基于PLY的Python解析-1

实验内容

1. 利用PLY实现简单的Python程序的解析

（1）示例程序位于**example**

（2）需要进行解析的文件为**example.py**

（3）需要完成以下内容的解析

- **赋值语句**
- **完整的四则运算**
- **print语句**

四则运算的无二义性下文法大致如下：

```
expr -> expr + term | expr - term | term

term -> term * factor | term / factor | factor

factor -> id | (expr)

（不需要消除二义性）
```

（4）解析结果以语法树的形式呈现

2. 编程实现语法制导翻译

（1）语法树上每个节点有一个属性value保存节点的值

（2）设置一个变量表保存每个变量的值

（3）基于深度优先遍历获取整个语法树的分析结果

实验步骤

1.读入数据

主程序中调用**util.py**中的**clear_text**函数读取数据：

```
from util import clear_text
text = clear_text(open('example.py', 'r').read())
```

被调用的函数如下：

```
def clear_text(text):
    lines=[]
    for line in text.split('\n'):
        line=line.strip()
        if len(line)>0:
            lines.append(line)
    return ' '.join(lines)
```

读入结果：`a=1 b=2 c=a+b d=c-1+a print(c) print(a,b,c)`

2. 编写py_lex.py进行序列标记

需要识别的符号标记**tokens**有以下几个：

```
# Tokens
tokens = ('VARIABLE', 'NUMBER', 'PRINT')
literals = ['=', '+', '-', '*', '(', ')', '{', '}', '<', '>', ',']
```

随后，对tokens中的每个标记定义

定义匹配规则时，需要以**t_**为前缀，紧跟在t_后面的单词，必须**跟标记列表中的某个标记名称对应**，同时使用**正则表达式**

同时定义句子之间的**间隔**和**错误处理**

```
# Define of tokens
def t_NUMBER(t):
    r'[0-9]+'
    return t

def t_PRINT(t):
    r'print'
    return t

def t_VARIABLE(t):
    r'[a-zA-Z]+'
    return t

# Ignored
t_ignore = " \t"

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

lex.lex()
```

3. 编写py_pacc.py进行语法分析

根据实验要求中的文法：

```
expr -> expr + term | expr - term | term

term -> term * factor | term / factor | factor

factor -> id | (expr)
```

对于每个产生式，可以根据**输入的标记数量**和**操作符号**可以判断出具体是哪一条产生式，从而可以生成语法分析树

例如，对于**term -> term * factor | term / factor | factor** 这一个产生式：

- 如果输入标记长度为2，那么一定是**term ->factor**这个产生式
- 如果输入标记长度为4，那么需要判断**第三个字符**是 ***** 还是 **/** 就可以判断是哪一个产生式

具体代码如下：

```
#! /usr/bin/env python
# coding=utf-8
import ply.yacc as yacc
from py_lex import *
from node import node, num_node

# YACC for parsing Python
def p_program(t):
    '''program : statements'''
    if len(t) == 2:
        t[0] = node('[PROGRAM]')
        t[0].add(t[1])

def p_statements(t):
    '''statements : statements statement
                  | statement'''
    if len(t) == 3:
        t[0] = node('[STATEMENTS]')
        t[0].add(t[1])
        t[0].add(t[2])
    elif len(t) == 2:
        t[0] = node('[STATEMENTS]')
        t[0].add(t[1])

def p_statement(t):
    ''' statement : assignment
                  | operation
                  | print'''
    if len(t) == 2:
        t[0] = node('[STATEMENT]')
        t[0].add(t[1])

def p_assignment(t):
    '''assignment : VARIABLE '=' NUMBER'''
    if len(t) == 4:
        t[0] = node('[ASSIGNMENT]')
        t[0].add(node(t[1]))
```

```
        t[0].add(node(t[2]))
        t[0].add(num_node(t[3]))

def p_operation(t):
    '''operation : VARIABLE '=' expr'''
    if len(t) == 4:
        t[0] = node('[OPERATION]')
        t[0].add(node(t[1]))
        t[0].add(node(t[2]))
        t[0].add(t[3])

def p_expr(t):
    '''expr : expr '+' term
            | expr '-' term
            | term
    '''
    t[0] = node('[expr]')
    if len(t) == 2:
        t[0].add(t[1])
    else:
        t[0].add(t[1])
        t[0].add(node(t[2]))
        t[0].add(t[3])

def p_term(t):
    '''term : term '*' factor
            | factor
    '''
    t[0] = node('[term]')
    if (len(t) == 2):
        t[0].add(t[1])
    else:
        t[0].add(t[1])
        t[0].add(node(t[2]))
        t[0].add(t[3])

def p_factor(t):
    '''factor : VARIABLE
              | NUMBER
    '''
    t[0] = node('[factor]')
    if (t[1].isdigit()):
        t[0].add(num_node(eval(t[1])))
    else:
        t[0].add(node(t[1]))

def p_print(t):
    '''print : PRINT '(' values ')' '''
    t[0] = node('[PRINT]')
    t[0].add(node(t[1]))
    t[0].add(node(t[2]))
    t[0].add(t[3])
    t[0].add(node(t[4]))

def p_values(t):
    '''values : values ',' VARIABLE
              | VARIABLE'''
    t[0] = node('[values]')
```

```
    if len(t) == 2:
        t[0].add(node(t[1]))
    else:
        t[0].add(t[1])
        t[0].add(node(t[2]))
        t[0].add(node(t[3]))

def p_error(t):
    print("Syntax error at '%s'" % t.value)

yacc.yacc()
```

其中的结点是自定义的**node**类，其中包含了**结点类型_data**，**子结点列表_children**和**结点的值_value**：

```
#! /usr/bin/env python
#coding=utf-8
class node:
    def __init__(self, data):
        self._data = data
        self._children = []
        self._value=None
    def getdata(self):
        return self._data
    def setvalue(self,value):
        self._value=value
    def getvalue(self):
        return self._value
    def getchild(self,i):
        return self._children[i]
    def getchildren(self):
        return self._children
    def add(self, node):
        self._children.append(node)
    def print_node(self, prefix):
        print('  '*prefix,'+',self._data)
        for child in self._children:
            child.print_node(prefix+1)
def num_node(data):
    t=node(data)
    t.setvalue(float(data))
    return t
```

4. 编写translation.py语法制导翻译

该程序的目的是：基于**深度优先遍历**获取整个语法树的分析结果

在深度优先遍历的同时，需要有一个**v_table**字典来实时存储各个变量的值

同时也需要同步更新上面得出的**语法树的每个结点的值**

v_table定义如下，update_v_table用于更新v_table的值：

```
v_table = {}  # variable table

def update_v_table(name, value):
    v_table[name] = value
```

trans(node)工作原理如下：

首先接收传入的参数node，表示当前深度优先搜索**到达的点**

随后判断当前的node点**在语法树中的类型**，并作出相应操作。

例如，如果当前结点对应的是**statement : VARIABLE '=' NUMBER**时，

我们需要获得NUMBER结点的值，将它赋给VARIABLE结点，并在v_table表中更新变量的实时值

代码如下：

```
def trans(node):
    for c in node.getchildren():
        trans(c)

    # Translation
    # Assignment
    if node.getdata() == '[ASSIGNMENT]':
        ''' statement : VARIABLE '=' NUMBER'''
        value = node.getchild(2).getvalue()
        node.getchild(0).setvalue(value)
        # update v_table
        update_v_table(node.getchild(0).getdata(), value)

    # Operation
    elif node.getdata() == '[OPERATION]':
        '''operation : VARIABLE '=' expr '''
        value = node.getchild(2).getvalue()
        node.getchild(0).setvalue(value)
        # update v_table
        update_v_table(node.getchild(0).getdata(), value)

    elif node.getdata() == '[expr]':
        '''expr : expr '+' term
            | expr '-' term
            | term
        '''
        if (len(node.getchildren()) == 1):
            node.setvalue(node.getchild(0).getvalue())
        else:
            arg0 = node.getchild(0).getvalue()
            arg1 = node.getchild(2).getvalue()
            op = node.getchild(1).getdata()
            if op == '+':
                value = arg0 + arg1
            else:
                value = arg0 - arg1
            # update v_table
            node.setvalue(value)

    elif node.getdata() == '[term]':
        '''term : term '*' factor
```

```
                | term '/' factor
                | factor '''
        if (len(node.getchildren()) == 1):
            node.setvalue(node.getchild(0).getvalue())
        else:
            arg0 = node.getchild(0).getvalue()
            arg1 = node.getchild(2).getvalue()
            op = node.getchild(1).getdata()
            if op == '*':
                value = arg0 * arg1
            else:
                value = arg0 / arg1
            node.setvalue(value)

    elif node.getdata() == '[factor]':
        """factor : id
                  | '(' expr ')'  """
        if (len(node.getchildren()) == 1):
            if isinstance(node.getchild(0).getdata(), int):
                value = float(node.getchild(0).getdata())
            else:
                value = v_table[node.getchild(0).getdata()]
            node.getchild(0).setvalue(value)
            node.setvalue(value)
        else:
            node.setvalue(node.getchild(1).getvalue())

    # Print
    elif node.getdata() == '[values]':
        '''values : values ',' VARIABLE
                  | VARIABLE'''
        if len(node.getchildren()) == 1:
            arg0 = v_table[node.getchild(0).getdata()]
            print()
        else:
            arg0 = v_table[node.getchild(2).getdata()]
        print(arg0, end=" ")
```

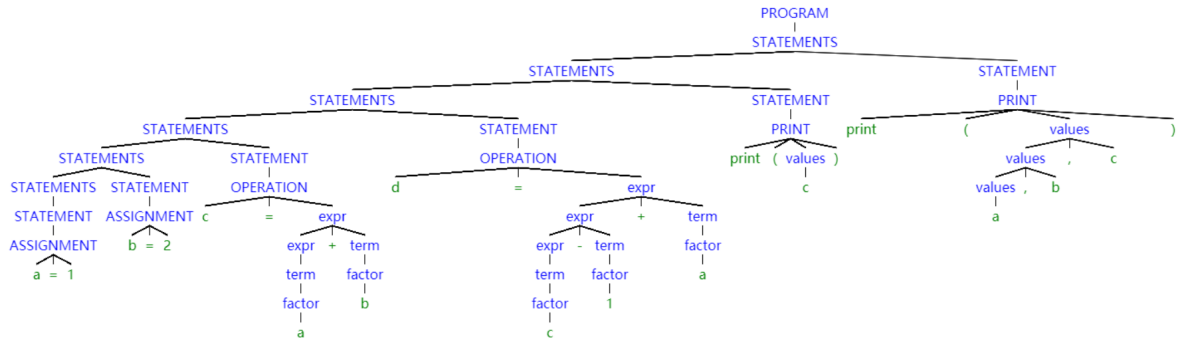
5. 完善主程序

补充准换成指定格式语法树的函数tree_to_str()

```
tree=""
def tree_to_str(node):
    global tree
    if node:
        temp = str(node._data)
        temp = temp.replace("'", "").replace("[", "").replace("]", "")
        tree += temp
    if node._children:
        for i in node._children:
            tree += "["
            tree_to_str(i)
            tree += "]"
```

运行结果

1.语法树




```
+ [PROGRAM]
  + [STATEMENTS]
    + [STATEMENTS]
      + [STATEMENTS]
        + [STATEMENTS]
          + [STATEMENTS]
            + [STATEMENTS]
              + [STATEMENTS]
                + [STATEMENT]
                  + [ASSIGNMENT]
                    + a
                    + =
                    + 1
                + [STATEMENT]
                  + [ASSIGNMENT]
                    + b
                    + =
                    + 2
            + [STATEMENT]
              + [OPERATION]
                + c
                + =
                + [expr]
                  + [expr]
                    + [term]
                      + [factor]
                        + a
                + +
```

```
            + [term]
              + [factor]
                + b
    + [STATEMENT]
      + [OPERATION]
        + d
        + =
        + [expr]
          + [expr]
            + [expr]
              + [term]
                + [factor]
                  + c
            + -
            + [term]
              + [factor]
                + 1
          + +
          + [term]
            + [factor]
              + a
  + [STATEMENT]
    + [PRINT]
      + print
      + (
      + [values]
        + c
```

```
        + )
    + [STATEMENT]
      + [PRINT]
        + print
        + (
        + [values]
          + [values]
            + [values]
              + a
            + ,
            + b
          + ,
          + c
        + )
```

2. example.py经过main.py语法制导后的输出

```
3.0
1.0 2.0 3.0
```

3. v_table保存每个变量值的变量表

```
{'a': 1.0, 'b': 2.0, 'c': 3.0, 'd': 3.0}
```