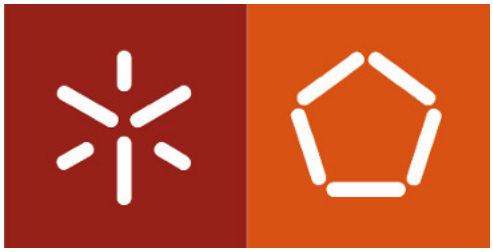

Universidade do Minho
Licenciatura em Engenharia Informática

Processamento de Linguagens

Trabalho Prático 2 - Grupo 52

Armando Silva - A87949 Joana Oliveira - A87956

Maio 2022

Conteúdo

1 Introdução

O seguinte projeto foi elaborado no âmbito da UC de Processamento de Linguagens e tinha como objetivo o desenvolvimento de um compilador capaz de transformar um ficheiro em sintaxe PLY-simple num ficheiro com a sintaxe normal de PLY. Foi nos fornecido um ficheiro base exemplo que posteriormente sofreu alterações de modo a facilitar a sua leitura. A linguagem de programação utilizada foi Python, com recurso às ferramentas lex e yacc.

Ao longo do relatório irá ser explicado mais detalhadamente o problema, bem como todo o processo de implementação da gramática responsável pela conversão do ficheiro.

2 Análise e Especificação do Problema

2.1 Descrição Informal

Desenvolver um tradutor que permita transformar uma sintaxe PLY-simple, criada pelo grupo, nas respetivas funções PLY convenientes.

2.2 Requisitos

De modo a cumprir com o objetivo deste projeto foi necessário estabelecer os requisitos necessários para a sua realização. Nomeadamente, a elaboração de uma **boa sintaxe** que nos permita alternar entre estados facilmente, a análise léxica através do **lex** e análise sintática com o uso do **yacc** e, por fim, a criação de uma **gramática de tradução** e **independente do contexto**.

3 Desenho da Solução

Chegamos à fase do desenho da solução, onde será apresentada a sintaxe PLY-simple. Iremos explicar a forma como esta foi definida, bem como apresentar a gramática desenvolvida, de modo a fazer a sua tradução para PLY.

3.1 Sintaxe PLY-simple

Sintaxe desenvolvida pelo grupo com o objetivo de simplificar a sintaxe PLY original. Foi usada como base a sintaxe fornecida no enunciado e de seguida foram adicionadas e feitas algumas alterações, que serão apresentadas de seguida.

Começamos então por falar das várias divisões do ficheiro, que permitiram ajudar no desenvolvimento da gramática:

- %% LEX - Início da parte referente ao Lex

- %% FUNCTIONS - Funções do Lex
- %% YACC - Início da parte referente ao Yacc
- %% ERS - Funções da gramática do Yacc
- %% PYTHON - Conteúdo na linguagem Python
- %% - Fim do ficheiro

Dentro de cada divisão poderá ser ainda feita a identificação de variáveis e comentários. As variáveis encontram-se em seguida de um caracter % e os comentários após um # ou ##, sendo que estes apenas podem estar presentes na linha acima daquilo a que se referem ou logo a seguir à atribuição, no caso das variáveis.

Na divisão dos ERS, optamos por dividir a expressão regular, que se encontrava inicialmente toda junta, em duas partes, através do uso de aspas. Assim sendo, a parte que se encontra antes dos dois pontos será utilizada na nomeação da função respetiva e será depois adicionada à parte que se encontra entre aspas para criar uma expressão regular.

No anexo presente de seguida encontra-se o exemplo dado no enunciado com as alterações mencionadas.

Listagem 1. Sintaxe *PLY-simple*

```
1 %% LEX
2 %literals = "+-/*=()" ## a single char
3 %ignore = " \t\n"
4 %tokens = [ 'VAR','NUMBER' ]
5
6 %% FUNCTIONS
7 [a-zA-Z_][a-zA-Z0-9_]* return('VAR', t.value )
8 \d+(\.\d+)?            return('NUMBER', float(t.value) )
9 .                      error(f"Illegal character '{t.value[0]}', [{t
    .lexer.lineno}]",
10                              t.lexer.skip(1) )
11
12 %% YACC
13
14 %precedence = [ ('left','+','-'), ('left','*','/'), ('right','UMINUS
    '), ]
15
16 # symboltable : dictionary of variables
17 %ts = {}
18
19 %% ERS
20 stat : "VAR '=' exp"          { ts[t[1]] = t[3] }
21 stat : "exp"                  { print(t[1]) }
22 exp : "exp '+' exp"          { t[0] = t[1] + t[3] }
23 exp : "exp '-' exp"          { t[0] = t[1] - t[3] }
24 exp : "exp '*' exp"          { t[0] = t[1] * t[3] }
25 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]) }

%% PYTHON

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)

%y=yacc()

%% PYTHON

y.parse("3+4*7")

%%
```

3.2 Gramática definida

Após definida a sintaxe é então necessário construir uma gramática que permita traduzi-la e fazer a respetiva tradução para sintaxe normal de PLY. A gramática desenvolvida tem recursividade à esquerda, seguindo os exemplos das aulas práticas e encontra-se apresentada de seguida. A sua explicação mais pormenorizada será feita no capítulo seguinte, referente à implementação.

Listagem 2. Gramática contruída

```
P0  : Programa -> lex yacc

P1  : lex ->
P2  :     | LEX vars FUNCTIONS funcs

P3  : yacc ->
P4  :      | YACC vars ERS ers PYTHON python vars PYTHON python END

P5  : vars ->
P6  :      | vars comment var vcomment

P7  : var -> PERC ID '=' STRING
P8  :     | PERC ID '=' LIST
P9  :     | PERC ID '=' EMPTYLIST
P10 :     | PERC ID '=' YACC
P11 :     | PERC ID '=' NUM
P12 :     | PERC ID '=' REGEX

P13 : funcs ->
```

```
20 P14 :         | funcs comment func
21
22 P16 : func -> ER RETURN PAL PAL
23 P17 :        | PONTO ERROR PAL STRING PAL
24 P18 :        | PONTO ERROR STRING PAL
25 P19 :        | END
26
27 P20 : python ->
28 P21 :          | python TEXT
29
30 P22 : ers ->
31 P23 :       | ers comment er
32
33 P24 : er -> EXP STRING EXP
34
35 P25 : vcomments ->
36 P26 :             | VCOMMENT
37
38 P27 : comments ->
39 P28 :            | COMMENT
```

4 Implementação

4.1 Lex

Para a implementação do analisador léxico começamos por definir os símbolos terminais. Estes podem ser literais (*literals*) ou não literais (*tokens*), dependendo da sua função no programa.

Listagem 3. Tokens e Literals

```
1 tokens = ['LEX', 'YACC', 'FUNCTIONS', 'ERS',
2         "COMMENT", 'ID', 'STRING', 'PERC', "ER",
3         "RETURN", "PAL", "ERROR", "LIST", "END",
4         "EMPTYLIST", "EXP", "TEXT", "PYTHON",
5         "PONTO", "VCOMMENT", "NUM", "REGEX"]
6
7 literals = ['=']
```

Além disso foram também definidos diversos estados que representam diferentes secções da sintaxe.

Listagem 4. Estados

```
1 states = [("var", "exclusive"), ("func", "exclusive"), ("er", "
     exclusive"), ("python", "exclusive")]
```

Cada *token* tem uma expressão regular associada que o permite identificar e estas são apresentadas de seguida. Estes permitem denotar identificadores de variáveis e estados, expressões regulares, comentários, listas, etc.

Listagem 5. Expressões regulares associadas a cada token

```
def t_LEX(t):
    r'%%\sLEX'
    return t

def t_YACC(t):
    r'%%\sYACC'
    return t

def t_PYTHON(t):
    r'%%\sPYTHON'
    t.lexer.begin("python")
    return t

def t_FUNCTIONS(t):
    r'%%\sFUNCTIONS'
    t.lexer.begin("func")
    return t

def t_ERS(t):
    r'%%\sERS'
    t.lexer.begin("er")
    return t

def t_VCOMMENT(t):
    r'\#{2}.*'
    return t

def t_COMMENT(t):
    r'\#.*'
    return t

def t_func_COMMENT(t):
    r'\#.*'
    return t

def t_PERC(t):
    r'%'
    t.lexer.begin("var")
    return t

def t_var_REGEX(t):
    r'r\'.*\''
    t.lexer.begin("INITIAL")
    return t

def t_var_NUM(t):
    r'[0-9]+'
    t.lexer.begin("INITIAL")
    return t

def t_var_YACC(t):
    r'yacc\(\)'
```

```
53      t.lexer.begin("INITIAL")
54      return t
55
56  def t_var_ID(t):
57      r'[a-zA-Z_][a-zA-Z0-9_\.]*'
58      return t
59
60  def t_var_STRING(t):
61      r'\".*\"'
62      t.lexer.begin("INITIAL")
63      return t
64
65  def t_var_LIST(t):
66      r'\[.*\]'
67      t.lexer.begin("INITIAL")
68      return t
69
70  def t_var_EMPTYLIST(t):
71      r'\{\}'
72      t.lexer.begin("INITIAL")
73      return t
74
75  def t_func_RETURN(t):
76      r'return'
77      return t
78
79  def t_func_ERROR(t):
80      r'error'
81      return t
82
83  def t_func_YACC(t):
84      r'%%\sYACC'
85      t.lexer.begin("INITIAL")
86      return t
87
88  def t_func_STRING(t):
89      r'\".*\"'
90      return t
91
92  def t_func_PONTO(t):
93      r'\.'
94      return t
95
96  def t_func_PAL(t):
97      r'[a-zA-Z0-9\.\(\)]+'
98      return t
99
100 def t_func_ER(t):
101     r'[^\s]+'
102     return t
103
104 def t_er_STRING(t):
105     r'\".*\"'
106     return t
```

```
107
108 def t_er_PYTHON(t):
109     r'%%\sPYTHON'
110     t.lexer.begin("python")
111     return t
112
113 def t_er_EXP(t):
114     r'[^\}\"\:]+'
115     return t
116
117 def t_python_END(t):
118     r'%%'
119     t.lexer.begin("INITIAL")
120     return t
121
122 def t_python_PERC(t):
123     r'%'
124     t.lexer.begin("var")
125     return t
126
127 def t_python_TEXT(t):
128     r'.+'
129     return t
130
131 t_ignore = " \t\n\r"
132 t_var_ignore = " \t\n\r"
133 t_func_ignore = " \t\n\r,'()"
134 t_er_ignore = " \t\n\r}{:"
135 t_python_ignore = "\n\t\r"
136
137 def t_var_error(t):
138     print("Illegal character: ", t.value[0])
139     t.lexer.skip(1)
140
141 def t_func_error(t):
142     print("Illegal character: ", t.value[0])
143     t.lexer.skip(1)
144
145 def t_er_error(t):
146     print("Illegal character: ", t.value[0])
147     t.lexer.skip(1)
148
149 def t_python_error(t):
150     print("Illegal character: ", t.value[0])
151     t.lexer.skip(1)
152
153 def t_error(t):
154     print("Illegal character: ", t.value[0])
155     t.lexer.skip(1)
156
157 lexer = lex.lex()
```

4.2 Yacc

O analisador sintático permitiu fazer o reconhecimento da gramática desenvolvida. Para tal, foi necessário associar a cada produção reconhecida, uma ação a realizar. Estas ações consistem em armazenar o conteúdo que deverá estar depois no ficheiro resultado. Foi então necessário recorrer a uma string *output* associada ao parser, que irá conter todo o código Python respetivo. Além disso, existe também uma outra estrutura de dados, a *erfunc*, que consiste num dicionário que associa a cada nome de função um contador que nos permite enumerar as diferentes funções existentes.

Iremos então agora apresentar e analisar individualmente cada um dos símbolos não terminais implementados.

4.2.1 Programa

O primeiro símbolo não terminal é o símbolo '*programa* e é aquele que permite inicar toda a gramática. Este é composto pelos símbolos '*lex*' e '*yacc*, que permitem identificar as diferentes partes do programa.

Listagem 6. Programa

```
1 def p_programa(p):
2     'programa : lex yacc'
3     p[0] = p[1] + p[2]
4     parser.output += p[0]
```

4.2.2 Lex

Parte da gramática que define toda a componente do Lex. Contém os símbolos '*LEX*', '*vars*', '*FUNCTIONS*' e '*funcs*'. Estes correspondem ao símbolo que identifica o início do Lex, às atribuições de variavéis, ao símbolo identificador do início das funções e à escrita das mesmas, respetivamente.

Listagem 7. Lex

```
1 def p_lex(p):
2     'lex : LEX vars FUNCTIONS funcs'
3     p[0] = "import ply.lex as lex\n\n" + p[2] + "\n" + p[4] + "lexer
      = lex.lex()\n\n\n"
```

4.2.3 Yacc

Corresponde à parte Yacc do programa. Esta é iniciada pelo símbolo '*YACC*', seguida de '*vars*', que contém todas as variáveis associadas a esta parte do programa. Além disso, esta é também composta pelos símbolos que identificam o início dos diferentes estados, bem como o fim do programa, tais como '*ERS*', '*PYTHON*' e '*END*'. Entre estes é possível encontrar os símbolos presentes entre cada um dos estados tais

como '*vars*', '*ers*' e '*python*'.

Listagem 8. Yacc

```
def p_yacc(p):
    'yacc : YACC vars ERS ers PYTHON python vars PYTHON python END'
    p[0] = "import ply.yacc as yacc\n\n" + p[2] + "\n" + p[4] + p[6]
    + p[7] + "\n" + p[9]
```

4.2.4 Vars

O símbolo '*vars*' é recursivamente chamado à esquerda enquanto existirem variáveis a ler. Além disso, aqui estão também incluídos os símbolos '*comment*' e 'vcomment' que identificam, respetivamente, os comentários existentes na linha anterior e os comentários que se encontram logo a seguir a cada variável.

Listagem 9. Vars

```
def p_vars_var(p):
    'vars : vars comment var vcomment'
    p[0] = p[1] + p[2] + p[3] + " " + p[4]

def p_vars_empty(p):
    'vars : '
    p[0] = ''
```

4.2.5 Var

O símbolo 'var' contém todos os tipos de atribuições aceitáveis. É sempre incializado pelo símbolo '*PERC*', correspondente à percentagem. A atribuição a ser feita a cada variável pode uma '*STRING*', '*LIST*', '*EMPTYLIST*', '*YACC*', '*NUM*' ou 'REGEX'. De realçar que '*EMPTYLIST*' equivale a '{}' e 'YACC' a 'yacc()'.

Listagem 10. Var

```
def p_var_number(p):
    "var : PERC ID '=' NUM"
    p[0] = p[2] + ' = ' + p[4]

def p_var_regex(p):
    "var : PERC ID '=' REGEX"
    p[0] = f"t_{p[2]}" + ' = ' + p[4]

def p_var_string(p):
    "var : PERC ID '=' STRING"
    if p[2] == "ignore":
        p[0] = f"t_{p[2]}" + ' = ' + p[4]
    else :
        p[0] = p[2] + ' = ' + p[4]
```

```
15
16 def p_var_yacc(p):
17     "var : PERC ID '=' YACC"
18     p[0] = "\n" + p[2] + ' = ' + "yacc." + p[4] + "\n"
19
20 def p_var_lista(p):
21     "var : PERC ID '=' LIST"
22     p[0] = p[2] + ' = ' + p[4]
23
24 def p_var_emptylista(p):
25     "var : PERC ID '=' EMPTYLIST"
26     p[0] = p[2] + ' = ' + p[4]
```

4.2.6 Funcs

À semelhança do que acontece com o símbolo '*vars*', também o símbolo '*funcs*' é chamado recursivamente à esquerda consoante existirem funções. Neste caso a existência de comentários apenas poderá ser feita antes de cada função, sendo isso representado pelo símbolo '*func*' seguido de '*comment*'.

Listagem 11. Funcs

```
1 def p_funcs_list(p):
2     'funcs : funcs comment func'
3     p[0] = p[1] + p[2] + p[3] + "\n"
4
5 def p_funcs_empty(p):
6     'funcs : '
7     p[0] = ''
```

4.2.7 Func

Existem dois tipos de funções diferentes no nosso programa. As normais, compostas por uma expressão regular, o respetivo nome e um valor a retornar. Estas representam o primeiro caso apresentado e recorrem aos símbolos '*ER*','*RETURN*' e 'PAL'. O segundo tipo de função corresponde às funções de erro. Devido à maneira como foram definidos os nossos tokens e a conseguirmos englobar diferentes tipos de prints, optamos por criar dois casos distintos. O primeiro caso corresponde a um print formatado que é iniciado com o caracter "f"e que é reconhecido pelo símbolo 'PAL'. O segundo caso apenas se trata de um print normal. Em ambos os casos este tipo de funções é identificado pela existência de um 'PONTO' no início da linha, bem como a respetiva 'STRING' e a 'PAL' a serem impressas.

Listagem 12. Func

```
1 def p_func_ret(p):
2     "func : ER RETURN PAL PAL"
3     p[0] = "def " + "t_" + p[3] + "(t):" + "\n\t" + f"r'{p[1]}'\n\t"
      + f"t.value = {p[4]}\n\t" + f"{p[2]} t\n"
4
5 def p_func_errorf(p):
```

```
6       "func : PONTO ERROR PAL STRING PAL"
7       p[0] = "def t_error(t):\n\t" + f"print({p[3]}{p[4]})\n\t" + p[5]
        + "\n"
8
9   def p_func_error(p):
10      "func : PONTO ERROR STRING PAL"
11      p[0] = "def t_error(t):\n\t" + f"print({p[2]})\n\t" + p[3] + "\n
       "
12
13  def p_func_end(p):
14      'func : END'
15      p[0] = ''
```

4.2.8 Python

O símbolo '*python*' está associado à parte do programa em que o texto lá contido já se encontra na sua forma final e ao qual não terão de ser feitas quaisquer alterações. Para tal este símbolo é usado recursivamente à esquerda, juntamente com o símbolo 'TEXT' que nos permite ler todo o conteúdo de uma linha.

Listagem 13. Python

```
1   def p_python_list(p):
2       "python : python TEXT"
3       p[0] = p[1] + p[2] + "\n"
4
5   def p_python_empty(p):
6       "python : "
7       p[0] = ''
```

4.2.9 Ers

Parte da gramática relativa às funções do yacc, compostas por expressões regulares. O símbolo 'ers' é chamado recursivamente à esquerda até não encontrar mais nenhuma função. A existência de comentários é também possível graças ao símbolo 'comment', sendo que estes terão de ser sempre feitos antes da função respetiva.

Listagem 14. Ers

```
1   def p_ers_empty(p):
2       "ers : "
3       p[0] = ''
4
5   def p_ers_list(p):
6       "ers : ers comment er"
7       p[0] = p[1] + p[2] + p[3] + "\n"
```

4.2.10 Er

O símbolo 'er' representa uma função do yacc. Todas as funções têm a mesma sintaxe, tal como é apresentado de seguida. O primeiro símbolo '*EXP*' representa o

nome que irá ser dado à função. Como o mesmo nome poderá estar a ser associado a diferentes funções, recorremos a um dicionário que a cada nome associa um contador. O número contido nesse contador será depois utilizado para numerar as diferentes funções. O símbolo '*STRING*' representa parte da expressão regular que, juntamente com o símbolo mencionado anteriormente, irá formar a expressão regular associada à função em questão. Por fim, o último símbolo '*EXP*' representa qualquer outra coisa a ser incluída no corpo da função.

Listagem 15. Er

```
def p_er(p):
    "er : EXP STRING EXP"
    var = p[1][:-1]
    if var in p.parser.erfunc:
        p.parser.erfunc[var] += 1
        p[1] = var + f'_{p.parser.erfunc[var]}'
    else:
        p.parser.erfunc[var] = 1
        p[1] = var + f'_{p.parser.erfunc[var]}'

    p[0] = f"def p_{p[1]}(t):\n\t" + f"\"{var} : {p[2][1:]}\n\t{p
   [3]}\n"
```

4.2.11 Comment e VComment

Os símbolos 'comment' e 'vcomment' representam ambos comentários, sendo que a única coisa que os distingue é o local onde se encontram. Os 'comment' aparecem na linha anterior relativamente aquilo a que se referem, sendo esta identificação feita através do símbolo 'COMMENT'. Já os 'vcomment' são comentários que se encontram na mesma linha que a variável que foi definida e à qual se referem.

Listagem 16. Comment e VComment

```
def p_vcomment_com(p):
    "vcomment : VCOMMENT"
    p[0] = p[1] + "\n"

def p_vcomment_empty(p):
    "vcomment : "
    p[0] = "\n"

def p_comment_com(p):
    "comment : COMMENT"
    p[0] = p[1] + "\n"

def p_comment_empty(p):
    "comment : "
    p[0] = ''
```

5 Testes

De forma a provar o bom funcionamente do programa efetuamos dois testes distintos e que se encontram apresentados de seguida. O primeiro teve como base o exemplo fornecido no enunciado, apenas com pequenas alterações e o segundo foi criado usando uma abordagem um pouco diferente e inspirado num exercício feito durante as aulas práticas.

5.1 1º Teste

Ao correr o programa sem qualquer argumento, este irá, por omissão, utilizar o ficheiro da Listagem 1. Apresentamos então o ficheiro Python obtido.

Listagem 17. Ficheiro Python obtido no 1º teste

```
1  import ply.lex as lex
2
3  literals = "+-/*=()" ## a single char
4  t_ignore = " \t\n"
5  tokens = [ 'VAR','NUMBER' ]
6
7  def t_VAR(t):
8   r'[a-zA-Z_][a-zA-Z0-9_]*'
9   t.value = t.value
10  return t
11
12 def t_NUMBER(t):
13  r'\d+(\.\d+)?'
14  t.value = float(t.value)
15  return t
16
17 def t_error(t):
18  print(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
19  t.lexer.skip(1)
20
21 lexer = lex.lex()
22
23
24 import ply.yacc as yacc
25
26 precedence = [ ('left','+','-'), ('left','*','/'), ('right','UMINUS'
      ), ]
27 # symboltable : dictionary of variables
28 ts = {}
29
30 def p_stat_1(t):
31  "stat : VAR '=' exp"
32  ts[t[1]] = t[3]
33
34 def p_stat_2(t):
```

```
35  "stat : exp"
36  print(t[1])
37
38 def p_exp_1(t):
39  "exp : exp '+' exp"
40  t[0] = t[1] + t[3]
41
42 def p_exp_2(t):
43  "exp : exp '-' exp"
44  t[0] = t[1] - t[3]
45
46 def p_exp_3(t):
47  "exp : exp '*' exp"
48  t[0] = t[1] * t[3]
49
50 def p_exp_4(t):
51  "exp : exp '/' exp"
52  t[0] = t[1] / t[3]
53
54 def p_exp_5(t):
55  "exp : '-' exp %prec UMINUS"
56  t[0] = -t[2]
57
58 def p_exp_6(t):
59  "exp : '(' exp ')'"
60  t[0] = t[2]
61
62 def p_exp_7(t):
63  "exp : NUMBER"
64  t[0] = t[1]
65
66 def p_exp_8(t):
67  "exp : VAR"
68  t[0] = getval(t[1])
69
70 def p_error(t):
71     print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
72 def getval(n):
73     if n not in ts: print(f"Undefined name '{n}'")
74     return ts.get(n,0)
75
76 y = yacc.yacc()
77
78
79 y.parse("3+4*7")
```

Fazendo a excecução deste ficheiro Python obtivemos o seguinte resultado, o que permite comprovar a correta conversão do programa de uma sintaxe para outra.

```
armando@ASUS:~/Documents/LEI/3ano/PL/TP/PL2122/TP2$ python3 sintaxe.py
Generating LALR tables
31.0
```

Figura 1. Output do ficheiro Python anterior

5.2 2º Teste

O ficheiro do segundo teste, tal como já mencionado anteriormente, utilizou como base um exemplo das aulas práticas e foi convertido pelo grupo para PLY-simple. Esse ficheiro pode ser visto de seguida.

Listagem 18. Ficheiro input utilizado para o 2ºteste

```
1  %% LEX
2  %tokens = ["NUM", "PAL"] ##tokens
3  %literals = ["[" , "]" , ","] ##literals
4  %ignore = " \n\t\r"
5  %NUM = r'\d+'
6  %PAL = r'[a-zA-Z]+'
7
8  %% FUNCTIONS
9  # ERRO
10 .               error(f"Illegal character {t.value[0]}", t.lexer.
       skip(1) )
11
12 %% YACC
13
14 %% ERS
15 lista : "'[' conteudo ']'" { pass }
16 conteudo : "  " { pass }
17 conteudo : "elementos" { pass }
18 elementos : "elem" { pass }
19 elementos : "elem ',' elementos" { pass }
20 elem : "NUM" { t.parser.contador_num += 1 }
21 elem : "PAL" { t.parser.contador_pal += 1 }
22 elem : "lista" { pass }
23
24 %% PYTHON
25
26 # ERRO
27 def p_error(t):
28     print("Syntax error!")
29
30 %parser=yacc()
31
32 # counter of NUMS
33 %parser.contador_num = 0
34
35 # counter of PALS
36 %parser.contador_pal = 0
37
```

```
%% PYTHON

parser.parse("[1,A,2]")
print("N mero de NUMs:", parser.contador_num)
print("N mero de PALs:", parser.contador_pal)

%%
```

Executando o nosso conversor, obtivemos o seguinte resultado.

Listagem 19. Ficheiro Python obtido no 2º teste

```
import ply.lex as lex

tokens = ["NUM", "PAL"] ##tokens
literals = ["[" , "]" , ","] ##literals
t_ignore = " \n\t\r"
t_NUM = r'\d+'
t_PAL = r'[a-zA-Z]+'

# ERRO
def t_error(t):
 print(f"Illegal character {t.value[0]}")
 t.lexer.skip(1)

lexer = lex.lex()


import ply.yacc as yacc


def p_lista_1(t):
 "lista : '[' conteudo ']'"
 pass

def p_conteudo_1(t):
 "conteudo :   "
 pass

def p_conteudo_2(t):
 "conteudo : elementos"
 pass

def p_elementos_1(t):
 "elementos : elem"
 pass

def p_elementos_2(t):
 "elementos : elem ',' elementos"
 pass

def p_elem_1(t):
 "elem : NUM"
 t.parser.contador_num += 1
```

```
43
44 def p_elem_2(t):
45  "elem : PAL"
46  t.parser.contador_pal += 1
47
48 def p_elem_3(t):
49  "elem : lista"
50  pass
51
52
53 # ERRO
54 def p_error(t):
55     print("Syntax error!")
56
57 parser = yacc.yacc()
58
59 # counter of NUMS
60 parser.contador_num = 0
61 # counter of PALS
62 parser.contador_pal = 0
63
64 parser.parse("[1,A,2]")
65 print("N mero de NUMs:", parser.contador_num)
66 print("N mero de PALs:", parser.contador_pal)
```

Ao executar o ficheiro Python, obtivemos este resultado que, mais uma vez, comprova a correta conversão do programa.

```
armando@ASUS:~/Documents/LEI/3ano/PL/TP/PL2122/TP2$ python3 test_1.py
Número de NUMs: 2
Número de PALs: 1
```

Figura 2. Output do ficheiro Python anterior

6 Conclusão

Terminada a realização deste trabalho prático, consideramos pertinente fazer uma apreciação crítica, realçando os seus pontos positivos e negativos, demonstrando as dificuldades e obstáculos que surgiram durante a realização do mesmo.

Começando pelos pontos positivos, consideramos que este programa cumpre com os seus requisitos, sendo capaz de fazer uma tradução acertada da sintaxe PlY-simple para PLY e usando uma gramática que não apresenta qualquer tipo de conflito, sendo também simples e fácil de entender.

No entanto, o grupo sente que é possível que a gramática não tenha em conta todos os casos possíveis da sintaxe PLY-simple e que esta possa talvez ser simplificada um pouco em alguns casos.

Por fim, o grupo está satisfeito com o trabalho desenvolvido, pois este cumpre os requisitos pedidos. Para além disso, a realização deste trabalho ajudou-nos a consolidar a matéria dada relativamente à criação e manipulação de gramáticas tradutoras e independentes de contexto.