ply (continuación)

Teoría de Lenguajes

Ramiro Camino

Universidad de Buenos Aires

Junio 2015

Introducción

- Ya vimos como generar un AST.
- Ahora vamos a ver cómo darle semántica.

Gramática de ejemplo

$$G =< \{E, T, F\}, \{+, *, num, (,)\}, P, E >$$

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & F \\ F & \rightarrow & num \\ F & \rightarrow & (E) \end{array}$$

Expresiones

- Una vez que tengamos el AST armado, queremos evaluar la expresión aritmética.
- Vamos a pensar cada nodo del AST como una expresión evaluable.

```
class Expression(object):

    def evaluate(self):
        # Aca se implementa cada tipo de expresion.
        raise NotImplementedError
```

Números

- Los nodos más simples son las expresiones numéricas.
- Cuando se evalúan simplemente devuelven el valor del token.

```
class Number(Expression):

    def __init__(self, value):
        self.value = value

    def evaluate(self):
        return self.value
```

Operaciones

- Para este ejemplo la suma y la multiplicación se pueden considerar como operaciones binarias, es decir, que aplican una operación a dos operandos.
- En ambos casos, para evaluar la operación primero necesitamos evaluar cada operando.
- Luego podemos aplicar la operación.

Operaciones binarias

```
class BinaryOperation(Expression):

    def __init__(self, left, right, operator):
        self.left = left
        self.right = right
        self.operator = operator

    def evaluate(self):
        res_l = self.left.evaluate()
        res_r = self.right.evaluate()
        return self.operator(res_l, res_r)
```

Reglas

```
from operator import add, mul
from expressions import BinaryOperation, Number

def p_expression_plus(expr):
    'expression : expression PLUS term'
    expr[0] = BinaryOperation(expr[1], expr[3], add)

def p_term_times(expr):
    'term : term TIMES factor'
    expr[0] = BinaryOperation(expr[1], expr[3], mul)

def p_factor_number(expr):
    'factor : NUMBER'
    expressions[0] = Number(expr[1])
```

Ejecución completa

- Generamos el árbol como antes.
- El nodo raíz representa la expresión completa.
- La evaluamos para saber el resultado total.

```
text = "(14 + 6) * 2"

lexer = lex(module=lexer_rules)
parser = yacc(module=parser_rules)

expression = parser.parse(text, lexer)

result = expression.evaluate()
print result

>> 40
```