

Intro a PLY (Python Lex-Yacc)

Teoría de Lenguajes

Fac. de Ciencias Exactas y Naturales, Universidad de Buenos Aires

2do. cuatrimestre 2016

Introducción

- Sirve para construir analizadores léxicos y sintácticos
- Implementación de *lex* y *yacc* en python
- Sitio oficial: <http://www.dabeaz.com/ply/>

Gramática de ejemplo

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

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow num$$

$$F \rightarrow (E)$$

Lexer: Introducción

Para generar un lexer, vamos a crear un archivo llamado `lexer_rules.py`, en el cual vamos a definir los tokens y las reglas para detectarlos.

Lexer: Tokens

Tenemos que definir una variable llamada `tokens` que guarde la lista de tipos de tokens que vamos a buscar. Cada tipo se representa con un string:

```
tokens = [  
    'NUMBER',  
    'PLUS',  
    'TIMES',  
    'LPAREN',  
    'RPAREN'  
]
```

Lexer: Reglas

- Cada regla posee una expresión regular para indicar la cadena que se desea capturar.
- Se prueba con todas las reglas, y se utiliza la que más símbolos captura.
- Si dos reglas capturan la misma cantidad de símbolos, se utiliza la que esté definida más arriba en el archivo.
- Una vez seleccionada la regla, se decide el valor del token.
- Comienza de nuevo con el resto de la cadena.
- Termina cuando no queden mas símbolos para leer.

Lexer: Reglas 'simples'

- Solo definen el tipo y la expresión regular.
- El valor queda implícito: es exactamente la cadena que se captura.
- Se definen con variables con el prefijo `t_`.
- El resto del nombre de la variable debe coincidir con el tipo del token.

Lexer: Reglas 'simples' (ejemplo)

```
t_PLUS = r"\+"  
t_TIMES = r"\*"  
t_LPAREN = r"\("  
t_RPAREN = r"\)"
```

- Las cadenas que empiezan con `r` en python se leen en crudo, esto nos permite escribir las `\` sin escaparlas.
- Hay que escapar el `+`, `*`, `(` y el `)` porque son símbolos que forman parte de la sintaxis de expresiones regulares.

Lexer: Reglas 'complejas'

- Se definen con funciones con el prefijo `t_`
- El resto del nombre de la función debe coincidir con el tipo del token
- La expresión regular de la regla se define en el *docstring* de la función (ver ejemplo)
- Recibe un token como parámetro, cuyo valor es la cadena capturada
- El valor se puede modificar
- Se devuelve el token para generarlo, o no se devuelve nada para descartarlo

Lexer: Reglas 'complejas' (ejemplo)

```
def t_NUMBER(token):  
    r"[1-9][0-9]*"  
    token.value = int(token.value)  
    return token
```

- El docstring de la función va inmediatamente después de su definición
- En realidad (como lo dice su nombre) esto se usa en python para documentar la función, pero ply abusa de este feature

Lexer: Reglas 'especiales'

- Se pueden crear reglas con el sufijo `t_ignore_` para ignorar tokens capturados por expresiones regulares, ejemplo:

```
t_ignore_WHITESPACES = r"[ \t]+"
```

- Opcionalmente se puede definir la variable `t_ignore`, una cadena (no una expresión regular) que define los caracteres que se ignoran:

```
t_ignore = " \t"
```

Lexer: Manejo de líneas

```
def t_NEWLINE(token):  
    r"\n+"  
    token.lexer.lineno += len(token.value)
```

- El lexer por defecto no sabe cuándo queremos incrementar el contador de líneas
- Podemos hacer una regla que maneje los saltos de línea e incremente el contador
- No devolvemos el token para ignorarlo

Lexer: Manejo de errores

```
def t_error(token):  
    message = "Token desconocido:"  
    message = "\nntype:" + token.type  
    message += "\nvalue:" + str(token.value)  
    message += "\nline:" + str(token.lineno)  
    message += "\nposition:" + str(token.lexpos)  
    raise Exception(message)
```

- La función `t_error` define qué hacer cuando no se puede *matchear* ninguna regla
- Ignoramos espacios, tabs y saltos de línea
- Lanzamos una excepción cuando no se puede *matchear* ninguna regla

Ejemplo: Ejecutar lexer solo

```
import lexer_rules

from ply.lex import lex

text = "(14 + 6) * 2"
lexer = lex(module=lexer_rules)
lexer.input(text)

token = lexer.token()

while token is not None:
    print token.value
    token = lexer.token()
```

Parser: Introducción

- Para generar un parser, vamos a crear un archivo llamado `parser_rules.py`, en el cual vamos a definir las producciones de la gramática y cómo generar el árbol.
- En este archivo ply nos pide que importemos la lista de tokens que definimos antes:

```
from lexer_rules import tokens
```

- Por otro lado, cada producción debe devolver un nodo del árbol, con lo cual vamos a crear en un archivo llamado `expressions.py` para definir una clase (muy simple) para cada tipo de nodo del árbol, y las incluimos:

```
from expressions import *
```

Parser: Reglas

```
def p_expression_plus(subexpr):  
    'expression : expression PLUS term'  
    subexpr[0] = Add(subexpr[1], subexpr[3])
```

- Se define una función por cada producción que debe contener el sufijo `p_`, el resto del nombre no importa.
- Esta vez el *docstring* contiene la producción.
- En minúsculas están los no terminales.
- En mayúsculas están los terminales (tokens).
- La función recibe la lista de subexpresiones ya parseadas.
- Los *dos puntos* no forman parte de la lista de subexpresiones.
- El valor de la producción (el nodo parseado) se devuelve modificando el lado izquierdo de la producción.

Parser: Errores

```
def p_error(subexpr):  
    raise Exception("Syntax error.")
```

- Pueden definir una función llamada `p_error` para manejar errores.
- Con `subexpr.lineno(i)` pueden obtener el número de línea de la subexpresión `i`.
- Con `subexpr.lexpos(i)` pueden obtener la posición de la subexpresión `i`.
- Por defecto el parser genera un log llamado `parser.out` que pueden mirar.
- Se pueden hacer cosas mas complejas, como intentar corregir los errores o hacer producciones extra que capturen casos de sintaxis errónea conocidos. Hay mas ejemplos en el manual.

Ejemplo: Ejecutar parser

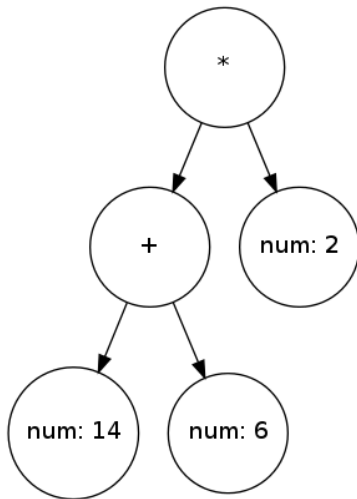
```
import lexer_rules
import parser_rules

from ply.lex import lex
from ply.yacc import yacc

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

text = "(14 + 6) * 2"
ast = parser.parse(text, lexer)
```

Ejemplo: Abstract Syntax Tree



Introducción

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

Gramática de ejemplo

$$G = \langle \{E, T, F\}, \{+, *, \textit{num}, (,)\}, P, E \rangle$$

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow \textit{num}$$

$$F \rightarrow (E)$$

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
```