TP11 : PLY : Python Lex & Yacc

31 Mars 2014

1 Le Lexer

Il est utilisé pour tokeniser une chaîne de caractères. Imaginons :

$$x = 3 + 42 * (s - t)$$

Si on tokenise cette entrée, on obtiendra :

$$'x', '=', '3', '+', '42', '*', '(', 's', '-', 't', ')'$$

Généralement, il est préférable de donner des noms aux tokens afin de les définir.

Prenons le cas de PLY :

```
# List of token names.   This is always required
#Commentaire : On definit ici la liste de tous les tokens
#que le lexer va pouvoir prendre en charge
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
   'COMMENT'
)

# Regular expression rules for simple tokens
#Commentaire : On specifie ici a quoi correspondent les differents tokens
#On peut utiliser des expressions regulieres comme ci-dessous
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
#Commentaire: On peut aussi utiliser des fonctions qui debutent
# par une expression reguliere. Ainsi dans ce code ci-dessous
# on s'assure que le nombre sera pris en compte par Python comme
#etant un entier.

# La fonction prend un seul et unique argument :
# une instance de LexToken (appartenant au module)
# On a acces a differentes choses :
# - t.type --> type du token (un string) qui est en fait son nom
# - t.value --> valeur du token (ici on veut un entier)
# - t.lineno --> le numero de ligne courant (important pour gerer les erreurs)
```

```python
# - t.lexpos --> la position du token dans la chaine d'entree
# Si on modifie quoi que ce soit, il faut retourner t.
# Si la fonction ne retourne rien, alors le token n'est pas conserve.
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Added ignore between t_ and the token name will ignore it.
t_ignore_COMMENT = r'\#.*'

# Error handling rule
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)

#Specify litterals characters

literals = [ '+', '-', '*', '/' ]

# Build the lexer
lexer = lex.lex()

# Data to test
data = "3 + 42 * (10 - 0)"

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok: break      # No more input
    print tok
```

On peut aussi utiliser une approche orientée objet pour définir le lexer :

```python
class MyLexer:
    # List of token names.   This is always required
    tokens = (
       'NUMBER',
       'PLUS',
       'MINUS',
       'TIMES',
       'DIVIDE',
       'LPAREN',
       'RPAREN',
    )

    # Regular expression rules for simple tokens
    t_PLUS    = r'\+'
```

```
    t_MINUS   = r'-'
    t_TIMES   = r'\*'
    t_DIVIDE  = r'/'
    t_LPAREN  = r'\('
    t_RPAREN  = r'\)'

    # A regular expression rule with some action code
    # Note addition of self parameter since we're in a class
    def t_NUMBER(self,t):
        r'\d+'
        t.value = int(t.value)
        return t

    # Define a rule so we can track line numbers
    def t_newline(self,t):
        r'\n+'
        t.lexer.lineno += len(t.value)

    # A string containing ignored characters (spaces and tabs)
    t_ignore  = ' \t'

    # Error handling rule
    def t_error(self,t):
        print "Illegal character '%s'" % t.value[0]
        t.lexer.skip(1)

    # Build the lexer
    def build(self,**kwargs):
        self.lexer = lex.lex(module=self, **kwargs)

    # Test it output
    def test(self,data):
        self.lexer.input(data)
        while True:
             tok = lexer.token()
             if not tok: break
             print tok

# Build the lexer and try it out
m = MyLexer()
m.build()           # Build the lexer
m.test("3 + 4")     # Test it
```

2 Le parser

On utilise une grammaire sous forme Backus-Naur (BNF). Par exemple la grammaire ETF serait représentée de la sorte :

```
expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

Dans la grammaire ci-dessus, $+$, $-$, $*$, $/$, (,) et $NUMBER$ sont des littéraux et correspondent à des tokens d'entrée. Les identifiants tels que *term* ou $factor$, quant à eux, réfèrent aux règles de la grammaire.

Pour mettre en place la sémantique du langage, on utilise la technique dite *Syntax-driven translation*. Pour chaque symbole de la grammaire, on lui attache un attribut et on lui assigne une action. L'action décrit ce qu'il faut faire. Reprenons l'exemple ci-dessus :

```
        Grammar                             Action
--------------------------------    --------------------------------------------
expression0 : expression1 + term    expression0.val = expression1.val + term.val
            | expression1 - term    expression0.val = expression1.val - term.val
            | term                  expression0.val = term.val

term0       : term1 * factor        term0.val = term1.val * factor.val
            | term1 / factor        term0.val = term1.val / factor.val
            | factor                term0.val = factor.val

factor      : NUMBER                factor.val = int(NUMBER.lexval)
            | ( expression )        factor.val = expression.val
```

Enfin, YACC est un analyseur LR (pour être précis : LALR).

Prenons maintenant un exemple de règles de grammaire implémentées via PLY.

```python
# Yacc example

#import yacc
import ply.yacc as yacc

# Get the token map from the lexer.  This is required.
from calclex import tokens

# On remarque que chaque regle de grammaire est representee par une fonction
# en python avec un attribut p qui est une sequence contenant
# les valeurs de chaque symbole pour la regle corresponde
#
# On definit une regle dans le docstring de la fonction
# de fait pour la regle p_expression_plus, on retrouve
# notre regle : expression : expression + term
#
# On rappelle que + a ete nomme PLUS dans notre lexer

def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]
```

```
def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
#
# On utilise la fonction que pour recuperer les erreurs de syntaxe
def p_error(p):
    print "Syntax error in input!"

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print result
```

On peut élaborer les règles de grammaire de la sorte :

```
# Plutot que d'avoir deux fonctions differentes

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

#On peut n'en avoir qu'une

def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

Par ailleurs, on peut vouloir représenter les ε-productions de la sorte :

```
# On definit epsilon qui ne fait strictement rien
def p_empty(p):
    'empty :'
    pass

# On peut donc utiliser empty en tant qu'epsilon
def p_optitem(p):
    'optitem : item'
    '        | empty'
```

Concernant l'axiome de la grammaire, la première règle trouvée par YACC dans le fichier est considérée comme l'axiome. Ce comportement par défaut n'est pas gênant, mais on veut pouvoir le modifier.

```
# En definissant une variable start avec un nom de regle
# on explique a YACC que cette regle sera l'axiome
start = 'foo'
```

Enfin, on ne traite pas ici du cas des grammaires ambigües et de la précédence (nécessaire avec ETF par exemple), mais on pourra se référer à la section 6.6 de la documentation.

3 Exercices

On définit un langage qui permet de réprésenter des arbres à la figure 1. L'arbre résultant est représenté à la figure 2.

racine	:	F ;
F	:	A, B, E ;
A	:	;
B	:	D ;
D	:	;
E	:	C, G ;
C	:	;
G	:	;

Figure 1 – Langage d'arbre

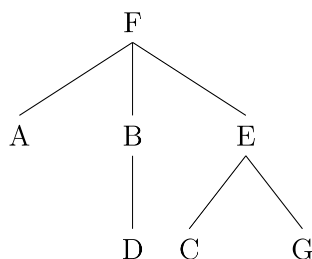

Figure 2 – Arbre résultant

1. Proposer une grammaire qui reconnaisse ce langage.
2. Ajouter un attribut et des actions sémantique pour calculer le nombre de fils que l'arbre possède.