Processamento de Linguagens (3º ano de LEI)

Trabalho Prático II

Relatório - Grupo 39

Henrique José Fernandes Alvelos (a93316)

Bohdan Malanka (a93300)

Diogo da Silva Rebelo (a93278)

Julho de 2022

Resumo

Este segundo trabalho prático no âmbito da unidade curricular de Processamento de Linguagens consistiu na elaboração de um tradutor de uma linguagem com recurso a gramáticas independentes de contexto e tradutoras. Com recurso às ferramenta yacc e lex do python, esta gramática deve ser capaz de gerar funções PLY a partir de uma sintaxe mais simples "PLY-simple". No presente relatório explicamos como desenvolvemos o tradutor para a linguagem e as diversas produções implementadas na gramática.

Conteúdo

Capítulo 1

Introdução

1.1 Enquadramento e Contextualização

Um gerador de parser é um algoritmo, um componente de *software* ou uma aplicação que gera o código-fonte de um analisador sintático, interpretador ou compilador de uma linguagem de programação. Geralmente ele é alimentado com a descrição sintática e semântica da linguagem independente de arquitetura, com uma descrição do conjunto de instruções da arquitetura independente de linguagem de programação.

Nesse sentido, gerador de parsers PLY, embora poderoso tem uma sintaxe um pouco complexa, portanto, criarmos um tradutor que a partir de uma sintaxe mais "limpa"PLY-simple, gere as funções PLY convenientes.

1.2 Problema e Objetivos

Em geral o projeto pretende aprofundar e cimentar conceitos já abordados nas aulas, tais como:

a) 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);

b) desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;

c) desenvolver um compilador gerando código para um objetivo específico;

d) 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.

1.3 Estrutura do documento

O presente relatório visa ilustrar o trabalho realizado. Para isso, estruturamos o relatório em diferentes capítulos:

O primeiro capítulo é a **Introdução**. Aqui serão abordados tópicos como o enquadramento e contextualização do tema proposto, o problema que se pretende resolver e o seu objetivo e também será exposto o modo de estruturação do relatório.

A seguir, no capítulo 2 o foco será na **Análise e Especificação** do problema, onde será efetuada uma descrição informal do problema seguida da especificação dos requisitos, que permitirá abordar em detalhe as especificações dos requisitos para uma gramática GIC e CT.

No terceiro capítulo apresentamos o **Concepção/desenho da Resolução**. De forma a exemplificar a solução obtida, esta secção está subdividida em cinco temas principais: Sintaxe PLY-Simple, Gramática com Produções, Analisador Léxico, Estrutura de Dados para o YACC, Analisador Sintático.

O capítulo 4 assenta na **Codificação e Testes**, que se subdivide em Compilação e Execução e Testes realizados e Resultados.

No capítulo 5 é efetuada uma **Conclusão** e análise crítica do trabalho efetuado, realçando aspetos positivos da implementação e aspetos a melhorar.

Por fim, existe também uma última secção **Apêndice A** onde está presente o **Código do Programa**.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Para um conjunto de instruções que representa uma sintaxe mais simple do módulo PLY de Puthon, pretende-se criar um compilador que traduza essa sintaxe (PLY-simple) em PLY. A partir da leitura de cada instrução do ficheiro de *input* é possível transforma-las em respetivas funções do **lex** e **yacc**. Nesse sentido, esse ficheiro é constituído por três conjuntos de instruções, que fazem parte respetivamente do *lex*, *yacc* e código *python*.

2.2 Especificação do Requisitos

Como forma de cumprir com o objetivo do problema apresentado é necessário analisar e especificar os dados e requisitos do projeto. Para isso, é fundamental ter em consideração as ferramentas fornecidas pelo *python*, tais como o analisador léxico **lex** e o analisador sintático **yacc**. Além disso, para a implementação do programa também foi necessário ter em consideração os conceitos de gramática independente de contexto(GIC) e gramática tradutora. Por fim, tivemos também em consideração estruturas de dados do *python* como os dicionários para a implementação da solução.

Capítulo 3

Concepção/desenho da Resolução

Dada por concluída a fase de análise e especificação, chegamos agora a etapa de implementação, ou por outras palavras, desenho da resolução. Nesta etapa é importante realçar tanto o trabalho desenvolvido no analisador léxico como no analisador sintático. No primeiro, falar em particular das suas especificações(tokens,literais, ignore,etc ...), e no segundo, das estruturas de dados utilizadas, da gramática que tem por base e suas produções, que consequentemente derivam em regras de tradução para o próprio PLY.

3.1 Sintaxe PLY-simple

Para a implementação do programa foi necessário ter em conta algumas particularidades da sintaxe da linguagem PLY-simple considerada. A nossa linguagem possui a mesma estrutura apresentada no enunciado, mas com algumas distinções. Nomeadamente a alteração das plicas de ´ para ' e correções no código, como por exemplo uma vírgula a mais na lista *precedence* e adição de um parêntese para fechar no *return*, no segmento do *lexer*.

Deste modo, segue se um exemplo que o grupo considerou para a tradução:

```
%% LEX
%literals = "+-/*=()" ## a single char
%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')
]
ts = { }
# symboltable : dictionary of variables
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)
y=yacc()
y.parse("3+4*7")
```

3.2 Gramática com produções

Para implementar o tradutor foi necessário criar uma gramática tradutora para conseguir interpretar a sintaxe referida anteriormente traduzindo-a em código *python* por diversas produções. A seguir encontra-se a gramática que será utilizada pelo tradutor criado.

```
p0:     S' -> program
p1:     program -> <empty>
p2:              | exp program

p3:     exp -> COMENTARIO

p4:     lista -> '[' listcont
p5:            | '[' ']'

p6:     listcont -> STRING ']'
p7:               | STRING , listcont
p8:               | tuplo ']'
p9:               | tuplo , listcont

p10:    tupcont -> STRING ')'
p11:              | STRING , tupcont
p12:              | tuplo ')'
p13:              | tuplo , tupcont

p14:    tuplo -> '(' tupcont
p15:           | '(' ')'
```

```
22
23  p16:    exp -> EXP GRAM TODO
24  p17:         | '%' '%' LEX
25  p18:         | '%' '%' YACC
26  p19:         | INITCODE
27  p20:         | PYCODE
28  p21:         | '%' LITERALS lista
29  p22:         | '%' LITERALS STRING
30  p23:         | '%' TOKENS lista
31  p24:         | '%' TOKENS STRING
32  p25:         | '%' LEXIGNORE STRING
33  p26:         | REGEX OPENPARENTESES STRING codigo
34  p27:         | REGEX OPENPARENTESES FSTRING codigo
35  p28:         | ERROR OPENPARENTESES STRING codigo
36  p29:         | ERROR OPENPARENTESES FSTRING codigo
37
38  p30:    codigo -> PYCODE CLOSEPARENTESES
39  p31:            | PYCODE OPENPARENTESES codigo CLOSEPARENTESES
40
41  p32:    exp -> '%' PRECEDENCE lista
42  p33:         | VARIAVEL = dict
43
44  p34:    dict -> '{' dictcont
45  p35:          | '{' '}'
46
47  p36:    dictcont -> STRING ':' STRING '}'
48  p37:              | STRING ':' STRING , dictcont
```

3.3 Analisador Léxico

Para a implementação do analisador léxico foi necessário analisar e ter em consideração os símbolos terminais que a linguagem irá reconhecer. Estes símbolos são denotados por uma lista de *tokens* ou *litarals* no **lex** conforme seja necessário, ou não, uma expressão regular mais complexa que não esteja implícita no próprio símbolo. Deste modo, na seguinte figura estão apresentados os símbolos literais considerados:

```
1  literals = ["(", ")", "[", "]", ":", "=", ",", "{", "}", "%", "\n"]
```

Analogamente, seguem os símbolso não literais (*tokens*):

```
1  tokens = ["LEX", "YACC", "LITERALS", "LEXIGNORE", "TOKENS", "STRING", "FSTRING", "ERROR", "REGEX",
2            "VARIAVEL","PRECEDENCE", "COMENTARIO", "GRAM", "TODO", "PYCODE",  "INITCODE", "EXP",
3            "OPENPARENTESES","CLOSEPARENTESES"]
```

Estados podem ajudar imenso para poder captar fragmentos de cada linha, daí que usamos 4 estados. Reparamos que, na secção do **Yacc**, cada linha pertencente à gramática podia ser dividida em 3 partes: **Expressão**, **Gramática** e **Código Python**. Conseguíamos ver que a Expressão acabava sempre em **":"**,

a Gramática em **"{"** e o código ia até ao fim da linha. Sendo assim, criamos dois estados (**GRAMATICA** e **TODOTHINGS**) que representam a ideia referida anteriormente. Quanto ao **CODE**, reparamos que, a partir da linha *%%*, começava o código em Python. Ao ser ativo, o Lexer capta toda a linha e assume como código. Por último, nas regras do Lexer, notamos que, depois da palavra *"return"*, o conteúdo estava entre dois parênteses. Por haver a possibilidade de haver mais parênteses dentro, implementamos o método de níveis: caso fosse **(**, aumentava de nível; caso fosse **)**, descia de nível. Quando o nível atingia 0, significava que o conteúdo do *"return"* já estava captado, voltando ao estado inicial

```
states = [("GRAMATICA", "exclusive"), ("TODOTHINGS", "exclusive"), ("CODE", "exclusive"),
          ("RETURN", "exclusive")]
```

3.4 Estrutura de Dados para o YACC

Para a implementação do programa foi necessário considerar 1 dicionário como estrutura de dados. Assim, o dicionário serviu para manter a numeração das funções da gramática.

3.5 Analisador Sintático

Com recurso ao analisador sintático, YACC, foi realizado o reconhecimento da gramática implementada. Assim, para cada produção reconhecida era necessário traduzi-la numa ação que dependerá de produção para produção. Esta ação, de grosso modo, será armazenada no *parser* do analisador sintático, quais as operações e funções em modo *string* serão armazenados no ficheiro de output passado como parâmetro ao programa. As respetivas produções em YACC seguem-se no código anexado em baixo.

Capítulo 4

Codificação e Testes

4.1 Compilação e Execução

O programa implementado este para complicar recebe como parâmetros o caminho do ficheiro de *input* e o caminho do ficheiro de *output*, da seguinte forma

python3 parser.py -i input.txt -o output.py

4.2 Testes realizados e Resultados

Teste realizado com o ficheiro de entrada subracitado gerou o segunte *output*:

```
%% LEX
%literals = "+-/*=()" ## a single char
%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')
]
# 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 : '(' 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)
y=yacc()
y.parse("3+4*7")
```

Assim, executando o ficheiro *python* traduzido optemos a seguinte solução:

```
31.0
Generating LALR tables

Process finished with exit code 0
```

Capítulo 5

Conclusão

Dada por coarguida a realização do trabalho prático, consideramos boa prática fazer uma apreciação crítica realçando não só os aspetos positivos como também as dificuldades que surgiram e o modo como estas foram colmatadas.

No que diz respeito aos pontos fortes, pensamos que o trabalho está bem concluído. Isto porque conseguimos traduzir o exemplo do enunciado para um ficheiro que é compilado com sucesso.

Durante a realização deste trabalho surgiram algumas dificuldades, dos quais destacamos a realização da gramática. Não foi um processo fácil porque voltamos muitas vezes à estaca zero visto que, as gramáticas anteriores chegavam a ser ambíguas.

Desta forma, pretendemos explicar que a realização deste projeto foi um aspeto essencial para aprimorar e melhor cimentar os conhecimentos sobre escrita de gramáticas independentes de contexto e tradutoras. Além disso, visto que grande parte das dificuldades foram ultrapassadas e o programa está operacional concluímos que o balanço do resultado foi positivo.

Apêndice A

Código do Programa

A.1 A.1 Programa LEX

```
from ply import lex

tokens = ["LEX", "YACC", "LITERALS", "LEXIGNORE", "TOKENS", "STRING", "FSTRING", "ERROR", "REGEX",
          "VARIAVEL", "PRECEDENCE", "COMENTARIO", "GRAM", "TODO", "PYCODE",  "INITCODE", "EXP",
          "OPENPARENTESES", "CLOSEPARENTESES"]
literals = ["(", ")", "[", "]", ":", "=", ",", "{", "}", "%", "\n"]
states = [("GRAMATICA", "exclusive"), ("TODOTHINGS", "exclusive"), ("CODE", "exclusive"),
          ("RETURN", "exclusive")]

t_ANY_ignore = ' \n\r\t'
t_CODE_ignore = '\n\r'


def t_INITIAL_LEX(t):
    r"""LEX"""
    return t


def t_INITIAL_YACC(t):
    r"""YACC"""
    return t


def t_INITIAL_LITERALS(t):
    r"""literals\ *\="""
    return t


def t_INITIAL_LEXIGNORE(t):
    r"""ignore\ *\="""
    return t

```

```
33
34  def t_INITIAL_TOKENS(t):
35      r"""tokens\ *\="""
36      return t
37
38
39  def t_INITIAL_RETURN_FSTRING(t):
40      r"""(f\'[^\']*\')|(f\"[^\"]*\")"""
41      return t
42
43
44  def t_INITIAL_RETURN_STRING(t):
45      r"""(\'[^\']*\')|(\"[^\"]*\")"""
46      return t
47
48
49  def t_INITIAL_REGEX(t):
50      r""".*\ return"""
51      t.lexer.begin("RETURN")
52      t.lexer.level = 0
53      return t
54
55
56  def t_RETURN_PYCODE(t):
57      r"""[^(^)]+"""
58      return t
59
60
61  def t_RETURN_OPENPARENTESES(t):
62      r"""\("""
63      t.lexer.level += 1
64      return t
65
66
67  def t_RETURN_CLOSEPARENTESES(t):
68      r"""\)"""
69      t.lexer.level -= 1
70      if t.lexer.level == 0:
71          t.lexer.begin("INITIAL")
72
73      return t
74
75
76  def t_INITIAL_ERROR(t):
77      r"""\.\ *error"""
78      t.lexer.begin("RETURN")
79      return t
80
81
```

```
82  def t_INITIAL_EXP(t):
83      r"""\w+\ *\:"""
84      t.lexer.begin("GRAMATICA")
85      return t
86
87
88  def t_INITIAL_PRECEDENCE(t):
89      r"""precedence\ *\="""
90      return t
91
92
93  def t_INITIAL_VARIAVEL(t):
94      r"""[\w\d]+"""
95      return t
96
97
98  def t_INITIAL_COMENTARIO(t):
99      r"""\#.*"""
100     return t
101
102
103 def t_GRAMATICA_GRAM(t):
104     r"""[^\]]*\{"""
105     t.lexer.begin("TODOTHINGS")
106     return t
107
108
109 def t_TODOTHINGS_TODO(t):
110     r"""[^\}]*\}"""
111     t.lexer.begin("INITIAL")
112     return t
113
114
115 def t_INITCODE(t):
116     r"""\%\%\n"""
117     t.lexer.begin("CODE")
118     return t
119
120
121 def t_CODE_PYCODE(t):
122     r""".+"""
123     return t
124
125
126 def t_ANY_error(t):
127     print(f"Illegal character '{t.value[0]}'")
128     t.lexer.skip(1)
129
130
```

```
lexer = lex.lex()
lexer.level = 0
lexer.begin("INITIAL")
```

A.2 A.2 Programa YACC

```
import re
import ply.yacc as yacc
import sys
from lexer import tokens, literals

f = None
dgram = {}

file = None
args = sys.argv[1:]

if len(args) == 2 and args[0] == "-i":
    try:
        file = open(f"input/{args[1]}", encoding="utf-8", mode="r")
        args[1].replace(".txt", ".py")
        f = open(f"output/{args[1]}", "w")
    except FileNotFoundError:
        print("Ficheiro não encontrado")
        exit()
elif len(args) == 4 and args[0] == "-i" and args[2] == "-o":
    try:
        file = open(f"input/{args[1]}", encoding="utf-8", mode="r")
        f = open(f"output/{args[3]}", "w")
    except FileNotFoundError:
        print("Ficheiro não encontrado")
        exit()

data = file.read()
file.close()


def p_program(p):
    """program : empty
               | program exp"""
    if len(p) > 2:
        f.write(p[2])


def p_empty(p):
    """empty : """
```

```
41
42
43  def p_exp_comentario(p):
44      """exp : COMENTARIO"""
45      p[0] = p[1] + "\n"
46
47
48  def p_lista(p):
49      """lista : '[' listcont
50               | '[' ']'"""
51      p[0] = p[1] + p[2]
52
53
54  def p_tuplo(p):
55      """tuplo : '(' tupcont
56               | '(' ')'"""
57      p[0] = p[1] + p[2]
58
59
60  def p_dict(p):
61      """dict : '{' dictcont
62                  | '{' '}'"""
63      p[0] = p[1] + p[2]
64
65
66  def p_listcont(p):
67      """listcont : STRING ']'
68                  | STRING ',' listcont
69                  | tuplo ']'
70                  | tuplo ',' listcont"""
71
72      p[0] = p[1] + p[2] + p[3] if len(p) == 4 else p[1] + p[2]
73
74
75  def p_tupcont(p):
76      """tupcont : STRING ')'
77                  | STRING ',' tupcont
78                  | tuplo ')'
79                  | tuplo ',' tupcont"""
80      p[0] = p[1] + p[2] + p[3] if len(p) == 4 else p[1] + p[2]
81
82
83  def p_dictcont(p):
84      """dictcont : STRING ':' STRING '}'
85                  | STRING ':' STRING ',' dictcont"""
86      p[0] = p[1] + p[2] + p[3] + p[4] + p[5] if len(p) == 6 else p[1] + p[2] + p[3] + p[4]
87
88
89  def p_exp_yaccline(p):
```

```
    """exp : EXP GRAM TODO"""
    p[1] = p[1][:-2]
    global dgram
    if not p[1] in dgram.keys():
        dgram[p[1]] = 0
    else:
        dgram[p[1]] += 1
    p[2] = p[2][:-1]
    p[3] = p[3][:-1]
    p[0] = f"def p_{p[1]}_{dgram[p[1]]}(t):\n\tr\"{p[1]} : {p[2]}\"\n\t{p[3]}\n"


def p_exp_lex(p):
    """exp : '%' '%' LEX"""
    p[0] = "from ply import lex" + "\n"


def p_exp_yacc(p):
    """exp : '%' '%' YACC"""
    p[0] = "\nlex = lex.lex()\nfrom ply.yacc import yacc" + "\n"


def p_exp_initcode(p):
    """exp : INITCODE"""
    p[0] = "#code" + "\n"


def p_exp_pycode(p):
    """exp : PYCODE"""
    p[0] = p[1] + "\n"


def p_exp_lexvars(p):
    """exp : '%' LITERALS lista
            | '%' LITERALS STRING
            | '%' TOKENS lista
            | '%' TOKENS STRING
            | '%' LEXIGNORE STRING"""
    p[0] = p[2] + p[3] + "\n"


def p_exp_lexRules(p):
    """exp : REGEX OPENPARENTESES STRING codigo
            | REGEX OPENPARENTESES FSTRING codigo"""
    p[0] = f"def t_{p[3][1:-1]}(t):\n\tr\"{p[1][:-7]}\"\n\tt.value={p[4][2:-1]}\n\treturn t" + "\n"


def p_exp_lexError(p):
    """exp : ERROR OPENPARENTESES STRING codigo
```

```
            | ERROR OPENPARENTESES FSTRING codigo"""
    match = re.match(r"[^, ]*,(.+)\)", p[4])
    codigo = None
    if match:
        codigo = match.group(1)
    if codigo:
        p[0] = f"def t_error(t):\n\tprint({p[3]})\n\t{codigo}\n"
    else:
        p[0] = f"def t_error(t):\n\tprint({p[3]})\n"


def p_codigo(p):
    """codigo : PYCODE CLOSEPARENTESES
                | PYCODE OPENPARENTESES codigo CLOSEPARENTESES"""
    p[0] = p[1] + p[2] + p[3] + p[4] if len(p) == 5 else p[1] + p[2]


def p_exp_precedence(p):
    """exp : '%' PRECEDENCE lista"""
    p[0] = f"{p[2]} {p[3]}\n"


def p_exp_variavelAtrib(p):
    """exp : VARIAVEL '=' dict"""
    p[0] = f"{p[1]} {p[2]} {p[3]}\n"


# Error rule for syntax errors
def p_error(p):
    print("Erro de sintaxe: ", p)
    parser.success = False


# Build the parser
parser = yacc.yacc()
parser.success = True

res = parser.parse(data)

if parser.success:
    print("Conversão efetuada!")
f.close()
```