Processamento de Linguagens
(3º ano de Licenciatura em Engenharia Informática)

Trabalho Prático 2

Relatório de Desenvolvimento

Rui Monteiro (a93179)

Rodrigo Rodrigues (a93201)

Daniel Azevedo (a93324)

11 de maio de 2022

Resumo

Este relatório aborda o desenvolvimento de um compilador que traduz PLY-simple para PLY, no contexto do 2º trabalho prático de Processamento de Linguagens.

Conteúdo

Capítulo 1

Introdução

Através do desenvolvimento deste trabalho prático, pretende-se criar um compilador que seja capas de transformar PLY-simple, linguagem que especifica um *parser* e um lex de um determinado programa, para PLY.

Para atingir esse objetivo, primeiramente, foi definido um analisador léxico, utilizando a biblioteca lex que se encontra definida na biblioteca ply. Seguidamente foi desenvolvido um modulo, que recorre à biblioteca yacc da biblioteca especifica anteriormente, que será o analisador sintático, este irá utilizar os *tokens*, assim como os *literals*, que forram especificados no analisador léxico.

Capítulo 2

Lexer

Através do *Lexer* serão definidos os *tokens* e também os *literals* que serão usados pelo analisador sintático.

2.1 Tokens

Nesta secção serão especificados os *tokens* que foram definidos no analisador léxico. Este tem como funcionalidade analisar um conjunto de caracteres que serão traduzido em símbolos léxico, denominados por *tokens*.

2.1.1 PAL

```
1    t_PAL = r"[A-Za-z\.\(\_0-9]+"
```

Através da utilização deste *token*, podemos captar um conjunto que contenha **letras, números ou os símbolos . (_ .**

2.1.2 TOKEN

```
1    t_TOKEN = r"\'[^\']+\'"
```

Este *token* é capaz de analisar um conjunto de carácter que inicie e termine com **'**.

2.1.3 SIMB

```
1    t_SIMB = r"\"[^\"]+\""
```

À semelhança do *tokens* especificado anteriormente, este encontra conjuntos que iniciem e terminem com **"**.

2.1.4 PRECEDENCE

```
1    t_PRECEDENCE = r"\((\'[^\']+\'\,)+(\'[^\']+\')+\)"
```

O *token PRECEDENCE* é utilizado para analisar a lista de precedência, que se encontra no ficheiro de PLY-simple, este é capaz de encontra um ou mais elementos dessa lista.

2.1.5 NEWLINE

```
1        t_NEWLINE = r"\n"
```

Por fim, é apresentado o *token NEWLINE* este é responsável por encontrar novas linhas, dentro do ficheiro a analisar.

2.1.6 Caracteres que serão ignorados

```
1        t_ignore = " \t"
```

Através da especificação t_ignore podemos definir o que o analisador irá descartar do ficheiro de input, neste caso serão ignorados os espaços e os *tabs*.

2.2 Literals

Diferente dos *tokens*, a lista de *Literals*, capta caracteres que são estáticos. Seguidamente são indicados o *literals* que forma usados no analisador léxico.

```
1        literals =['+','-','*','/','=','%','_','[',']',',','{','}','.',':','(',')']
```

Capítulo 3

Parser

Neste capitulo serão apresentados os símbolos terminais e não terminais assim como a definição da gramática.

3.1 Gramática

De modo a se definir a gramática, que será usado no analisador sintático, serão usadas 59 produções. Esta encontra-se dividida em duas partes, a primeira corresponde à parte responsável por analisar o Lex, que se encontra no ficheiro que especifica a linguagem PLY-simple, por sua vez a segunda parte analisa o yacc.

```
1  P0    S' -> Gramatica
2  P1    Gramatica -> Lex Yacc
3  P2    Lex -> % % PAL ListNewLines List Defs Erro
4  P3    List -> Literals Ignore Tokens
5  P4    Literals -> % PAL = SIMB ListNewLines
6  P5    Ignore -> % PAL = SIMB ListNewLines
7  P6    Tokens -> % PAL = ListTokens ListNewLines
8  P7    ListTokens -> [ TOKEN ListTokens ]
9  P8    ListTokens -> , TOKEN
10 P9    ListTokens ->
11 P10   Defs -> Def Defs
12 P11   Defs ->
13 P12   Def -> SIMB PAL { TOKEN , PAL Fim } ListNewLines
14 P13   Erro -> - PAL SIMB , PAL Fim ) ListNewLines
15 P14   Yacc -> % % PAL ListNewLines Precedence Gramar Code
16 P15   Precedence -> % PAL = [ ListNewLines ListPrecedence ] ListNewLines
17 P16   ListPrecedence -> PRECEDENCE , ListNewLines ListPrecedence
18 P17   ListPrecedence ->
19 P18   Gramar -> Productions Gramar
20 P19   Gramar ->
21 P20   Productions -> PAL : Exp { Id } ListNewLines
22 P21   Exp -> PAL Exp
23 P22   Exp -> TOKEN Exp
24 P23   Exp -> % PAL Exp
25 P24   Exp ->
26 P25   Id -> Atr Math ListAtr Fim
27 P26   Atr -> PAL [ Atr ]
28 P27   Atr -> PAL
29 P28   ListAtr -> Atr Math ListAtr
30 P29   ListAtr ->
```

```
31 P30     Fim -> )
32 P31     Fim ->
33 P32     Math -> =
34 P33     Math -> +
35 P34     Math -> -
36 P35     Math -> *
37 P36     Math -> /
38 P37     Math -> _
39 P38     ListNewLines -> NEWLINE ListNewLines
40 P39     ListNewLines ->
41 P40     Math -> = +
42 P41     Math -> = -
43 P42     Math ->
44 P43     Code -> % % ListNewLines ListDefsCode
45 P44     ListDefsCode -> DefsCode ListNewLines ListDefsCode
46 P45     ListDefsCode ->
47 P46     DefsCode -> PAL = PAL ) NEWLINE PAL SIMB ) ListNewLines
48 P47     DefsCode -> PAL PAL Fim : NEWLINE ListLinhaCode
49 P48     ListLinhaCode -> LinhaCode ListLinhaCode
50 P49     ListLinhaCode ->
51 P50     LinhaCode -> ListElem ) NEWLINE
52 P51     ListElem -> Elem ListElem
53 P52     ListElem ->
54 P53     Elem -> PAL
55 P54     Elem -> TOKEN
56 P55     Elem -> SIMB
57 P56     Elem -> :
58 P57     Elem -> ,
59 P58     Elem -> =p
```

3.2 Símbolos Terminais

Aqui serão apresentados os símbolos terminais que a gramática acima definida pode usar. Essa lista corresponde aos *tokens* juntamente com o *literals*.

```
1 %
2 )
3 *
4 +
5 ,
6 -
7 /
8 :
9 =
10 NEWLINE
11 PAL
12 PRECEDENCE
13 SIMB
14 TOKEN
15 [
16 ]
17 _
18 error
```

```
{
}
```

3.3 Símbolos não Terminais

Por sua vez, símbolos não terminais presentes na gramática são os seguintes:

```
Atr
Code
Def
Defs
DefsCode
Elem
Erro
Fim
Gramar
Gramatica
Ignore
List
ListDefsCode
ListElem
ListLinhaCode
ListNewLines
ListPrecedence
ListTokens
Literals
Precedence
Productions
Tokens
```

3.4 Main

A *main* encontra-se definida no ficheiro de *parse*. Esta começa por pedir, via terminal, o ficheiro escrito em PLY-simple. Posteriormente irá criar dois ficheiros um parser.py e um lexer.py e irão estar na diretoria out.

Capítulo 4

Conclusão

Com a realização deste trabalho prático, foi possível aprofundar os diversos conhecimentos de que foram lecionados durante as aulas de Processamento de linguagens. Desta forma, conseguimos desenvolver um *lexer* e um *parser* que são capazes de interpretar a linguagem PLY-simple e traduzi-la para PLY.

Apêndice A

Código do Programa

lexer.py

```
import ply.lex as lex

tokens = ['PAL','TOKEN','SIMB','PRECEDENCE','NEWLINE']
literals = ['+','-','*','/','=','%','_','[',']',',','{','}','.',':','(',')']

t_ignore = " \t"
t_PAL = r"[A-Za-z\.\(\_0-9]+"
t_TOKEN = r"\'[^\']+\'"
t_SIMB = r"\"[^\"]+\""
t_PRECEDENCE = r"\((\'[^\']+\'\,)+(\'[^\']+\')+\)"
t_NEWLINE = r"\n"

def t_error(t):
    print('Car ter ilegal: ', t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()
```

parser.py

```
1  import ply.yacc as yacc
2  from lexer import tokens
3
4
5  def split(palavra):
6      return [char for char in palavra]
7
8
9  def p_gramatica(p):
10     "Gramatica : Lex Yacc "
11     filename_out_parser = "out/parser.py"
12     filename_out_lexer = "out/lexer.py"
13     fileout1 = open(filename_out_parser,'w')
14     fileout2 = open(filename_out_lexer,'w')
15     fileout1.write(p[2])
16     fileout2.write(p[1])
17     fileout1.close()
18     fileout2.close()
19
20 #LEX
21 def p_lex(p):
22     "Lex : '%' '%' PAL ListNewLines List Defs Erro"
23     p[0] = "import ply.lex as lex\n\n" + p[5] + "\n"+ p[6] +"\n"+p[7]+"\n\n"+"lexer =
           lex.lex()"
24
25 def p_list(p):
26     "List : Literals Ignore Tokens"
27     p[0] = p[3] + "\n" +p[1] + "\n\n"+ p[2]+ "\n"
28
29 def p_literals(p):
30     "Literals : '%' PAL '=' SIMB ListNewLines"
31     p[0] = "literals = " + str(split(p[4][1:-1]))
32
33 def p_ignore(p):
34     "Ignore : '%' PAL '=' SIMB ListNewLines"
35     p[0] = "t_ignore =" + p[4]
36
37 def p_tokens(p):
38     "Tokens : '%' PAL '=' ListTokens ListNewLines"
39     p[0] = "tokens = [" + p[4] +"]"
40
41 def p_listtokens_with_value(p):
42     "ListTokens : '[' TOKEN ListTokens ']'"
43     p[0] = p[2] +p[3]
44
45 def p_listtokens_one(p):
46     "ListTokens : ',' TOKEN"
47     p[0] = p[1]+ p[2]
48
49 def p_listtokens_empty(p):
50     "ListTokens : "
51
52 def p_defs_with_value (p):
```

```
53      "Defs : Def Defs"
54      p[0] = p[1] + "\n\n" + p[2]
55
56  def p_defs_empty (p):
57      "Defs : "
58      p[0] = ""
59
60  def p_def (p):
61      "Def : SIMB PAL '{' TOKEN ',' PAL Fim '}' ListNewLines "
62      p[0] = "def t_" + p[4][1:-1] +"(t):\n\tr"+p[1] +"\n\tt.value = " + p[6]+p[7]+"\n\
            t"+p[2] + " t"
63
64  def p_erro (p):
65      "Erro : '-' PAL SIMB ',' PAL Fim ')' ListNewLines"
66      p[0] = "def t_error(t):\n\tprint(" + p[3]+")\n\t" + p[5]+p[6]
67
68
69  #YACC
70  def p_yacc (p):
71      "Yacc : '%' '%' PAL ListNewLines Precedence Gramar Code "
72      p[0] = "import ply.yacc as yacc\nfrom lexer import tokensp\n\n"+p[5]+"\n" + p[6]
            + p[7]
73
74  def p_precedence (p):
75      "Precedence : '%' PAL '=' '[' ListNewLines ListPrecedence ']' ListNewLines"
76      p[0] = p[2]+ p[3] + "(\n\t" +p[6]  + ")\n"
77
78  def p_listPrecedence_with_value(p):
79      "ListPrecedence : PRECEDENCE ',' ListNewLines ListPrecedence"
80      p[0] = p[1] + p[2] + "\n\t" + p[4]
81
82  def p_listPrecedence_empty(p):
83      "ListPrecedence : "
84      p[0] = ""
85
86  def p_gramar_with_values(p):
87      "Gramar : Productions Gramar"
88      p[0] = p[1]+p[2]
89
90  def p_gramar_empty(p):
91      "Gramar :"
92      p[0] = ""
93
94  def p_productions_Productions(p):
95      "Productions : PAL ':' Exp '{' Id  '}' ListNewLines"
96      p[0] = "def p_"+p[1] +"(p):\n\t\"" + p[1]+ " : " + p[3] + "\"\n\t"  + p[5]  +"\n\
            n"
97
98  def p_exp_with_values(p):
99      '''
100     Exp : PAL Exp
101         | TOKEN Exp
102     '''
103     p[0] = p[1] +" "+ p[2]
```

```
104
105 def p_exp_with_values_P(p):
106     "Exp : '%' PAL Exp"
107     p[0] = p[1] +" "+ p[2] +" "+ p[3]
108
109 def p_exp_empty(p):
110     "Exp :"
111     p[0] = ""
112
113 def p_id_at (p):
114     "Id : Atr Math  ListAtr Fim"
115     p[0] = p[1] + p[2] + p[3]+p[4]
116
117 def p_atr (p):
118     "Atr : PAL '[' Atr ']'"
119     p[0] = p[1] + p[2] + p[3] + p[4]
120
121 def p_atr_empty (p):
122     "Atr : PAL"
123     p[0] = p[1]
124
125 def p_listAtr_with_values(p):
126     "ListAtr : Atr Math ListAtr"
127     p[0] = p[1] + p[2] + p[3]
128
129 def p_listAtr_empty(p):
130     "ListAtr : "
131     p[0] = ""
132
133 def p_fim(p):
134     "Fim : ')' "
135     p[0] = p[1]
136
137 def p_fim_empty(p):
138     "Fim : "
139     p[0] = ""
140
141 def p_math_one (p):
142     '''
143     Math : '='
144          | '+'
145          | '-'
146          | '*'
147          | '/'
148          | '-'
149     '''
150     p[0] = p[1]
151
152 def p_list_newlines_with_value(p):
153     "ListNewLines : NEWLINE ListNewLines"
154
155 def p_list_newline_empty(p):
156     "ListNewLines : "
157
```

```
158 def p_math_two (p):
159     '''
160     Math : '=' '+'
161          | '=' '-'
162     '''
163     p[0] = p[1] + p[2]
164
165 def p_math_empty (p):
166     "Math : "
167     p[0] = ''
168
169 def p_code(p):
170     "Code : '%' '%' ListNewLines ListDefsCode "
171     p[0] = p[4]
172
173 def p_listdefscode_with_value(p):
174     "ListDefsCode : DefsCode ListNewLines ListDefsCode"
175     p[0] = p[1]+"\n"+p[3]
176
177 def p_listdefscode_empty(p):
178     "ListDefsCode : "
179     p[0] = ""
180
181 def p_DefsCode_Parser(p):
182     "DefsCode : PAL '=' PAL ')' NEWLINE PAL SIMB ')' ListNewLines"
183     p[0] = p[1] + p[2]+ "yacc." + p[3] + p[4] + "\n" + p[6] + p[7] + p[8] + '\n'
184
185 def p_DefsCode(p):
186     "DefsCode : PAL PAL Fim ':' NEWLINE ListLinhaCode"
187     p[0] = p[1]+" "+p[2]+p[3]+p[4]+p[6]+"\n"
188
189 def p_listLinhaCode_with_value(p):
190     "ListLinhaCode : LinhaCode ListLinhaCode "
191     p[0] = "\n" + p[1] + p[2]
192
193 def p_listLinhaCode_empty(p):
194     "ListLinhaCode : "
195     p[0] = ""
196
197 def p_linhaCode_with_value(p):
198     "LinhaCode : ListElem ')' NEWLINE"
199     p[0] = "\t" + p[1] + p[2]
200
201 def p_ListElem_with_value(p):
202     "ListElem : Elem ListElem"
203     p[0] = p[1] + p[2]
204
205 def p_ListElem_empty(p):
206     "ListElem : "
207     p[0] = ""
208
209 def p_elem(p):
210     '''
211     Elem : PAL
```

```
212             | TOKEN
213             | SIMB
214             | ':'
215             | ','
216             | '='
217     '''
218     p[0] = p[1] + " "
219
220
221 def p_error(p):
222     print('Erro sint tico: ', p)
223     parser.success = False
224
225 # Build the parser
226 parser = yacc.yacc()
227
228 # Read line from input and parse it
229
230 import sys
231
232 def main():
233     print("Choose the file to process, press 'q' to quit.")
234     for line in sys.stdin:
235         filename_in = line.rstrip('\n')
236         if 'q' == filename_in.rstrip():
237             break
238         filein = open(filename_in,'r')
239         if not filein:
240             print("Invalid file, try again.")
241             break
242         else:
243             content = filein.read()
244             parser.parse(content)
245             filein.close()
246             print("Done, see you later!")
247             break
248
249 main()
```

Apêndice B

Exemplos e Resultados

Input

```
%% LEX
%literals = "+-/*=()"
%ignore = "\t\n"
%tokens =  [ 'VAR','NUMBER' ]

"[a-zA-Z_][a-zA-Z0-9_]*"      return{'VAR', t.value}
"\d+(\.\d+)?"                 return{'NUMBER', float(t.value)}
-                             error(f"Illegal character '{t.value[0]}', [{t.lexer.
    lineno}]",t.lexer.skip(1))

%% YACC

%precedence = [
        ('left','+','-'),
        ('left','*','/'),
        ('right','UMINUS'),
        ]

stat : VAR '=' exp { ts[t[1]] = t[3] }
stat : exp  { print(t[1]) }
exp : exp '+' exp { t[0] = t[1] + t[3] }
exp : exp '-' exp { t[0] = t[1] - t[3] }
exp : exp '*' exp { t[0] = t[1] * t[3] }
exp : exp '/' exp { t[0] = t[1] / t[3] }
exp : '-' exp %prec UMINUS { t[0] = -t[2] }
exp : '(' exp ')' { t[0] = t[2] }
exp : NUMBER { t[0] = t[1] }
exp : VAR { t[0] = getval(t[1]) }

%%
def p_error(t):
        print(f"Syntax error at    {t.value}    , [{t.lexer.lineno}]")

def getval(n):
        if n not in ts: print(f"Undefined name    {n}    ")
        return ts.get(n,0)

```

```
37 y=yacc()
38 y.parse("3+4*7")
```

Output (Lexer)

```
import ply.lex as lex

tokens = ['VAR','NUMBER']
literals = ['+', '-', '/', '*', '=', '(', ')']

t_ignore ="\t\n"

def t_VAR(t):
        r"[a-zA-Z_][a-zA-Z0-9_]*"
        t.value = t.value
        return t

def t_NUMBER(t):
        r"\d+(\.\d+)?"
        t.value = float(t.value)
        return t


def t_error(t):
        print("Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
        t.lexer.skip(1)

lexer = lex.lex()
```

Output (Parser)

```
import ply.yacc as yacc
from lexer import tokensp

precedence=(
        ('left','+','-'),
        ('left','*','/'),
        ('right','UMINUS'),
        )

def p_stat(p):
        "stat : VAR '=' exp "
        ts[t[1]]=t[3]

def p_stat(p):
        "stat : exp "
        print(t[1])

def p_exp(p):
        "exp : exp '+' exp "
        t[0]=t[1]+t[3]

def p_exp(p):
        "exp : exp '-' exp "
        t[0]=t[1]-t[3]

def p_exp(p):
        "exp : exp '*' exp "
        t[0]=t[1]*t[3]

def p_exp(p):
        "exp : exp '/' exp "
        t[0]=t[1]/t[3]

def p_exp(p):
        "exp : '-' exp %prec UMINUS "
        t[0]=-t[2]

def p_exp(p):
        "exp : '(' exp ')' "
        t[0]=t[2]

def p_exp(p):
        "exp : NUMBER "
        t[0]=t[1]

def p_exp(p):
        "exp : VAR "
        t[0]=getval(t[1])

def p_error(t):
        print(f "Syntax error at    {t.value}    , [{t.lexer.lineno}]" )

def getval(n):
```

```
54          if n not in ts : print(f "Undefined name     {n}     " )
55          return ts.get(n , 0 )
56
57 y=yacc.yacc()
58 y.parse("3+4*7")
```