PyTiger2C

Breve descripción de Python Lex-Yacc

Yasser González Fernández
yglez@uh.cu

Ariel Hernández Amador
gnuaha7@uh.cu

1. Introducción

Python Lex-Yacc ó PLY es una implementación, desarrollada completamente en Python, de las herramientas tradicionales para la construcción de compiladores Lex y Yacc. PLY intenta seguir de manera bastante fiel la forma en que trabajan estas herramientas, incluyendo el soporte para gramáticas LALR(1), ε-producciones, resolución de ambigüedades mediante reglas de precedencia y recuperación de errores.

Esta herramienta fue creada por David M. Beazley en el año 2001 para ser utilizado en un curso de Introducción a Compiladores impartido por el autor en la Universidad de Chicago. Fue utilizada por los estudiantes de este curso para construir compiladores para un lenguaje simple con características semejantes a Pascal. Debido a su origen relacionado con la docencia, se dedicó en su implementación un gran esfuerzo a proveer una extensiva comprobación de errores.

PLY no soporta funcionalidades presentes en otras herramientas para la construcción de compiladores como construcción automática del árbol de sintaxis abstracta ni aspectos relacionados con la comprobación semántica. PLY sólo se ocupa del análisis lexicográfico y sintáctico.

PLY está compuesto por un paquete Python llamado `ply` que contiene dos módulos `lex.py` y `yacc.py`. El módulo `lex.py` se encarga del análisis lexicográfico, es decir, separa el flujo de entrada en una secuencia de *tokens* especificados mediante un conjunto de expresiones regulares. `yacc.py` se ocupa de reconocer la sintaxis del lenguaje especificada en la forma de una gramática libre del contexto. `yacc.py` realiza un análisis ascendente o LR y genera las tablas utilizando LALR(1) y opcionalmente SLR. Los dos módulos funcionan de manera conjunta, específicamente `lex.py` presenta una función `token()` como forma de interfaz externa que es ejecutada repetidamente por `yacc.py` para obtener los *tokens* y ejecutar las acciones asociadas a las producciones de la gramática.

La principal diferencia entre `yacc.py` y la herramienta `yacc` de UNIX es que `yacc.py` no requiere una fase de generación de código a partir de la descripción de la gramática. Las especificaciones dadas a PLY son programas Python válidos, esto significa que no se requieren archivos de código fuente adicionales ni un paso especial en el proceso de construcción del compilador para generar el código Python del compilador. PLY hace uso de los *docstrings* de Python para asociar las reglas de la gramática con sus acciones correspondientes. Como la generación de la tabla LALR(1) puede ser un proceso relativamente costoso, PLY guarda la tabla resultante en un archivo y en una próxima ejecución del compilador se comprobará si la gramática ha sido modificada para cargar la tabla generada anteriormente en lugar de volverla a generar.

2. Análisis lexicográfico

`lex.py` se utiliza para separar el flujo de caracteres de la entrada en una secuencia de *tokens*. La descripción de cada uno de los *tokens* válidos se realiza mediante expresiones regulares.

Todos los *tokens* válidos se deben especificar previamente en una lista llamada `tokens`. A continuación se muestra un fragmento de código Python en el que se definen *tokens* relacionados con operaciones aritméticas.

```
# Lista que define los tokens válidos.
tokens = ('NUMBER', 'PLUS', 'MINUS')

# Expresiones regulares para cada uno de los tokens.
t_NUMBER = r'\d+'
t_PLUS = r'\+'
t_MINUS = r'-'
```

Todos los *tokens* tienen asociado un nombre determinado por los caracteres luego del prefijo `t_` y un valor que generalmente es el lexema asociado al *token*.

Si es necesario asociar una acción a la regla de un *token* se puede especificar el *token* mediante una función. A continuación se muestra una regla asociada al *token* `NUMBER` que convierte la cadena al entero correspondiente.

```
def t_NUMBER(token):
    r'\d+'
    token.value = int(token.value)
    return token
```

Para construir el *lexer* se utiliza la función `lex.lex()` que retorna una instancia de la clase `Lexer`. Una vez que ha sido construido, se pueden utilizar dos funciones para interactuar con el *lexer*.

- `lexer.input(data)` Permite especificar el flujo de caracteres de entrada.
- `lexer.token()` Retorna una instancia de `LexToken` que representa el siguiente *token* en el flujo de entrada. Si todos los *tokens* han sido consumidos retornará `None`.

3. Análisis sintáctico

`yacc.py` se utiliza para reconocer la sintaxis del lenguaje. Las producciones de la gramática se especifican utilizando notación BNF en los *docstrings* de las funciones que corresponden a las acciones asociadas a estas. La primera producción de la gramática especificada se tomará como el símbolo inicial.

A continuación se muestra un ejemplo de una producción que forma parte de un evaluador de expresiones cuya implementación completa se muestra en la sección 4.

```
def p_expr_plus(symbs):
    r'expr : expr PLUS term'
    symbs[0] = symbs[1] + symbs[3]
```

La función acepta un único argumento `symbs` que es una secuencia que contiene los valores asociados a los símbolos de la gramática presentes en la producción. Las posiciones en la secuencia se asignan de izquierda a derecha. Es posible asignar cualquier objeto Python como valor asociado a un símbolo en una producción.

Para construir el *parser* se utiliza la función `yacc.yacc()` que recibe un *lexer* y una cadena para reconocer y retorna una instancia de la clase `Parser`. Una vez que ha sido construido, se puede utilizar el método `parse()` para indicar que comience a reconocer la cadena y ejecutar las acciones asociadas a las producciones.

4. Ejemplo

A continuación se muestra un programa Python que utiliza PLY para implementar un pequeño evaluador de expresiones aritméticas con números. Cuando este programa se ejecuta muestra un *prompt* y queda a la espera de que se introduzca una expresión para evaluarla y mostrar su valor.

```
# -*- coding: utf-8 -*-

"""
Ejemplo de un evaluador de expresiones aritméticas con PLY.
"""

from ply import lex, yacc

# Reglas del lexer.

tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN')

def t_NUMBER(token):
    r'\d+'
```

```
    token.value = int(token.value)
    return token

t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
t_ignore  = ' \t'


# Reglas del parser.

def p_expr_plus(symbs):
    r'expr : expr PLUS term'
    symbs[0] = symbs[1] + symbs[3]

def p_expr_minus(symbs):
    r'expr : expr MINUS term'
    symbs[0] = symbs[1] - symbs[3]

def p_expr_term(symbs):
    r'expr : term'
    symbs[0] = symbs[1]

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

def p_term_divide(symbs):
    r'term : term DIVIDE factor'
    symbs[0] = symbs[1] / symbs[3]

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

def p_factor_number(symbs):
    r'factor : NUMBER'
    symbs[0] = symbs[1]

def p_factor_paren(symbs):
    r'factor : LPAREN expr RPAREN'
    symbs[0] = symbs[2]


def main():
    """
    Función principal del script.
    """
```

```
    lexer = lex.lex()
    parser = yacc.yacc()
    while True:
        try:
            expression = raw_input('expression: ')
        except EOFError:
            break
        if expression:
            value = parser.parse(expression, lexer=lexer)
            print 'value: ', value


if __name__ == '__main__':
    main()
```