



Escola Politécnica
da Universidade de
São Paulo

Segunda Prova

PCS-2056 : Linguagens e Compiladores

Autores:

Eduardo Russo
Helio Kazuo Nagamachi

Professor:

Ricardo Rocha

9 de dezembro de 2010

Sumário

1	Objetivo	2
2	Gramática	2
3	Análise léxica	4
4	Análise sintática	7
5	Definição do ambiente de execução	18
6	Estrutura de dados e algoritmos do compilador	18
7	Semântica dinâmica	18
8	Integração das rotinas semânticas	18

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 = "," | ";".
```

```

32 opbin = "+" | "-" | "*" | "/" | "^" | "%" | ".." |
33      "<" | "<=" | ">" | ">=" | "==" | "~=" |
34      "and" | "or" .
35 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.

```

1 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)}{, ("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)].
2
3 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:

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

Considerando essas palavras reservadas, mas as possíveis sequências de palavras, foram criados autômatos para reconhecer e retornar os *tokens*.

A construção do léxico utilizou os códigos apresentados na figura 1.

A classe principal do analisador léxico, que chama as funções dos outros códigos encontra-se no código 3.

Código 3: Classe “Analyzer.java” que 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;

```

9 itens, 123,6 GB disponível(eis)			
Nome	Data de Modific.	Tamanho	Tipo
Analyzer.java	06/12/2010 20:05	4 KB	Java Source File
automaton	Ontem, 16:23	--	Pasta
Automaton.java	05/12/2010 11:23	4 KB	Java Source File
NumberAutomaton.java	05/12/2010 11:40	4 KB	Java Source File
StringAutomaton.java	06/12/2010 08:05	4 KB	Java Source File
WordAutomaton.java	06/12/2010 20:05	12 KB	Java Source File
ReservedWords.java	05/12/2010 12:57	4 KB	Java Source File
Token.java	06/12/2010 08:06	4 KB	Java Source File
TokenType.java	05/12/2010 20:37	4 KB	Java Source File

Figura 1: Códigos utilizados para construir o analisador léxico.

```

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<
17         Automaton>();
18     private FileReader fileReader;
19     private int line;
20     private int now;
21
22     /**
23      * Cria uma instancia do analisador lexico
24      */
25     public Analyzer(File file) throws FileNotFoundException,
26         IOException {
27         initAutomatas();
28         this.fileReader = new FileReader(file);
29         line = 1;
30         now = 0;
31         readNextChar();
32     }
33
34     /**
35      * Retorna o proximo token do arquivo.
36      * @return
37      */
38     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 automatos, 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 Baraúna¹ que aplica o algoritmo de conversão e gera os APEs automaticamente.

A tabela de transição dos autômatos encontra-se a seguir²:

Código 4: Transições do Wirth de “comandos”.

```

1 initial: 0
2 final: 17, 18, 33, 40
3 (0, "Nome") -> 1
4 (0, "(") -> 2

```

¹Encontra-se em <http://radiant-fire-72.herokuapp.com/>

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


```

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
30 (6, exp) -> 72
31 (7, "Nome") -> 47
32 (8, "Nome") -> 41
33 (9, "Nome") -> 18
34 (9, "function") -> 19
35 (10, exp) -> 45
36 (10, ")") -> 17
37 (11, exp) -> 93
38 (12, "Nome") -> 1
39 (13, "Nome") -> 42
40 (14, "Nome") -> 54
41 (14, "(") -> 55
42 (15, "Nome") -> 24
43 (15, exp) -> 25
44 (15, "[" ) -> 26
45 (15, "}") -> 17
46 (16, exp) -> 40
47 (17, "(") -> 10
48 (17, "[" ) -> 20
49 (17, ".") -> 21
50 (17, ":") -> 13
51 (17, "{") -> 15
52 (17, "Cadeia") -> 17
53 (18, ",") -> 39

```

```

54 (18, "=") -> 16
55 (19, "Nome") -> 22
56 (20, exp) -> 44
57 (21, "Nome") -> 43
58 (22, "(") -> 23
59 (23, "Nome") -> 27
60 (23, ")") -> 3
61 (23, "...") -> 28
62 (24, "=") -> 34
63 (25, ",") -> 15
64 (25, ";") -> 15
65 (25, "}") -> 17
66 (26, exp) -> 29
67 (27, ")") -> 3
68 (27, ",") -> 38
69 (28, ")") -> 3
70 (29, "]"") -> 24
71 (30, ";") -> 3
72 (30, comando) -> 30
73 (30, "return") -> 31
74 (30, "break") -> 32
75 (30, "end") -> 33
76 (31, exp) -> 36
77 (31, ";") -> 35
78 (31, "end") -> 33
79 (32, ";") -> 35
80 (32, "end") -> 33
81 (34, exp) -> 25
82 (35, "end") -> 33
83 (36, ",") -> 37
84 (36, ";") -> 35
85 (36, "end") -> 33
86 (37, exp) -> 36
87 (38, "Nome") -> 27
88 (38, "...") -> 28
89 (39, "Nome") -> 18
90 (40, ",") -> 16
91 (41, "(") -> 23
92 (41, ".") -> 8
93 (41, ":") -> 19
94 (42, "(") -> 10
95 (42, "{") -> 15
96 (42, "Cadeia") -> 17
97 (43, "[") -> 20
98 (43, ".") -> 21
99 (43, ",") -> 14
100 (43, "=") -> 16
101 (44, "]"") -> 43
102 (45, ")") -> 17

```

```

103 (45, ",") -> 46
104 (46, exp) -> 45
105 (47, ",") -> 48
106 (47, "=") -> 49
107 (47, "in") -> 50
108 (48, "Nome") -> 71
109 (49, exp) -> 52
110 (50, exp) -> 51
111 (51, ",") -> 50
112 (51, "do") -> 3
113 (52, ",") -> 53
114 (53, exp) -> 56
115 (54, "(") -> 59
116 (54, "[" ) -> 60
117 (54, ".") -> 61
118 (54, ":") -> 62
119 (54, ",") -> 14
120 (54, "{") -> 63
121 (54, "=") -> 16
122 (54, "Cadeia") -> 64
123 (55, exp) -> 57
124 (56, ",") -> 4
125 (56, "do") -> 3
126 (57, ")") -> 58
127 (58, "(") -> 59
128 (58, "[" ) -> 60
129 (58, ".") -> 61
130 (58, ":") -> 62
131 (58, "{") -> 63
132 (58, "Cadeia") -> 64
133 (59, exp) -> 81
134 (59, ")") -> 64
135 (60, exp) -> 90
136 (61, "Nome") -> 54
137 (62, "Nome") -> 77
138 (63, "Nome") -> 65
139 (63, exp) -> 66
140 (63, "[" ) -> 67
141 (63, "}") -> 64
142 (64, "(") -> 59
143 (64, "[" ) -> 20
144 (64, ".") -> 21
145 (64, ":") -> 62
146 (64, "{") -> 63
147 (64, "Cadeia") -> 64
148 (65, "=") -> 69
149 (66, ",") -> 63
150 (66, ";") -> 63
151 (66, "}") -> 64

```

```

152 (67, exp) -> 68
153 (68, "]" ) -> 65
154 (69, exp) -> 66
155 (70, "do") -> 3
156 (71, ",") -> 48
157 (71, "in") -> 50
158 (72, "then") -> 73
159 (73, comando) -> 74
160 (73, "return") -> 75
161 (73, "break") -> 76
162 (73, "end") -> 33
163 (73, "elseif") -> 6
164 (73, "else") -> 3
165 (74, ";") -> 73
166 (74, comando) -> 74
167 (74, "return") -> 75
168 (74, "break") -> 76
169 (74, "end") -> 33
170 (74, "elseif") -> 6
171 (74, "else") -> 3
172 (75, exp) -> 79
173 (75, ";") -> 78
174 (75, "end") -> 33
175 (75, "elseif") -> 6
176 (75, "else") -> 3
177 (76, ";") -> 78
178 (76, "end") -> 33
179 (76, "elseif") -> 6
180 (76, "else") -> 3
181 (77, "(" ) -> 59
182 (77, "{" ) -> 63
183 (77, "Cadeia") -> 64
184 (78, "end") -> 33
185 (78, "elseif") -> 6
186 (78, "else") -> 3
187 (79, ",") -> 80
188 (79, ";") -> 78
189 (79, "end") -> 33
190 (79, "elseif") -> 6
191 (79, "else") -> 3
192 (80, exp) -> 79
193 (81, ")" ) -> 64
194 (81, ",") -> 82
195 (82, exp) -> 81
196 (83, ";") -> 5
197 (83, comando) -> 83
198 (83, "return") -> 84
199 (83, "break") -> 85
200 (83, "until") -> 86

```

```

201 (84, exp) -> 88
202 (84, ";") -> 87
203 (84, "until") -> 86
204 (85, ";") -> 87
205 (85, "until") -> 86
206 (86, exp) -> 33
207 (87, "until") -> 86
208 (88, ",") -> 89
209 (88, ";") -> 87
210 (88, "until") -> 86
211 (89, exp) -> 88
212 (90, "]"") -> 54
213 (91, ")"") -> 92
214 (92, "("") -> 10
215 (92, "["") -> 11
216 (92, "."") -> 12
217 (92, ":"") -> 13
218 (92, "{"") -> 15
219 (92, "Cadeia") -> 17
220 (93, "]"") -> 1

```

Código 5: Transições do Wirth de “funções”.

```

1  initial: 0
2  final: 10
3  (0, "function") -> 1
4  (1, "(") -> 2
5  (2, "Nome") -> 3
6  (2, "...") -> 4
7  (2, ")"") -> 5
8  (3, ",") -> 6
9  (3, ")"") -> 5
10 (4, ")"") -> 5
11 (5, comando) -> 7
12 (5, "return") -> 8
13 (5, "break") -> 9
14 (5, "end") -> 10
15 (6, "Nome") -> 3
16 (6, "...") -> 4
17 (7, comando) -> 7
18 (7, ";") -> 5
19 (7, "return") -> 8
20 (7, "break") -> 9
21 (7, "end") -> 10
22 (8, ";") -> 11
23 (8, exp) -> 12
24 (8, "end") -> 10
25 (9, ";") -> 11
26 (9, "end") -> 10

```

```

27 (11, "end") -> 10
28 (12, ",") -> 13
29 (12, ";") -> 11
30 (12, "end") -> 10
31 (13, exp) -> 12

```

Código 6: Transições do Wirth de “expressões”.

```

1  initial: 0
2  final: 1, 2, 12, 26
3  (0, "nil") -> 1
4  (0, "false") -> 1
5  (0, "true") -> 1
6  (0, "Numero") -> 1
7  (0, "Cadeia") -> 1
8  (0, "...") -> 1
9  (0, funcao) -> 1
10 (0, "Nome") -> 2
11 (0, "(") -> 3
12 (0, "{") -> 4
13 (0, "-") -> 5
14 (0, "not") -> 5
15 (0, "#") -> 5
16 (1, "-") -> 0
17 (1, "+") -> 0
18 (1, "*") -> 0
19 (1, "/") -> 0
20 (1, "^") -> 0
21 (1, "%") -> 0
22 (1, "..") -> 0
23 (1, "<") -> 0
24 (1, "<=") -> 0
25 (1, ">") -> 0
26 (1, ">=") -> 0
27 (1, "==" ) -> 0
28 (1, "~=") -> 0
29 (1, "and") -> 0
30 (1, "or") -> 0
31 (2, "Cadeia") -> 12
32 (2, "(") -> 13
33 (2, "[") -> 14
34 (2, ".") -> 15
35 (2, ":") -> 16
36 (2, "{") -> 17
37 (2, "-") -> 0
38 (2, "+") -> 0
39 (2, "*") -> 0
40 (2, "/") -> 0
41 (2, "^") -> 0

```

```

42 (2, "%") -> 0
43 (2, "..") -> 0
44 (2, "<") -> 0
45 (2, "<=") -> 0
46 (2, ">") -> 0
47 (2, ">=") -> 0
48 (2, "==" ) -> 0
49 (2, "~=") -> 0
50 (2, "and") -> 0
51 (2, "or") -> 0
52 (3, exp) -> 11
53 (4, "Nome") -> 6
54 (4, exp) -> 7
55 (4, "[") -> 8
56 (4, "}") -> 1
57 (5, exp) -> 1
58 (6, "=") -> 10
59 (7, ",") -> 4
60 (7, ";") -> 4
61 (7, "}") -> 1
62 (8, exp) -> 9
63 (9, "]" ) -> 6
64 (10, exp) -> 7
65 (11, ")") -> 2
66 (12, "Cadeia") -> 12
67 (12, "(") -> 13
68 (12, "[") -> 21
69 (12, ".") -> 22
70 (12, ":") -> 16
71 (12, "{") -> 17
72 (12, "-" ) -> 0
73 (12, "+" ) -> 0
74 (12, "*" ) -> 0
75 (12, "/" ) -> 0
76 (12, "^" ) -> 0
77 (12, "%" ) -> 0
78 (12, "..") -> 0
79 (12, "<") -> 0
80 (12, "<=") -> 0
81 (12, ">") -> 0
82 (12, ">=") -> 0
83 (12, "==" ) -> 0
84 (12, "~=") -> 0
85 (12, "and") -> 0
86 (12, "or") -> 0
87 (13, exp) -> 28
88 (13, ")") -> 12
89 (14, exp) -> 30
90 (15, "Nome") -> 2

```

```

91 (16, "Nome") -> 25
92 (17, "Nome") -> 18
93 (17, exp) -> 19
94 (17, "[" -> 20
95 (17, "}") -> 12
96 (18, "=") -> 24
97 (19, ",") -> 17
98 (19, ";") -> 17
99 (19, "}") -> 12
100 (20, exp) -> 23
101 (21, exp) -> 27
102 (22, "Nome") -> 26
103 (23, "]" -> 18
104 (24, exp) -> 19
105 (25, "Cadeia") -> 12
106 (25, "(" -> 13
107 (25, "{") -> 17
108 (26, "[" -> 21
109 (26, "." -> 22
110 (26, "-" -> 0
111 (26, "+" -> 0
112 (26, "*" -> 0
113 (26, "/" -> 0
114 (26, "^" -> 0
115 (26, "%" -> 0
116 (26, ".." -> 0
117 (26, "<" -> 0
118 (26, "<=" -> 0
119 (26, ">" -> 0
120 (26, ">=" -> 0
121 (26, "==" -> 0
122 (26, "~=" -> 0
123 (26, "and") -> 0
124 (26, "or") -> 0
125 (27, "]" -> 26
126 (28, ")" -> 12
127 (28, "," -> 29
128 (29, exp) -> 28
129 (30, "]" -> 2

```

Para a construção do analisador sintático, foram usados os códigos apresentados na figura 2.

A classe principal do analisador sintático, que chama as funções dos outros códigos encontra-se no código 7.

Código 7: Classe “Analyzer.java” que controla o processo de análise sintática.

```

1 package syntax;
2 import java.io.File;
3 import java.io.FileNotFoundException;

```


8 itens, 123,6 GB disponível(eis)			
Nome	Data de Modific.	Tamanho	Tipo
Analyzer.java	06/12/2010 21:14	4 KB	Java Source File
AutomataStack.java	06/12/2010 20:08	4 KB	Java Source File
config	Ontem, 16:23	--	Pasta
Parser.java	06/12/2010 20:16	12 KB	Java Source File
FiniteAutomata.java	Ontem, 02:43	8 KB	Java Source File
StructedAutomata.java	06/12/2010 21:12	4 KB	Java Source File
Transition.java	05/12/2010 20:24	4 KB	Java Source File
TransitionType.java	05/12/2010 20:24	4 KB	Java Source File

Figura 2: Códigos utilizados para construir o analisador sintático.

```

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" ,
17         "funcao"};
18
19     public Analyzer(File file) throws FileNotFoundException ,
20         IOException {
21         //makes the parser listing
22
23         lexical = new lex.Analyzer(file);
24         structedAutomata = new StructedAutomata(3);
25         structedAutomata.init(getFilePaths());
26
27     }
28
29     public void reset(){
30         structedAutomata.setAutomataAndState(0, 0);
31         AutomataStack stack = AutomataStack.getInstance();
32         while(!stack.isEmpty()){
33             stack.pop();
34         }
35     }
36 }

```

```

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 "
47                          + token);
47          result = structuredAutomata.nextStep(token);
48          if(!result){
49              if(structuredAutomata.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
65   * files vector should have the names of the files
66   * @return array containing the paths
67   */
68  private String[] getFilePaths(){
69      String[] result = new String[files.length];
70      int index = 0;
71      while(index < files.length){
72          result[index] = Analyzer.class.getResource("/syntax/
73                          config/" + files[index]).getFile();
74          index ++;
75      }
76      return result;
77  }

```

5 Definição do ambiente de execução

O objetivo deste projeto é modelar e construir um compilador que utilize uma linguagem de programação imperativa.

6 Estrutura de dados e algoritmos do compilador

O objetivo deste projeto é modelar e construir um compilador que utilize uma linguagem de programação imperativa.

7 Semântica dinâmica

O objetivo deste projeto é modelar e construir um compilador que utilize uma linguagem de programação imperativa.

8 Integração das rotinas semânticas

O objetivo deste projeto é modelar e construir um compilador que utilize uma linguagem de programação imperativa.