Processamento de Linguagens (3º ano de LEI)

Trabalho Prático

Solitários & Companhia

(Inês Ferreira A97040) (José Ferreira A97642) (Marta Sá A97158)

28 de maio de 2023

Resumo

Este projeto visa recriar parte de um conversor de Pug em HTML definindo analisadores léxico e sintático para esta finalidade. Desta forma, o grupo consolida conceitos e técnicas aprendidas na Unidade Curricular de *Processamento de Linguagens*.

Conteúdo

Capítulo 1

Introdução

Este trabalho prático pretende aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT), desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora, desenvolver um compilador gerando código para um objetivo específico e, por fim, utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

Capítulo 2

Tema

O grupo optou pelo conversor que aceite um subconjunto da linguagem Pug e gera o HTML correspondente. A título de exemplo vejamos o seguinte código Pug e a sua conversão em HTML do lado direito:

Pug	HTML

`html(lang="en")`
`  head`
`  title= pageTitle`
`  script(type='text/javascript')`
`  .`
`  if (foo) bar(1+5)`
`  body`
`  h1 Pug - node template engine`
`  #container.col`
`  if youAreUsingPug`
`  p You are amazing`
`  else`
`  p Get on it!`
`  p.`
`  Pug is a terse and simple templating`
`  language with a strong focus on`
`  performance and powerful features`
	`<html lang="en">`
	`  <head><title></title>`
	`  <script type="text/javascript">`
	`  if (foo) bar(1 + 5)</script>`
	`  </head>`
	`  <body>`
	`  <h1>Pug - node template engine</h1>`
	`  <div class="col" id="container">`
	`  <p>Get on it!</p>`
	`  <p>Pug is a terse and simple templating language with a strong focus on performance and powerful features</p>`
	`  </div>`
	`  </body>`
	`</html>`

Capítulo 3

Estrutura de um Compilador

Antes de avançar para a explicação do que foi feito em termos do trabalho desenvolvido pelo grupo, é importante notar que este projeto tem como objetivo a implementação de uma parte do que compõe um compilador. Portanto, a seguinte imagem ilustra todas as componentes essenciais de um compilador, sendo destacado que neste trabalho são abordadas apenas as partes que correspondem à análise léxica e sintática.

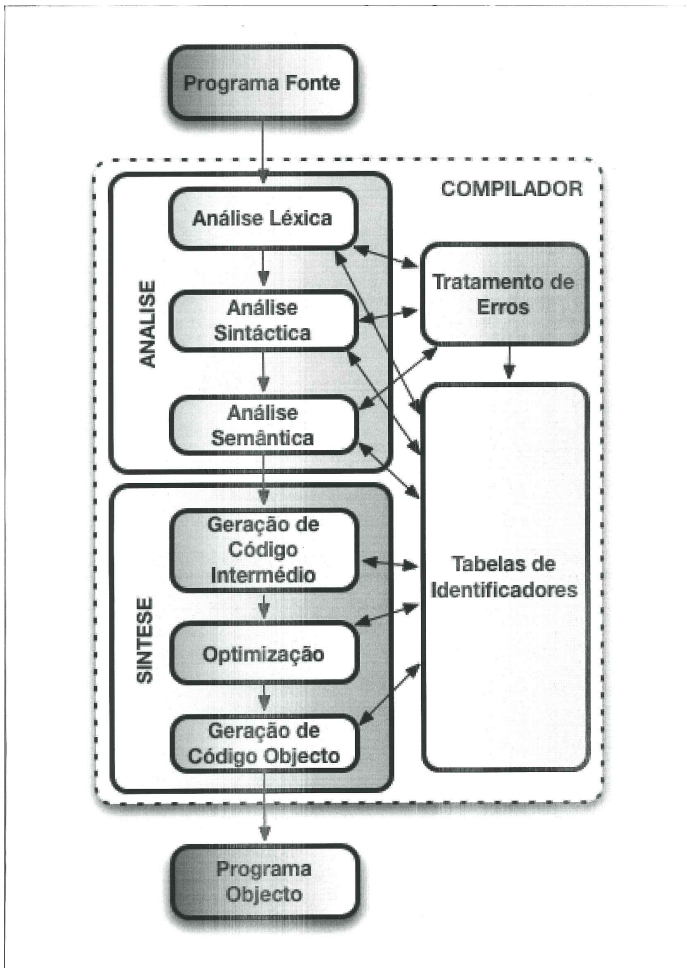

Figura 3.1: Estrutura de um compilador.

Capítulo 4

Analisador Léxico

4.1 Definição

O analisador léxico, que passaremos a denominar como tokenizer, é um dos processos de compilação de um programa. Este converte código-fonte de um programa numa sequência de tokens como palavras-chave, identificadores, símbolos e literais. O objetivo principal é identificar e agrupar os caracteres em tokens para facilitar as etapas subsequentes do compilador.

4.2 Funcionamento

Essencialmente, nesta fase definem-se regras utilizando expressões regulares. Essas expressões regulares descrevem padrões de caracteres que correspondem a cada tipo de token presente na linguagem de programação. O tokenizer percorre o código-fonte caractere por caractere, tentando verificar correspondência com as expressões que as regras definem para asssim gerar uma sequência estruturada de tokens que serão enviados para o analisador sintático.

4.3 Conceção

Para iniciar a definição do analisador léxico é importante definir quais tokens podem ser encontrados e, para tal definiram-se os seguintes:

Listing 4.1: Definição de Tokens

```
1  tokens = (
2      'ATTRIBUTE',
3      'QUESTION_MARK',
4      'COMMA',
5      'TWO_POINTS',
6      'ATTRIBUTE_VALUE',
7      'ATTRIBUTE_VAR',
8      'PA',
9      'PF',
10     'TAG',
11     'HASHTAG',
12     'ID',
13     'POINT',
14     'CLASS',
```

```
15      'TEXT',
16      'BLOCK_TEXT',
17      'IF',
18      'ELSE',
19      'VAR_JS',
20      'VAR_NAME',
21      'VAR_VALUE',
22      'EQUALS',
23      'PLUS',
24      'MENOS',
25      'MAIOR',
26      'MENOR',
27      'DIF',
28      'MULT',
29      'DIV',
30      'CONJ',
31      'DIJ',
32      'NEG',
33      'EQUIVALENCIA',
34      'MAIORIGUAL',
35      'MENORIGUAL',
36      'MODULO',
37      'VAR_COND',
38      'VALUE_COND',
39      'WHILE',
40      'INDENT',
41      'DEDENT'
42  )
```

Como se faz a leitura de Pug para converter em HTML foi importante definir tokens para a identação e, desidentação que é uma das características principais de Pug, para além disto definiu-se um token para as tags e, caso existam, respetivos atributos, valores e delimitadores de abertura e fecho (PA-'(' e PF-')'). Os tokens para expressões aritméticas também foram criados porque dentro das condições muitas vezes estão envolvidos, a par do léxico que representa condições como if, else, while, etc...

Foi importante, posteriormente, usar estados para controlar o fluxo de interpretação das regras, para tal codificaram-se os seguintes estados:

Listing 4.2: Definição dos estados

```
1 states = (
2     ('pointState','exclusive'),
3     ('barState','exclusive'),
4     ('atributeState', 'exclusive'),
5     ('conditionalState', 'exclusive'),
6     ('dedent', 'exclusive')
7 )
```

Em Pug existem blocos de texto, isto é, um pedaço de conteúdo que surge imediatamente a seguir às tags script ou p (pode seguir-se um ponto, senão distinguem-se porque o texto inicia-se com barra vertical) e que não deve ser interpretado pelas regras, para esse efeito definiram-se os estados pointState e barState. Relativamente, aos estados atributeState e conditionalState estes foram feitos com o intuito de, quando surgirem condições ou atributos, as regras que estão a ser aplicadas até ao momento não sejam sobrepostas com as que se pretendem executar a partir de então. O último estado, dedent, criou-se para solucionar a questão da desidentação gerando assim um correto número de tokens para desidentação.

Por fim, estabeleceram-se as regras que permitem realizar a análise pretendida de acordo com a correspondência entre tokens e expressões regulares.

Capítulo 5

Analisador Sintático

5.1 Definição

O analisador sintático, que denominaremos por parser, é uma parte do processo de compilação de um programa. Este analisa a estrutura gramatical do código-fonte, verificando se o programa está sintaticamente correto de acordo com as regras definidas por uma gramática. É construída ao longo dessa análise uma árvore designada como árvore de derivação que representa a estrutura hierárquica do programa.

5.2 Funcionamento

O trajeto da análise sintática começa pela receção de uma sequência de tokens gerada pelo analisador léxico. O parser verifica a correspondência que encontra entre a sequência de tokens e as regras que são estabelecidas na estrutura gramatical instituída aplicando assim as regras de produção para construir a árvore de análise sintática. Uma vez que a análise sintática é concluída com sucesso, o analisador sintático pode gerar uma saída estruturada, como uma árvore de análise sintática anotada ou uma representação intermediária para ser utilizada por processos da compilação subsequentes.

5.3 Conceção

No analisador sintático foi útil usar estruturas auxiliares, as variáveis guardam em memória as variáveis e, respetivo, valor ao longo da análise, a par disto as self_closing_tags foram necessárias para encerrar um bloco associado a uma tag, isto significa que no analisador léxico não diferenciamos quando se abre ou fecha blocos associados a tags e, assim isto permite encerrá-los corretamente.

Listing 5.1: Estruturas auxiliares

```
1 variaveis = {}
2 self_closing_tags = ["area", "base", "br", "col", "embed", "hr", "img", "input", "
    link", "meta", "param", "source", "track", "wbr"]
```

Fez-se uso de uma grande vantagem do Yacc que é a definição de precedências, assim escrevereram-se as seguintes:

Listing 5.2: Declaração de precedência

```
1 precedence = (
2     ('left', 'DIJ'),
```

```
    ('left', 'CONJ'),
    ('left', 'MAIORIGUAL'),
    ('left', 'MENORIGUAL'),
    ('left', 'EQUIVALENCIA'),
    ('left', 'DIF'),
    ('left', 'NEG'),
    ('left', 'MAIOR'),
    ('left', 'MENOR'),
    ('left', 'MULT'),
    ('left', 'DIV'),
    ('left', 'MODULO'),
    ('left', 'PLUS'),
    ('left', 'MENOS'),
)
```

Quanto a este analisador, as precedências aplicam-se, essencialmente, para cobrir as prioridas que existem associadas aos operadores aritméticos.

De acordo com o código que se apresenta em apêndice a gramática tem a seguinte estrutura:

Listing 5.3: Estrutura da Gramática

```
    S' -> pug
    pug -> elemList
    elemList -> elemList elem
    elemList -> elem
    elem -> TAG PA atributos PF TEXT INDENT elemList DEDENT
    elem -> TAG PA atributos PF TEXT INDENT elemList
    elem -> TAG TEXT INDENT elemList DEDENT
    elem -> TAG TEXT INDENT elemList
    elem -> TAG PA atributos PF INDENT elemList DEDENT
    elem -> TAG PA atributos PF INDENT elemList
    elem -> TAG PA atributos PF INDENT blocks DEDENT
    elem -> TAG PA atributos PF INDENT blocks
    elem -> TAG INDENT elemList DEDENT
    elem -> TAG INDENT elemList
    elem -> TAG INDENT blocks DEDENT
    elem -> TAG INDENT blocks
    elem -> TAG PA atributos PF TEXT
    elem -> TAG TEXT
    elem -> TAG PA atributos PF
    elem -> TAG
    elem -> TAG literals INDENT elemList DEDENT
    elem -> TAG literals INDENT elemList
    elem -> TAG literals
    elem -> literals INDENT elemList DEDENT
    elem -> literals INDENT elemList
    elem -> literals
    elem -> TAG literals TEXT INDENT elemList DEDENT
    elem -> TAG literals TEXT INDENT elemList
    elem -> TAG literals TEXT
    elem -> literals TEXT INDENT elemList DEDENT
    elem -> literals TEXT INDENT elemList
    elem -> literals TEXT
    elem -> IF cond INDENT elemList DEDENT else_if ELSE INDENT elemList DEDENT
    elem -> IF cond INDENT elemList DEDENT ELSE INDENT elemList DEDENT
```

```
35      elem -> IF cond INDENT elemList DEDENT
36      elem -> WHILE cond INDENT elemList DEDENT
37      else_if -> else_if ELSE IF cond INDENT elemList DEDENT
38      else_if -> ELSE IF cond INDENT elemList DEDENT
39      elem -> VAR_JS VAR_NAME EQUALS VAR_VALUE
40      elem -> TAG EQUALS VAR_VALUE
41      blocks -> blocks INDENT BLOCK_TEXT DEDENT
42      blocks -> blocks INDENT BLOCK_TEXT
43      blocks -> blocks BLOCK_TEXT
44      blocks -> BLOCK_TEXT
45      literals -> HASHTAG ID POINT CLASS
46      literals -> HASHTAG ID
47      literals -> POINT CLASS
48      atributos -> atributos atributo
49      atributos -> atributos COMMA atributo
50      atributos -> atributo
51      atributo -> ATTRIBUTE atr QUESTION_MARK atr TWO_POINTS atr
52      atributo -> ATTRIBUTE atr PLUS atr
53      atributo -> ATTRIBUTE atr
54      atr -> ATTRIBUTE_VALUE
55      atr -> ATTRIBUTE_VAR
56      cond -> cond CONJ cond
57      cond -> cond DIJ cond
58      cond -> cond EQUIVALENCIA cond
59      cond -> cond DIF cond
60      cond -> cond MENOR cond
61      cond -> cond MENORIGUAL cond
62      cond -> cond MAIOR cond
63      cond -> cond MAIORIGUAL cond
64      cond -> express
65      express -> express PLUS term
66      express -> express MENOS term
67      express -> term
68      term -> term MULT factor
69      term -> term DIV factor
70      term -> factor
71      factor -> PA express PF
72      factor -> VAR_COND
73      factor -> VALUE_COND
```

Seguidamente, apresenta-se o conjunto de tokens não-terminais

Listing 5.4: Tokens Não-Terminais

```
1 atr
2 atributo
3 atributos
4 blocks
5 cond
6 elem
7 elemList
8 else_if
9 express
10 factor
11 literals
```

```
12 pug
13 term
```

Por último, apresenta-se o conjunto de tokens terminais

Listing 5.5: Tokens Terminais

```
1 ATTRIBUTE
2 ATTRIBUTE_VALUE
3 ATTRIBUTE_VAR
4 BLOCK_TEXT
5 CLASS
6 COMMA
7 CONJ
8 DEDENT
9 DIF
10 DIJ
11 DIV
12 ELSE
13 EQUALS
14 EQUIVALENCIA
15 HASHTAG
16 ID
17 IF
18 INDENT
19 MAIOR
20 MAIORIGUAL
21 MENOR
22 MENORIGUAL
23 MENOS
24 MODULO
25 MULT
26 NEG
27 PA
28 PF
29 PLUS
30 POINT
31 QUESTION_MARK
32 TAG
33 TEXT
34 TWO_POINTS
35 VALUE_COND
36 VAR_COND
37 VAR_JS
38 VAR_NAME
39 VAR_VALUE
40 WHILE
41 error
```

Capítulo 6

Testes

Fazendo uso do exemplo fornecido no enunciado do problema mostramos o resultado obtido:

Output esperado	Output Obtido
`<html lang="en"> <head><title></title> <script type="text/javascript"> if (foo) bar(1 + 5)</script> </head> <body> <h1>Pug – node template engine</h1> <div class="col" id="container"> <p>Get on it!</p> <p>Pug is a terse and simple templating language with a strong focus on performance and powerful features</p> </div> </body> </html>`	`<html lang="en"> <head> <title></title> <script type="text/javascript"> if (foo) bar(1 + 5) </script> </head> <body> <h1> Pug – node template engine </h1> <div class="col" id="container"> <p> Get on it! </p> <p> Pug is a terse and simple templating language with a strong focus on performance and powerful features </p> </div> </body> </html>`

Capítulo 7

Alternativas, Decisões e Problemas de Implementação

Relativamente, a este capítulo pode-se realçar o seguinte ponto:
Há outros tokens definidos pelo analisador léxico que o grupo desejava ter implementado a sua correta interpretação no analisador sintático.

Capítulo 8

Conclusão

De forma geral, o grupo desenvolveu a parte que se pretendia cobrir no enunciado. Infelizmente, não conseguiu avançar noutras partes que idealizava fazê-lo por falta de apoio da equipa docente, relativamente, a esta componente prática.

Apêndice A

Código do Programa

Lista-se a seguir o código do tokenizer que foi desenvolvido.

Listing A.1: Código do Lexer

```
1  import ply.lex as lex
2
3  total_idents = 0
4  total_dedents = 0
5
6  states = (
7      ('pointState','exclusive'),
8      ('barState','exclusive'),
9      ('atributeState', 'exclusive'),
10     ('conditionalState', 'exclusive'),
11     ('dedent', 'exclusive')
12 )
13
14
15 # Definicao dos tokens
16 tokens = (
17     'ATTRIBUTE',
18     'QUESTION_MARK',
19     'COMMA',
20     'TWO_POINTS',
21     'ATTRIBUTE_VALUE',
22     'ATTRIBUTE_VAR',
23     'PA',
24     'PF',
25     'TAG',
26     'HASHTAG',
27     'ID',
28     'POINT',
29     'CLASS',
30     'TEXT',
31     'BLOCK_TEXT',
32     'IF',
33     'ELSE',
34     'VAR_JS',
35     'VAR_NAME',
36     'VAR_VALUE',
```

```
37      'EQUALS',
38      'PLUS',
39      'MENOS',
40      'MAIOR',
41      'MENOR',
42      'DIF',
43      'MULT',
44      'DIV',
45      'CONJ',
46      'DIJ',
47      'NEG',
48      'EQUIVALENCIA',
49      'MAIORIGUAL',
50      'MENORIGUAL',
51      'MODULO',
52      'VAR_COND',
53      'VALUE_COND',
54      'WHILE',
55      'INDENT',
56      'DEDENT'
57  )
58
59  # Express es regulares para cada token
60  def t_ANY_enter_pointState(t):
61      r'\.(?=\n)'
62      t.lexer.block_indent = True
63      t.lexer.push_state('pointState')
64
65  def t_ANY_enter_barState(t):
66      r'(?<=\t)\|'
67      if(t.lexer.current_state() == 'INITIAL'):
68          t.lexer.push_state('barState')
69
70  def t_pointState_barState_BLOCK_TEXT(t):
71      r'[^\t\n]+'
72      return t
73
74  def t_HASHTAG(t):
75      r'\#'
76      return t
77
78  def t_ID(t):
79      r'(?<=\#)[a-zA-Z0-9\_]+'
80      return t
81
82  def t_POINT(t):
83      r'\.'
84      return t
85
86  def t_CLASS(t):
87      r'(?<=\.)\w+'
88      return t
89
90  def t_IF(t):
```

```
91      r'if'
92      t.lexer.push_state('conditionalState')
93      return t
94
95  def t_WHILE(t):
96      r'while'
97      t.lexer.push_state('conditionalState')
98      return t
99
100 def t_ELSE(t):
101     r'else'
102     return t
103
104 def t_ANY_EQUIVALENCIA(t):
105     r'=='
106     return t
107
108 def t_ANY_EQUALS(t):
109     r'='
110     return t
111
112 def t_ANY_PLUS(t):
113     r'\+'
114     return t
115 def t_ANY_DIJ(t):
116     r'\|\|'
117     return t
118
119 def t_ANY_DIF(t):
120     r'!='
121     return t
122
123
124 def t_ANY_DIV(t):
125     r'\/'
126     return t
127
128 def t_ANY_MULT(t):
129     r'\*'
130     return t
131
132 def t_ANY_CONJ(t):
133     r'&&'
134     return t
135
136 def t_ANY_MAIOR(t):
137     r'\>'
138     return t
139
140 def t_ANY_MENOR(t):
141     r'\<'
142     return t
143
144 def t_ANY_NEG(t):
```

```
145     r'\!'
146     return t
147
148 def t_ANY_MODULO(t):
149     r'\%'
150     return t
151 def t_ANY_MAIORIGUAL(t):
152     r'>='
153     return t
154 def t_ANY_MENORIGUAL(t):
155     r'<='
156     return t
157 def t_atributeState_ATTRIBUTE(t):
158     r'(?<=[, \t\(])[ ]?\w+=[ ]?'
159     return t
160
161 def t_atributeState_ATTRIBUTE_VALUE(t):
162     r"(?<=[=  ,?:])['\"][^\n ]+['\"]"
163     return t
164
165 def t_atributeState_ATTRIBUTE_VAR(t):
166     r"(?<=[=  ,?:])[\w\-']+"
167     return t
168
169 def t_VAR_JS(t):
170     r'-[ ]?var(?=\s)|-'
171     return t
172
173 def t_VAR_NAME(t):
174     r'(?<=\bvar\s)\w+|(?<=-\s)\w+|(?<=-)\w+|(?<=\bif\s)\w+'
175     return t
176
177 def t_VAR_VALUE(t):
178     r"((?<=\=)|(?<==\s))['\"]?[^\n\t\+\-\*\&\|]+[\"']?"
179     return t
180
181 def t_TAG(t):
182     r'[a-z]\w*(?=[\(\.\=:])|(?<=\t)[a-z]\w+|[a-z]\w*'
183     return t
184
185 def t_atributeState_COMMA(t):
186     r','
187     return t
188
189 def t_atributeState_QUESTION_MARK(t):
190     r'\?'
191     return t
192
193 def t_ANY_MENOS(t):
194     r'\-'
195     return t
196
197 def t_ANY_TWO_POINTS(t):
198     r':'
```

```
199     return t
200
201 def t_conditionalState_INITIAL_PA(t):
202     r'\('
203     if t.lexer.current_state() == 'INITIAL':
204         t.lexer.begin('atributeState')
205     return t
206
207 def t_atributeState_conditionalState_PF(t):
208     r'\)'
209     if t.lexer.current_state() == 'atributeState':
210         t.lexer.begin('INITIAL')
211     return t
212
213 def t_conditionalState_VAR_COND(t):
214     r'[a-z]\w*'
215     return t
216
217 def t_conditionalState_VALUE_COND(t):
218     r'\d+'
219     return t
220
221 def t_TEXT(t):
222     r'((?<= )|(?<=\|))[^\n]+'
223     return t
224
225 # Define a rule so we can track line numbers
226 def t_ANY_newline(t):
227     r'\n+'
228     t.lexer.lineno += len(t.value)
229
230     tabs_count = 0
231     for char in t.lexer.lexdata[t.lexer.lexpos:]:
232         if char == ' ':
233             tabs_count += 1
234         elif char == '\t':
235             tabs_count += 4
236         else:
237             break
238
239     if t.lexer.current_state() == 'pointState':
240         if t.lexer.block_indent:
241             t.lexer.indent = t.lexer.tabs
242             t.lexer.block_indent = False
243         if tabs_count <= t.lexer.indent:
244             t.lexer.pop_state()
245     elif tabs_count != t.lexer.tabs and t.lexer.current_state() == 'barState':
246         t.lexer.pop_state()
247     elif t.lexer.current_state() == 'conditionalState':
248         t.lexer.pop_state()
249
250     global total_indents, total_dedents
251     if tabs_count > t.lexer.tabs:
252         t.type = 'INDENT'
```

```
253         t.value = tabs_count - t.lexer.tabs
254         t.lexer.tabs = tabs_count
255         return t
256     elif tabs_count < t.lexer.tabs:
257         total_dedents = (t.lexer.tabs - tabs_count) / 4
258         t.lexer.push_state('dedent')
259         t.lexer.tabs = tabs_count
260 
261 
262 def t_dedent_DEDENT(t):
263     r'(.|\n)'
264     global total_dedents
265     t.lexer.lexpos -= 1
266     if total_dedents > 0:
267         total_dedents -= 1
268         t.value = 4
269         return t
270     else:
271         t.lexer.pop_state()
272 
273 
274 # Ignora espa os em branco e tabula  es
275 t_ANY_ignore = ' \t'
276 
277 def t_ANY_error(t):
278     print('Illegal character: ', t.value[0], ' Line: ', t.lexer.lineno)
279     t.lexer.skip(1)
280 
281 lexer = lex.lex()
282 
283 lexer.tabs = 0
284 lexer.block_indent = False
285 lexer.indent = 0
286 
287 lexer.input(pug)
288 
289 #while tok := lexer.token():
290     #print(tok)
```

Lista-se a seguir o código do parser que foi desenvolvido.

Listing A.2: Código do Parser

```
from programa_lex import tokens

variaveis = {}
self_closing_tags = ["area", "base", "br", "col", "embed", "hr", "img", "input", "
    link", "meta", "param", "source", "track", "wbr"]


precedence = (
    ('left', 'DIJ'),
    ('left', 'CONJ'),
    ('left', 'MAIORIGUAL'),
    ('left', 'MENORIGUAL'),
    ('left', 'EQUIVALENCIA'),
    ('left', 'DIF'),
    ('left', 'NEG'),
    ('left', 'MAIOR'),
    ('left', 'MENOR'),
    ('left', 'MULT'),
    ('left', 'DIV'),
    ('left', 'MODULO'),
    ('left', 'PLUS'),
    ('left', 'MENOS'),

)

# Regras de producao da gramatica
def p_pug(p):
    'pug : elemList'
    p[0] = p[1]

def p_elemList(p):
    """elemList : elemList elem
                | elem
    """
    if len(p) == 3:
        p[0] = p[1] + p[2]
    else:
        p[0] = p[1]


def p_elem_tag_atr_text(p):
    """
    elem : TAG PA atributos PF TEXT INDENT elemList DEDENT
         | TAG PA atributos PF TEXT INDENT elemList
    """
    p[0] = f"<{p[1]} {p[3]}> {p[5]} {p[7]} </{p[1]}>"

def p_elem_tag_text(p):
    """
    elem : TAG TEXT INDENT elemList DEDENT
         | TAG TEXT INDENT elemList
```

```
51      """
52      p[0] = f"<{p[1]}> {p[2]} {p[4]} </{p[1]}>"
53
54  def p_elem_tag_atr(p):
55      """
56      elem : TAG PA atributos PF INDENT elemList DEDENT
57           | TAG PA atributos PF INDENT elemList
58      """
59      p[0] = f"<{p[1]} {p[3]}> {p[6]} </{p[1]}>"
60
61  def p_elem_tag_atr_btext(p):
62      '''
63      elem : TAG PA atributos PF INDENT blocks DEDENT
64           | TAG PA atributos PF INDENT blocks
65      '''
66      p[0] = f"<{p[1]} {p[3]}> {p[6]} </{p[1]}>"
67
68  def p_elem_tag(p):
69      """
70      elem : TAG INDENT elemList DEDENT
71           | TAG INDENT elemList
72      """
73      p[0] = f"<{p[1]}> {p[3]} </{p[1]}>"
74
75  def p_elem_tag_btext(p):
76      '''
77      elem : TAG INDENT blocks DEDENT
78           | TAG INDENT blocks
79      '''
80      p[0] = f"<{p[1]}> {p[3]} </{p[1]}>"
81
82  def p_elem_tag_text_sem_in(p):
83      """
84      elem : TAG PA atributos PF TEXT
85           | TAG TEXT
86      """
87      if len(p) == 6:
88          p[0] = f"<{p[1]} {p[3]}> {p[5]} </{p[1]}>"
89      else:
90          p[0] = f"<{p[1]}> {p[2]} </{p[1]}>"
91
92  def p_elem_tag_atr_sem_in(p):
93      """
94      elem : TAG PA atributos PF
95           | TAG
96      """
97      if len(p) == 5:
98          p[0] = f"<{p[1]} {p[3]}></{p[1]}>"
99      else:
100         p[0] = f"<{p[1]}></{p[1]}>"
101
102 def p_elem_tag_literals(p):
103     """
104     elem : TAG literals INDENT elemList DEDENT
```

```
105         | TAG literals INDENT elemList
106         | TAG literals
107     """
108     if len(p) == 6:
109         p[0] = f"<{p[1]} {p[2]}> {p[4]} </{p[1]}>"
110     if len(p) == 5:
111         p[0] = f"<{p[1]} {p[2]}> {p[4]} </{p[1]}>"
112     if len(p) == 3:
113         p[0] = f"<{p[1]} {p[2]}></{p[1]}>"
114
115 def p_elem_literals(p):
116     """
117     elem : literals INDENT elemList DEDENT
118          | literals INDENT elemList
119          | literals
120     """
121     if len(p) == 5:
122         p[0] = f"<div {p[1]}> {p[3]} </div>"
123     elif len(p) == 4:
124         p[0] = f"<div {p[1]}> {p[3]} </div>"
125     else:
126         p[0] = f"<div {p[1]}></div>"
127
128
129 def p_elem_tag_literals_text(p):
130     """
131         elem : TAG literals TEXT INDENT elemList DEDENT
132              | TAG literals TEXT INDENT elemList
133              | TAG literals TEXT
134         """
135     if len(p) == 7:
136         p[0] = f"<{p[1]} {p[2]}> {p[3]} {p[5]} </{p[1]}> "
137     if len(p) == 6:
138         p[0] = f"<{p[1]} {p[2]}> {p[3]} {p[5]} </{p[1]}>"
139     if len (p) == 4:
140         p[0] = f"<{p[1]} {p[2]}> {p[3]} </{p[1]}>"
141
142 def p_elem_literals_text(p):
143     """
144     elem : literals TEXT INDENT elemList DEDENT
145          | literals TEXT INDENT elemList
146          | literals TEXT
147     """
148     if len(p) == 6:
149         p[0] = f"<div {p[1]}> {p[2]} {p[4]} </div> "
150     if len(p) == 5:
151         p[0] = f"<div {p[1]}> {p[2]} {p[4]} </div> "
152     if len (p) == 3:
153         p[0] = f"<div {p[1]}> {p[2]} </div> "
154
155 def p_elem_condition(p):
156     """
157     elem : IF cond INDENT elemList DEDENT else_if ELSE INDENT elemList DEDENT
158          | IF cond INDENT elemList DEDENT ELSE INDENT elemList DEDENT
```

```
159         | IF cond INDENT elemList DEDENT
160         | WHILE cond INDENT elemList DEDENT
161     """
162
163     if len(p) == 10:
164         if p[2]:
165             p[0] = p[4]
166         else:
167             p[0] = p[8]
168     if len(p) == 6:
169         if p[2]:
170             if p.slice[1] == "IF":
171                 p[0] = p[4]
172             else:
173                 p[0] = ""
174                 while p[2]:
175                     p[0]+=p[4]
176
177 def p_else_if(p):
178     '''else_if : else_if ELSE IF cond INDENT elemList DEDENT
179                | ELSE IF cond INDENT elemList DEDENT
180     '''
181     p[0]=""
182
183 def p_elem_var(p):
184     """
185     elem : VAR_JS VAR_NAME EQUALS VAR_VALUE
186          | TAG EQUALS VAR_VALUE
187     """
188     if len(p) == 5:
189         if p[4] == "true":
190             p[4] = True
191         elif p[4] == "false":
192             p[4] = False
193         else:
194             try:
195                 p[4] = float(p[4])
196             except:
197                 p[4] = p[4].replace("'","")
198                 p[4] = p[4].replace('"',"")
199         p[0] = ""
200         variaveis[p[2]] = p[4]
201     if len(p) == 4:
202         p[0] = f"<{p[1]}></{p[1]}>"
203
204 def p_blocks(p):
205     """
206     blocks : blocks INDENT BLOCK_TEXT DEDENT
207            | blocks INDENT BLOCK_TEXT
208            | blocks BLOCK_TEXT
209            | BLOCK_TEXT
210     """
211     if len(p) == 3:
212         p[0] = p[1] + '\n' + p[2]
```

```
213     elif len(p) == 2:
214         p[0] = p[1]
215     else:
216         p[0] = p[1] + '\n' + p[3]
217 
218 def p_literals(p):
219     """
220     literals : HASHTAG ID POINT CLASS
221              | HASHTAG ID
222              | POINT CLASS
223     """
224     if len(p) == 5:
225         p[0] = " class=" + '"' + p[4] + '"' + " id=" + '"' + p[2] + '"'
226     elif len(p) == 3:
227         p[0] = p.slice[2].type.lower() + "=" + '"' + p[2] + '"'
228 
229 def p_atributos(p):
230     """
231     atributos : atributos atributo
232               | atributos COMMA atributo
233               | atributo
234     """
235     if len(p) == 3:
236         p[0] = p[1] + " " + p[2]
237     elif len(p) == 4:
238         p[0] = p[1] + " " + p[3]
239     elif len(p) == 2:
240         p[0] = p[1]
241 
242 def p_atributo(p):
243     """
244     atributo : ATTRIBUTE atr QUESTION_MARK atr TWO_POINTS atr
245              | ATTRIBUTE atr PLUS atr
246              | ATTRIBUTE atr
247     """
248     if len(p) == 7:
249         if p[2] == True:
250             p[0] = p[1] + '"' + p[4] + '"'
251         else:
252             p[0] = p[1] + '"' + p[6] + '"'
253     elif len(p) == 5:
254         p[0] = p[1] + '"' + p[2] + p[4] + '"'
255     elif len(p) == 3:
256         p[0] = p[1] + '"' + p[2] + '"'
257 
258 def p_atr_value(p):
259     'atr : ATTRIBUTE_VALUE'
260     p[1] = p[1].replace("'",'')
261     p[1] = p[1].replace('"','')
262     p[0] = p[1]
263 
264 def p_atr_var(p):
265     'atr : ATTRIBUTE_VAR'
266     if p[1] in variaveis.keys():
```

```
267         p[0] = variaveis[p[1]]
268
269 def p_cond(p):
270     """
271     cond : cond CONJ cond
272          | cond DIJ cond
273          | cond EQUIVALENCIA cond
274          | cond DIF cond
275          | cond MENOR cond
276          | cond MENORIGUAL cond
277          | cond MAIOR cond
278          | cond MAIORIGUAL cond
279          | express
280     """
281     if len(p) == 4:
282         if p.slice[2].type == 'CONJ':
283             p[0] = p[1] and p[3]
284         elif p.slice[2].type == 'DIJ':
285             p[0] = p[1] or p[3]
286         elif p.slice[2].type == "EQUIVALENCIA":
287             p[0] = p[1] == p[3]
288         elif p.slice[2].type == 'DIF':
289             p[0] = p[1] != p[3]
290         elif p.slice[2].type == 'MENORIGUAL':
291             p[0] = p[1] <= p[3]
292         elif p.slice[2].type == 'MAIORIGUAL':
293             p[0] = p[1] >= p[3]
294         elif p.slice[2].type == 'MENOR':
295             p[0] = p[1] < p[3]
296         elif p.slice[2].type == 'MAIOR':
297             p[0] = p[1] > p[3]
298     elif len(p) == 2:
299         p[0] = p[1]
300
301 def p_express(p):
302     """
303     express : express PLUS term
304             | express MENOS term
305             | term
306     """
307
308     if len(p) == 4 and p.slice[2].type == "PLUS":
309         p[0] = p[1] + p[3]
310     if len(p) == 4 and p.slice[2].type == "MENOS":
311         p[0] = p[1] - p[3]
312     elif len(p) == 2:
313         p[0] = p[1]
314
315 def p_term(p):
316     """
317     term : term MULT factor
318          | term DIV factor
319          | factor
320     """
```

```
321     if len(p) == 4 and p.slice[2].type == "MULT":
322         p[0] = p[1] * p[3]
323     if len(p) == 4 and p.slice[2].type == "DIV":
324         p[0] = p[1] / p[3]
325     elif len(p) == 2:
326         p[0] = p[1]
327
328 def p_factor(p):
329     """
330     factor : PA express PF
331            | VAR_COND
332            | VALUE_COND
333     """
334     if len(p) == 4:
335         p[0] = p[2]
336     elif len(p) == 2:
337         if p.slice[1].type == "VAR_COND":
338             if p[1] in variaveis.keys():
339                 p[0] = variaveis[p[1]]
340             else:
341                 p[0] = False
342         else:
343             p[0] = int(p[1])
344
345 # Define a rule so we can track line numbers
346 #def p_newline(p):
347 #     p.lineno += len(p.value)
348
349 def p_error(p):
350     print('Syntax error: ', p, ' Line: ', p.lineno)
351
352 ##########################################################################
353
354 parser = yacc.yacc()
355
356 def parse_file(file_path):
357     with open(file_path, 'r') as f:
358         return f.read()
359
360
361 def pug_to_html(pug_file):
362     pug_code = parse_file(pug_file)
363     html_code = parser.parse(pug_code, tracking=False, debug=False)
364
365     with open("outputs/output1.html", "w", encoding="utf-8") as f:
366         f.write(html_code)
367
368 pug_to_html('datasets/ex1.pug')
```