**编译原理 实验报告**

|  |  |  |
| --- | --- | --- |
| **评 语** | **成绩** |  |
| 教 师： 邓岳  年 月 日 | | |

**学院班级： 1613012**

**学生学号： 16130120191、16130120158**

**学生姓名： 罗阳豪、杨轹文**

**实验日期： 2018.12**

# 一 实验目的

通过实验加深对编译器构造原理和方法的理解，巩固所学知识。

1. 会用正规式设计简单语言的词法；
2. 会用产生式设计简单语言的语法；
3. 会用递归下降子程序编写语言的解释器。

# 二 实验环境

CPython 3.6.7

# 三 实验内容

因为对于这个函数绘图语言本身的设计已经很详细了，所以主要描述一下解释器程序的设计。

## 软件体系结构设计

### 技术选型

首先这个解释器是用 Python 编写的，考虑到程序编译是计算密集型应用，而 Python 又不是执行性能非常出色的语言，为什么我们还会选择 Python 作为该解释器的实现语言呢。我们主要有以下几点考虑

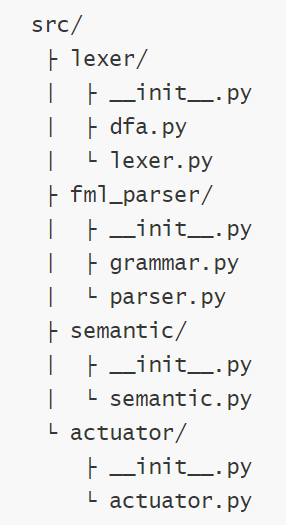
1. Python对于字符串处理操作比较方便。因为解释器是需要与源代码交互的，源代码作为字符串输入，所以Python对字符串处理的便利性的优势就很明显。
2. Python 的类型系统比较灵活。因为解释器实现上需要实现几种数据结构，而 Python 自带的类型就可以很方便地实现这些数据结构，比如 Python的字典比起其他语言的结构体、Map、类都要方便，Python 的元组可以通过嵌套很方便地实现树形结构。Python的列表可以很容易实现 栈、队列。这些类型为实现模块间交互的数据结构提供了非常多的便利。
3. Python 本身就是一门解释型语言，有强大的运行时异常机制，这对于简化我们实现的解释器的异常处理非常有帮助
4. Python 有非常简单易用而且强大的图像处理和绘图库（PIL）

基于以上几点考虑，我们选择了Python。而且考虑到重要的在于了解程序编译的原理，而不在于实现一个真正用于生产环境的解释器。

对于绘图工具，我们选择了pillow，不同于Python其他的很多绘图库，比如matplotlib，这些库主要是用于数据可视化的，而pillow就是一个纯粹的图像处理与绘图库，而且他绘图的方式可以很好地契合题目要求，即给出点坐标，逐点绘制。

### 模块设计

该解释器项目目录如下



我们将程序按照功能分为4个模块，lexer 词法分析器，fml\_parser 语法分析器，semantic 语义分析器和actuator执行器，和一个主程序main.py

dfa.py定义了枚举类Token和每个DFA的状态迁移表、和DFA类

lexer.py 定义了词法分析器类 Lexer

grammar.py 定义了枚举类 Terminals和NonTerminals，表示终结符和非终结符

parser.py 定义了语法分析器类 Parser，该类对递归下降子程序做了简单封装

semantic.py 定义了语义分析器类 Semantic，和枚举类 Operation 表示6种操作（设置origin、设置rot、设置scale、设置绘图颜色、设置背景颜色、画点）

actuator.py 定义执行器类 Actuator，负责维护origin、rot、scale、color、background五个属性的值，和计算记录要绘制的点的真实坐标，提供生成所绘制图片的方法

### 接口传递数据结构

由于有4个模块，他们由主程序逐一调用，前一模块的输出将作为后一模块的输入，因此需要设计一些统一的数据结构使各个模块能够正常交互

1. **源代码**

源代码是词法分析器的输入，直接以字符串的形式输入。但是为了能够实现缓冲输入，允许源代码任意截断，按顺序调用lexer.append()方法分批或逐一输入，可以是字符串或字符串的列表的形式。

1. **记号流**

记号流是词法分析器的输出和语法分析器的输入。首先将记号表示为元组 (Token.TYPE, value) Token.TYPE 是枚举类 Token 的任一取值，value是所识别的源码中的单词（字符串格式）。记号流表示为记号的列表。

1. **语法树**

语法树是语法分析器的输出和语义分析器的输入。为了表示树形结构，用嵌套的二元组表示。非终结符节点表示为 (NonTerminal.TYPE, (...)) ，第一个元素是非终结符类型，第二个元素是包含的子树的元组。终结符节点表示为 (Terminals.TYPE, token) ，第一个元素是终结符类型，第二个元素是表示记号的二元组（表示该终结符的值）。由此按照语法树逐层嵌套。

1. **操作序列**

操作序列是语义分析器的输出和执行器的输入。首先表示原子操作，原子操作被分为5类，表示为元组 (Operation.TYPE, arg1, arg2, ...) ，第一个元素是操作类型，可以是设置origin、设置rot、设置scale、设置背景颜色、设置绘图颜色和绘点，根据操作类型不同，后面给出1 - 3 个参数，参数都是整数或浮点数，比如点的坐标，缩放的值等。然后操作序列是按照操作次序排列的原子操作的列表。

## 词法分析器模块设计

词法分析器以源代码作为输入，输出记号流，识别记号的工具主要就是DFA。我们实现的词法分析器是一种表驱动型的词法分析器。在初始化一个词法分析器时，他会初始化一个DFA集合，每个DFA对应一种记号，他们有 move() 成员方法。

词法分析器在运行时需要维护2个队列，和3个列表，分别是输入队列（字符串队列），输出队列（记号队列），已处理源码列表，已识别记号列表，已发现错误记号列表。列表是留作复查用的，实际与模块外部交互的主要是输入队列和输出队列。

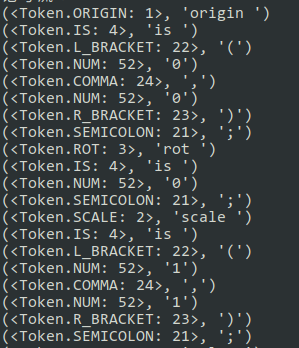
主程序通过调用词法分析器的append()方法向词法分析器输入一个或一批字符串（源码），词法分析器会将输入加入到输入队列的末尾，并且启动一次词法分析。

词法分析时，拷贝DFA集创建“活跃DFA集”，对活跃DFA集中的DFA逐一调用move() 方法，若move失败则从活跃DFA集移除，若move进入接受状态集则做一次记录，当所有DFA都被移除出活跃集时，一次记号识别完成，将缓存的记号记录中最长的一个加入到输出队列尾部。循环执行上述过程，直到输入队列为空，将未识别完全的字符放回输入队列尾部，结束此次分析。

在最后一次append时，在参数中设置eof字段为True，则词法分析器会将最终未能完整识别的字符放入错误记号中，并在输出队列尾部加入空记号表示程序结束。

主程序通过检查错误记号列表可以知道是否有词法错误，以决定下一步操作。

词法分析器部分输出如下



## 语法分析器模块设计

语法分析器以记号流作为输入，输出语法树。为了简便起见，我们是用递归下降子程序实现的。由于简化的，消除左递归、左因子、二义性的文法的EBNF描述已经在文档给出了。所以这部分的实现只需要写出递归下降的子程序。

语法分析器部分输出结果如下：

图片包含 文字, 就坐

自动生成的说明

## 语义分析器模块设计

语义分析器以语法树为输入，输出中间代码。所谓中间代码并没有确定的格式，所以结合该解释器的功能，我们设计了我们的中间代码。将程序划分为6种原子操作，设置origin、设置scale、设置rot、设置背景颜色、设置绘图颜色，绘制点。其中前5种操作对应5种语句，最后一种操作是for语句的展开。

语义分析器遍历语法树，对表达式求值，展开for语句，得到原子操作的序列。

语义分析器部分输出如下：

图片包含 文字

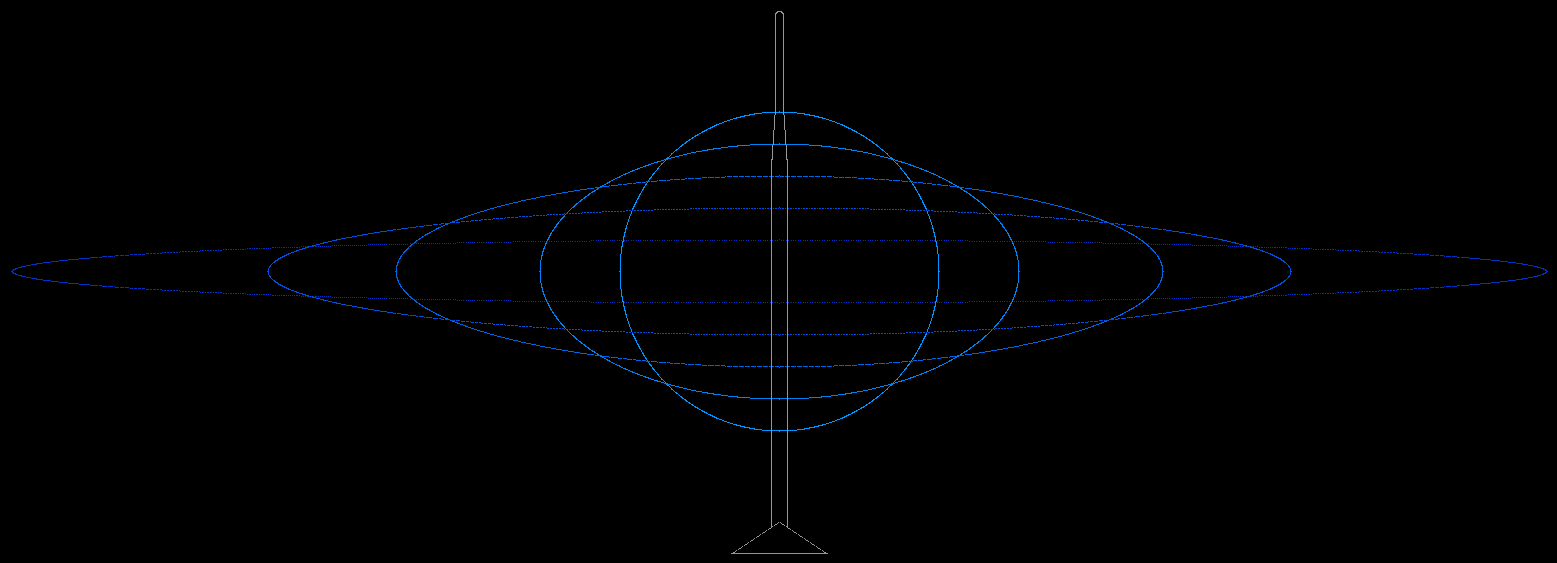
自动生成的说明

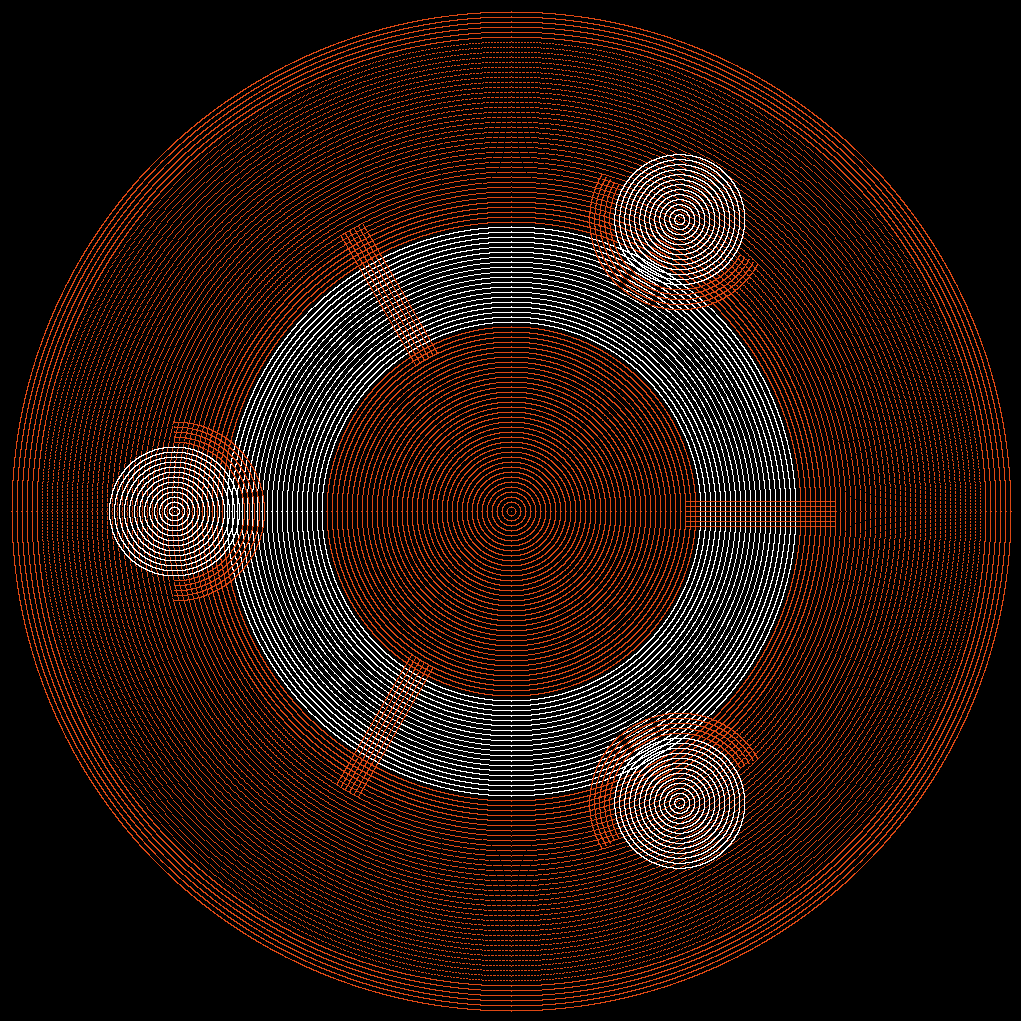
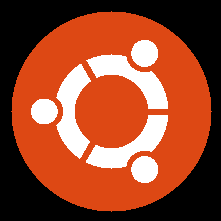
## 执行器模块设计

执行器以操作序列为输入，执行绘图操作。前五种操作是对应5种属性的，属性通过成员变量维护。第6种操作，即绘制点，有两个参数，对应点的坐标。执行器首先会将该点坐标根据origin、rot、scale转换为绘图所用坐标系的坐标，然后记录在绘制点队列中。

当调用执行器的create\_image() 方法时，执行器会根据所有点的坐标的取值范围，生成一个能够容纳所有点的空白图像，然后给每个点加上一个坐标偏移，使其坐标为正（即能落在图片中），然后调用绘图库逐点绘制，保存并显示图片。

绘制图片效果展示：

图片包含 烟火, 户外艺术系列

自动生成的说明

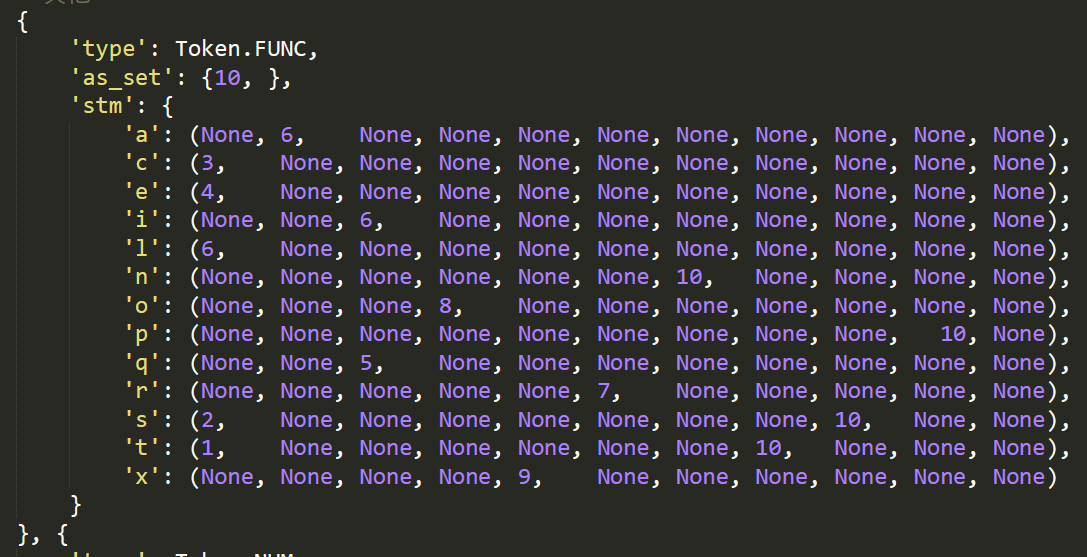
# 四 心得体会

第一次写解释器还是十分有挑战的，期间主要遇到这么几个问题

## DFA的表示方法

由于不想硬编码DFA，所以期望将DFA作为一个类，通过“某些数据”实例化一个能够识别某种特定记号的DFA。但是如何表示这些“数据”就是个问题。

首先考虑一个DFA至关重要的几个参数。首先DFA只有1个初态，所以不需要显式地指明初态，我们定义0状态为初态。然后是字典，由于我们可以在状态迁移表中隐含所接受的输入字符，所以这一部分也不需要指定，那么就只剩下状态迁移表和接受状态集了。接受状态集直接用Python的内建类型set表示。而状态迁移表就比较麻烦，首先可以用二维列表表示，但是二维列表就还需要额外的表头，而且每次顺序查找也很费时。所以我们考虑DFA工作时的情形，move()函数接受一个字符作为参数，然后查找当前状态下，该输入，应该跳转到什么状态。那么我们可以这么实现，将输入字符作为map的键，然后当前状态对应的下一状态存储在一个一维列表中。这样，一个是哈希、一个是直接寻址，就能定位到状态迁移表的某行某列，效率上不错。下面是我们实现的DFA初始化数据的表示的一个例子



'type' 是该DFA识别的记号类型，'as\_set' 是接受状态集（Access State Set）， 'stm' 是状态迁移矩阵。

Python 灵活的内建类型也能很方便地支持我们的这个设计

不足之处的话，一个在于对于比较复杂的DFA这样描述的状态转移矩阵看起来会非常庞大，实际上这个表绝大部分都是None，所以应该考虑作为稀疏矩阵，或者树形结构存储的。

## 错误记号的处理

词法分析的时候遇到词法错误如何处理。首先先考虑要达到什么效果，我们希望的是，当遇到词法错误，就跳过若干字符，直到遇到一个空字符（空格或者换行），并且把跳过的字符记录为一个错误记号，然后继续进行词法分析。如何实现这种效果呢。

我们使用了一种比较有趣的方法，我们尽量不把错误记号特殊处理，像其他记号一样，我们为错误记号定义一个DFA，这个DFA将会匹配从任意非空字符开始的任意非空符字符串，这个DFA会跟其他DFA一同对输入进行匹配。但是在选择最后识别成功的记号时，我们做一个差别对待，即非错误记号优于错误记号，长的记号优于短的。这样，当不能识别出任何一个正常记号时，就会被记录为错误记号。

好处在于编码简单，我们仅仅加了一个DFA，和一两行判断，其余地方我们对正常和错误记号都一样处理。

考虑这样设计的不足的话，我觉得主要是，异常处理机制应该单独作为一个模块，并且能够有一种切实有效的方法在解释的全过程给出错误和定位。这样的话，我们现在的设计就与词法分析器的耦合过于紧密，不易于后期对这方面功能的完善。

## 中间代码的优化

基于多种考虑，我们没有使用语法制导的语义分析。所以我们有一个与语法分析器完全独立的语义分析器模块，并且他能够产生我们定义的中间代码。这原本就为优化提供了很多空间，但是由于时间限制，很可惜的是我们没有在这方面做任何工作。

事实上，现在再来考虑的话，可以优化的空间是很多的。

优化首先在于减少绘点的数量，我们实现的语义分析器是忠实地展开for语句生成绘点指令的，但是可能会有这么几种可能。一个是for语句的step设置过小，导致两个点的坐标在取整（因为像素不能有半个）后实际上是一个点，那么这种绘制就是没有必要的。这个是相对容易发现的，还有比较隐秘的在于，两个相交的曲线，或者两个重叠的曲线，他们属于不同的for语句，这里似乎可以节省不少。还有包括没有意义的反复设置属性，这些模式应该都被尽量发现，并且消除。

另一个优化方向是对于最终绘制效果的，比如适当增加点，或调整点的位置，使绘制的曲线的锯齿没那么明显，更圆润、清晰，而这方面其实也是字符渲染经常要面对的问题。