automaton.NumberAutomaton;
9  import lex.automaton.StringAutomaton;
10 import lex.automaton.WordAutomaton;
11 import static utils.ArrayUtils.whiteSpace;
12 import static utils.ArrayUtils.charIsOnArray;
13
14 public class Analyzer {
```

```
15
16      private ArrayList<Automaton> automatas = new ArrayList<
            Automaton>();
17      private FileReader fileReader;
18      private int line;
19      private int now;
20
21      /**
22       * Cria uma instancia do analizador lexico
23       */
24      public Analyzer(File file) throws FileNotFoundException,
            IOException {
25          initAutomatas();
26          this.fileReader = new FileReader(file);
27          line = 1;
28          now = 0;
29          readNextChar();
30      }
31
32      /**
33       * Retorna o proximo token do arquivo.
34       * @return
35       */
36      public Token getNextToken() throws IOException {
37          Token token = null;
38          while (charIsOnArray((char) now, whiteSpace)) {
39              // killing the white space stuff.
40              readNextChar();
41          }
42          boolean result;
43          boolean gotToken = false;
44          while (!gotToken) {
45              if (automatas.isEmpty()) {
46                  if (now == -1) {
47                      Token eof = new Token(TokenType.EOF, -1);
48                      eof.setLine(line);
49                      return eof;
50                  }
51                  initAutomatas();
52              }
53              ArrayList<Automaton> newAutomataList = new ArrayList<
                    Automaton>();
54              for (Automaton automaton : automatas) {
55  //                System.out.println( "Automata " + automatas.
        indexOf(automaton));
56                  result = automaton.process((char) now);
57  //                System.out.println(" retornou " + result + "
        com " + (char) now);
58                  if (!result) {
```

```
59                      //checks if it is on final state.
60                      if (automaton.isOnFinalState()) {
61  //                          System.out.println( "Automata " +
    automatas.indexOf(automaton) +"em estado final");
62                          token = automaton.getToken();
63                          gotToken = true;
64                          newAutomataList.add(automaton);
65                          break;
66                      }
67                  } else {
68                      newAutomataList.add(automaton);
69                  }
70              }
71              automatas = newAutomataList;
72              if (!gotToken) {
73                  readNextChar();
74              }
75          }
76          if (now == '\n') {
77              token.setLine(line - 1);
78          } else {
79              token.setLine(line);
80          }
81          if (token.getType() == TokenType.IDENTIFIER) {
82  //              System.out.println(((WordAutomaton) automatas.get
    (0)).getName());
83          }
84          // all to start again
85          initAutomatas();
86          return token;
87
88      }
89
90      /**
91       * Inicializa / reseta os automatos do analisador lÃ©xico
92       */
93      private void initAutomatas() {
94          automatas = new ArrayList<Automaton>();
95          automatas.add(new WordAutomaton());
96          automatas.add(new NumberAutomaton());
97          automatas.add(new StringAutomaton());
98      }
99
100     /**
101      * Le o proximo caractere do arquivo.
102      * @throws IOException
103      */
104     private void readNextChar() throws IOException {
105         now = fileReader.read();
```

```
106             if (now == '\n') {
107                 line++;
108             }
109         }
110 }
```

4 Analise sintática

Como visto anteriormente, a gramática em notação de Wirth foi reduzida de forma a ficarmos com 3 autômatos: um de "comandos", um de "funções" e o terceiro de "expressões".

Esta gramática foi transformada em autômatos finitos determinísticos mínimos com o uso da ferramenta desenvolvida por Fabio Sendoda Yamate e Hugo Baraúna[1] que aplica o algoritmo de conversão e gera os APEs automaticamente.

As tabelas de transição dos autômatos encontram-se a seguir[2]:

Código 5: *Transições do Wirth de "comandos".*

```
1  initial: 0
2  final: 17, 18, 33, 40
3  (0, "Nome") -> 1
4  (0, "(") -> 2
5  (0, "do") -> 3
6  (0, "while") -> 4
7  (0, "repeat") -> 5
8  (0, "if") -> 6
9  (0, "for") -> 7
10 (0, "function") -> 8
11 (0, "local") -> 9
12 (1, "(") -> 10
13 (1, "[") -> 11
14 (1, ".") -> 12
15 (1, ":") -> 13
16 (1, ",") -> 14
17 (1, "{") -> 15
18 (1, "=") -> 16
19 (1, "Cadeia") -> 17
20 (2, exp) -> 91
21 (3, comando) -> 30
22 (3, "return") -> 31
23 (3, "break") -> 32
24 (3, "end") -> 33
25 (4, exp) -> 70
26 (5, comando) -> 83
27 (5, "return") -> 84
28 (5, "break") -> 85
29 (5, "until") -> 86
```

[1] Encontra-se em http://radiant-fire-72.heroku.com/

[2] Os APEs não são apresentados pois ficaram muito confusos de visualizar.

```
(6, exp) -> 72
(7, "Nome") -> 47
(8, "Nome") -> 41
(9, "Nome") -> 18
(9, "function") -> 19
(10, exp) -> 45
(10, ")") -> 17
(11, exp) -> 93
(12, "Nome") -> 1
(13, "Nome") -> 42
(14, "Nome") -> 54
(14, "(") -> 55
(15, "Nome") -> 24
(15, exp) -> 25
(15, "[") -> 26
(15, "}") -> 17
(16, exp) -> 40
(17, "(") -> 10
(17, "[") -> 20
(17, ".") -> 21
(17, ":") -> 13
(17, "{") -> 15
(17, "Cadeia") -> 17
(18, ",") -> 39
(18, "=") -> 16
(19, "Nome") -> 22
(20, exp) -> 44
(21, "Nome") -> 43
(22, "(") -> 23
(23, "Nome") -> 27
(23, ")") -> 3
(23, "...") -> 28
(24, "=") -> 34
(25, ",") -> 15
(25, ";") -> 15
(25, "}") -> 17
(26, exp) -> 29
(27, ")") -> 3
(27, ",") -> 38
(28, ")") -> 3
(29, "]") -> 24
(30, ";") -> 3
(30, comando) -> 30
(30, "return") -> 31
(30, "break") -> 32
(30, "end") -> 33
(31, exp) -> 36
(31, ";") -> 35
(31, "end") -> 33
```

```
(32, ";") -> 35
(32, "end") -> 33
(34, exp) -> 25
(35, "end") -> 33
(36, ",") -> 37
(36, ";") -> 35
(36, "end") -> 33
(37, exp) -> 36
(38, "Nome") -> 27
(38, "...") -> 28
(39, "Nome") -> 18
(40, ",") -> 16
(41, "(") -> 23
(41, ".") -> 8
(41, ":") -> 19
(42, "(") -> 10
(42, "{") -> 15
(42, "Cadeia") -> 17
(43, "[") -> 20
(43, ".") -> 21
(43, ",") -> 14
(43, "=") -> 16
(44, "]") -> 43
(45, ")") -> 17
(45, ",") -> 46
(46, exp) -> 45
(47, ",") -> 48
(47, "=") -> 49
(47, "in") -> 50
(48, "Nome") -> 71
(49, exp) -> 52
(50, exp) -> 51
(51, ",") -> 50
(51, "do") -> 3
(52, ",") -> 53
(53, exp) -> 56
(54, "(") -> 59
(54, "[") -> 60
(54, ".") -> 61
(54, ":") -> 62
(54, ",") -> 14
(54, "{") -> 63
(54, "=") -> 16
(54, "Cadeia") -> 64
(55, exp) -> 57
(56, ",") -> 4
(56, "do") -> 3
(57, ")") -> 58
(58, "(") -> 59
```

```
(58, "[") -> 60
(58, ".") -> 61
(58, ":") -> 62
(58, "{") -> 63
(58, "Cadeia") -> 64
(59, exp) -> 81
(59, ")") -> 64
(60, exp) -> 90
(61, "Nome") -> 54
(62, "Nome") -> 77
(63, "Nome") -> 65
(63, exp) -> 66
(63, "[") -> 67
(63, "}") -> 64
(64, "(") -> 59
(64, "[") -> 20
(64, ".") -> 21
(64, ":") -> 62
(64, "{") -> 63
(64, "Cadeia") -> 64
(65, "=") -> 69
(66, ",") -> 63
(66, ";") -> 63
(66, "}") -> 64
(67, exp) -> 68
(68, "]") -> 65
(69, exp) -> 66
(70, "do") -> 3
(71, ",") -> 48
(71, "in") -> 50
(72, "then") -> 73
(73, comando) -> 74
(73, "return") -> 75
(73, "break") -> 76
(73, "end") -> 33
(73, "elseif") -> 6
(73, "else") -> 3
(74, ";") -> 73
(74, comando) -> 74
(74, "return") -> 75
(74, "break") -> 76
(74, "end") -> 33
(74, "elseif") -> 6
(74, "else") -> 3
(75, exp) -> 79
(75, ";") -> 78
(75, "end") -> 33
(75, "elseif") -> 6
(75, "else") -> 3
```

```
(76, ";") -> 78
(76, "end") -> 33
(76, "elseif") -> 6
(76, "else") -> 3
(77, "(") -> 59
(77, "{") -> 63
(77, "Cadeia") -> 64
(78, "end") -> 33
(78, "elseif") -> 6
(78, "else") -> 3
(79, ",") -> 80
(79, ";") -> 78
(79, "end") -> 33
(79, "elseif") -> 6
(79, "else") -> 3
(80, exp) -> 79
(81, ")") -> 64
(81, ",") -> 82
(82, exp) -> 81
(83, ";") -> 5
(83, comando) -> 83
(83, "return") -> 84
(83, "break") -> 85
(83, "until") -> 86
(84, exp) -> 88
(84, ";") -> 87
(84, "until") -> 86
(85, ";") -> 87
(85, "until") -> 86
(86, exp) -> 33
(87, "until") -> 86
(88, ",") -> 89
(88, ";") -> 87
(88, "until") -> 86
(89, exp) -> 88
(90, "]") -> 54
(91, ")") -> 92
(92, "(") -> 10
(92, "[") -> 11
(92, ".") -> 12
(92, ":") -> 13
(92, "{") -> 15
(92, "Cadeia") -> 17
(93, "]") -> 1
```

Código 6: *Transições do Wirth de "funções".*

```
initial: 0
final: 10
```

```
(0, "function") -> 1
(1, "(") -> 2
(2, "Nome") -> 3
(2, "...") -> 4
(2, ")") -> 5
(3, ",") -> 6
(3, ")") -> 5
(4, ")") -> 5
(5, comando) -> 7
(5, "return") -> 8
(5, "break") -> 9
(5, "end") -> 10
(6, "Nome") -> 3
(6, "...") -> 4
(7, comando) -> 7
(7, ";") -> 5
(7, "return") -> 8
(7, "break") -> 9
(7, "end") -> 10
(8, ";") -> 11
(8, exp) -> 12
(8, "end") -> 10
(9, ";") -> 11
(9, "end") -> 10
(11, "end") -> 10
(12, ",") -> 13
(12, ";") -> 11
(12, "end") -> 10
(13, exp) -> 12
```

Código 7: *Transições do Wirth de "expressões".*

```
initial: 0
final: 1, 2, 12, 26
(0, "nil") -> 1
(0, "false") -> 1
(0, "true") -> 1
(0, "Numero") -> 1
(0, "Cadeia") -> 1
(0, "...") -> 1
(0, funcao) -> 1
(0, "Nome") -> 2
(0, "(") -> 3
(0, "{") -> 4
(0, "-") -> 5
(0, "not") -> 5
(0, "#") -> 5
(1, "-") -> 0
(1, "+") -> 0
```

```
(1, "*") -> 0
(1, "/") -> 0
(1, "^") -> 0
(1, "%") -> 0
(1, "..") -> 0
(1, "<") -> 0
(1, "<=") -> 0
(1, ">") -> 0
(1, ">=") -> 0
(1, "==") -> 0
(1, "~=") -> 0
(1, "and") -> 0
(1, "or") -> 0
(2, "Cadeia") -> 12
(2, "(") -> 13
(2, "[") -> 14
(2, ".") -> 15
(2, ":") -> 16
(2, "{") -> 17
(2, "-") -> 0
(2, "+") -> 0
(2, "*") -> 0
(2, "/") -> 0
(2, "^") -> 0
(2, "%") -> 0
(2, "..") -> 0
(2, "<") -> 0
(2, "<=") -> 0
(2, ">") -> 0
(2, ">=") -> 0
(2, "==") -> 0
(2, "~=") -> 0
(2, "and") -> 0
(2, "or") -> 0
(3, exp) -> 11
(4, "Nome") -> 6
(4, exp) -> 7
(4, "[") -> 8
(4, "}") -> 1
(5, exp) -> 1
(6, "=") -> 10
(7, ",") -> 4
(7, ";") -> 4
(7, "}") -> 1
(8, exp) -> 9
(9, "]") -> 6
(10, exp) -> 7
(11, ")") -> 2
(12, "Cadeia") -> 12
```

```
(12, "(") -> 13
(12, "[") -> 21
(12, ".") -> 22
(12, ":") -> 16
(12, "{") -> 17
(12, "-") -> 0
(12, "+") -> 0
(12, "*") -> 0
(12, "/") -> 0
(12, "^") -> 0
(12, "%") -> 0
(12, "..") -> 0
(12, "<") -> 0
(12, "<=") -> 0
(12, ">") -> 0
(12, ">=") -> 0
(12, "==") -> 0
(12, "~=") -> 0
(12, "and") -> 0
(12, "or") -> 0
(13, exp) -> 28
(13, ")") -> 12
(14, exp) -> 30
(15, "Nome") -> 2
(16, "Nome") -> 25
(17, "Nome") -> 18
(17, exp) -> 19
(17, "[") -> 20
(17, "}") -> 12
(18, "=") -> 24
(19, ",") -> 17
(19, ";") -> 17
(19, "}") -> 12
(20, exp) -> 23
(21, exp) -> 27
(22, "Nome") -> 26
(23, "]") -> 18
(24, exp) -> 19
(25, "Cadeia") -> 12
(25, "(") -> 13
(25, "{") -> 17
(26, "[") -> 21
(26, ".") -> 22
(26, "-") -> 0
(26, "+") -> 0
(26, "*") -> 0
(26, "/") -> 0
(26, "^") -> 0
(26, "%") -> 0
```

```
(26, "..") -> 0
(26, "<") -> 0
(26, "<=") -> 0
(26, ">") -> 0
(26, ">=") -> 0
(26, "==") -> 0
(26, "~=") -> 0
(26, "and") -> 0
(26, "or") -> 0
(27, "]") -> 26
(28, ")") -> 12
(28, ",") -> 29
(29, exp) -> 28
(30, "]") -> 2
```

Para a construção do analisador sintático, foram usados os códigos apresentados na figura 2 e sua classe principal – que chama as funções dos outros códigos – encontra-se no código 8.

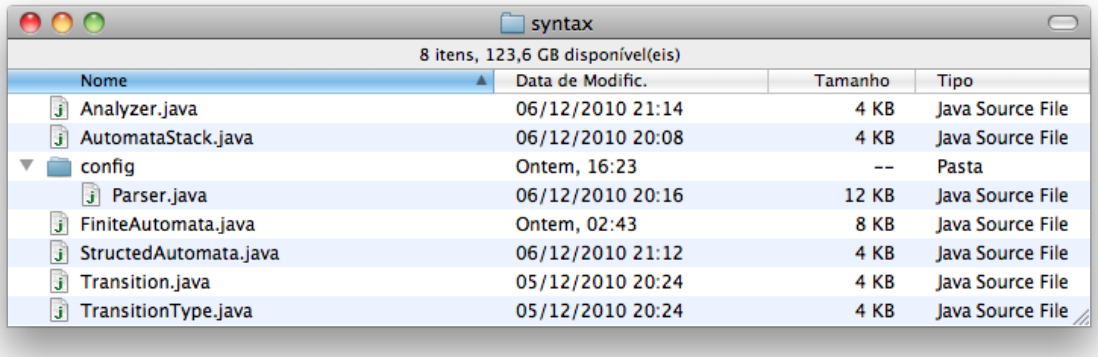

Figura 2: *Códigos utilizados para construir o analisador sintático.*

Código 8: *Classe "Analyzer.java" do pacote "syntax", controla o processo de análise sintática.*

```
1  package syntax;
2  import java.io.File;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  import lex.Token;
6  import lex.TokenType;
7
8  /**
9   * This class takes cares on analyzing
10  * a code file syntax.
11  */
12 public class Analyzer {
13
```

```
14      private lex.Analyzer lexical;
15      private StructedAutomata structedAutomata;
16      public static final String[] files = { "comando" ,  "exp"  ,
           "funcao"};
17
18      public Analyzer(File file) throws FileNotFoundException,
           IOException {
19          //makes the parser listing
20
21          lexical = new lex.Analyzer(file);
22          structedAutomata = new  StructedAutomata(3);
23          structedAutomata.init(getFilePaths());
24
25      }
26
27
28      public void reset(){
29          structedAutomata.setAutomataAndState(0, 0);
30          AutomataStack stack = AutomataStack.getInstance();
31          while(!stack.isEmpty()){
32              stack.pop();
33          }
34      }
35
36      /**
37       * returns true if the program is valid.
38       * @return
39       */
40      public boolean analyze() throws IOException{
41          Token token ;
42  //          SymbolTable table = Semantic.latestTable;
43          token = lexical.getNextToken();
44          while(token.getType() != TokenType.EOF){
45              boolean result;
46              System.out.println("Syntax analyzer got this token "
                   + token);
47              result = structedAutomata.nextStep(token);
48              if(!result){
49                  if(structedAutomata.accepted()){
50                      System.out.println("Valid Program");
51                      return true;
52                  }
53                  System.out.println("Invalid Program");
54                  return false;
55              }
56              System.out.println("getting next token");
57              token = lexical.getNextToken();
58          }
59          return false;
```

```
60      }
61
62
63      /**
64       * Gets the paths to the files, using the files vector, the
             files vector should have the names of the files
65       * @return array containing the paths
66       */
67      private String[] getFilePaths(){
68          String[] result = new String[files.length];
69          int index = 0;
70          while(index < files.length){
71              result[index] = Analyzer.class.getResource("/syntax/
                    config/" + files[index]).getFile();
72              index ++;
73          }
74          return result;
75      }
76  }
```

5 Análise semântica

Para a construção do analisador semântico, foram usados os códigos apresentados na figura 3.

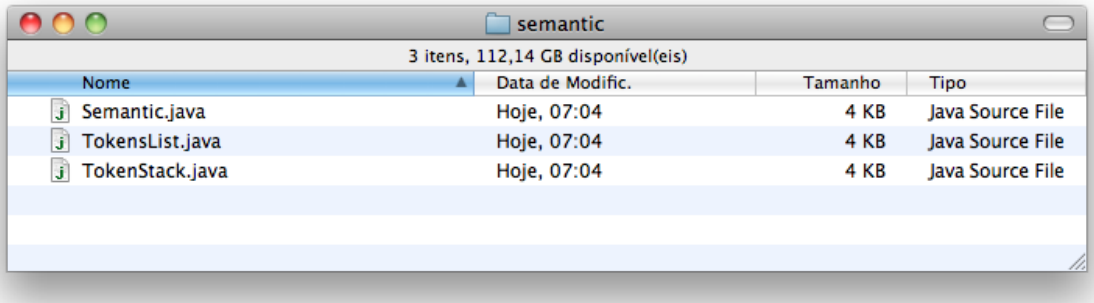

Figura 3: *Códigos utilizados para construir o analisador semântico.*

Cada uma dessas classes apresentam-se nos códigos 9, 10 e 11.

Código 9: *Classe "Semantic.java" do pacote "semantic", controla o processo de análise semântica.*

```
1  package semantic;
2
3  import codeGeneration.Coder;
4  import java.util.ArrayList;
5  import lex.Token;
```

```
6  import lex.TokenType;
7  import utils.CompilerException;
8  import static lex.ReservedWords.getByIndex;
9  import utils.SymbolTable;
10
11 /**
12  * This class will gives the support to all code that is related
       with
13  * the semantics of the compiler
14  */
15 public class Semantic {
16
17     protected static ArrayList<Token> tokenList = new ArrayList<
          Token>();
18     private Token latestToken;
19     public static SymbolTable rootTable = new SymbolTable();
20     public static SymbolTable latestTable;
21     private static Coder coder;
22     private static String operator = "";
23
24
25     public static void initCoder() {
26         coder = Coder.getInstance();
27     }
28
29     public Semantic() {
30         System.out.println("Semantic Constructor called");
31         if (latestTable == null) {
32             latestTable = rootTable;
33         }
34     }
35
36     /**
37      * Adds a token to the list, this list will be used by the
           actions to do stuff.
38      * @param token
39      */
40     public void addToken(Token token) {
41         tokenList.add(token);
42         latestToken = token;
43     }
44
45     public void runAction(String name) throws CompilerException {
46         if (name.equals("num")) {
47             int valor = latestToken.getValue();
48             coder.putOnBuffer("iconst_" + valor + "\n", false);
49             coder.putOnBuffer(operator + "\n", false);
50             operator = "";
51         } else if(name.equals("new_op")){
```

```
52              int value = latestToken.getValue();
53              String word = getByIndex(value);
54              if(word.equals("+")){
55                  operator = "iadd";
56              }else if(word.equals("-")){
57                  operator = "isub";
58              }else if(word.equals("/")){
59                  operator = "idiv";
60              }else if(word.equals("*")){
61                  operator = "imul";
62              }
63          }
64          else {
65
66          }
67
68      }
69
70      public void runFinalAction(int machineNumber) {
71          //When the machine will return to another one, it can
               execute a final action;
72          System.out.println("Running the final action to the
               machine " + machineNumber);
73
74          switch (machineNumber) {
75              case 0:
76                  //PROGRAM;
77
78              case 1:
79                  //Code block
80                  break;
81              case 2:
82
83                  // <editor-fold defaultstate="collapsed" desc="
                       Expression ">
84
85                  break;
86
87          }
88      }
89
90
91  }
```

Código 10: *Classe "TokenList.java" do pacote "semantic".*

```
1  package semantic;
2
3  import lex.Token;
```

```
4
5  public class TokensList {
6
7      private Element first;
8      private int count;
9
10     public TokensList() {
11         count = 0;
12     }
13
14     public int getCount() {
15         return count;
16     }
17
18     /**
19      * Adds a token to the list
20      * @param token the token to be added.
21      */
22     public void addToken(Token token) {
23         if (count == 0) {
24             first = new Element(token);
25         } else {
26             int index = 1;
27             Element last = first;
28             while (index < count) {
29                 last = last.getNext();
30                 index++;
31             }
32             last.setNext(new Element(token));
33         }
34         count++;
35     }
36
37
38     @Override
39     public String toString(){
40         String result = "";
41         if(count == 0){
42             result = "Token list is empty";
43         }else{
44             int index = 0;
45             result = "Token List:\n";
46             Element element = first;
47             while(index < count){
48                 result += element.token.toString();
49                 result += "\n";
50                 index ++;
51                 element = element.next;
52             }
```

```
53          }
54          return result;
55      }
56      /**
57       * Clears the list.
58       */
59      public void clear() {
60          System.out.println("Tokens list cleared");
61          count = 0;
62          first = null;
63      }
64
65      /**
66       * Gets the list as an array.
67       * @return
68       */
69      public Token[] getArray() {
70          Token[] result = new Token[count];
71          int index = 0;
72          Element element = first;
73          while (index < count) {
74              result[index] = element.getToken();
75              element = element.getNext();
76              index++;
77          }
78          return result;
79      }
80
81      private class Element {
82
83          private Element next;
84          private Token token;
85
86          public Element(Token token) {
87              this.token = token;
88          }
89
90          public Element getNext() {
91              return next;
92          }
93
94          public void setNext(Element next) {
95              this.next = next;
96          }
97
98          public Token getToken() {
99              return token;
100         }
101     }
```

```
102 }
```

Código 11: *Classe "TokenStack.java" do pacote "semantic".*

```
1  package semantic;
2
3  import lex.Token;
4
5  public class TokenStack {
6
7      private Element top;
8
9      public TokenStack() {
10     }
11
12     public boolean isEmpty(){
13         return top == null;
14     }
15
16     /**
17      * Gets the top of the stack.
18      * @return null if the stack is empty.
19      */
20     public Token pop(){
21         if(top == null){
22             return null;
23         }else{
24             Element result = top;
25             top = result.getPrevious();
26             return result.getToken();
27         }
28     }
29
30     /**
31      * Like pop, but doesn't remove the top from stack
32      * @return
33      */
34     public Token peek(){
35         if(top == null){
36             return null;
37         }
38         return top.getToken();
39     }
40
41
42     @Override
43     public String toString(){
44         System.out.println("Called to string from the Token stack
               ");
```

```
45          String result = "";
46          Element element;
47          element = top;
48          while(element != null){
49              result = element.getToken() + " \n" + result;
50              element = element.getPrevious();
51          }
52          return result;
53      }
54
55      /**
56       * Adds a token to the stack.
57       * @param token
58       */
59      public void push(Token token){
60          Element element = new Element(token);
61          element.setPrevious(top);
62          top = element;
63      }
64
65      private class Element {
66
67          private Element previous;
68          private Token token;
69
70          public Element(Token token) {
71              this.token = token;
72          }
73
74          public Element getPrevious() {
75              return previous;
76          }
77
78          public void setPrevious(Element previous) {
79              this.previous = previous;
80          }
81
82          public Token getToken() {
83              return token;
84          }
85
86
87      }
88  }
```