**编译原理 实验报告**

|  |  |  |
| --- | --- | --- |
| **评 语（4号楷体）** | **成绩** |  |
| 教 师： 邓岳  年 月 日 | | |

**学院班级： 1803052**

**学生学号：18130500143、18130500139、18130500201**

**学生姓名： 陈凌灏、王若禹、何维杰**

**实验日期： 2020.12**

1. **实验目的**

通过实验实现一个简短绘图语言解释器；通过实验实现一个词法分析器；通过实验实现一个语法分析器；实现函数绘图语言的语义分析。

1. **实验环境**

**操作系统**: Windows10

**集成开发环境**：Visual Studio Code

**Requirement**：Python >= 3.8.0

Numpy >= 1.18.5

Matplotlib >= 3.2.1

1. **实验内容**

**3.1 整个实验的文件目录树**

│-- run.py （测试代码）

│-- test.txt （测试样例）

│-- testparser.py （测试语法分析器）

│-- testscanner.py（测试词法分析器）

├─Parser （语法分析器）

│ │-- parserParser.py

│ │-- parser\_exprnode.py

├─scanner （语法分析器）

│ │-- scannerprocessor.py

│ │-- token.py

└─semantic （语义分析）

│-- semantic.py

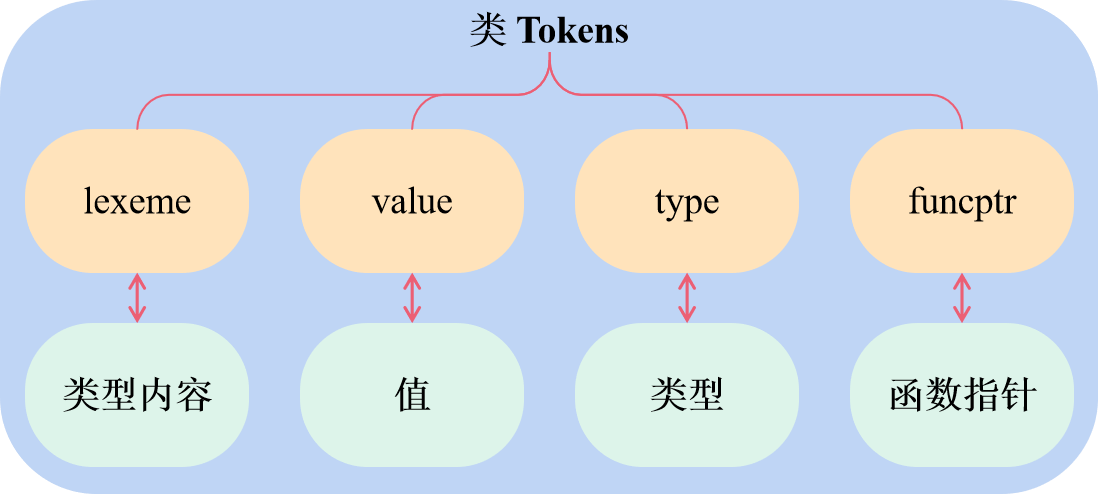
**3.2 词法分析器**

该模块定义了符号表及正规式，如表1所示。其中letter=[a-zA-Z]、digit=[0-9]。

**表1 符号表及正规式**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **符号** | **COMMENT** | **WHITE\_SPACE** | **SEMICO** | **L\_BRACKET** | **R\_BRACKET** |
| **正规式** | "//"|"--" | (" "|\t|\n|\r|\f|\v)+ | ";" | "(" | ")" |
| **符号** | **COMMA** | **PLUS** | **MINUS** | **MUL** | **DIV** |
| **正规式** | "," | "+" | "-" | "\*" | "/" |
| **符号** | **POWER** | **CONST\_ID** | | **ID** | |
| **正规式** | "\*\*" | digit+("."digit\*)? | | letter+(letter|digit)\* | |

对于每一种类型的token，需要定义一种抽象数据类型进行表示存储，本文定义了其数据结构为Tokens类，包括4个成员：type、lexeme、value、funcptr，分别用于表示token的类型、内容，值和函数指针。类的成员如图1所示。

****

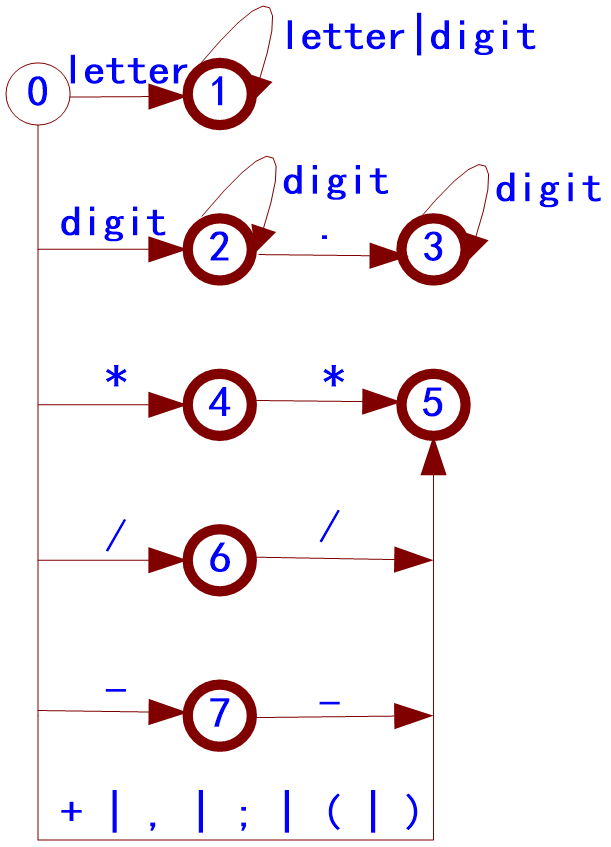
**图1 Tokens类的成员结构图**

涉及到的保留关键字如代码1中所示：

**代码1 系统保留关键字**

|  |  |
| --- | --- |
| **1:** | **enum Token\_Type**{ *// 记号的类别*  ORIGIN, SCALE, ROT, IS, *// 保留字（一字一码）*  TO, STEP, DRAW,FOR, FROM, *// 保留字*  T, *// 参数*  SEMICO, L\_BRACKET, R\_BRACKET, COMMA, *// 分隔符*  PLUS, MINUS, MUL, DIV, POWER, *// 运算符*  FUNC, *// 函数（调用）*  CONST\_ID, *// 常数*  NONTOKEN, *// 空记号（源程序结束）*  ERRTOKEN *// 出错记号（非法输入）*  }; |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |

在词法分析器中，需要确定词法分析器的DFA，其DFA的设计如图2所示。

****

**图2 词法分析器的DFA**

根据DFA确定词法分析的算法。相关算法的实现核心是**scanner**中**scannerprocessor.py**里的GetToken方法代码的编写。**我的设计思路**是：

* **首先**，读取一个非空格的字符进入Char中，并通过addToBuffer函数将其存储在TokenBuffer中（TokenBuffer在读取完一个token之后清空为空串），下面对读取到的这个字符进行分析。相关代码如代码2所示。

**代码2 读取一个字符**

|  |  |
| --- | --- |
| **1:** | **while** **True**:  Char = self.GetChar()  **if** Char == '':  token = st.Tokens(st.Token\_Type.NONTOKEN, Char, 0.0, **None**)  **return** token  **if** Char == '**\n**':  self.LineNo += 1  **if** ~Char.isspace():  **break**  self.addToBuffer(Char) |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |

* 如果这个字符是一个**字母**时，就通过循环识别其是否是字母打头的字母数字串，当不是的时候跳出循环。相关代码如代码3所示。

**代码2 处理字母开头的数字字符串**

|  |  |
| --- | --- |
| **1:** | **if** Char.isalpha(): *# 字母打头的字母数字串*  **while** **True**:  Char = self.GetChar()  **if** Char.isalnum(): *# 是否是字母数字串*  self.addToBuffer(Char) *# 加入token字符串*  **else**:  **break**  self.BackChar(Char)  token = self.JudgeKeyToken() *# 识别token类型*  token.lexme = self.TokenBuffer *# token的内容*  **return** token |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |

* 如果这个字符是一个**数字**时，识别其是否为整数或小数。相关代码如代码4所示。

**代码4 处理小数**

|  |  |
| --- | --- |
| **1:** | **elif** Char.isdigit(): *# 整数或小数*  **while** **True**:  Char = self.GetChar()  **if** Char.isdigit():  self.addToBuffer(Char)  **else**:  **break**  **if** Char == '.': *# 识别小数*  self.addToBuffer(Char)  **while** **True**: *# 识别小数部分*  Char = self.GetChar()  **if** Char.isdigit():  self.addToBuffer(Char)  **else**:  **break**  self.BackChar(Char)  token = st.Tokens(st.Token\_Type.CONST\_ID, self.TokenBuffer,float(self.TokenBuffer), **None**)  **return** token |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |
| **12:** |
| **13:** |
| **14:** |
| **15:** |
| **16:** |
| **17:** |
|  |
| **18:** |

* 如果这个字符是：**+、/、，、**；时，直接识别。**这里存在一些例外**，\*、-、/运算符均是有可能**造成二义**的。\*\*表示乘方运算，--和//表示行注释。这种情况下，需要再往下读一个字符，如果还是一样的话，表示乘方或注释；如果不一样的话，表示是乘法、减法、除法运算符。解决二义性相关部分代码如代码5所示（以**乘方**为例）。

**代码5 解决二义性相关部分代码**

|  |  |
| --- | --- |
| **1:** | **elif** Char == '\*':  Char = self.GetChar()  **if** Char == '\*': *# 乘方*  token = st.Tokens(st.Token\_Type.POWER, '\*\*', 0.0, **None**)  **else**: *# 乘法运算符*  self.BackChar(Char)  token = st.Tokens(st.Token\_Type.MUL, '\*', 0.0, **None**) |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |

**我的实现细节：**

**1、在token表中的构造过程中，本文使用的是numpy库中的函数，而不是使用math库中的函数。**原因是：本文对于需要绘图的x轴变量生成的是一个numpy.array类型数组，y轴的变量据此再通过函数调用计算出数组，针对数组调用函数计算，就要使用numpy的函数(如cos、sin)对numpy.array进行计算。

**2、在词法分析模块Scanner中，本文定义了一个方法BackChar，实现文件读取过程中的字符回退操作。**设计该方法的目的是在识别字符串的过程中，是按字符进行读取，因此需要在遇到空格时候或者非DFA路径中字符的时候方可知道读取完一个token，即需要超前读取一个字符。而在下一次使用的时候又要回退这个超前读取的字符，重新从超前读取的字符开始读取。使用的代码为：self.filepointer.seek(self.filepointer.tell() - 1)。

**3.3 语法分析器**

语法分析器是本次实验过程中最难的部分。针对词法分析器提取的token，对其进行语法分析。根据函数绘图语言语法，得到其遵循的**EBNF文法**为文法1。

**文法1 函数绘图语言遵循的EBNF文法**

|  |
| --- |
| **Program →** { Statement SEMICO }  **Statement →** OriginStatment | ScaleStatment | RotStatment |ForStatment  **OriginStatment →** ORIGIN IS L\_BRACKET Expression COMMA ExpressionR\_BRACKET  **ScaleStatment →** SCALE IS L\_BRACKET Expression COMMA Expression R\_BRACKET  **ForStatment →** FOR T FROM Expression TO Expression STEP Expression DRAW L\_BRACKET Expression COMMA Expression R\_BRACKET  **RotStatment →** ROT IS Expression  **Expression →** Term { ( PLUS | MINUS ) Term }  **Term →** Factor { ( MUL | DIV ) Factor }  **Factor →** ( PLUS | MINUS ) Factor | Component  **Component →** Atom [ POWER Component ]  **Atom →** CONST\_ID  | T  | FUNC L\_BRACKET Expression R\_BRACKET  | L\_BRACKET Expression R\_BRACKET |

为了构建语法树，本文设计了语法树结点的抽象数据类型为ExprNode，可以灵活实现函数、运算符、操作数的定义，其定义如表2所示。如果该结点是双目运算符，则分配左右孩子;如果是常数/变量结点，就是叶子结点，只需要value(变量是Tvalue，一个np.array);如果是函数类型，则分配函数运算和操作数middle。

**表2 ExprNode的数据结构**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **结点类型** | **数据成员1** | **数据成员2** | **数据成员3** | **数据成员4** |
| **双目运算符** | item | left | right | value |
| **常数/变量** | item | value | — | — |
| **函数** | item | Functionpointer | middle | value |

对于Parser结点类，本文设计了一个成员方法GetValue，在构建双目运算符和函数为父节点时，将调用它求得父节点的值（相当于直接规约为计算结果），**实现更加简单**，该方法的定义如代码6所示。

**代码6 GetValue方法实现**

|  |  |
| --- | --- |
| **1:** | **def** GetValue(self): *# 依据不同的操作符进行运算*  **if** self.item == st.Token\_Type.PLUS:  self.value = self.left.value + self.right.value  **elif** self.item == st.Token\_Type.MUL:  self.value = self.left.value \* self.right.value  **elif** self.value == st.Token\_Type.POWER:  self.value = self.left.value \*\* self.right.value  **elif** self.item == st.Token\_Type.DIV:  self.value = self.left.value / self.right.value  **elif** self.item == st.Token\_Type.MINUS:  self.value = self.left.value - self.right.value  **elif** self.item == st.Token\_Type.FUNC:  self.value = self.functionpointer(self.middle.value)  **return** self.value |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |
| **12:** |
| **13:** |
| **14:** |

在Parser类中，本文定义了一个成员方法Expression，用于计算表达式。在建树的过程中，将left指向新构造的结点，通过此进行递归地构造，最终将left结点返回。其实现如代码7所示。

**代码7 Expression方法的实现**

|  |  |
| --- | --- |
| **1:** | **def** Expression(self):  self.getinto("The Expression")  left = self.Term()  **while** self.token.type == st.Token\_Type.PLUS **or** self.token.type== st.Token\_Type.MINUS:  tmp = self.token.type  self.MatchToken(tmp)  right = self.Term()  left = self.MakeExprNode(tmp, left, right) *# 递归建树*  self.printtree(left) *# 打印表达式语法树*  self.exitmodule("The Expression")  **return** left |
| **2:** |
| **3:** |
| **4:** |
|  |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |

**我实现的亮点：**

* 我在实现的过程中通过MatchToken方法预取下一个token，实现和此法分析器的连接，返回给语法分析器token。特别值得一提的是，我的方法可以通过self.scanner.LineNo计算行号，并且可以在token.type != ttype时，进行报错.。

**代码8 MatchToken实现与词法分析器连接并报错部分代码实现**

|  |  |
| --- | --- |
| **1:** | **def** MatchToken(self, ttype):  **if** self.token.type != ttype:  print(self.token.type)  print("Error line number：", self.scanner.LineNo," dosen't expected token ", self.token.lexeme)  self.scanner.Exit\_Scanner()  sys.exit(1)  **if** ttype == st.Token\_Type.SEMICO:  self.scanner.filepointer.readline()  last = self.scanner.filepointer.tell()  next\_line=self.scanner.filepointer.readline()*#读取到分号*  **if** len(next\_line) == 0:  **return**  **else**:  self.scanner.filepointer.seek(last)  self.GetToken() *# 取下个操作符放入self.token* |
| **2:** |
| **3:** |
| **4:** |
|  |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |
| **12:** |
| **13:** |
| **14:** |
| **15:** |

* 在实现过程中，为了可以清晰地看到Expression的语法树结构，我们通过设计前序遍历算法打印语法树，相关算法的简化形式为代码9所示。

**代码9 打印语法树算法实现**

|  |  |
| --- | --- |
| **1:** | **def** PreorderPrintTree(self, node, index\_range):  **for** i **in** range(index\_range):  print('--- ', end=' ')  **if** node.item == st.Token\_Type.PLUS:  print('+ ')  **elif** node.item == st.Token\_Type.MINUS:  print('- ')  ......  *# 省略了部分条件分支*  **else**:  print("Type error.")  sys.exit(0)  **if** node.item == st.Token\_Type.CONST\_ID **or** node.item == st.Token\_Type.T:  **return**  **elif** node.item == st.Token\_Type.FUNC:  self.PreorderPrintTree(node.middle, index\_range + 1)  **else**:  self.PreorderPrintTree(node.left, index\_range + 1)  self.PreorderPrintTree(node.right, index\_range + 1) |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |
| **11:** |
| **12:** |
| **13:** |
| **14:** |
| **15:** |
| **16:** |
| **17:** |
| **18:** |
| **19:** |

**3.4 语义分析**

在解释器的实现中，语义分析的部分和语法分析的部分紧密相连，因此，我们设计的**语义分析类Semantic类是对语法分析类Parser的继承**。这样的操作充分利用了面向对象的特性，使代码更加简介。

在Semantic类中，我对2个方法进行了重写。分别是ForStatement方法和分析的入口方法Parser方法。根据ROT、SCALE、Origin方法提取的参数翻入对象的成员变量中，在通过FOR语句实现绘图。

在Parser中，我实现了绘图功能，将存储于成员变量中的内容进行绘图，只增加的一个语句：plt.show()进行展示。

在ForStatement方法中，我添加了Draw()方法进行绘。其中setting方法实现了对函数横纵坐标的旋转、平移等转化的实现，相关代码如代码10 所示。最后通过self.ax.scatter(x, y)绘制散点图。

**代码10 setting方法的实现**

|  |  |
| --- | --- |
| **1:** | **def** setting(self, x, y):  x \*= self.x\_scale  y \*= self.y\_scale  y = -y *# y轴是反向的*  tmp = x \* np.cos(self.rot) + y \* np.sin(self.rot)  y = - x \* np.sin(self.rot) + y \* np.cos(self.rot)  x = tmp *# 这里不能写两句，要分开写，用tmp寄存*  x += self.x\_origin  y += self.y\_origin  **return** x, y |
| **2:** |
| **3:** |
| **4:** |
| **5:** |
| **6:** |
| **7:** |
| **8:** |
| **9:** |
| **10:** |

**我的实现细节：**

在代码10中，对x,y坐标的旋转遵循的公式如公式1所示，在实现过程中，**必须使用中间变量tmp对x进行暂存**。如果不进行暂存，那么y在进行旋转计算的过程中如果直接使用x进行计算，那么此时的x已经不是x了，而是x’。

. (1)

**3.5 实验结果测试**

**我测试了许多的测试样例，通过不同的语句进行绘图，并加入了注释和复杂的表达式计算和绘图指令。**

**①样例1**

rot is 0;

origin is (0, 0);

scale is (2,2+2\*5+9-1);

for T from 1 to 3\*(1+3\*\*3) step 1 draw (t,-ln(t));

scale is (20,0.1);

for T from 0 to 8 step 0.1 draw (t,exp(t));

scale is (2,1);

for T from 0 to 300 step 1 draw (t,0);

for T from 0 to 300 step 1 draw (0,t);

for T from 0 to 120 step 1 draw (t,t); --sss

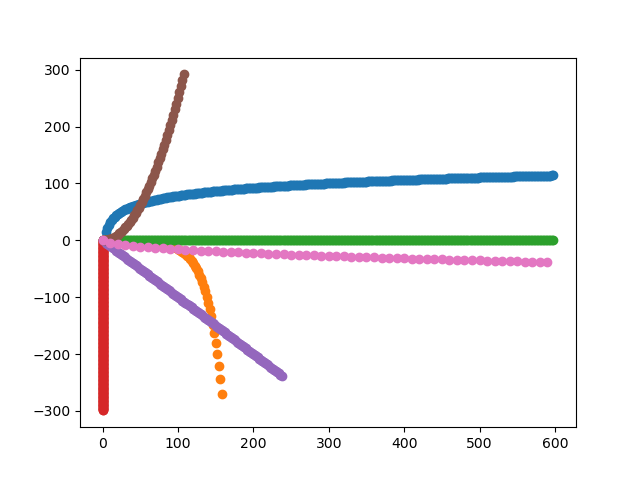
scale is (2,0.1);

for T from 0 to 55 step 1 draw (t,-(t\*t)); //sss

scale is (10,5);

for T from 0 to 60 step 1 draw (t,sqrt(t));

实验结果如图3所示：

****

**图3 测试样例1的测试结果**

**②样例2**

rot is 0;

origin is (0, 0);

scale is (100,100);

for t from 0 to pi\*20 step Pi/50 draw ((1-1/(10/7))\*cos(T)+1/(10/7)\*cos(-T\*(((10/7)-1))), (1-1/(10/7))\*sin(T)+1/(10/7)\*sin(-T\*((10/7)-1)));

origin is (500,500);

scale is (100,100/3);

rot is pi/2;

for T from -pi to pi step pi/50 draw (cos(t),sin(t));

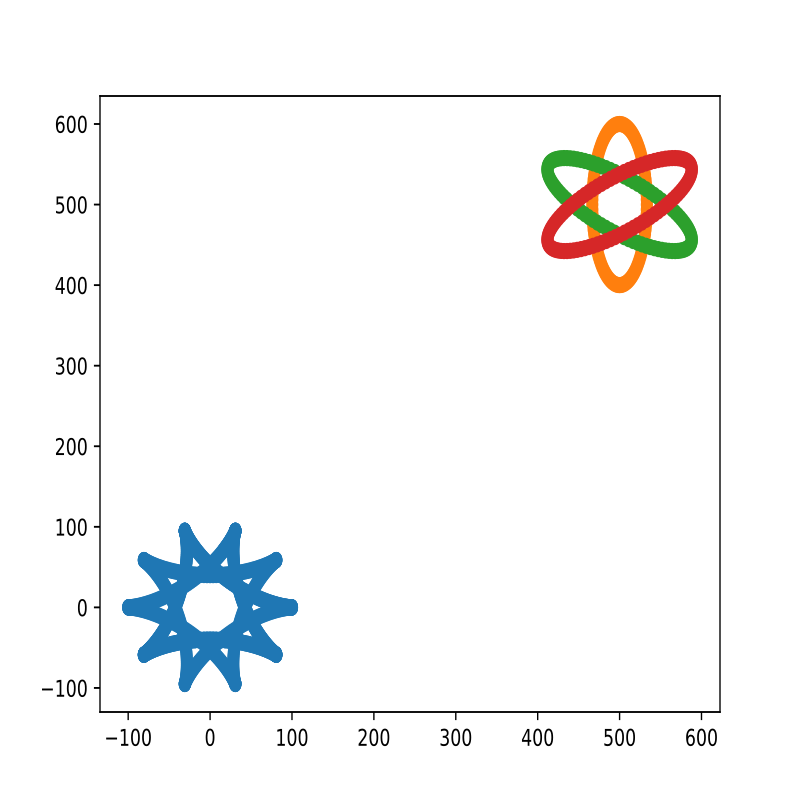
rot is pi/2 + 2\*pi/3;

for T from -pi to pi step pi/50 draw (cos(t),sin(t));

rot is pi/2 - 2\*pi/3;

for T from -pi to pi step pi/50 draw (cos(t),sin(t));

实验结果如图4所示：



**图4 测试样例2的测试结果**

1. **心得体会**

该实验的主要目标是使用通过编程语言来实现功能性绘图语言解释器。我之所以选择Python作为程序设计语言进行设计，是因为一方面，我认为我熟悉Python相关的面向对象方法；另一方面，也是因为Python提供了绘图和矩阵运算的第三方库，例如numpy和matplotlib。

整个实现过程最难的部分就是**语法分析器**的部分，这里涉及到函数之间的简介递归调用关系，非常难以实现，在设计函数之间的调用关系的时候还需要非常精心地设计一些数据结构作为函数返回值和树结点。虽然平时也有刷**LeetCode**等算法平台，但是这么大型的算法编写起来还是相当有挑战性的。完成了这次实验，让我收获巨大，非常有成就感。

通过这次实验，我更加透彻地理解了**面向对象程序设计的编程思想**，虽然平时也有承担一些项目的开发工作，但是并没有非常注重代码的质量、模块间的耦合关系等一些软件工程中需要注意的问题。这次实验加深了我对这部分的认识和理解。

在程序设计过程中，**接口是封装**尤为重要，并且调用模块的成员也更为方便，所以在代码编写过程中接口的封装非常重要。

我的抽象数据类型是经过**精心设计**的，我的语义分析模块就是对语法分析模块数据类型的继承，并进行方法的重写，提高了代码的质量。

本次实验提升了我发现问题并解决问题的能力，收益颇多。