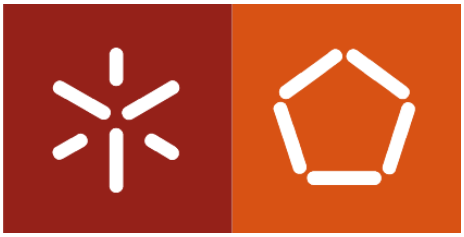

Universidade do Minho

Processamento de Linguagens

Trabalho em Grupo - TP2

Tradutor PLY-simple para PLY

Pedro Aquino Martins de Araújo - A90614
Raul Arieira Parente - A88321
Sergio Díaz Mediavilla - E10271

Braga, Portugal
21 de maio de 2022

Índice

1 Introdução

Assim como o ser humano, as máquinas possuem capacidade de realizar a análise sintática de texto, decompondo um conjunto de dados de entrada em unidades estruturais, a partir de uma gramática tradutora. Este método, também conhecido como parsing, é precedido de uma análise léxica, na qual uma sequência de carateres é convertida numa sequência de tokens.

Neste projeto iremos, com o auxílio da linguagem de programação Python e do módulo ply, desenvolver um programa capaz de fazer a análise léxica e sintática de uma linguagem imperativa desenvolvida por nós.

2 Desenvolvimento

2.1 Descrição do Trabalho

Para a realização deste projeto foi proposto realizar um tradutor do PLY-Simples para PLY.

Considerando uma versão do PLY-Simple, ou seja, uma versão mais limpa que executa as funções PLY convenientes, temos que criar uma versão análoga em PLY. Esta versão análoga executará o programa especificado em PLY-Simple, para isso temos que deduzir um esquema de tradução entre as duas versões e implementar um compilador(Lexer e Parser) que realize essa tradução.

2.2 PlySimple

Para realizar este exercício, temos que usar uma sintaxe PLY-simples, a partir da qual gerar uma versão análoga em PLY. A sintaxe usada para PLY-simples é a seguinte:

```
%% LEX

literals = §+-=/*()§
tokens = ['VAR', 'NUMBER']
ignore = ' \t\n'

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))

%% YACC

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

atrib -> stat : VAR '=' exp => { t.y.ts[t[1]] = t[3] }
statExp -> stat : exp => { print(t[1]) }
plus -> exp : exp '+' exp => { t[0] = t[1] + t[3] }
minus -> exp : exp '-' exp => { t[0] = t[1] - t[3] }
mult -> exp : exp '*' exp => { t[0] = t[1] * t[3] }
div -> exp : exp '/' exp => { t[0] = t[1] / t[3] }
uminus -> exp : '-' exp %prec UMINUS => { t[0] = -t[2] }
expPar -> exp : '(' exp ')' => { t[0] = t[2] }
expNumber -> exp : NUMBER => { t[0] = t[1] }
expVAR -> exp : VAR => { t[0] = getval(t[1]) }

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

@
def getval(n):
    if n not in y.ts:
        print(f"Undefined name '{n}'")
    return y.ts.get(n,0)

y=yacc.yacc()
y.ts = {}
y.parse("3+4*7")
@
```

Figura 1: Sintaxe do PLY-Simple

Fizemos algumas alterações em relação ao exemplo proposto no enunciado ao decorrer do projeto, onde achamos necessário. Dentre elas , é possível destacar algumas:

```
literals = §+-=/*()§
```

Figura 2: caracter '§' delimitador

Para a leitura do campo dos "literals", com o objetivo de não conflitar com caracteres "literals"já definidos , foi utilizado um caracter diferente: '§'

```
atrib -> stat : VAR '=' exp => { t.y.ts[t[1]] = t[3] }
```

Figura 3: Alteração nos campos das gramáticas

Foi acrescentado ao início o nome do estado que irá ser definido a gramática, e colocando a gramática entre a sequencia de caracteres definida acima , com o objetivo de facilitar na criação do ficheiro output a definicão do nome da função, e os delimitadores a fim de obter a gramática de uma melhor forma.

```
@
def getval(n):
    if n not in y.ts:
        print(f"Undefined name '{n}'")
    return y.ts.get(n,0)

y=yacc.yacc()
y.ts = {}
y.parse("3+4*7")
@
```

Figura 4: Utilização do caracter especial '@'

Para a leitura do fim da parte do YACC , como é uma parte considerada muito semelhante ao do ply , e a fim de colocar o conteúdo aprendido nas aulas acerca dos estados do lex, foi utilizado o caracter '@' com a funcionalidade de entrar em um estado onde se lê qualquer tipo de caracter além dele.

No desenvolvimento deste projeto definimos como prioridade tornar o tradutor o mais genérico possível, podendo aceitar outros tipos de ply-simple seguindo a sintaxe definida. E com o intuito de testar outros ply-simples, foi definido outro ply-simple a partir de um programa realizado nas aulas sobre compilação de diretorias, se encontra este na diretoria input/plysimple2.in .

2.3 Modelação da Gramática

Após ser definido a sintaxe do ficheiro PLY-Simple, foi realizado a modeleção inicial da gramática para processar o arquivo.

```
S -> parser

parser -> lex yacc

lex -> lexStats lexDefs

lexStats -> lexStats lexStat
lexStats -> lexStat
lexStat -> LIT '=' simbols
lexStat -> TOK '=' '[' tokens ']'
lexStat -> IGNORE '=' VALUE
simbols -> simbols simbol
simbols -> simbol
tokens -> tokens ',' token
tokens -> token

lexDefs -> lexDefs lexDef
lexDefs -> lexDef
lexDef -> REGEX RETURN TORETURN
lexDef -> ERROR

yacc -> precedence yaccGram yaccFim
precedence -> PRECEDENCE
precedence ->
yaccGram -> yaccGram yaccStat
yaccGram -> yaccStat
yaccStat -> NOME GRAM FUNC
yaccStat -> ERROR
yaccFim -> SAMEASPLY
```

Figura 5: Modelação da Gramática

Neste, começamos com um estado inicial S que dá origem ao analisador, que é dividido na parte LEX e na parte YACC.

Na parte lexical podemos ter:

- Uma série de declarações, ou seja, a definição dos literais, dos tokens e do ignore.
- Um conjunto de definições para os tokens definidos anteriormente.

Na parte YACC temos:

- A parte do precedence.
- A gramática criada para processar as operações.
- A parte final, com a definição do parser.

Nesta gramática que implementamos, a recursividade está sempre à esquerda, e é uma implementação do método BottomUp. Nesta estratégia, os casos mais específicos são implementados primeiro até que o caso mais geral seja alcançado.

2.4 Estrutura

Nosso programa está estruturado em um arquivo para o lexer e outro para o parser, além das pastas "etc"(contém ficheiros porduzidos pelo ply), "input"(ficheiros ply-simple) e "output"(ficheiros no formato ply produzidos).

2.4.1 Lexer

Para atingir os símbolos terminais, foram desenvolvidos através do LEX a criação dos tokens, que são os seguintes:

- GRAMM: para capturar a expressão que define uma linha da gramática, que processa a forma como as operações são realizadas.
- ID: para identificar qualquer sequência textual.
- ASPAS2: para capturar as aspas duplas.
- ASPAS: para capturar as aspas.
- REGEX: define a estrutura de uma expressão regular.
- RETVALUE: para capturar as expressões que retornam um valor (Exemplo: *t.value*).
- TYPE: para pegar o tipo pretendido de *t.value*.
- TOIGNORE: leitura do valor de : ignore = 'valor'.
- SPACE: leitura de qualquer espaço que se encontre.
- ERROR: para capturar a mensagem de erro.
- FUNC: para capturar as funções desempenhadas por cada linha da gramática (Exemplo: t[0] = t[1] + t[3]).
- ANY: token que lê qualquer sequencia de caracteres desde que não seja '@'.
- BEGINEND: lê o caracter '@' responsável começar e acabar o estado 'readall'.
- PRECED: para buscar o campo do precedence.

Também foram definidos os símbolos literais que usaremos no parser. Estes são os seguintes:

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

Figura 6: Literais

Para além da definição dos obrigatórios tokens, e da definição dos literals, foram também definidas palavras reservadas(reserved).

Estas palavras reservadas permitem evitar conflitos na interpretação de texto. Quando o lexer lê um conjunto de carateres procura por esse conjunto exato num dicionário de palavras reservadas. Se encontar uma correspondência, atribui ao token lido o tipo correspondente à chave encontrada. Caso isto não aconteça, o token irá corresponder a uma variável.

2.4.2 Parser

Aqui implementamos toda a sintaxe definida na gramática para realizar o processamento do arquivo PLY-Simple através do "yacc"disponibilizado pela biblioteca ply, e assim foi o compilador responsável por ler e analisar o ficheiro input.

Para fazer isso, definimos funções para cada linha da gramática. Foi implementada a gramática definida anteriormente mas de forma completa, não só apenas com os símbolos não terminais, mas também com os terminais, esses produzidos pelos tokens, literais e palavras reservadas definidas no Lexer.

2.5 Semântica

O objetivo deste projeto além de ler o ficheiro PLY-Simple cumprindo a sintaxe definida , também é proposto traduzi-lo de forma que gere um outro ficheiro equivalente, porém com a sintaxe do ply.

A fim de cumprir este objetivo, é possível incluir semanticamente no "yacc"sentenças que retornem em cada estado os valores desejados. Foi utilizado o standard output como forma de imprimir o resultado do parser, e passando assim o standard output para o ficheiro output pretendido. Como foi realizado o parser para o ficheiro todo, foi explorado manipulações de strings e a produção da semântica principalmente nos símbolos terminais onde possuem os valores necessários para a definição do ply.

Foi também utilizado na semântica a lista "parser.tokensList"responsável por guardar os tokens lidos, para assim realizar uma verificação nas definições dos tokens, verificando se o toker a definir já foi listado anteriormente.

```
87      "tokens : tokens ',' token"
88      p[0] = p[1] + p[2] + p[3]
89
90  def p_token(p):
91      "token : ASPAS ID ASPAS"
92      p[0] = '"' + p[2] + '"'
93      parser.tokensList.append(p[2])
94
95  def p_lex_defs(p):
96      "lexDefs : lexDefs lexDef"
97      p[0] = p[1] + p[2]
98
99  def p_lex_defs_one(p):
100     "lexDefs : lexDef"
101     p[0] = p[1]
102
103 def p_lex_def(p):
104     "lexDef : REGEX RET '(' ASPAS2 ID ASPAS2 ',' valueRet ')' "
105     if p[5] in parser.tokensList:
106         result = "def t_" + p[5] + "(t):\n\t" + p[1] + "\n"
107         if p[8] != "t.value":
108             result+= "\tt.value = " + p[8] + "(t.value)\n"
109         p[0] = result + "\treturn t\n\n"
110     else:
111         print(f"(error) token '{p[5]}' não definido\n")
112         p.parser.error = True
113
114 def p_lex_def_error(p):
115     "lexDef : ERROR"
116     p[0] = "def t_error(t):\n\tprint(" + p[1] + ")\n\n"
117
118 def p_lex_valueret_simple(p):
119     "valueRet :  RETVALUE "
120     p[0] = "t.value"
121
122 def p_lex_withType(p):
123     "valueRet :  TYPE '(' RETVALUE ')' "
124     p[0] = p[1]
125
```

Figura 7: Parte da semântica, do ficheiro parser.py

2.6 Resultados

A gramática final obtida foi:

```
Grammar

Rule 0     S' -> parser
Rule 1     parser -> lex yacc
Rule 2     lex -> % % LEX lexStats lexDefs
Rule 3     lexStats -> lexStats lexStat
Rule 4     lexStats -> lexStat
Rule 5     lexStat -> LIT = § simbols §
Rule 6     lexStat -> TOK = [ tokens ]
Rule 7     lexStat -> IGNORE = TOIGNORE
Rule 8     simbols -> simbols simbol
Rule 9     simbols -> simbol
Rule 10    simbol -> +
Rule 11    simbol -> -
Rule 12    simbol -> =
Rule 13    simbol -> /
Rule 14    simbol -> *
Rule 15    simbol -> (
Rule 16    simbol -> )
Rule 17    simbol -> [
Rule 18    simbol -> ]
Rule 19    tokens -> token
Rule 20    tokens -> tokens , token
Rule 21    token -> ASPAS ID ASPAS
Rule 22    lexDefs -> lexDefs lexDef
Rule 23    lexDefs -> lexDef
Rule 24    lexDef -> REGEX RET ( ASPAS2 ID ASPAS2 , valueRet )
Rule 25    lexDef -> ERROR
Rule 26    valueRet -> RETVALUE
Rule 27    valueRet -> TYPE ( RETVALUE )
Rule 28    yacc -> % % YAC precedence yaccGram yaccFim
Rule 29    precedence -> PRECED
Rule 30    precedence -> <empty>
Rule 31    yaccGram -> yaccStat
Rule 32    yaccGram -> yaccGram yaccStat
Rule 33    yaccStat -> ID GRAMM FUNC
Rule 34    yaccStat -> ERROR
Rule 35    yaccFim -> ANY
```

Figura 8: Gramática final

Foram gerados também 71 estados, 32 símbolos terminais e 15 não terminais, dados e a gramática identificados através do auxílio do ficheiro parser.out gerado pelo ply.

O ficheiro output do ply equivalente produzido foi:

```
import ply.lex as lex

literals = ["+","-","=","/","*","(",")"]
tokens = ["VAR","NUMBER"]
t_ignore = ' \t\n'
def t_VAR(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    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')
]

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


def p_statExp(t):
    " stat : exp "
    print(t[1])
```

Figura 9: Parte do ficheiro output "plyEnunciado.py"

A partir do ficheiro produzido é possível corre-lo, o resultado:

```
pedraraujo@DESKTOP-2JT3RSE:/mnt/d/Uminho/PL/Projeto_PL/tp2/output$ python3 ply
Enunciado.py
31.0
```

Figura 10: resultado ao correr "plyEnunciado.py"

Vale destacar que o ficheiro output do outro exemplo plySimple também está disponível na pasta ./output/ .

3 Conclusão

Em suma, neste segundo trabalho, o grupo considera que aprofundou muito o seu conhecimento sobre as ferramentas usadas nesta unidade curricular. Foi dada novamente a oportunidade de trabalhar com expressões regulares e desenvolver processadores de linguagens regulares, e também, aprofundar os nossos conhecimentos da linguagem YACC.

Consideramos que o trabalho desenvolvido há aspetos à serem melhorados , como exemplos : permitir mais funcionalidades de especificações à gramática do ply-simple, podendo aceitar "ifs and elses", ciclos e etc. , também definir uma sintaxe que consiga ler a parte final do ficheiro ply-simple. Contudo, foi cumprido com o objetivo do trabalho que era a criação de um compilador explorando os conteúdos aprendidos nas aulas, e obteve-se então impacto positivo.

Por fim, com estes dois projetos concluídos, o objetivo será implementar o conhecimento obtido nesta unidade curricular e aplicá-lo em problemas de programação futuros.