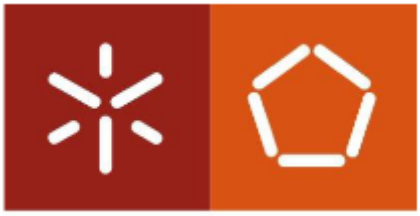

Processamento de Linguagens
(3º ano LEI)

Trabalho Prático 2
Enunciado 2 (Tradutor PLY-simple para PLY)

Relatório de Desenvolvimento

Grupo 45

a93241 Francisco Reis Izquierdo
a89526 Duarte Augusto Rodrigues Lucas
a83920 Afonso Trindade Araújo de Pascoal Faria

15 de maio de 2022

Conteúdo

Listings

Lista de Figuras

1 Introdução

No âmbito da unidade curricular de Processamento de Linguagens, foi proposto ao grupo de trabalho a escolha e consequente implementação de um dos enunciados de trabalho dispostos pela equipa docente. O enunciado escolhido pelo grupo tem como especificação um tradutor de uma linguagem intitulada ***PLY-simple*** num subconjunto da linguagem **Python** que interage com a biblioteca ***PLY***. A implementação deste tradutor deve ser feita através da linguagem Python.

Posto isto, o presente relatório tem como objetivo dar a conhecer toda a informação acerca da abordagem tomada para com o problema em estudo, bem como estratégias adotadas e focar o desenvolvimento e implementação face ao enunciado escolhido e descrito acima.

Além disso, o grupo de trabalho adotou algumas das metodologias abordadas nas aulas teóricas e práticas de forma a compactuar com o pedido no enunciado e aplicar os conceitos prestados pela equipa docente, de entre os quais podemos referir o uso das expressões regulares como o principal conceito que foi abordado em detalhe ao longo do projeto respetivo ao relatório em questão.

Por fim, na linha de raciocínio previamente mencionada, foram usadas ferramentas no auxílio da implementação do projeto, nomeadamente módulos que permitissem a utilização dos conceitos previamente referidos.

2 Análise do Problema

2.1 Descrição do problema

Tal como referido anteriormente, o enunciado escolhido pelo grupo de trabalho tem como especificação a tradução de ficheiros característicos em formato PLY-simple em ficheiros cujo formato final é Python + PLY.

De forma prática, o programa desenvolvido pela equipa teria de ser capaz de, passado o nome de um ficheiro em PLY-simple pela linha de comandos, gerar um ficheiro de *output* capaz de ser interpretado por Python, com recurso à biblioteca PLY.

2.2 Formato PLY-simple

Um ficheiro no formato PLY-simple está dividido em três secções: **LEX**, **YACC** e **Python**.

2.2.1 LEX

Na secção LEX, que começa com o símbolo `%% LEX`, são definidos os atributos **literals**, **ignore** e **tokens**.

literals é uma *string* na qual cada caráter é retornado "como está"quando encontrado pelo *lexer*. Os *literals* são verificados após todas as regras de expressões regulares definidas. Assim, se uma regra começar com um dos caráteres presente nos *literals*, terá sempre precedência.

ignore também é uma *string* na qual cada caráter, aquando da sua ocorrência no *input*, é ignorado pelo *lexer*. Normalmente é utilizado para ignorar caráteres *whitespace*, como `' '` (espaço) e `\t` (*tab*).

tokens é uma lista de *strings*, em que cada *string* corresponde ao nome dado a cada *token*.

Também é definido um conjunto de ações semânticas para cada padrão. Os padrões são descritos através de expressões regulares, e as ações semânticas através de sintaxe de Python.

2.2.2 YACC

Na secção YACC, que começa com o símbolo `%% YACC`, são definidas as regras de precedência e as regras de produção.

As regras de precedência seguem a mesma sintaxe de Python, e a mesma semântica da biblioteca PLY. Isto é, são representadas através de uma variável global com o nome `precedence`, que se trata de uma lista de tuplos.

Cada tuplo é composto por, no mínimo duas *strings*. A primeira *string* representa a associatividade (`'left'` ou `'right'`) de um ou mais operadores. Esses operadores são os seguintes elementos do tuplo.
Por exemplo, o tuplo `('left', '+', '-')` significa que os operadores `'+'` e `'-'` são associativos à esquerda.

Para além disso, o índice de cada tuplo nesta lista determina linearmente o seu nível de precedência, ou seja, quanto maior o índice, maior a precedência.
Por exemplo, a lista `[('left', '+'), ('left', '*')]` descreve dois operadores `'+'` e `'*'`, ambos associativos à esquerda, na qual o operador `'*'` tem uma maior precedência que operador `'+'`.

Por fim, nesta secção também são descritas as regras de produção do *parser*, e as ações semânticas associadas.
As regras de produção seguem a mesma sintaxe descrita pela biblioteca PLY.
As ações semânticas são código Python rodeado por chavetas.

2.2.3 Python

A secção de Python começa com o símbolo `%%`. Nesta secção, como o nome indica, qualquer código Python é válido, e é diretamente copiado para o ficheiro de *output*, sem qualquer modificação.

3 Estratégias Adotadas

3.1 Alterações feitas à sintaxe PLY-simple

O grupo optou por alterar os seguintes aspetos da sintaxe PLY-simple:

- as *keywords* **literals**, **ignore**, **tokens** e **precedence** não começam com %;
- as expressões regulares da secção **LEX** são rodeadas por `r''`.
  Por exemplo,
  `\d+(\.\d+)?`
  fica
  `r'\d+(\.\d+)?'`
- as regras de produções da secção YACC são rodeadas por `<>`.
  Por exemplo,
  `exp : NUMBER { t[0] = t[1] }`
  fica
  `exp : < NUMBER > { t[0] = t[1] }`

3.2 LEX

Para começar, definimos as variáveis `literals`, `reserved`, `tokens`, `states` que serão utilizadas pela biblioteca PLY para gerar o lexer.

```
literals = "=\'[]\",(){}.:+-*/<>?"
```

Listing 1: Variável `literals`.

```
reserved = {
  "literals" : "Literals",
  "tokens" : "Tokens",
  "ignore" : "Ignore",
  "return" : "Return",
  "right" : "Right",
  "left" : "Left",
  "precedence" : "Precedence",
  "error" : "Error"
}
```

Listing 2: Variável `reserved`.

```
tokens = [
    "Comment",
  "Info",
  "END",
  "Section",
  "Regex",
  "Id",
  "Value",
  "Chars",
  "Tvalue",
  "Index",
  "Python",
  "FuncElem"
] + list(reserved.values())
```

Listing 3: Variável `tokens`.

```
states = [
    ("Comment", "exclusive"),
    ("Section", "exclusive"),
    ("Python", "exclusive"),
    ("Function", "exclusive")
]
```

Listing 4: Variável `states`.

Cada um dos estados tem o seguinte significado:

3.2.1 INITIAL

O estado *default* do lexer. É a partir deste estado que todos os *tokens* presentes na variável `tokens` são interpretados.

Neste estado, os caráteres ␣ (espaço), `\n` (*newline*) e `\t` (*tab*) são ignorados.

Também é definida a função `t_error`, que é executada quando é encontrada uma expressão inválida.

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

Listing 5: Variável `t_ignore`.

```
1 def t_error(t):
2   print(f"<Error Message>: Illegal character '{t.value[0]}', in
      line {t.lexer.lineno}\n")
3   print("----------------Please fix your file!----------------")
4   sys.exit()
```

Listing 6: Função `t_error`.

3.2.2 Comment

O estado no qual o lexer se encontra quando está a ser interpretado um comentário.

Este estado é iniciado quando é encontrado o *token* `Comment`, associado ao caráter `#`, e é terminado quando é encontrado o *token* `Comment_END`, associado ao caráter `\n` (*newline*).

Qualquer caráter diferente de `\n` (*newline*) é válido no contexto de um comentário, e nenhum caráter é ignorado.

Também é definida a função `t_Comment_error`, que é executada quando é encontrada uma expressão inválida no contexto de um comentário.

```
1  # Beginning of a comment.
2  def t_Comment(t):
3    r'\#'
4    t.lexer.begin("Comment")
5    return t
6
7  # Content of a comment.
8  def t_Comment_Info(t):
9    r'.+'
10   return t
11
12 # End of a comment.
13 def t_Comment_END(t):
14   r'\n'
15   t.lexer.begin("INITIAL")
16   return t
```

Listing 7: *Tokens* e ações semânticas associadas a comentários.

```
1  t_Comment_ignore = ""
```

Listing 8: Variável `t_Comment_ignore`.

```
1  def t_Comment_error(t):
2    print(f"<Error Message>: Illegal character '{t.value[0]}' in
       comment (line: " + str(t.lexer.lineno) + ")\n")
3    print("----------------Please fix your file!----------------")
4    sys.exit()
```

Listing 9: Função `t_Comment_error`.

3.2.3 Section

O estado no qual o lexer se encontra quando está a ser interpretada uma linha que inicia uma secção de PLY-simple, como por exemplo `%% LEX` ou `%% YACC`.

Neste estado, o valor da secção é associado ao `import` de Python correspondente. Isto é, uma secção `%% LEX` corresponde a `import ply.lex as lex` e uma secção `%% YACC` corresponde a `import ply.yacc as yacc`.

Este estado é terminado quando é encontrado o *token* `t_Section_END`, associado ao caráter `\n` (*newline*).

Também é definida a função `t_Section_error`, que é executada quando é encontrada uma expressão inválida no contexto de uma secção.

```
1  def t_Section(t):
2    r'%%.+'
3    t.lexer.begin("Section")
4
5    regex = r'%%(.+)'
6    regexExp = re.compile(regex)
7    imp = regexExp.search(t.value).group(1)
8    imp = imp.replace(" ", "")
9
10   t.value = f"import ply.{imp.lower()} as {imp.lower()}"
11
12   return t
13
14 def t_Section_END(t):
15   r'\n'
16   t.lexer.begin("INITIAL")
17   return t
```

Listing 10: *Tokens* e ações semânticas associadas a secções.

```
1 t_Section_ignore = ""
```

Listing 11: Variável `t_Section_ignore`.

```
1 def t_Section_error(t):
2   print(f"<Error Message>: Illegal character '{t.value[0]}' in code
       section (line: " + str(t.lexer.lineno) + ")\n")
3   print("----------------Please fix your file!----------------")
4   sys.exit()
```

Listing 12: Função `t_Section_error`.

3.2.4 Python

O estado no qual o lexer se encontra quando está a ser interpretado qualquer código Python.

Este estado é iniciado quando é encontrada a expressão `%%\n`.

Também é definida a função `t_Python_error`, que é executada quando é encontrada uma expressão inválida no contexto de código Python.

```
1 def t_Python(t):
2   r'%%\n'
3   t.lexer.begin("Python")
4   return t
5
6
7 def t_Python_Info(t):
8   r'[A-Za-z0-9\!\"\#\$\%\&\'\(\)\*\+\,\-\.\/\:\;\<\>\=\?\@
      \[\]\{\}\\\\^\_\`\~\n\t ]+'
9   return t
```

Listing 13: *Tokens* e ações semânticas associadas à secção de Python.

```
1 t_Python_ignore = ""
```

Listing 14: Variável `t_Python_ignore`.

```
1 def t_Python_error(t):
2   print(f"<Error Message>: Illegal character '{t.value[0]}' in
      Pyhton (line: " + str(t.lexer.lineno) + ")\n")
3   print("----------------Please fix your file!----------------")
4   sys.exit()
```

Listing 15: Função `t_Python_error`.

3.2.5 Function

```
def t_Function_Info(t):
  r'[A-Za-z0-9\!\"\#\$\%\&\'\(\)\*\+\,\-\.\/\:\;\<\>\=\?\@
    \[\]\{\}\\\\^\_\`\~\t ]+'
  return t


def t_Function_END(t):
  r'\n'
  t.lexer.begin("INITIAL")
  return t
```

Listing 16: *Tokens* e ações semânticas associadas a funções.

```
t_Function_ignore = ""
```

Listing 17: Variável `t_Function_ignore`.

```
def t_Function_error(t):
  print(f"<Error Message>: Illegal character '{t.value[0]}' in
    Error message (line: " + str(t.lexer.lineno) + ")\n")
  print("----------------Please fix your file!----------------")
  sys.exit()
```

Listing 18: Função `t_Function_error`.

Para além dos *tokens* já descritos nos estados anteriores, restam os seguintes:

3.2.6 Info

```
def t_Info(t):
  r'\{[^\n\t]+\}'
  return t
```

Listing 19: Função `t_Info`.

3.2.7 Regex

```
def t_Regex(t):
  r'r\'[^\n\t]+\''
  return t
```

Listing 20: Função `t_Regex`.

3.2.8 Id

```
def t_Id(t):
  r'([a-zA-Z\%][a-zA-Z0-9_]+(\.[a-zA-Z][a-zA-Z0-9_])?)+'
  t.type = reserved.get(t.value, 'Id')
  if t.type == "Error":
    t.lexer.begin("Function")

  return t
```

Listing 21: Função `t_Id`.

3.2.9 Value

```
def t_Value(t):
  r'[\-\+]?[0-9]+(\.[0-9]+)?'
  return t
```

Listing 22: Função `t_Value`.

3.2.10 Chars

```
def t_Chars(t):
  r'\"[^\t\n]+\"|f\"[^\t\n]+\"'
  return t
```

Listing 23: Função `t_Chars`.

3.2.11 Tvalue

```
def t_Tvalue(t):
  r'((float\(|int\(|double\(|str\()t\.value\)|t\.value)'
  return t
```

Listing 24: Função `t_Tvalue`.

3.2.12 Index

```
def t_Index(t):
  r'[a-zA-Z][a-zA-Z0-9_]*\[\d+\]'
  return t
```

Listing 25: Função `t_Index`.

3.2.13 FuncElem

```
def t_FuncElem(t):
  r'([a-zA-Z][a-zA-Z0-9_]*\.)*[a-zA-Z][a-zA-Z0-9_]*\(\)'
  return t
```

Listing 26: Função `t_FuncElem`.

3.3 YACC

Nesta secção, começamos por definir as seguintes regras de produção:

```
ply : commentary
    | section
    | atribution
    | atribution commentary
    | lex
    | lex commentary
    | yacc
    | yacc commentary
    | python
    |
```

Listing 27: Variável *states*

Cada um dos termos destas regras de produção tem o seguinte significado:

3.3.1 commentary

Esta produção define o conjunto de *tokens* de forma a reconhecer um comentário.

```
def p_commentary(p):
  '''
  commentary : Comment Info END
  '''
  p[0] = p[1] + p[2]
```

Listing 28: Função `p_commentary`.

3.3.2 section

Esta produção define o conjunto de *tokens* de forma a reconhecer um *import* associado a uma *code section*.

```
def p_section(p):
  '''
  section : Section END
  '''
  p[0] = p[1] + "\n"
```

Listing 29: Função `p_section`.

3.3.3 atribution

Esta produção define o conjunto de *tokens* de forma a reconhecer um atributo, de forma a ser possível, em qualquer lado do ficheiro, criar atribuições de diversos tipos à semelhança de *Python*, tal como listas, dicionários, *strings*.

```
def p_atribution(p):
  '''
  atribution : Id '=' exp
         | Id '=' FuncElem
         | Id '=' Chars
         | Id '=' list
         | Id '=' dic
         | Index '=' exp
         | Index '=' Chars
         | Index '=' list
         | Index '=' dic
         | Index '=' Index
  '''
  p[0] = p[1] + " " + p[2] + " " + p[3] + "\n"
```

Listing 30: Função `p_atribution`.

3.3.4 lex

Esta produção define o conjunto de *tokens* de forma a reconhecer qualquer padrão relativo à parte léxica de código, isto é, de forma a reconhecer o código definido e possível associado à *code section Lex*.

```
def p_lex(p):
  '''
  lex : regex
    | reservedWordsLex
    | erro
  '''
  p[0] = p[1]
```

Listing 31: Função `p_lex`.

3.3.5 yacc

Esta produção define o conjunto de *tokens* de forma a reconhecer qualquer padrão relativo à parte sintática de código, isto é, de forma a reconhecer o código definido e possível associado à *code section Yacc*.

```
def p_yacc(p):
  '''
  yacc : reservedWordsYacc
     | productions
  '''
  p[0] = p[1]
```

Listing 32: Função `p_yacc`.

3.3.6 python

Esta produção define o conjunto de *tokens* de forma a reconhecer qualquer padrão relativo à parte *Python* de código, isto é, de forma a reconhecer o código definido e possível associado a *Python*.

```
1 def p_Pyhton(p):
2   '''
3   python : Python
4          | Info
5   '''
6   if p[1] == "%%\n":
7     p[0] = ""
8   else:
9     p[0] = p[1]
```

Listing 33: Função `p_Pyhton`.

4 Leitura de ficheiros

A leitura de ficheiros é feita linha a linha. Para cada linha lida é efetuado *lexing* e *parsing*. O resultado destas transformações é automaticamente escrito para o ficheiro de *output*.

Assim, é importante realçar que não são usadas quaisquer estruturas de dados para representar informação intermediária. Esta escolha foi tomada de modo a facilitar a implementação do tradutor. Observamos que a introdução de representações intermediárias iria causar complexidade desnecessária na implementação.

```
1 for line in inputFile:
2   parser.parse(line)
```

Listing 34: Excerto de código "alto nível"responsável pela tradução linha a linha.

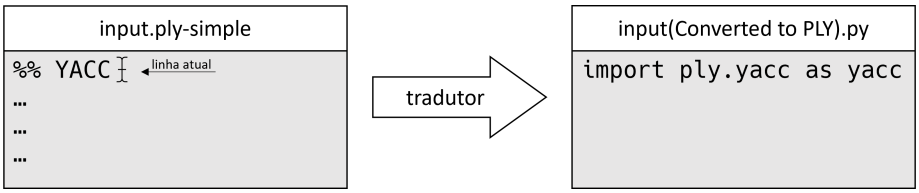

Figura 1: Conversão direta, linha a linha, de um ficheiro de *input* para um ficheiro de *output*.

No futuro, se fossem adicionados mais alvos de tradução para além de Python + PLY, seria possível definir uma tal representação intermediária. Isto permitiria reutilizar a *backend* do tradutor para os diferentes alvos de tradução, reduzindo otimisticamente o trabalho de desenvolvimento em metade.

5 Exemplos de utilização

Uma vez elaborado e explicado todo o projeto que foi planeado, desenvolvido e implementado pelo grupo de trabalho, convém demonstrar a aplicação do mesmo, com exemplos de ficheiros **PLY-simple** de entrada e a demonstração do respetivo ficheiro de saída **Python + PLY**.
Primeiramente iremos explicar o processo de como é executado o programa gerado através do projeto implementado. Assim, basta correr no terminal a invocação do programa da seguinte maneira:

```
$ python3 input.ply-simple
```

No final do qual é gerado o ficheiro de *output* com o nome (neste caso) `input(Converted to PLY).py`.

Por exemplo, o seguinte ficheiro de *input*

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

#? = "t"

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

lexer = lex.lex()

%% YACC

precedence = [('left', "+", "-" ), ('left', "*" , "/"), ('right', "
    UMINUS")]    # List of precedence

# symboltable : dictionary of variables
ts = { }

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 : < 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)

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

Listing 35: Ficheiro de *input* em PLY-simple.

é traduzido no seguinte ficheiro:

```
import ply.lex as lex

literals = "+-/*=()"  ## a single char
t_ignore = " \t\n"
tokens = [ "VAR", "NUMBER" ]

#? = "t"

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(
        f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]", t.
    lexer.skip(1)
    )

lexer = lex.lex()

import ply.yacc as yacc

precedence =  [
    ( "left", "+", "-" ),
    ('left', "*" , "/"),
    ('right', "UMINUS" )
] # List of precedence

# symboltable : dictionary of variables
ts = { }

def p_production0(t):
  '''
  stat : VAR "=" exp
  '''
  ts[t[1]] = t[3]

def p_production1(t):
  '''
  stat : exp
  '''
  print(t[1])

def p_production2(t):
  '''
  exp : exp "+" exp
  '''
  t[0] = t[1] + t[3]

def p_production3(t):
```

```
56    '''
57    exp : exp "-" exp
58    '''
59    t[0] = t[1] - t[3]
60
61  def p_production4(t):
62    '''
63    exp : exp "*" exp
64    '''
65    t[0] = t[1] * t[3]
66
67  def p_production5(t):
68    '''
69    exp : exp "/" exp
70    '''
71    t[0] = t[1] / t[3]
72
73  def p_production6(t):
74    '''
75    exp : "-" exp %prec UMINUS
76    '''
77    t[0] = -t[2]
78
79  def p_production7(t):
80    '''
81    exp : NUMBER
82    '''
83    t[0] = t[1]
84
85  def p_production8(t):
86    '''
87    exp : VAR
88    '''
89    t[0] = getval(t[1])
90
91  def p_error(t):
92    print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
93
94  def getval(n):
95    if n not in ts:
96          print(f"Undefined name '{n}'")
97    return ts.get(n, 0)
98
99  y=yacc.yacc()
100
101 y.parse("3+4*7")
```

Listing 36: Ficheiro de *output* em Python + PLY.

6 Conclusão

A resolução deste trabalho prático demonstrou ser bastante desafiante e uma mais-valia para o nosso conhecimento, pois foi possível consolidar a matéria lecionada ao longo das aulas teóricas e práticas, permitindo explorar um pouco mais sobre a biblioteca PLY da linguagem de programação Python.

Por outro lado, foi possível praticar e aprofundar o nosso conhecimento sobre gramáticas, regras de produção e regras sintáticas e estados de *lexing*. Além disso, o trabalho prático mostrou ser bastante crítico na vertente em que o mesmo exigiu ao grupo de trabalho todo um foco no planeamento relativamente ao modo como poderíamos ler e escrever os ficheiros pretendidos.

Por último, o exemplo fornecido no enunciado, que foi adaptado pelo grupo, mostrou ser uma mais valia, na medida em que o mesmo permitiu melhorar e refinar a implementação de todo o projeto desenvolvido.