Processamento de Linguagens Fase 2
PLY-SIMPLE

André Gonçalves Pinto a93173 and Rui Pedro Chaves Lousada a93252

Universidade do Minho `Grupo 58`

Abstract. Esta fase tem como objetivo desenvolver um projeto da cadeira de **Processamento de Linguagens**, capaz de converter um ficheiro com as nossas regras sintáticas em código executável. Para executar tal tarefa é necessário a extensa compreensão em gramáticas e analisadores léxicos. É requerido experência em desenvolver compiladores sofisticados capazes de detetar bem a gramática, assim como situações de erro

Keywords: Regular Expressions · Python · Text Filter · State Machines · Languague Processing

1 Tradutor PLY SIMPLE para PLY

1.1 Objetivo

O gerador de parsers PLY, embora poderoso tem uma sintaxe um pouco complexa. Pretendemos criar um tradutor que a partir de uma sintaxe mais "limpa" **PLY-simple**, gere as funções PLY convenientes.

1.2 Compilador de código LEX

A nossa abordagem inicial para o problema foi começar a partir do exemplo do **LEX** fornecido pelos docentes e a partir desse ponto aumentar a complexidade do nosso compilador

Ficheiro exemplo fornecido pelos professores:

```
%% 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) )
```

De início reparamos que esta sintaxe fornecida pelos professores era demasiado complexa e desnecessáriamente dificil de construir, devido ao facto de termos de analisar o que está dentro do return, conseguir separar as virgulas normais das virgulas de separação de argumentos

Dessa forma optamos por um sintaxe mais simples:

```
% LeX
%literals = "+-/*=()"
%ignore = " \t\n"
%tokens = "[ 'VAR','NUMBER' ]"

VAR "[a-zA-Z_][a-zA-Z0-9_]*" ""print("Found a ID")""

NUMBER "\d+(\.\d+)?" ""t.value = float(t.value)""

error   "" print(f"Illegal character {t.value[0]}")
```

```
t.lexer.skip(1)
""
```

Com a seguinte sintaxe foi muito mais facil criar os Tokens para o analisador léxico.

- LEX reconhece a palavra LEX iniciada por %
- STR_ATRIB texto capturado entre " "
- CODE_EXPRESSION texto captura entre "" ""
- PARAMETER palavra capturada que inicia -se por %
- TOKEN_ID palavra capturada

Tendo os tokens bem definidos foi só uma questão de implementar corretamente a gramática do ***YACC*** para reconhecer que tipo de pedaço código deve ser gerado

Dessa forma temos já o seguinte código Python gerado a partir da nossa sintaxe definida:

```
#------------------
# Lexer Code
#------------------

import ply.lex as lex

literals = []
tokens = ['VAR', 'NUMBER']
ignore = " \t\n"
def t_VAR(t):
        r'[a-zA-Z_][a-zA-Z0-9_]*'
        print("Found a ID")
        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.skip(1)


lexer = lex.lex()
```

1.3 Compilador de código YACC

Tendo inicialmente já gerado o codigo LEX, de seguida dedicamo-nos a conseguir gerar código YACC.

Como habitual primeiro observamos a implementação fornecida pelos docentes para uma sintaxe em YACC.

```
%% 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")
```

Novamente encontramos uma sintaxe desnecessáriamente complexa, sendo assim, criamos a nossa sintaxe mais simples de escrever e também mais fácil de implementar no analisador léxico e gramatical

```
% YAcc

%precedence = ""[
('left','+','-'),
('left','*','/'),
]""

""ts = { }""

%""prog : comandos"" "" ""
%""comandos : comando comandos"" "" ""
%"" comandos :"" "" print("End of file") ""

%"" comando : stat "" "" ""
 %""stat : VAR "=" exp"" "" ts[p[1]] = p[3] ""
%""stat : exp"" "" print(p[1]) ""

%""exp : exp '+' exp "" "" p[0] = p[1] + p[3]
print("found a +") ""

%""exp : exp '-' exp "" "" p[0] = p[1] - p[3] ""
%""exp : '(' exp ')' "" "" p[0] = p[2] ""
%""exp : NUMBER "" "" p[0] = p[1] ""
%""exp : VAR "" "" p[0] = getval(p[1]) ""
```

A partir desta sintaxe para o YACC foi necessário apenas adicionar um novo token

- YACC reconhece a palavra YACC iniciada por %
- GRAMMAR texto que inicia-se por %"" e acaba em ""

Tendo assim gerado o seguinte código YACC

```
#------------------
# Yacc Code
#------------------
import ply.yacc as yacc
precedence = [
('left','+','-'),
('left','*','/'),
]
ts = { }
def p_grammar0(p):
        r'prog : comandos'
def p_grammar1(p):
        r'comandos : comando comandos'
def p_grammar2(p):
        r' comandos :'
        print("End of file")
def p_grammar3(p):
        r' comando : stat '
def p_grammar4(p):
        r'stat : VAR "=" exp'
        ts[p[1]] = p[3]
def p_grammar5(p):
        r'stat : exp'
        print(p[1])
def p_grammar6(p):
        r"exp : exp '+' exp "
        p[0] = p[1] + p[3]
        print("found a +")
def p_grammar7(p):
        r"exp : exp '-' exp "
        p[0] = p[1] - p[3]
def p_grammar8(p):
        r"exp : '(' exp ')' "
        p[0] = p[2]
def p_grammar9(p):
        r'exp : NUMBER '
        p[0] = p[1]
def p_grammar10(p):
        r'exp : VAR '
        p[0] = getval(p[1])
def p_error(p):
    print("Error in",p)
parser = yacc.yacc()
```

Como podemos observar o nosso script de Python já está a conseguir a identificar corretamente os tokens, conseguir distinguir e atribuir significados à gramatica, e por fim gerar código válido.

1.4 Extras

Ainda assim achamos necessário conseguir tornar a nossa sintaxe ainda mais simples, dessa forma sendo o compilador a deduzir as nossas ações.

Comentários

Quando é encontrado um # fora de aspas esse pedaço de texto até ao final da linha vai ser ignorado, dessa forma não irá aparecer no código final

Não é necessário declarar tokens

Criamos outro analisador léxico que faz uma passagem **extra** pelo texto para descobrir os **tokens** que vão ser utilizados. Dessa maneira é desnecessário a declaração no LEX:

```
%tokens = ['VAR','NUMBER']
```

Não é necessário declarar literals

Na passagem extra pelo ficheiro também são encontrados os **caracteres que fazem partem da definição da gramática** e são corretamente adicionados à estrutura de dados.

Dessa forma é desnecessária a declaração no LEX:

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

Função sem valor de retorno no LEX

Basta inserir um `!` no inico da declaração de código de uma função do analisador lexico que o compilador não irá gerar o pedaço de código `return t`

Não há necessidade de declarar imports, nem de construir o analisador, nem a gramática

Os imports são adicionados logo no inicio do programa. O analisador léxico é construido quando é detetado um token **YACC**. O parser do YACC é construido depois da ultima gramatica ter sido inserida, assim como a **função error** que por default é construida

Dessa forma removemos a necessidade de escrever:

```
import ply.lex as lex
#lexer code...
lexer = lex.lex()

import ply.yacc as yacc
# yacc code...
def p_error(p):
    print("Error in",p)
parser = yacc.yacc()
```

Dualidade na escrita da gramática

É válido escrever a seguinte gramatica:

```
%""exp : exp '+' exp ""
```

como também a seguinte

```
 %""stat : VAR "=" exp""
```

o compilador consegue detetar esses casos e gerar o devido código

```
def p_grammar4(t):
        r'stat : VAR "=" exp'

def p_grammar6(t):
        r"exp : exp '+' exp "
```

Argumentos de entrada e saida

O programa é flexivel sobre os argumentos de entrada e saida, sendo os seguintes comandos completamente válidos

```
python3 yacc.py input.txt

python3 yacc.py input.txt outfile.py

cat input.txt| python3 yacc.py > outfile.py
```

2 Conclusão

Desenvolver um compilador para processar texto foi uma tarefa bastante difícil no início. Requeriu muito planeamento e conhecimento da teoria para reduzir a refactorização do código ao longo do projeto.

Com este projeto senti-mos muito mais confiantes a desenvolver gramáticas independentes de contexto, processadores de texto, que definitivamente irá ser uma ferramenta muito útil no nosso futuro.