



UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

Processamento de linguagens - Trabalho Prático #2  
Ano Letivo 2021/2022

Gonçalo Pereira (a93168)

Tiago Silva (a93277)

22 de setembro de 2022

## 1 Introdução

O presente relatório refere-se à realização do trabalho prático 2, cujo o objetivo era de um total de quatro problemas escolher um e proceder à implementação de uma solução. O enunciado escolhido "Tradutor PLY-simple para PLY" que, resumidamente, requer a criação de uma linguagem mais "limpa" e simplificada inspirada no PLY, chamada PLY-simple, deduzir um esquema de tradução e criar um compilador que traduza esta linguagem em PLY. Para isso utilizamos 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.

## 2 Sintaxe da linguagem Ply-simple

Começamos por definir a sintaxe a usar na nova linguagem Ply-simple. Embora tenhamos recebido uma sugestão de uma sintaxe possível, esta serviu apenas para inspiração já que a versão final da nossa sintaxe de Ply-simple tem algumas diferenças. Dividindo a sintaxe em dois (Lexer e Yacc) começaremos por mostrar cada uma das partes em mais detalhe.

## 2.1 Lexer

Exemplo:

```
1  LEX:
2  literals = "+-*=()"
3  ignore   = " \t\n "
4  tokens = ['VAR','NUMBER']
5
6  lfunc: [a-zA-Z\_][a-zA-Z0-9\_]* : return('VAR', t.value)
7  lfunc: \d+(\.\d+)? : return('NUMBER', float(t.value))
8  lerror(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]",
9  t.lexer.skip(1) )
```

Inicialmente no lexer caso pretendido pode-se, não sendo obrigatório, definir os literals e a regra de ignore, por outro lado, tokens tem que estar sempre definido. Além disso a definição das funções do lexer seguem a seguinte estrutura: *lfunc*: para inciar uma função do lexer, seguido de a expressão regular que pretende usar, seguido de *:* que marca o fim a expressão regular e inicia o *return* que contém o nome da função e do valor a retornar.

## 2.2 Yacc

Exemplo:

```
1  YACC:
2
3  precedend = [
4      ('left','+', '-'),
5      ('left','*', '\'),
6      ('right','UMINUS'),
7  ]
8
9  ts = {}
10
11  grammar:
12  stat : VAR '=' exp { ts[p[1]] = p[3] }
13  stat : exp { print(p[1]) }
14  exp : exp '+' exp { p[0] = p[1] + p[3] }
15  exp : exp '-' exp { p[0] = p[1] - p[3] }
16  exp : exp '*' exp { p[0] = p[1] * p[3] }
17  exp : exp '\' exp { p[0] = p[1] / p[3] }
18  exp : '-' exp %prec UMINUS { p[0] = -p[2] }
19  exp : '(' exp ')' { p[0] = p[2] }
20  exp : NUMBER { p[0] = p[1] }
21  exp : VAR { p[0] = getval(p[1]) }
22
23  yfuncs:
24  def p_error(t){
25      print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
26  }
27  def getval(n){
28      if n not in ts: print(f"Undefined name '{n}'")
29      return ts.get(n,0)
30  }
```

Na definição do Yacc é possível definir precedência como pode ser visto acima, de forma semelhante, pode-se iniciar variáveis de estado que podem ser listas, dicionários, tuplos, strings e valores numéricos. Depois chega à parte principal que é a da definição da gramática. A palavra-chave *grammar*: marca o início da definição da gramática. A gramática segue uma estrutura comum, em que cada produção é caracterizada por *Simbolos não terminais*: *Conjunto de símbolos terminais ou não terminais* seguida de *Ações semânticas*. Por fim usando *yfunc*: inicia-se a definição das funções do Yacc tal como se mostra no exemplo.

## 3 Solução implementada

### 3.1 Analisador léxico

#### 3.1.1 Tokens

Na imagem a seguir podemos ver os tokens que foram definidos e as suas respectivas expressões regulares.

LEX -> r'LEX:'	PA -> r'\'
LTS -> r'literals'	PF -> r'\'
IG -> r'ignore'	PCA -> r'['
TKS -> r'tokens'	PCF -> r']'
LFUNC -> r'lfunc'	PRA -> r'['
RT -> r'return'	PRF -> r']'
ER -> r'ererror\'	DS -> r':'
TVAL -> r't.value'	asval -> r'"((\\") (['^0-9n"])+)"'
TYPE -> r'(float) (int) {str}'	pelval -> r'"((\\") (['^0-9n"])+)"'
YACC -> r'YACC:'	cod -> r'(((["^+"]) (\\["^v]+)))((['^0-9\\V\\W\\]])+)'
PRCD -> r'precedent'	id -> r"[A-Za-z]+"
LTRG -> r'('left' 'right')'	nt -> r"[A-Za-z]+"
GRM -> r'grammar:'	symbol -> r'"([A-Za-z]+)(' \"')"
PREC -> r'%prec'	name -> r'[A-Za-z]w**'
YFUNCS -> r'yfuncs'	num -> r'(\'+ -)?d+(\.\d+)?'
DEF -> r'def'	rgx -> r'((\\) [.:])+'

\*formato python raw string

## Tokens

De seguida necessitamos de definir estados, e por isso, segue uma imagem com todos os estados definidos e os todos tokens associados a eles.

**INITIAL:**  
LTS IG TKS aspval  
pelval LFUNC RT  
PA PF ER YACC  
PRCD PRA PRF  
LTRG PCA PCF  
id num GRM DEF  
YFUNCS

**SYMBOLS:**  
symbol PREC PCA

**RGX:**  
rgx DS

**GRAMMAR:**  
nt DS

**YFUNCS:**  
name PA PF PCA

**CODE:**  
cod PA PF PCA  
PCF PRA PRF

## Estados

Por fim, temos uma imagem que nos ajuda a ver os tokens resultantes de um input exemplo, e que inclusive, ajuda a perceber como a gramática foi construída.

## PLY-SIMPLE example:

```
LEX:
literals = "+-/*=0"
ignore = "\t\n"
tokens = [ 'VAR', 'NUMBER' ]

lfunc: [a-zA-Z][a-zA-Z0-9_]; return( 'VAR', t.value )
lfunc: [0-9]+(\.[0-9]+)?; return( 'NUMBER', float(t.value) )
error: [^]; print("Illegal character '{t.value[0]}' at line {t.lex.lineno}"); t.lex.skip(1)
YACC:

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

ts = {}
td = [5, 'ola', 'adeus']

grammar:
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] = ts[t[1]] }

yfuncs:
def p_error(t):
    print("Syntax error at '{t.value}' at line {t.lex.lineno}")
def getval(n):
    if n not in ts: print("Undefined name '{n}'")
    return ts.get(n, 0)
```

CAPTION (LEXER STATES):

INITIAL	SYMBOLS	YFUNC
CODE	REGEX	GRAMMAR

## RESULT TOKENS:

```
LEX
LTS = aspvall
IG = aspvall
TKS = [ pelval, pelval ]

LFUNC rgx: RT ( pelval, TVAL )
LFUNC rgx: RT ( pelval, TYPE ( TVAL ) )
ER cod { cod { cod } } cod { { cod } } cod { cod }
YACC

PRCD = [
    (LTRG, pelval, pelval),
    (LTRG, pelval, pelval),
    (LTRG, pelval, )
]

id = {}
id = [ num, aspvall, pelval ]

GRM
nt : symbol symbol symbol { cod { cod { cod } } cod { cod } }
nt : symbol { cod { cod { cod } } }
nt : symbol symbol symbol { cod { cod } cod { cod } cod { cod } }
nt : symbol symbol symbol { cod { cod } cod { cod } cod { cod } }
nt : symbol symbol symbol { cod { cod } cod { cod } cod { cod } }
nt : symbol symbol symbol { cod { cod } cod { cod } cod { cod } }
nt : symbol symbol PREC symbol { cod { cod } cod { cod } }
nt : symbol symbol symbol { cod { cod } cod { cod } }
nt : symbol { cod { cod } cod { cod } }
nt : symbol { cod { cod } cod { cod } }

YFUNCS
DEF name ( name ) {
    cod { cod { cod } cod { { cod } } }
DEF name ( name ) {
    cod { cod { cod } cod }
    cod { cod } }
```

## Ply-simple example

### 3.2 Analisador semântico

#### 3.2.1 Gramática

Gramática definida:

Ply : Lexer Yc

Lexer : LEX Literals Ignore Tokens Lfuncs Lerror

Yc : YACC Precedents Declaration Grammar Yfs

Literals : LTS '=' aspvall

Literals :

Ignore : IG '=' aspvall

Ignore :

Tokens : TKS '=' PRA Tokl PRF

Tokl : Tokl ',' pelval

Tokl : pelval  
 Lfuncs : Lfuncs Lfunc  
 Lfuncs :  
 Lfunc : LFUNC rgx DS RT PA pelval ‘,’ Tv PF  
 Tv : TVAL  
 Tv : TYPE PA TVAL PF  
 Lerror : ER Codes PF  
 Lerror :  
  
 Precedents : PRCD ‘=’ PRA Predlist PRF  
 Precedents :  
 Declarations : Declarations Declaration  
 Declarations :  
 Declaration : id ‘=’ Values  
 Grammar : GRM Productions  
 Yfs : YFUNCS Funcs  
  
 Productions : Productions Production  
 Productions :  
 Production : nt DS Symbols PCA Codes PCF  
 Symbols : Symbols S  
 Symbols :  
 S : symbol  
 S : PREC symbol  
  
 Predlist : Predlist PA LTRG ‘,’ Pelvals pelval PF ‘,’  
 Predlist :  
 Pelvals : Pelvals pelval ‘,’  
 Pelvals :  
 Funcs : Funcs Func  
 Funcs :  
 Func : DEF name PA name PF PCA Codes PCF  
 Codes : Codes Code  
 Codes :  
 Code : cod  
 Code : PA Codes PF  
 Code : PRA Codes PRF  
 Code : PCA Codes PCF  
 Values : Values ‘,’ Type  
 Values : Type  
 Type : num  
 Type : aspval  
 Type : pelval  
 Type : PRA Cont PRF  
 Type : PA Cont PF  
 Type : PCA Cont PCF  
 Cont : Values  
 Cont :

## 4 Verificação de erros

Uma importante característica de um gerador de código é a capacidade de detetar erros e de informar o utilizador dos mesmos. Contudo só implementamos duas verificações, uma que verifica tokens definidos mas não usados e outra que verifica se estão a ser utilizados símbolos não terminais que não foram definidos. Para a implementação destas verificações são usadas variáveis de estado *tt* - tabela de terminais, *tntDef* - tabela de não terminais definidos e *tntUsed* - tabela de não terminais utilizados. Em *tt* são guardados todos os tokens e literais definidos, em *tntDef* são guardados todos os não terminais definidos (lado esquerdo de uma produção) e um campo que contabiliza quantos produções têm um dado não terminal, por fim em *tntUsed* guarda-se todos os não terminais (usados no lado direito de uma produção). A primeira verificação traduz-se num *warning* para o utilizador enquanto que a segunda traduz-se num *error* não havendo geração de código.

## 5 Exemplo de Utilização

Segue-se um exemplo de linguagem PLY-Simple usada para definir S expressions e o código PLY gerado.

```
1  LEX:
2
3  literals = "()+-"
4  ignore = "\t\n "
5  tokens = ['num']
6
7  lfunc: \d+ : return('num',int(t.value))
8
9  YACC:
10
11  grammar:
12  Z : Sexp {print(p[1])}
13  Sexp : '(' '+' Lista ')' {p[0] = somatorio(p[3])}
14  Sexp : '(' '*' Lista ')' {p[0] = produtorio(p[3])}
15  Sexp : num {p[0] = p[1]}
16  Lista : Lista Sexp {p[0] = p[1] + [p[2]]}
17  Lista : Sexp Sexp {p[0] = [p[1],p[2]] }
18
19  yfuncs:
20  def somatorio(lista){
21      res = 0
22      for n in lista:
23          res += n
24      return res
25  }
26  def produtorio(lista){
27      res = 1
28      for n in lista:
29          res *= n
30      return res
31  }
```

```

1  import ply.lex as lex
2  literals = ['(', ')', '+', '*', '-']
3  t_ignore = "\t\n "
4  tokens = ['num']
5
6  def t_num(t):
7      r'\d+ '
8      t.value = int(t.value)
9      return t
10 def t_ANY_error(t):
11     print(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
12     t.lexer.skip(1)
13
14 lexer = lex.lex()
15
16 import ply.yacc as yacc
17
18 def p_Z(p):
19     "Z : Sexp "
20     print(p[1])
21
22 def p_Sexp(p):
23     "Sexp : '(' '+' Lista ')' "
24     p[0] = somatorio(p[3])
25
26 def p_Sexp_1(p):
27     "Sexp : '(' '*' Lista ')' "
28     p[0] = produtorio(p[3])
29
30 def p_Sexp_2(p):
31     "Sexp : num "
32     p[0] = p[1]
33
34 def p_Lista(p):
35     "Lista : Lista Sexp "
36     p[0] = p[1] + p[2]
37
38 def p_Lista_1(p):
39     "Lista : Sexp Sexp "
40     p[0] = [p[1], p[2]]
41
42 def somatorio(lista):
43     res = 0
44     for n in lista:
45         res += n
46     return res
47
48 def produtorio(lista):
49     res = 1
50     for n in lista:
51         res *= n
52     return res
53
54 parser = yacc.yacc()

```



## 6 Conclusão

Com este trabalho sentimos que alcançamos os principais objetivos propostos tais como • 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; • 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. Tendo em conta ao que nos foi proposto acreditamos ter feito um bom trabalho, embora tenhamos consciência que a nossa aplicação ficou longe de ser perfeita, uma vez que não implementamos os estados do ply e fazemos poucas verificações a nível do código em Python.