# 基于PLY的Python解析-4

## 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学生信息stu.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

本次课主要需要实现类的解析。主要需要实现：

- （1） 类的解析
- （2） 类中变量的翻译
- （3） 类中函数的翻译

## 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**stu.py**代码，我们可以得出需要解析的tokens有以下这些：

```
tokens = ('VARIABLE', 'NUMBER', 'CLASS', 'SELF', 'STR', 'PRINT', 'DEF')
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**stu.py**，可以得出以下文法：

```
program -> statements
statements -> statements statement
            | statement
statement -> assignment
           | operation
           | print
           | function
           | run_function
           | class
operation -> VARIABLE = expression
           | self = expression
           | VARIABLE + = expression
           | VARIABLE - = expression
           | VARIABLE [ expression ] = expression
assignment -> VARIABLE = NUMBER
            | self = VARIABLE
            | VARIABLE = VARIABLE
            | VARIABLE [ expression ] = NUMBER
            | VARIABLE = VARIABLE [ expression ]
            | VARIABLE = VARIABLE ( expressions )
expressions ->(empty)
             | expression
             | expressions , expression
expression -> term
            | expression + term
            | expression - term
term -> factor
     | term * factor
     | term / factor
factor -> STR
       | self
       | NUMBER
       | VARIABLE
       | ( expression )
       | VARIABLE [ expression ]
print -> PRINT ( variables )
function -> DEF VARIABLE ( variables ) { statements }
         | DEF VARIABLE ( SELF ) { statements }
         | DEF VARIABLE ( SELF , variables ) { statements }
run_function -> VARIABLE ( expressions )
              | VARIABLE . VARIABLE ( expressions )
variables ->(empty)
           | VARIABLE
           | self
           | variables , VARIABLE
           | variables , self
class -> CLASS VARIABLE { statements }
self -> SELF . VARIABLE
```

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和固定字符等省略掉，不再对括号等进行解析，也不需要将它们翻译到语法树中。

本次实验的关键是对于**类的定义和调用**，以及其他文法语句与类的结合如何进行解析：

**类的yacc解析：**

```
def p_class(t):
    '''class : CLASS VARIABLE '{' statements '}' '''
    if len(t) == 6:
        t[0] = node('[CLASS]')
        t[0].add(node(t[2]))
        t[0].add(t[4])

def p_self(t):
    '''self : SELF '.' VARIABLE '''
    if len(t) == 4:
        t[0] = node('[SELF]')
        t[0].add(node(t[3]))
    elif len(t) == 2:
        t[0] = node('[SELF]')
```

与其他文法结合（部分代码）：

```
def p_function(t):
    '''function : DEF VARIABLE '(' variables ')' '{' statements '}'
                | DEF VARIABLE '(' SELF ')' '{' statements '}'
                | DEF VARIABLE '(' SELF ',' variables ')' '{' statements '}' '''
    if len(t) == 9:
        if t[4] == 'self':
            ''' DEF VARIABLE '(' SELF ')' '{' statements '}' '''
            t[0] = node('[FUNCTION]')
            t[0].add(node(t[2]))
            t[0].add(node('[SELF]'))
            t[0].add(t[7])
        elif t[4].getdata() == '[VARIABLES]':
            ''' DEF VARIABLE '(' variables ')' '{' statements '}' '''
            t[0] = node('[FUNCTION]')
            t[0].add(node(t[2]))
            t[0].add(t[4])
            t[0].add(t[6])
            t[0].add(t[9])
    elif len(t) == 11:
        ''' DEF VARIABLE '(' SELF ',' variables ')' '{' statements '}'  '''
        t[0] = node('[FUNCTION]')
        t[0].add(node(t[2]))
        t[0].add(node('[SELF]'))
        t[0].add(t[6])
        t[0].add(t[9])
```

主要是在上次实验**基于PLY的Python解析-3**的基础上增加一个**self**的解析与处理

对于self的引进，我们需要判断它会在哪些位置出现，对于这些新的情况，程序的文法就需要相应更新，例如上面在定义解析函数的部分时，我们需要在引入两条文法语句：

```
function : DEF VARIABLE '(' SELF ')' '{' statements '}'
         | DEF VARIABLE '(' SELF ',' variables ')' '{' statements '}'
```

需要根据有没有**self**来判断这个函数是否是类中所定义的函数，以便于我们在**translation.py**中继续解析处理。

其余部分详见**code.zip**

## 3. 完善translation.py进行语法制导翻译

由于在类中也出现了函数，如果继续沿用实验3的方法，整个程序仅仅用一个v_table字典来存储变量，显然不能满足需求。

所以我们把每个在**stu.py**中出现的类都看作一个单独的**解析单元**，同时整个**stu.py**也是一个大的解析单元。

我们在**translation.py**中定义出这样的**解析单元**的类：**Tran（）**

对于每个解析单元，**Tran**类的实例，定义两个字典来分别存储当前解析单元的**变量的值**（v_table）和当前存储单元的**子函数**（f_table）

每个子函数也就是一个比较小的解析单元，一个大解析单元可能包含很多小解析单元。

如此递归往下，每个解析单元都可以拥有自己的存储空间，于是，问题得以解决。

部分代码如下：

```
class Tran:
    def __init__(self):
        self.v_table = {}  # variable table
        self.f_table = {}  # function table

    def trans(self, node):
        # Translation
        ...
        # Function
        elif node.getdata() == '[FUNCTION]':
            '''function : DEF VARIABLE '(' variables ')' '{' statements '}'
                        | DEF VARIABLE '(' SELF ')' '{' statements '}'
                        | DEF VARIABLE '(' SELF ',' variables ')' '{' statements
'}' '''
            if node.getchild(1).getdata() == '[VARIABLES]':
                ''' DEF VARIABLE '(' variables ')' '{' statements '}' '''
                self.trans(node.getchild(1))
                fname, vname = node.getchild(0).getdata(),
node.getchild(1).getvalue()
                self.f_table[fname] = (vname, node.getchild(2))
            elif node.getchild(1).getdata() == '[SELF]':
                if len(node.getchildren()) == 3:
                    ''' DEF VARIABLE '(' SELF ')' '{' statements '}' '''
                    fname, vname = node.getchild(0).getdata(), []
                    self.f_table[fname] = (vname, node.getchild(2))
                elif len(node.getchildren()) == 4:
                    '''  DEF VARIABLE '(' SELF ',' variables ')' '{' statements
'}' '''
                    self.trans(node.getchild(2))
                    fname, vname = node.getchild(0).getdata(),
node.getchild(2).getvalue()
```

```
                    self.f_table[fname] = (vname, node.getchild(3))
        # Run_function
        elif node.getdata() == '[RUN_FUNCTION]':
            '''run_function : VARIABLE '(' expressions ')'
                            | VARIABLE '.' VARIABLE '(' expressions ')' '''
            if len(node.getchildren()) == 2:
                ''' VARIABLE '(' expressions ')' '''
                self.trans(node.getchild(1))
                fname, vname1 = node.getchild(0).getdata(),
node.getchild(1).getvalue()
                vname0, fnode = self.f_table[fname]
                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])

            elif len(node.getchildren()) == 3:
                ''' VARIABLE '.' VARIABLE '(' expressions ')' '''
                self.trans(node.getchild(2))
                variable, fname, vname1 = node.getchild(0).getdata(),
node.getchild(1).getdata(), node.getchild(
                    2).getvalue()
                c = self.v_table[variable][1]
                vname0, fnode = c.f_table[fname]  # function_name :
(variable_names, function)
                for i in range(len(vname1)):
                    c.v_table[vname0[i]] = vname1[i]
                value = c.trans(fnode)
                if isinstance(value, list):
                    node.setvalue(value[1])
        # Class
        elif node.getdata() == '[CLASS]':
            '''class : CLASS VARIABLE '{' statements '}' '''
            if len(node.getchildren()) == 2:
                '''  CLASS VARIABLE '{' statements '}' '''
                cname = node.getchild(0).getdata()
                t = Tran()
                t.trans(node.getchild(1))
                c_table[cname] = t
        # Self
        elif node.getdata() == '[SELF]':
            '''self : SELF '.' VARIABLE'''
            if len(node.getchildren()) == 1:
                node.setvalue('self.' + node.getchild(0).getdata())
```

**其余详见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

main.py代码：

```
#! /usr/bin/env python
#coding=utf-8
```

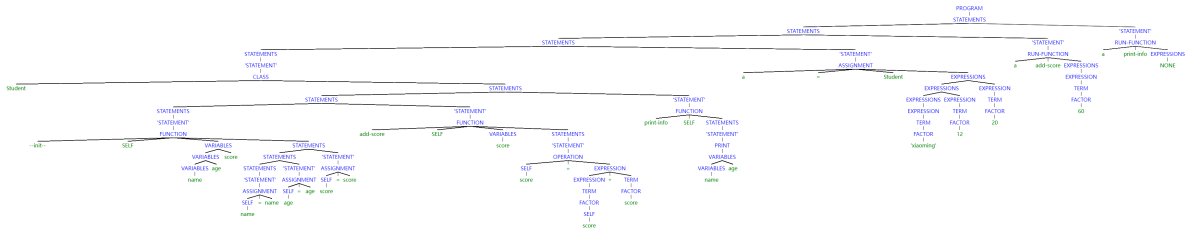
```
from py_yacc import yacc
from util import clear_text
from translation import Tran

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('stu.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

语法树：



解析stu.py运行结果：

```
xiaoming 12.0
```

print(t.v_table)：

```
{'a': ('Student', <translation.Tran object at 0x00000199533583A0>)}
```