TP2 - Tradutor PLY-simple para PLY

Trabalho realizado por:

Francisco Alves Andrade
A89513
Jaime Abreu Fernandes de Oliveira
A89598

Grupo 64
Processamento de Linguagens 2021/2022
Universidade do Minho

Conteúdo

1 Resumo

No âmbito da Unidade Curricular de Processamento de Linguagens e perante a escolha de quatro enunciados distintos, foi nos proposto, de modo a desenvolver os conhecimentos acerca dos conteúdos lecionados nas aulas práticas, o desenvolvimento de um tradutor que transforme um programa escrito numa linguagem simplificada do PLY num programa escrito em PLY. Deste modo, o objetivo deste trabalho pratico passa por aprofundar os conhecimentos relativos à Unidade Curricular em causa.

2 Introdução

Uma vez que o grupo frequenta o 3º ano do Mestrado Integrado de Engenharia Informática, no âmbito da Unidade Curricular de Processamento de Linguagens, foi necessário aprofundar os nossos conhecimentos na area.

Este trabalho prático tem como principais objetivos, o aumento da nossa experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT).

Para além disso, é proposto o desenvolvimento de processadores de linguagens segundo o método da tradução dirigida pela sintaxe a partir de uma gramática tradutora, e também a utilização de geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão do *PLY* do Python, completado pelo gerador de analisadores léxicos Lex, também versão *PLY* do *Python*.

Ao longo deste relatório, iremos apresentar todas as etapas e decisões tomadas durante o processo de elaboração deste projeto.

3 Explicação da linguagem definida e abordagem ao problema

3.1 Decisões e desenho da linguagem/gramática

Antes de definirmos a nossa gramática tivemos de tomar algumas decisões para que a mesma suportasse as funcionalidades pretendidas. Definimos então que o corpo do programa (body) seria constituíddo por três partes seguindo a ordem do enunciado fornecido. O primeiro bloco está destinado ao campo léxico (*lex*), o segundo à gramática (*yacc*) e o terceiro engloba o código em *Python* apresentado no fim do enunciado. Dentro dos primeiro dois blocos há, ainda, uma subdivisão nos seus vários componentes e na ultima componente que decidimos denominar por "regras".

```
[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) )
```

Figura 1: Regras do *lex* (retirado do enunciado)

```
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]) }
```

Figura 2: Regras do *yacc* (retirado do enunciado)

3.1.1 Abordagem ao Problema

Em primeiro lugar, foi necessária a definição da GIC, para especificar a linguagem pretendida, tendo definido os símbolos terminais e as produções necessárias.

No desenvolvimento desta gramática tivemos em consideração todas as funcionalidades que pretendíamos que a nossa linguagem suportasse. Para além disso, tivemos especial atenção em formar um estilo esteticamente apresentável e legível, para um utilizador comum.

De seguida é apresentado o código relativo ao mesmo, e sobre o qual assenta a definição.

```
tokens = ['LEX_T',
          'YACC_T',
          'LEFT',
          'RIGHT',
          'TS',
          'CHAVS',
          'FIM',
          'NEG',
          'UMINUS',
          'TVALUE',
          'QUOTES',
          'PRINT',
          'TVAR',
          'TLSKIP',
          'PERCENTAGEM',
          'PARRETOA',
          'PARRETOF',
          'ASPAS',
          'PRECEDENCE_T',
          'PLICA',
          'LITERALS_T',
          'IGNORE_T',
          'TOKENS_T',
          'GETVAL',
          'PALMA',
          'PALMI',
          'SPECIAL',
          'REGEX',
          'RETURN',
          'ERROR']
```

Figura 3: Tokens

```
literals = ['+','-','*','/', '%', '(', ')', ',', '.', '=', ':', '@']
```

Figura 4: Literals

Tokens e Literals

3.1.2 Definição de Tokens

```
def t_LEX_T(t):
    r'LEX'
    t.value = str(t.value)
    return t

def t_YACC_T(t):
    r'YACC'
    t.value = str(t.value)
    return t

def t_TVALUE(t):
    r',.*?t\.value.*?\)*'
    t.value = str(t.value)
    return t

def t_CHAVS(t):
    r'\{.*?\}'
    t.value = str(t.value)
    return t

def t_PRECEDENCE_T(t):
    r'precedence\ *'
    t.value = str(t.value)
    return t

def t_QUOTES(t):
    r'"\w.+"'
    t.value = str(t.value)
    return t

def t_ASPAS(t):
    r'"'
    t.value = str(t.value)
    return t

def t_PARRETOA(t):
    r'\['
    t.value = str(t.value)
    return t
```

```
def t_PARRETOF(t):
    r'\]'
    t.value = str(t.value)
    return t

def t_LEFT(t):
    r'left'
    t.value = str(t.value)
    return t

def t_RIGHT(t):
    r'right'
    t.value = str(t.value)
    return t

def t_NEG(t):
    r'neg\ *'
    t.value = str(t.value)
    return t

def t_UMINUS(t):
    r'UMINUS\ *'
    t.value = str(t.value)
    return t


def t_FIM(t):
    r'~~(.| |\n|\t)+?~~'
    t.value = str(t.value)
    return t

def t_PRINT(t):
    r'print'
    t.value = str(t.value)
    return t

def t_GETVAL(t):
    r'getval'
    t.value = str(t.value)
    return t
```

Figura 5: Tokens (1)

```python
def t_LITERALS_T(t):
    r'literals\ *'
    t.value = str(t.value)
    return t

def t_IGNORE_T(t):
    r'ignore\ *'
    t.value = str(t.value)
    return t

def t_TOKENS_T(t):
    r'tokens\ *'
    t.value = str(t.value)
    return t

def t_PLICA(t):
    r"'"
    t.value = str(t.value)
    return t

def t_PERCENTAGEM(t):
    r"%"
    t.value = str(t.value)
    return t

def t_TS(t):
    r'ts'
    t.value = str(t.value)
    return t

def t_TVAR(t):
    r't\[\d\]'
    t.value = str(t.value)
    return t

def t_TLSKIP(t):
    r't\.lexer\.skip\(1\)'
    t.value = str(t.value)
    return t
```

```python
def t_RETURN(t):
    r'return'
    t.value = str(t.value)
    return t

def t_ERROR(t): #erro da gramatica
    r'error'
    t.value = str(t.value)
    return t

def t_PALMA(t): #PALMA -> palavra maiuscula
    r'[A-Z]+'
    t.value = str(t.value)
    return t

def t_PALMI(t): #PALMI -> palavra minuscula
    r'[a-z]+'
    t.value = str(t.value)
    return t

def t_SPECIAL(t):  #\t \n ou espaço
    r'(\\[tn]|\ )'
    t.value = str(t.value)
    return t

def t_REGEX(t):
    r'@.+@'
    t.value = str(t.value)
    return t


def t_error(t): #erro do programa
    print('Illegal character: '+t.value[0])
    t.lexer.skip(1)
```

Figura 6: Tokens (2)

3.1.3 Definição da Linguagem Tradutora (Produções)

Uma vez definida a gramática, foram definidas as produções necessárias para a implementação da linguagem pretendida. De seguida estão apresentadas todas as produções definidas, tanto as respetivas à secção *lex*, como as respetivas à secção *yacc*:

```
def p_body(p):
    "body : lex yacc funcoes"
    p[0] = p[1] + "\n" + p[2] +"\n" + p[3]
```

Figura 7: Body

```
def p_lex(p):
    "lex : PERCENTAGEM PERCENTAGEM LEX_T literals ignore tokens regras_l "
    p[0] = "import ply.lex as lex\n\n" +  p[6] + p[4] + p[5] + p[7] + "\nlexer = lex.lex()"

def p_literals(p):
    "literals : PERCENTAGEM LITERALS_T '=' ASPAS simbolos ASPAS"
    aux = "literals = ["
    for i in range(0,len(p[5])-1):
        aux += "'" + p[5][i] + "'" +","
    aux += "'" + p[5][len(p[5])-1] + "'" + "]"
    p[0] = aux

def p_simbolos(p):
    "simbolos : simbolos simbolo"
    p[0] = p[1]+p[2]

def p_simbolo(p):
    """simbolo : '+'
               | '-'
               | '*'
               | '/'
               | '='
               | '('
               | ')'
    """
    p[0] = p[1]

def p_simbolos_vazio(p):
    "simbolos : "
    p[0] = ""

def p_ignore(p):
    "ignore : PERCENTAGEM IGNORE_T '=' ASPAS especiais ASPAS"
    p[0] = "\nt_ignore = " + p[4] + " " + p[5] + p[6] + "\n"

def p_especiais(p):
    "especiais : especiais SPECIAL"
    p[0] = p[1] + p[2]
```

Figura 8: Lex (1)

```
def p_especiais_vazio(p):
    "especiais : "
    p[0] = ""

def p_tokens(p):
    "tokens : PERCENTAGEM TOKENS_T '=' PARRETOA listatokens PARRETOF"
    p[0] = f"tokens = {p.parser.tokens}\n"

def p_tokens_vazio(p):
    "tokens : "

def p_listatokens_simples(p):
    "listatokens : PLICA PALMA PLICA "
    p.parser.tokens.append(p[2])

def p_listatokens_recursivo(p):
    "listatokens : listatokens ',' PLICA PALMA PLICA"
    p.parser.tokens.append(p[4])


def p_regras_l(p):
    "regras_l : regras_l regra_l"
    p[0] = p[1] + "\n" + p[2]

def p_regras_l_vazio(p):
    "regras_l : "
    p[0] = ""

def p_regra_l_return(p):
    "regra_l : REGEX RETURN '(' PLICA PALMA PLICA TVALUE  "
    aux = "def t_"+p[5]+"(t):\n"
    aux += "\tr'"+p[1][1:len(p[1])-1]+"'\n"
    aux += "\treturn "+p[7][1:len(p[7])-1]+"\n\n"
    p[0] = aux

def p_regra_error(p):
    "regra_l : REGEX ERROR '(' PALMI QUOTES ',' TLSKIP ')'"
    aux = "def t_error(t):\n"
    aux += "\tprint("+p[4] + p[5]+")\n"
    aux += f"\t{p[7]}\n\n"
    p[0] = aux
```

Figura 9: Lex (2)

```
def p_yacc(p):
    "yacc : PERCENTAGEM PERCENTAGEM YACC_T precedence symboltable regras_y"
    p[0] = p[6]

def p_palavra(p):
    """palavra : PALMI
               | PALMA
    """
    p[0] = p[1]

def p_regras_y(p):
    "regras_y : regras_y regra_y"
    p[0] = p[1] + "\n" +p[2]

def p_regras_y_vazio(p):
    "regras_y : "
    p[0] = ""


def p_precedence(p):
    "precedence : PERCENTAGEM PRECEDENCE_T '=' PARRETOA conteudo_prec PARRETOF"

def p_conteudo_prec(p):
    "conteudo_prec : conteudo_prec tuplo"

def p_conteudo_prec_vazio(p):
    "conteudo_prec : "

def p_tuplo(p):
    "tuplo : '(' lado ',' PLICA simbolo PLICA ',' PLICA simbolo PLICA ')'"

def p_tuplo_2(p):
    "tuplo : '(' lado ',' PLICA UMINUS PLICA ')'"

def p_lado_l(p):
    "lado : PLICA LEFT PLICA"

def p_lado_r(p):
    "lado : PLICA RIGHT PLICA"

def p_symboltable(p):
    "symboltable : TS '=' CHAVS"
```

Figura 10: Yacc (1)

```
def p_regra_y(p):
    "regra_y : PALMI ':' termo CHAVS"
    if p[1] not in p.parser.gramaticas:
        p.parser.gramaticas[p[1]]=0
    else:
        p.parser.gramaticas[p[1]]+=1
    i=p.parser.gramaticas[p[1]]
    if i==0:
        p[0] = f"def p_{p[1]}(t): \n"
    else:
        p[0] = f"def p_{p[1]}_{i}(t): \n"

    p[0] += f'\t"{p[1]} : {p[3]}" \n'
    p[0] += f'\t{p[4][1:len(p[4])-1].strip()} \n'

def p_simbolo_operacao(p):
    """simbolo_operacao : '+'
                        | '-'
                        | '*'
                        | '/'
                        | '='
    """
    p[0] = p[1]

def p_termo_fators(p):
    "termo : fator"
    p[0] = p[1]

def p_termo(p):
    "termo : termo PLICA simbolo_operacao PLICA fator"
    p[0] = p[1] + " " + "'" + p[3] + "'" + " " + p[5]


def p_fator(p):
    "fator : palavra "
    p[0] = p[1]

def p_fator_termo(p):
    "fator : PLICA '(' PLICA termo PLICA ')' PLICA"
    p[0] = p[4]


def p_funcoes(p):
    "funcoes : FIM"
    p[0] = p[1][2:len(p[1])-2]

def p_error(p):
    print("Syntax error! ",p)
    parser.success = False
```

Figura 11: Yacc (2)

3.1.4 Estruturas Auxiliares do Parser

- **parser.tokens**: dicionário para guardar os *tokens* apresentados no input fornecido.
- **parser.gramaticas**: dicionario para guardar as variáveis apresentadas na secção do yacc.

3.2 Input Teste

Como input de teste utilizamos o ficheiro disponibilizado no enunciado. Este era suscetível a alterações, sendo que a nossa única alteração foi a adição de dois delimitadores na última secção de código *Python*, onde acrescentamos "~~"ao início e ao fim da mesma, com vista a facilitar a sua identificação e previnir conflitos com tokens.

```
%%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')
]
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 ')' {  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")
~~
```

Figura 12: Yacc (2)

4 Conclusão

Tendo sido finalizada toda a implementação do projeto, consideramos que o trabalho proposto se demonstrou um excelente desafio, mas também extremamente recompensador, no sentido em que fomos capazes de criar programas com uma linguagem "inventada"por nós, com todas as funcionalidades advindas da mesma bem definidas e operacionais.

De um ponto de vista lúdico, foi bastante importante para reforçar os conteúdos abordados, quer nas aulas práticas quer nas teóricas. Houve algumas funcionalidades que gostaríamos de ter aprimorado, como por exemplo um controlo de erros mais avançado e aumentado a pilha de testes, mas ainda assim sentimos que temos um trabalho sólido.

Em suma, sentimos que a nossa solução para este problema se demonstra bastante capaz e eficaz e que nos revelamos aptos a superar o desafio proposto.