Python Lex/Yacc
PLY

Horst H. von Brand
vonbrand@inf.utfsm.cl

Departamento de Informática
Universidad Técnica Federico Santa María

Contenido

Lenguajes de programación contemporáneos

Suelen describirse en varios niveles:

- **Léxico:** Aspectos como operadores y puntuación, comentarios, constantes, identificadores y palabras clave.
- **Sintáctico:** Conformación de expresiones, estructuras de control y programas completos.
- **Semántica:** Significado de las anteriores.

Aspectos léxicos se representan por lenguajes regulares, sintácticos por gramáticas de contexto libre, mientras la semántica generalmente se expresa en código.

Las herramientas lex y yacc

Escribir un programa que reconoce el lenguaje generado por una gramática de contexto libre es una tarea tediosa. Muchas construcciones comunes (como expresiones aritméticas) requieren contorsiones para manejarlas mediante técnicas simples.
En los '70 en Unix se desarrolló la herramienta yacc (abreviatura del inglés *Yet Another Compiler Compiler*) utilizando nuevas técnicas de manejo de gramáticas de contexto libre. Pronto se le unió lex para manejar aspectos léxicos (símbolos básicos del lenguaje, como identificadores y operadores). Ambas son herramientas estándar en POSIX.

Python Lex-Yacc

Como herramienta didáctica (aunque ha hallado uso práctico) se desarrolló ply. A diferencia de sus inspiraciones, un programa que usa ply procesa la gramática cada vez que se ejecuta.

Convenciones de ply

Entidades léxicas (terminales) se nombran en mayúsculas, no-terminales en minúsculas. A cada símbolo le asigna un valor, producciones calculan el valor del no-terminal del lado izquierdo de los valores de los símbolos a lado derecho.
En esencia, calcula el valor del símbolo de partida en un recorrido en postorden del árbol de derivación.

Calculadora

Desarrollaremos una calculadora que maneje variables y números enteros con las cuatro operaciones además de potencia (que anotaremos **).
Leemos una línea y la ejecutamos. Valores asignados a variables se almacenan en un diccionario global, los valores de otras expresiones se imprimen.

Subherramienta lex

Terminales son funciones con nombres que comienzan 't_', una expresión regular Python define la representación. Símbolos de un único caracter y sin valor se definen en el arreglo literals . El símbolo especial t_ignore son caracteres ignorados. La función t_error se invoca en caso de error.

Calculadora — análisis léxico

Símbolos con nombre y de un caracter

```
tokens = ('NAME', 'NUMBER', 'POWER')

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

t_POWER    = r'\*\*'
t_NAME     = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

Calculadora — análisis léxico

Valores de símbolos

Valores de números son simplemente el valor entero respectivo.

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Calculadora — análisis léxico

Valores de símbolos

Variables son especiales: pueden usarse al lado izquierdo de una asignación (hay que darle un valor, posiblemente creando el espacio respectivo) o en expresiones, como por ejemplo al lado derecho de asignaciones (hay que recuperar el valor, o dar un error si la variable no está definida). Esto debe manejarse en detalle en la gramática.

Calculadora — análisis léxico

Caracteres ignorados, saltos de línea, errores

```
# Ignored characters
t_ignore = " \t"

def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")

def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)
```

Calculadora — análisis léxico

Crear el objeto lex

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

Subherramienta yacc

Producciones son funciones con nombres que comienzan 'p_'. Su documentación da la producción, código calcula el valor del no-terminal del lado izquierdo. Se pueden agrupar varias producciones para un no-terminal dado o dar una sola producción en la función del caso.

Para abreviar, se pueden definir precedencias y asociatividades de operadores y usar la gramática ambigua del caso.

En caso de error se invoca p_error.

Calculadora — análisis sintáctico

Precedencias y asociatividades, diccionario de variables

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/', '%'),
    ('right', 'UMINUS'),    # Just for precedence
    ('right', 'POWER'),
    )

# dictionary of names
names = { }
```

Calculadora — análisis sintáctico

Instrucciones

```
def p_statement_assign(t):
    '''statement : NAME '=' expression'''
    names[t[1]] = t[3]

def p_statement_expr(t):
    'statement : expression'
    print(t[1])
```

Calculadora — análisis sintáctico

Operaciones binarias

```
def p_expression_binop(t):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression
                  | expression '%' expression
                  | expression POWER expression'''
    if t[2] == '+'  : t[0] = t[1] + t[3]
    elif t[2] == '-': t[0] = t[1] - t[3]
    elif t[2] == '*': t[0] = t[1] * t[3]
    elif t[2] == '/': t[0] = t[1] / t[3]
    elif t[2] == '%': t[0] = t[1] % t[3]
    elif t[2] == '**': t[0] = t[1] ** t[3]
```

Calculadora — análisis sintáctico

Signo menos

```
def p_expression_uminus(t):
    '''expression : '-' expression %prec UMINUS'''
    t[0] = -t[2]
```

Calculadora — análisis sintáctico

Paréntesis

```
def p_expression_group(t):
    '''expression : "(" expression ")"'''
    t[0] = t[2]
```

Calculadora — análisis sintáctico

Números, variables

```
def p_expression_number(t):
    'expression : NUMBER'
    t[0] = t[1]

def p_expression_name(t):
    'expression : NAME'
    try:
        t[0] = names[t[1]]
    except LookupError:
        print(f"Undefined name '{t[1]}'")
        t[0] = 0
```

Calculadora — análisis sintáctico

Reportar errores

```
def p_error(t):
    print("Syntax error at '{t.value}'")
```

Calculadora — análisis sintáctico

Crear parser

```
import ply.yacc as yacc
parser = yacc.yacc()
```

Calculadora — ciclo principal

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

Resumen

- Hay herramientas que automatizan grandemente el manejo de sintaxis. Mostramos una muy simple de usar para Python.
- Como herramienta más que nada para uso didáctico, ply es más bien ineficiente y está limitado a trabajar con datos en memoria.
- Hay herramientas mucho más sofisticadas, que llegan a construir un árbol de derivación abreviado (un AST, *Abstract Syntax Tree*) y automatizan parcialmente su recorrido.