基于PLY的Python解析-3

实验内容

1. 利用PLY实现的Python程序的解析

*本次学习的语法是**函数语句**，需要注意的是本次使用的语法做了一些改进，不是纯粹的**python2**语法。*

需要结合上次课四则运算的解析程序

（1）示例程序位于**example4/**

（2）需要进行解析的文件为快速排序**quick_sort.py**

（3）解析结果以语法树的形式呈现

2.编程实现语法制导翻译

函数的解析分为2部分：

（1）函数的定义的解析：通过一个函数表来保存每个函数的信息

```
89          elif node.getdata()=='[FUNCTION]':
90              r'''function : DEF VARIABLE '(' VARIABLE ')' '{' statements RETURN VARIABLE '}' '''
91
92              fname=node.getchild(0).getdata()
93              vname=node.getchild(1).getdata()
94              f_table[fname]=(vname,node.getchild(2)) # function_name : (variable_names, function)
```

（2）函数的调用：当函数需要调用时，访问函数表，找到相应的函数名，并进行调用。

```
96          elif node.getdata()=='[RUNFUNCTION]':
97              r'''runfunction : VARIABLE '(' VARIABLE ')' '''
98
99              fname=node.getchild(0).getdata()
100             vname1=node.getchild(1).getdata()
101
102             vname0,fnode=f_table[fname]
103
104             t=Tran()
105             t.v_table[vname0]=self.v_table[vname1]
106
107             t.trans(fnode)
```

环境

- ***windows***
- ***python3.9***
- ***ply包***

运行

```
python main.py
```

实验步骤

1. 编写py_lex.py进行序列标记

通过观察**quick.py**代码，我们可以得出需要解析的tokens有以下这些：

```
tokens = ('VARIABLE', 'NUMBER', 'IF', 'WHILE', 'PRINT', 'DEF', 'RETURN', 'AND',
'LEN')
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**来进行匹配

- 如果变量是一个**字符串**，那么它被解释为一个正则表达式，匹配值是标记的值。
- 如果变量是**函数**，则其文档字符串包含模式，并使用匹配的标记调用该函数。

该函数可以自由地修改序列或返回一个新的序列来代替它的位置。 如果没有返回任何内容，则忽略匹配。 通常该函数只更改“value”属性，它最初是匹配的文本。

同时定义句子之间的**间隔**和**错误处理**。

由于example/中已经给出代码，所以这里不再展示。

2. 编写py_pacc.py进行语法分析

观察**quick_sort.py**,可以得出如下文法：

```
statements -> statements statement
            | statement
statement ->  (empty)
           | assignment
           | operation
           | print
           | if
           | while
           | function
           | run_function
           | return
assignment -> VARIABLE = NUMBER
            | VARIABLE = num_list
            | VARIABLE = VARIABLE
            | VARIABLE [ expression ] = NUMBER
            | VARIABLE = VARIABLE [ expression ]
operation -> VARIABLE = expression
           | VARIABLE + = expression
           | VARIABLE - = expression
           | VARIABLE [ expression ] = expression
expressions -> expression
             | expressions , expression
expression -> term
            | expression + term
            | expression - term
            | LEN ( factor )
term -> factor
      | term * factor
      | term / factor
factor -> NUMBER
```

```
        | VARIABLE
        | ( expression )
        | VARIABLE [ expression ]
variables -> (empty)
           | VARIABLE
           | variables , VARIABLE
print -> PRINT ( VARIABLE )
if -> IF ( condition ) { statements }
function -> DEF VARIABLE ( variables ) { statements }
run_function -> VARIABLE ( expressions )
return -> RETURN variables
condition -> factor > factor
           | factor < factor
           | factor < = factor
           | factor > = factor
conditions -> condition
            | condition AND condition
while -> WHILE ( conditions ) { statements }
num_list -> [ numbers ]
numbers -> NUMBER
         | numbers , NUMBER
```

本次实验在进行翻译条件语句和循环语句时，不能简单的进行深度优先遍历，要对于某些条件节点进行优先翻译。

为了简化步骤，这里将所有类型的括号省略掉，不再对括号进行解析，也不需要将括号翻译到语法树中。

本次实验的关键是对于函数定义和调用的解析，在**py_yacc.py**中代码如下：

```
def p_function(t):
    '''function : DEF VARIABLE '(' variables ')' '{' statements '}' '''
    if len(t) == 9:
        t[0] = node('[FUNCTION]')
        t[0].add(node(t[2]))
        t[0].add(t[4])
        t[0].add(t[7])

def p_run_function(t):
    '''run_function : VARIABLE '(' expressions ')' '''
    if len(t) == 5:
        t[0] = node('[RUN_FUNCTION]')
        t[0].add(node(t[1]))
        t[0].add(t[3])
```

其余代码与实验**基于PLY的Python解析-2**类似，不在这里展示，详细代码见code.zip

3. 完善translation.py进行语法制导翻译

本次实验的关键是完善函数定义和使用的解析部分，关键部分代码如下：

（1） 函数的定义的解析：通过一个函数表来保存每个函数的信息

```
# Function
elif node.getdata() == '[FUNCTION]':
    '''function : DEF VARIABLE '(' variables ')' '{' statements '}' '''
    fname = node.getchild(0).getdata()
    self.trans(node.getchild(1))
    vname = node.getchild(1).getvalue()
    f_table[fname] = (vname, node.getchild(2))   # function_name :
(variable_names, function)
```

（2） 函数的调用：当函数需要调用时，访问函数表，找到相应的函数名，并进行调用。

```
# Run_function
elif node.getdata() == '[RUN_FUNCTION]':
    '''run_function : VARIABLE '(' expressions ')' '''
    fname = node.getchild(0).getdata()
    self.trans(node.getchild(1))
    vname1 = node.getchild(1).getvalue()
    vname0, fnode = f_table[fname]   # function_name : (variable_names,
function)
    t = Tran()
    for i in range(len(vname1)):
        t.v_table[vname0[i]] = vname1[i]
    value = t.trans(fnode)
    if isinstance(value, list):
        node.setvalue(value[1])
```

其余代码与实验**基于PLY的Python解析-2**类似，不在这里展示，详细代码见code.zip

4. 调用编写好的语法制导翻译系统解析目标程序

由于本次实验需要输出指定格式的语法树，所以编写一个get_tree(node)函数来递归获取语法树字符串

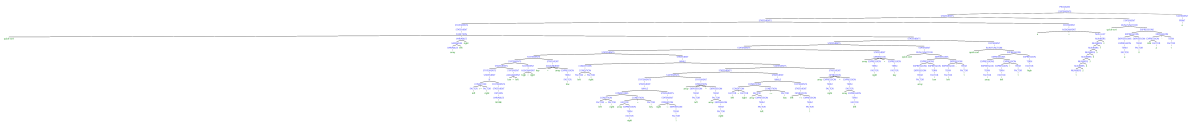
```
#! /usr/bin/env python
#coding=utf-8
from py_yacc import yacc
from util import clear_text
from translation import Tran

def get_tree(node):
    global ans
    if node:
        ans+=str(node.getdata()).replace("[", "").replace("]", "").replace("/'",
"")
    children=node.getchildren()
    if children:
        for child in children:
            ans+='['
            get_tree(child)
            ans+=']'

text=clear_text(open('quick_sort.py','r').read())
# syntax parse
root=yacc.parse(text)
root.print_node(0)
# get tree string
ans=""
```

```
get_tree(root)
print("["+ans+"]")
# translation
t=Tran()
t.trans(root)
print(t.v_table)
```

运行结果



quick_sort.py翻译运行输出：

```
[1.0, 2.0, 3.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

t.v_table：

```
{'a': [1.0, 2.0, 3.0, 3.0, 4.0, 5.0, 6.0, 7.0]}
```