

# 编译原理实验报告

## # Preface

本文是我们小组的三次编译原理实验报告综合而成的一篇总报告。

小组成员及学号：

- 陈逸飞-20009101290
- 刘旺旺-20009200053
- 林育铭-20009200211

## # 任务与目的

### 任务

- 题目：为函数式绘图语言编写一个解释器，要求输入为用函数绘图语言编写的源程序，若源程序中存在语法错误和语义错误则输出错误信息，若源程序编写正确则输出正确绘制的图像。

- 构建 词法分析器 对绘图源程序中的记号进行识别（可将记号的信息显示出来）。
- 构建 语法分析器 对记号流的语句进行识别（可将语句记号识别出来）。
- 进行 语句翻译，将绘图语句所描述的图像绘制并输出。

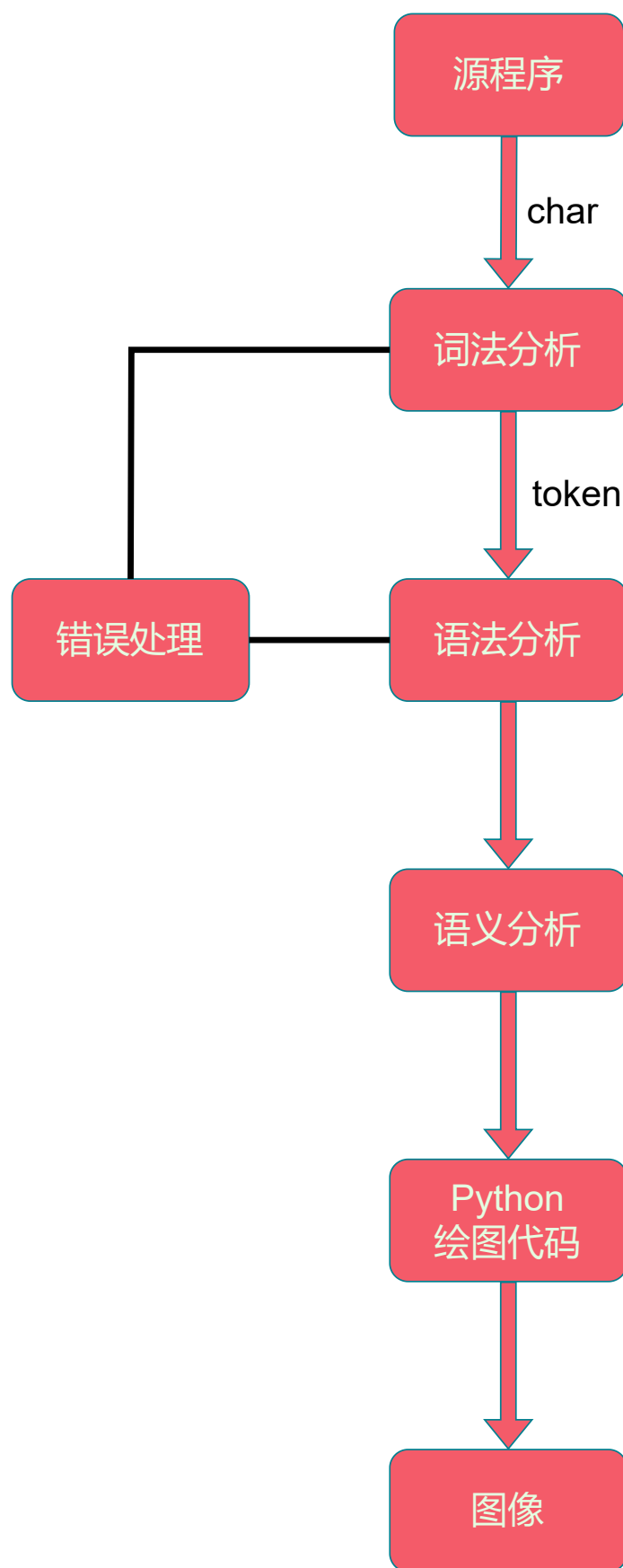
## 目的

- 通过自己编写解释器，掌握语言分析基本方法。
- 通过构建**词法分析器Scanner**学习
  - 记号 的设计
  - 正规式 的书写
  - 正规式-NFA-DFA-最小DFA 的 构建、转换 过程
  - DFA的 存图方式 和 状态转移算法
- 通过构建**语法分析器Parser**学习
  - 文法 的 初步构造 以及无二义&消除左递归和提取左因子后的文法的 改写，使其适合 递归下降分析
  - 适用于存放表达式的 语法树 的 结构设计和构建
  - 递归下降子程序 的设计
  - 递归下降分析记号流是否符合文法 的过程
- 通过构建**语义分析器Semantics**学习
  - 文法符号的 属性设计
  - 语法树 的计算 算法
  - 语法制导翻译的 辅助函数设计
  - 产生式的 语义规则设计

## # 软件设计

---

# 总体结构



# 词法分析器Scanner

该部分使用C++编写，完整代码见附件 `scanner.h` 和 `scanner.cpp`。

## Token

### Token Type

- `CONST_ID`：数值字面量 和 标识符形式的常量名 称为 常数 (`CONST_ID`)。
- `T`：本绘图语言中唯一的、已经被定义好的 变量名`T` 被称为 参数 (`T`)，它也是一个表达式。
- `FUNC`：指向某个函数地址的函数名 被称为 函数 (`FUNC`)。
- 保留字：语句中具有固定含义的标识符，包括：`ORIGIN`, `SCALE`, `ROT`, `IS`, `FOR`, `FROM`, `TO`, `STEP`, `DRAW`
- 运算符：`PLUS('+')`, `MINUS('-')`, `MUL('*')`, `DIV('/')`, `POWER '**')`
- 分隔符：`SEMICO(';')`, `COMMA(',')`, `L_BRACKET('(')`, `R_BRACKET('')`
- `COMMENT`：从`/**`或`--`后的第一个字符开始到行末均为注释内容

### Token Structure

Token的结构包括：

- `type`：表示token的类别
- `lexeme`：表示token的字符串值
- `value`：如果token为常数则表示常数的数值
- `funcPtr`：如果token为函数名则表示指向的函数地址
- `row`：token在源程序中的行号

```

1 struct token
2 {
3     tokenType type; //token类别
4     std::string lexeme; //token对应的字符串值
5     double value; //如果token为常数则表示常数的数值
6     double (*funcPtr)(double); //如果token为函数名则表示指向的
    函数地址
7     int row; //token所在行号
8 };

```

## Token Tab

```

1 struct token tokenTab[] =
2 {
3     {CONST_ID, "PI", 3.1415926, NULL}, //0
4     {CONST_ID, "E", 2.71828, NULL}, //1
5     {T, "T", 0.0, NULL}, //2
6     {FUNC, "SIN", 0.0, sin}, //3
7     {FUNC, "COS", 0.0, cos}, //4
8     {FUNC, "TAN", 0.0, tan}, //5
9     {FUNC, "LN", 0.0, log}, //6
10    {FUNC, "EXP", 0.0, exp}, //7
11    {FUNC, "SQRT", 0.0, sqrt}, //8
12    {ORIGIN, "ORIGIN", 0.0, NULL}, //9
13    {SCALE, "SCALE", 0.0, NULL}, //10
14    {ROT, "ROT", 0.0, NULL}, //11
15    {IS, "IS", 0.0, NULL}, //12
16    {FOR, "FOR", 0.0, NULL}, //13
17    {FROM, "FROM", 0.0, NULL}, //14
18    {TO, "TO", 0.0, NULL}, //15
19    {STEP, "STEP", 0.0, NULL}, //16
20    {DRAW, "DRAW", 0.0, NULL}, //17
21    {COMMENT, "//", 0.0, NULL}, //18
22    {COMMENT, "--", 0.0, NULL}, //19
23    {SEMICO, ";", 0.0, NULL}, //20
24    {L_BRACKET, "(", 0.0, NULL}, //21
25    {R_BRACKET, ")", 0.0, NULL}, //22
26    {COMMA, ",", 0.0, NULL}, //23

```

```
27      {PLUS, "+", 0.0, NULL}, //24
28      {MINUS, "-", 0.0, NULL}, //25
29      {MUL, "*", 0.0, NULL}, //26
30      {DIV, "/", 0.0, NULL}, //27
31      {POWER, "**", 0.0, NULL} //28
32  }; //符号表;
```

## DFA

### 正规式

letter = [a-zA-Z]

digit = [0-9]

ID = letter+

CONST\_ID = digit+ ("." digit\*)?

COMMENT = "/\*" "--"

SEMICO = ";"

L\_BRACKET = "("

R\_BRACKET = ")"

COMMA = ","

PLUS = "+"

MINUS = "-"

MUL = "\*"

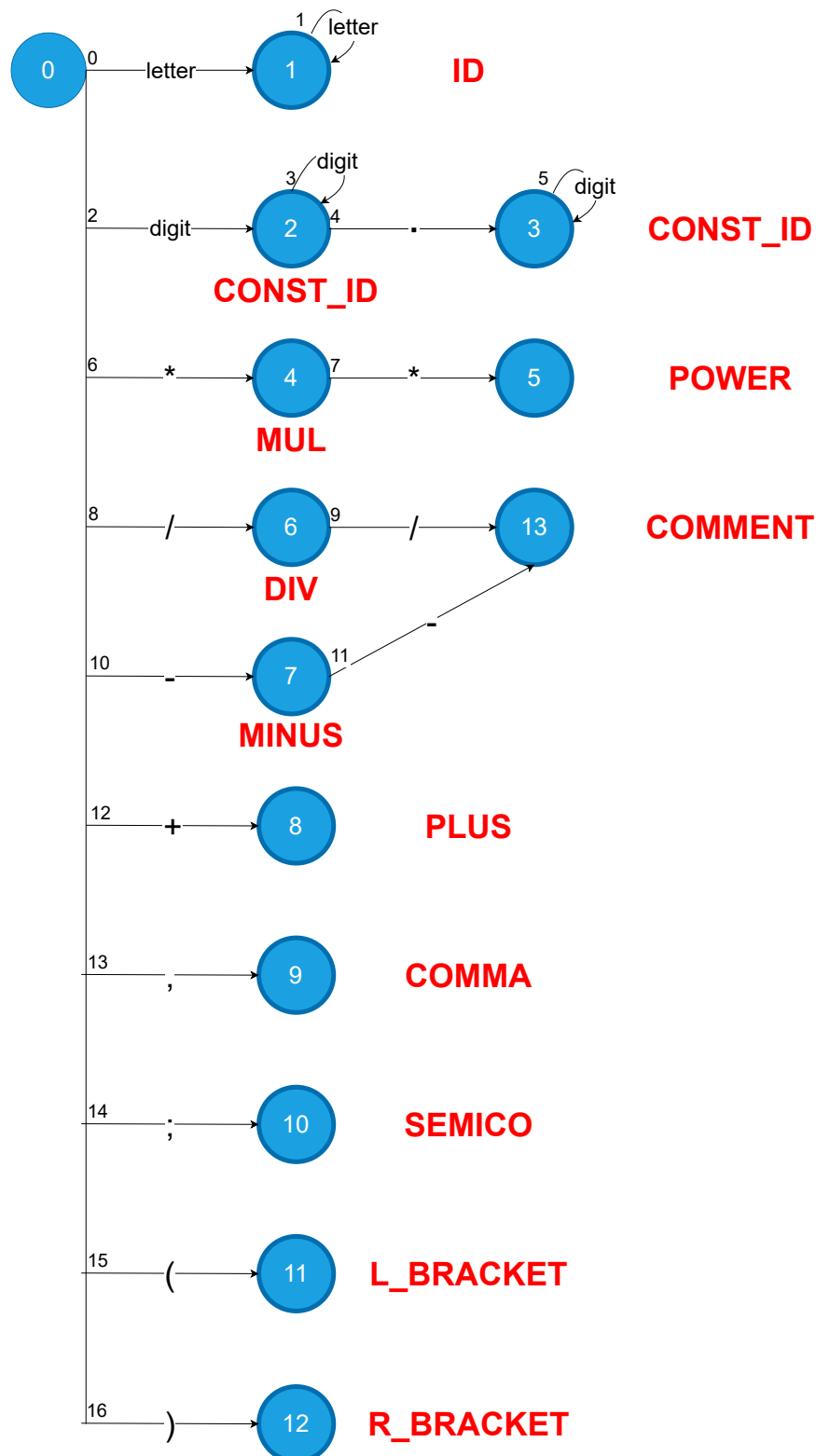
DIV = "/"

POWER = "\*\*"

## DFA Graph

图中边起始位置的数字代表边编号，边上符号表示该边对应的符号，终态附近的红色字表示识别到的Token Type。

需要特别指出的是，**FUNC类型**、**保留字类型**和**标识符形式的常量类型**的token均会被识别为**ID**，然后通过查符号表找到对应的具体含义，符号表中未查找到的则视为ERRORTOKEN。



# DFA链式前向星实现

结构体声明：

```
1 //graph-链式前向星
2 struct Node
3 {
4     bool isF; //该节点是否为终态
5     tokenType type; //如果节点为终态则表示对应的token类型
6 };
7
8 struct Edge
9 {
10     bool isD; //该边是否为数字
11     bool isL; //该边是否为字母
12     int tag; //该边的符号
13     int to; //通过该边到达的状态
14     int next;
15 };
```

存图：

```
1 //表示状态的节点
2 struct Node nds[] =
3 {
4     //isF,type
5     {false,T},
6     {true,ID},
7     {true,CONST_ID},
8     {true,CONST_ID},
9     {true,MUL},
10    {true,POWER},
11    {true,DIV},
12    {true,MINUS},
13    {true,PLUS},
14    {true,COMMA},
15    {true,SEMICO},
16    {true,L_BRACKET},
```



```

17     {true, R_BRACKET},
18     {true, COMMENT}
19 };
20
21 //表示字符的边
22 struct Edge eds[] =
23 {
24     //isD, isL, tag, to, next
25     {false, true, -2, 1, -1},
26     {false, true, -2, 1, -1},
27     {true, false, -2, 2, 0},
28     {true, false, -2, 2, -1},
29     {false, false, '.', 3, 3},
30     {true, false, -2, 3, -1},
31     {false, false, '*', 4, 2},
32     {false, false, '*', 5, -1},
33     {false, false, '/', 6, 6},
34     {false, false, '/', 13, -1},
35     {false, false, '-', 7, 8},
36     {false, false, '-', 13, -1},
37     {false, false, '+', 8, 10},
38     {false, false, ',', 9, 12},
39     {false, false, ';', 10, 13},
40     {false, false, '(', 11, 14},
41     {false, false, ')', 12, 15}
42 };
43
44 int head[] = {16, 1, 4, 5, 7, -1, 9, 11, -1, -1, -1, -1, -1};

```

## 函数接口声明

```

1 //function
2 extern void initScanner();
3 extern void closeScanner();
4 extern struct token getToken(); //获取下一个token
5 extern int prepare(struct token& pToken); //去掉前头的空白
   符, 返回第一个非空白字符
6 extern int scan(struct token& pToken,int firstChar); //根据
   第一个非空字符开始接收字符流
7 extern int move(int nownode,int nowc); //根据目前状态和字符在
   DFA中寻找下一步的状态
8 extern int GetChar(); //获取下一个字符, 如果是字母将其转换为大写
9 extern void setToken(struct token& tk,tokenType tp,double
   v,double (* funcP)(double),int r); //设置token的type,
   value, 函数指针和行号

```

## Scanner工作过程

1. 每次需要获取一个token时, 调用getToken函数。
2. 先进行空白符的去除: 获取当前游标处的字符, 若为空白符:
 

当前空白符为换行符时当前行号++, 然后继续读取下一个字符;

为文件结束符时返回-1;

其他情况, 继续读取下一个字符;

直到该字符非空, 返回第一个非空字符。
3. 将当前状态设置为初态0, 从第一个非空字符开始接收字符流, 每读取到一个字符就搜索当前状态的出边是否有和目前读取到的字符相对应的边
 

若找到对应边则将目前状态设为该边到达的状态;

若找不到对应边则判断目前状态是否为终态:

  - 若目前状态非终态, 继续读取下一个字符直到读到分隔符或运算符或空白符, 将当前读到的字符串标记为ERRORTOKEN。
  - 若目前状态为终态, 进行以下判定:

- 若当前字符为分隔符或运算符或空白符，则将当前经过的路径对应的字符串标记为目前状态对应的token type并返回当前状态。
- 若目前状态对应的token type为COMMENT，则将当前经过的路径对应的字符串标记为COMMENT并将游标移动到行末，返回当前状态。
- 若以上情况均不是，则继续读取下一个字符直到读到分隔符或运算符或空白符，将读到的字符串标记为ERRORTOKEN，返回-1。

4. 根据返回的最后状态对获取到的token进行进一步处理。

## 语法分析器Parser

该部分使用C++编写，完整代码见附件 `parser.h` 和 `parser.cpp`。

### EBNF文法

**Program**  $\rightarrow$  { **Statement** SEMICO }

**Statement**  $\rightarrow$  **OriginStatment** | **ScaleStatment** | **RotStatment** | **ForStatment**

**OriginStatment**  $\rightarrow$  ORIGIN IS L\_BRACKET **Expression** COMMA **Expression** R\_BRACKET

**ScaleStatment**  $\rightarrow$  SCALE IS L\_BRACKET **Expression** COMMA **Expression** R\_BRACKET

**RotStatment**  $\rightarrow$  ROT IS **Expression**

**Expression**  $\rightarrow$  **Term** { ( PLUS | MINUS ) **Term** }

**Term**  $\rightarrow$  **Factor** { ( MUL | DIV ) **Factor** }

**Factor**  $\rightarrow$  ( PLUS | MINUS ) **Factor** | **Component**

**Component**  $\rightarrow$  **Atom** [ POWER **Component** ]

**Atom** → CONST\_ID

| T

| FUNC L\_BRACKET **Expression** R\_BRACKET

| L\_BRACKET **Expression** R\_BRACKET

## Expression语法树

- 叶子节点：存储了常数值/参数值。
- 两个孩子的内部节点：存储了二元运算符  
MUL/MINUS/MUL/DIV/POWER，其左右子树为二元运算的左右表达式/常数/参数。
- 一个孩子的内部节点：其唯一的子节点为叶子节点，存储了指向的函数地址。

## 数据结构

```
1  struct ExprNode
2  {
3      tokenType opcode; //节点对应的token type
4      struct{ ExprNode *left,*right; }CaseOperator; //该节点
      为运算符时左右子树根节点指针
5      struct{ ExprNode *child;double (*funcPtr)(double);
      }CaseFunc; //该节点为函数名时对应的函数地址和函数自变量的表达式
6      double CaseConst; //该节点为常数时的常数值
7      double *CaseParmPtr; //该节点为参数时对应的参数值
8  };
9
10 typedef struct ExprNode* ExprNode_Ptr;
```

## 建树

1. 确定要建立的树的根节点的类型
2. 根据根节点类型存储相应的儿子信息
3. 返回根节点地址

建树代码如下：

```
1  struct ExprNode * MakeExprNode(tokenType
   opcode, ExprNode_Ptr left, ExprNode_Ptr right)
2  {
3      struct ExprNode* root = (struct ExprNode
   *)malloc(sizeof(struct ExprNode));
4      InitNode(root);
5      root->opcode = opcode;
6      switch(opcode)
7      {
8          case CONST_ID://常数节点
9              root->CaseConst = left->CaseConst;
10             break;
11          case T://参数节点
12              root->CaseParmPtr = &parameter;
13              break;
14          case FUNC://函数调用节点
15              root->CaseFunc.funcPtr = left->CaseFunc.funcPtr;
16              root->CaseFunc.child = right;
17              break;
18          default://二元运算符
19              root->CaseOperator.left = (struct
   ExprNode*)malloc(sizeof(struct ExprNode));
20              root->CaseOperator.right = (struct
   ExprNode*)malloc(sizeof(struct ExprNode));
21              root->CaseOperator.left = left;
22              root->CaseOperator.right = right;
23              break;
24      }
25      return root;
26  };
```

## 语法树接口

```

1 //语法树接口
2 struct ExprNode * MakeExprNode(tokenType
  opcode, ExprNode_Ptr left, ExprNode_Ptr right); //建树
3 void PrintSyntaxTree(ExprNode_Ptr root); //输出表达式语法树结
  构
4 void destroy_tree(ExprNode_Ptr root); //回收表达式语法树内存

```

## 递归下降子程序

根据构造的EBNF文法编写递归下降子程序，函数接口如下：

```

1 //递归下降子程序
2 void program();
3 void statement();
4 void for_statement();
5 void origin_statement();
6 void rot_statement();
7 void scale_statement();
8 ExprNode_Ptr expression();
9 ExprNode_Ptr term();
10 ExprNode_Ptr factor();
11 ExprNode_Ptr component();
12 ExprNode_Ptr atom();

```

## Parser工作过程

1. 初始化Parser，启动Scanner
2. 获取第一个token，进入program子程序
3. 判断当前token是否为结束符NONTOKEN，若是则输出整个记号流的语法结构和其中的表达式的语法树结构，工作完成，若不是则进入statement子程序。
4. 在statement子程序中根据当前的token type和文法产生式进入相应语句的子程序。

5. 在相应语句的子程序中根据文法产生式继续执行记号流的识别匹配，并在需要匹配表达式时构造表达式对应的语法树
6. 相应语句识别结束后，调用语义分析器的坐标变换功能和绘制功能进行绘图。
7. statement语句执行完后，匹配SEMICO(';'), 并进行下一个statement的识别（即跳转回步骤3）

## 语义分析器Semantics

该部分使用C++编写，完整代码见附件 `my_semantics.h` 和 `my_semantics.cpp`，其中绘图语句的中间代码使用Python。

### 设计属性

设计变量记录生成Python绘图语句时需要用到的**T参数数值**、**原点横纵坐标偏移距离**、**横纵坐标缩放比例**、**逆时针旋转弧度**。

需要指出的是，T参数数值是实时变化的，通过表达式语法树计算函数 `GetExprValue` 形参 `bias` 传递。其余属性存储在全局变量。

```
1 // 原点x, y轴平移长度 横纵坐标比例 旋转弧度(逆时针)
2 double Origin_x = 0.0, Origin_y = 0.0, Scale_x = 1,
   Scale_y = 1, Rot_rad = 0.0;
```

### 语义规则的设计与嵌入

对一个语法结构进行语法分析后，紧跟着执行为该结构设计的语义规则。

在我们编写的程序中，Semantics的功能调用嵌入在Parser的函数 `origin_statement/scale_statement/rot_statement/for_statement` 对各种statement分析完成后，函数接口声明和嵌入位置如下：

```

1 //设置原点x、y轴的偏移位置, 嵌入在origin_statement末尾
2 extern void setOrigin(ExprNode_Ptr x_ptr, ExprNode_Ptr
  y_ptr);
3 //设置绘制图像的横纵缩放比例, 嵌入在scale_statement末尾
4 extern void setScale(ExprNode_Ptr x_ptr, ExprNode_Ptr
  y_ptr);
5 //设置绘制图像的逆时针旋转角度, 嵌入在rot_statement末尾
6 extern void setRot(ExprNode_Ptr angle_ptr);
7 //for-draw语句绘图, 嵌入在for_statement末尾
8 extern void DrawLoop(ExprNode_Ptr start_ptr, ExprNode_Ptr
  end_ptr, ExprNode_Ptr step_ptr, ExprNode_Ptr x_ptr,
  ExprNode_Ptr y_ptr);

```

## 表达式语法树计算

从语法树根节点开始进行**后序遍历**计算表达式的值, 即先计算左右子树的值再计算根节点对应子树的值。

- 根节点类型为 PLUS/MINUS/MUL/DIV/POWER : 递归计算左右子树的值, 它们 加/减/乘/除/以右儿子为指数左儿子为底数求幂 的结果即为根节点对应子树的值, 将其返回
- 根节点类型为 FUNC : 递归计算其唯一子树的值, 将该值作为函数的自变量值输入函数指针指向的函数, 将得到的因变量值返回
- 根节点类型为 CONST\_ID : 直接返回常数值
- 根节点类型为 T : 直接返回当前的T参数值

代码如下:

```

1 // 计算表达式的值, 深度优先后序遍历语法树, bias为T参数当前的值
2 double GetExprValue(ExprNode_Ptr root, double bias) {
3     if (root == NULL)
4         return 0.0;
5     switch (root -> opcode) {
6         case PLUS :
7             return GetExprValue(root -> CaseOperator.left,
  bias) + GetExprValue(root -> CaseOperator.right, bias);

```



```

8         case MINUS:
9             return GetExprValue(root -> CaseOperator.left,
bias) - GetExprValue(root -> CaseOperator.right, bias);
10        case MUL:
11            return GetExprValue(root -> CaseOperator.left,
bias) * GetExprValue(root -> CaseOperator.right, bias);
12        case DIV:
13            return GetExprValue(root -> CaseOperator.left,
bias) / GetExprValue(root -> CaseOperator.right, bias);
14        case POWER:
15            return pow(GetExprValue(root ->
CaseOperator.left, bias), GetExprValue(root ->
CaseOperator.right, bias));
16        // 调用指定的函数
17        case FUNC:
18            return (root -> CaseFunc.funcPtr)
(GetExprValue(root -> CaseFunc.child, bias));
19        case CONST_ID:
20            return root -> CaseConst;
21        case T:
22            return bias;
23        default:
24            return 0.0;
25
26    }
27 }

```

## 实际绘制点坐标计算

- 1. 计算初始坐标：**调用表达式计算函数计算初始x、y坐标值
- 2. 比例变换：**将x、y坐标值乘以横纵坐标缩放比例
- 3. 旋转变换：**

$$\begin{aligned}
 x &= x \cdot \cos(\text{rot\_angle}) + y \cdot \sin(\text{rot\_angle}) \\
 y &= y \cdot \cos(\text{rot\_angle}) - x \cdot \sin(\text{rot\_angle})
 \end{aligned}$$

4. **平移变换**：将x、y坐标值加上相对应的平移量

5. 返回变换后的x、y坐标值

代码如下：

```
1 // 计算点的坐标：获取坐标后进行变换
2 void CalcCoord(ExprNode_Ptr x_ptr, ExprNode_Ptr y_ptr,
3 double * x_value, double * y_value, double bias) {
4     double x_val, y_val, temp;
5     // 初始坐标
6     x_val = GetExprValue(x_ptr, bias);
7     y_val = GetExprValue(y_ptr, bias);
8     // 比例变换
9     x_val *= Scale_x;
10    y_val *= Scale_y;
11    // 旋转变换
12    temp = x_val * cos(Rot_rad) + y_val * sin(Rot_rad);
13    y_val = y_val * cos(Rot_rad) - x_val * sin(Rot_rad);
14    x_val = temp;
15    // 平移变换
16    x_val += Origin_x;
17    y_val += Origin_y;
18    // 返回变换后坐标
19    if (NULL != x_value)
20        *x_value = x_val;
21    if (NULL != y_value)
22        *y_value = y_val;
```

## 单点绘图语句生成

将**实际绘制点坐标**作为**Python的tkinter库点绘制语句**的形参值，并将该点的绘制语句写入 `drawit.py` 文件中，单点绘图语句生成函数如下：

```
1 //py绘图文件初始化
```

```

2 void init_drawer()
3 {
4     draw_py_file = fopen("./drawit.py", "w");
5     if (!draw_py_file)
6     {
7         //err_exit("unable to create py file, draw
failed.", nullptr, -EFAULT);
8         printf("unable to create py.file, fail to
draw...\n");
9         exit(1);
10    }
11
12    drawer_init = true;
13
14    fprintf(draw_py_file, "from tkinter import *\n");
15    fprintf(draw_py_file, "tk = Tk()\n");
16    fprintf(draw_py_file, "tk.title(\"Final
Painting\")\n");
17    fprintf(draw_py_file, "canvas = Canvas(tk, width =
1024, height = 500)\n");
18    fprintf(draw_py_file, "canvas.pack()\n");
19 }
20
21 //单点绘制语句写入py文件
22 void DrawPixel(double x, double y, double step)
23 {
24     if (!drawer_init)
25         init_drawer();
26
27     if (fabs(x) != NAN && fabs(x) != INFINITY && fabs(y)
!= NAN && fabs(y) != INFINITY)
28         fprintf(draw_py_file, "canvas.create_oval(%lf,
%lf, %lf, %lf)\n", x, y, x, y);
29 }

```

## 循环绘制点坐标

循环绘制点坐标的步骤如下：

1. 初始化Python绘图文件
2. 调用 `GetExprValue` 函数获取参数T的初始值、结束值和step值
3. 检查T值是否超过结束值，若是则结束循环绘制，若不是则进入下一步。
4. 调用 `CalcCoord` 函数计算实际绘制点坐标
5. 调用 `DrawPixel` 生成在实际绘制点坐标处画点的Python语句并写入py文件中
6. 跳回步骤3

代码实现如下：

```
1 // 循环绘制点坐标
2 void DrawLoop(ExprNode_Ptr start_ptr, ExprNode_Ptr
  end_ptr, ExprNode_Ptr step_ptr, ExprNode_Ptr x_ptr,
  ExprNode_Ptr y_ptr) {
3     double x_val, y_val;
4     double start_val, end_val, step_val;
5     //double * p_T_value = getT();
6
7     if (!drawer_init)
8         init_drawer();
9
10    start_val = GetExprValue(start_ptr, 0);
11    end_val = GetExprValue(end_ptr, 0);
12    step_val = GetExprValue(step_ptr, 0);
13
14    for(double i = start_val; i <= end_val; i += step_val)
15    {
16        CalcCoord(x_ptr, y_ptr, &x_val, &y_val, i);
17        DrawPixel(x_val, y_val, step_val);
18    }
19 }
```

## 绘制图像生成

使用 `system("python ./drawit.py")` 语句通过控制台执行Python脚本文件 `drawit.py`，即可生成绘图语句所绘制的图像。

## # 软件测试

---

采用分模块测试，对Scanner、Parser和Semantics分别进行测试用例设计、测试结果分析，测试完成后解释器的所有源代码整合在目录 `./MyInterpreter` 中。

解释器主程序的源代码为 `main.cpp`

解释器各模块的编译、链接的Windows脚本已写入到批处理文件 `compile.bat` 中，打开即可生成解释器的可执行程序 `MyInterpreter.exe`。

## Scanner

Scanner测试程序和用例放在附件的 `./ScannerTest` 文件夹中

测试主程序为 `scannerMain.cpp`。

Scanner测试程序的编译、链接的Windows脚本已写入到批处理文件 `compile.bat` 中，打开即可生成Scanner测试程序的可执行程序 `ScannerTest.exe`。

测试主程序将输出一个token list，这个list中包含了测试用例中所有token的 token type, lexeme, value, 指向的函数名, 在源程序中的行号。

## 测试用例设计

该测试用例包含了一段正确的绘图语句和许多错误token，内容短小却十分具有代表性，测试用例 `1.txt` 如下。

```
--follows are correct tokens.  
rot is 0; //test  
origin is (50, 400);  
scale is (2, 1);  
for T from 0 to 60 step 0.01 draw (t+2, sin(t*t/2));  
//some error tokens in these sentences.  
for T from 0abc to abc60 step 1 draw (tt,3t);  
forT from123 to456 1step1 Ddraw *&^&%&%@!
```

该测试样例中

- 不仅有任何一个字符都不合法的token `*&^&%&%@!`，也有诸如 `forT`，`from123` 这样由两个正确token拼接成的错误token和诸如 `Ddraw` 这样字符串中包含一部分正确token的错误token。
- 样例囊括了所有正确的token类型

## 测试结果及分析

```
E:\Project\Cpp\myScanner\scannerMain.exe
Please input the filename you wanna compile:
l.txt
Token Type      Lexeme      Value      Function Pointer      Row
COMMENT         --          0          NULL                  1
ROT             ROT         0          NULL                  2
IS              IS          0          NULL                  2
CONST_ID        0           0          NULL                  2
SEMICO          ;           0          NULL                  2
COMMENT         //          0          NULL                  2
ORIGIN          ORIGIN      0          NULL                  3
IS              IS          0          NULL                  3
L_BRACKET       (           0          NULL                  3
CONST_ID        50          50         NULL                  3
COMMA           ,           0          NULL                  3
CONST_ID        400         400        NULL                  3
R_BRACKET       )           0          NULL                  3
SEMICO          ;           0          NULL                  3
SCALE           SCALE       0          NULL                  4
IS              IS          0          NULL                  4
L_BRACKET       (           0          NULL                  4
CONST_ID        2           2          NULL                  4
COMMA           ,           0          NULL                  4
CONST_ID        1           1          NULL                  4
R_BRACKET       )           0          NULL                  4
SEMICO          ;           0          NULL                  4
FOR             FOR         0          NULL                  5
T              T           0          NULL                  5
FROM           FROM        0          NULL                  5
CONST_ID        0           0          NULL                  5
TO             TO          0          NULL                  5
CONST_ID        60          60         NULL                  5
STEP           STEP        0          NULL                  5
CONST_ID        0.01        0.01       NULL                  5
DRAW           DRAW        0          NULL                  5
L_BRACKET       (           0          NULL                  5
T              T           0          NULL                  5
PLUS           +           0          NULL                  5
CONST_ID        2           2          NULL                  5
COMMA           ,           0          NULL                  5
FUNC           SIN         0          SIN                   5
L_BRACKET       (           0          NULL                  5
T              T           0          NULL                  5
MUL            *           0          NULL                  5
T              T           0          NULL                  5
DIV            /           0          NULL                  5
CONST_ID        2           2          NULL                  5
R_BRACKET       )           0          NULL                  5
R_BRACKET       )           0          NULL                  5
SEMICO          ;           0          NULL                  5

COMMENT         //          0          NULL                  6
FOR             FOR         0          NULL                  7
T              T           0          NULL                  7
FROM           FROM        0          NULL                  7
ERRTOKEN        OABC        0          NULL                  7
TO             TO          0          NULL                  7
ERRTOKEN        ABC60       0          NULL                  7
STEP           STEP        0          NULL                  7
CONST_ID        1           1          NULL                  7
DRAW           DRAW        0          NULL                  7
L_BRACKET       (           0          NULL                  7
ERRTOKEN        TT         0          NULL                  7
COMMA           ,           0          NULL                  7
ERRTOKEN        3T         0          NULL                  7
R_BRACKET       )           0          NULL                  7
SEMICO          ;           0          NULL                  7
ERRTOKEN        FORT       0          NULL                  8
ERRTOKEN        FROM123     0          NULL                  8
ERRTOKEN        TO456       0          NULL                  8
ERRTOKEN        1STEP1      0          NULL                  8
ERRTOKEN        DDRAW       0          NULL                  8
MUL            *           0          NULL                  8
ERRTOKEN        &`&%&%@!      0          NULL                  8
请按任意键继续. . .
```

可以看到测试结果中

- lexeme中的所有字母均被转换为大写
- 非ERRORTOKEN均能够被正确地识别出其token type, lexeme, value, 指向的函数名和在源程序中的行号
- COMMENT类型的token也能够被识别，并且COMMENT类型的token `//` 或 `--` 后的第一个字符到行末字符均被Scanner忽略。
- 每个ERRORTOKEN均能被识别

- Scanner能够在文本结束处停止识别，并且不会输出文本结束符  
NONTOKEN

由以上分析可得Scanner功能完整正确。

## Parser

Parser测试程序和用例放在附件的 `./ParserTest` 文件夹中。

需要指出的是，Parser测试程序中的parser.cpp和解释器中使用的parser.cpp不同，Parser测试程序中的parser.cpp没有嵌入Semantics。

测试主程序为 `parserMain.cpp`。

Parser测试程序的编译、链接的Windows脚本已写入到批处理文件 `compile.bat` 中，打开即可生成Parser测试程序的可执行程序 `ParserTest.exe`。

测试结果将输出：

- 根据文法和测试样例内容输出递归下降子程序的调用过程
- 遇到表达式时，输出表达式语法树结构，输出形式为先输出根节点，再输出左子树，再输出右子树。
- 遇到词法或语法错误时，输出错误token在测试用例中的行数，并结束识别。

### 测试用例设计

#### 1. 完全正确的测试用例 `1.txt`

该用例包含了所有类型的statement，且含有注释和复杂的表达式（用于观察语法树输出）

测试用例1内容如下：



```
rot is 0;  
origin is (50, 400);  
scale is (2, 1);  
//for T from 0 to 300 draw (t,0);  
for T from 0 to 60 step 0.01 draw (t+2, sin(t*t/2));
```

## 2. 具有多余结构的语法错误测试用例 2.txt

该用例在最后点坐标处多出了一个逗号 ',' 和 1维坐标表达式 "`cos(t)`"

测试用例2内容如下:

```
for T from 0 to 300 step 0.01 draw (t, -ln(t), cos(t));
```

## 3. 缺少一部分结构的语法错误测试用例 3.txt

该用例的for-draw语句缺少step token和step的值表达式, 在T的结束值表达式后紧随的是draw token

测试用例3内容如下:

```
for T from 0 to 300 draw (t, -ln(t));
```

## 4. 含有ERRORTOKEN的词法错误测试用例 4.txt

该用例中含有一个ERRORTOKEN "`%&#`"

测试用例4内容如下:

```
for T from 0 to 60 step 0.01 draw %&# (t+2, sin(t*t/2));
```

# 测试结果及分析

## 测试用例1

E:\Project\Cpp\ParserTest\parserMain.exe

Please input the filename you wanna compile:

1.txt

enter in program

enter in statement

enter in rot\_statement

MatchToken ROT

MatchToken IS

enter in expression

MatchToken 0

exit from expression

0.000000

exit from rot\_statement

exit from statement

MatchToken ;

enter in statement

enter in origin\_statement

MatchToken ORIGIN

MatchToken IS

MatchToken (

enter in expression

MatchToken 50

exit from expression

50.000000

MatchToken ,

enter in expression

MatchToken 400

exit from expression

400.000000

MatchToken )

exit from origin\_statement

exit from statement

MatchToken ;

enter in statement

enter in scale\_statement

MatchToken SCALE

MatchToken IS

MatchToken (

enter in expression

MatchToken 2

exit from expression

2.000000

MatchToken ,

enter in expression

MatchToken 1

exit from expression

1.000000

MatchToken )

exit from scale\_statement

exit from statement

MatchToken ;

```

enter in statement
enter from for_statement
MatchToken FOR
MatchToken T
MatchToken FROM
enter in expression
MatchToken 0
exit from expression
0.000000
MatchToken TO
enter in expression
MatchToken 60
exit from expression
60.000000
MatchToken STEP
enter in expression
MatchToken 0.01
exit from expression
0.010000
MatchToken DRAW
MatchToken (
enter in expression
MatchToken T
MatchToken +
MatchToken 2
exit from expression
+ 根
T 左子树根
2.000000 右子树根
MatchToken ,
enter in expression
MatchToken SIN
MatchToken (
enter in expression
MatchToken T
MatchToken *
MatchToken T
MatchToken /
MatchToken 2
exit from expression
MatchToken )
exit from expression
00406E7C SIN函数指针指向的地址
/ 根
* 左子树根
T 左子树左儿子
T 左子树右儿子
2.000000 右子树根
MatchToken )
exit from for_statement
exit from statement
MatchToken ;
exit from program请按任意键继续. . .

```

表达式语法树

表达式语法树

可以看到测试结果中

- 递归下降子程序的进入和退出过程输出正确
- 表达式语法树输出正确
- 能够正确输出函数指针指向的函数地址

## 测试用例2

```
E:\Project\Cpp\ParserTest\parserMain.exe
Please input the filename you wanna compile:
2.txt
enter in program
enter in statement
enter from for_statement
MatchToken FOR
MatchToken T
MatchToken FROM
enter in expression
MatchToken 0
exit from expression
0.000000
MatchToken TO
enter in expression
MatchToken 300
exit from expression
300.000000
MatchToken STEP
enter in expression
MatchToken 0.01
exit from expression
0.010000
MatchToken DRAW
MatchToken (
enter in expression
MatchToken T
exit from expression
T
MatchToken ,
enter in expression
MatchToken -
MatchToken LN
MatchToken (
enter in expression
MatchToken T
exit from expression
MatchToken )
exit from expression
-
0.000000
00406EDC
T
Unexpected token ", " at line 1!
请按任意键继续. . .
```

该用例在最后点坐标处多出了一个逗号 ',' 和1维坐标表达式 'cos(t)'

可以看到测试结果输出了第一个unexpected token为多出的逗号 ',' 以及其所在行号 **line 1**，由此可以反映出程序能够发现**多余结构的语法错误**。

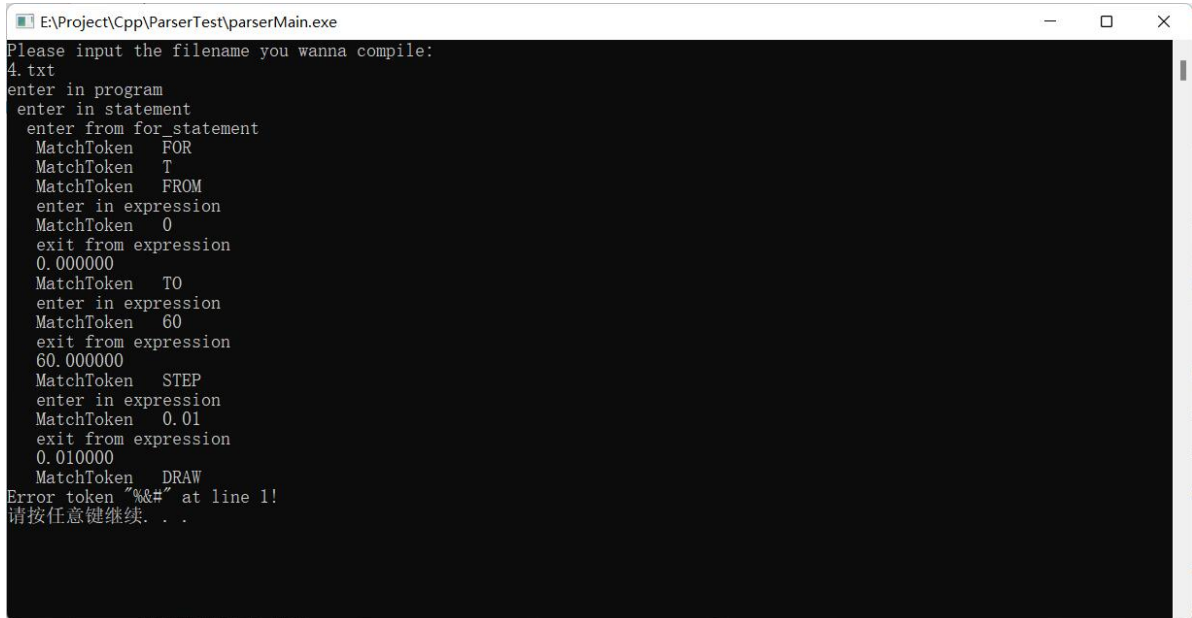
### 测试用例3

```
E:\Project\Cpp\ParserTest\parserMain.exe
Please input the filename you wanna compile:
3.txt
enter in program
enter in statement
enter from for_statement
MatchToken FOR
MatchToken T
MatchToken FROM
enter in expression
MatchToken 0
exit from expression
0.000000
MatchToken TO
enter in expression
MatchToken 300
exit from expression
300.000000
Unexpected token "DRAW" at line 1!
请按任意键继续. . .
```

该用例的for-draw语句缺少step token和step的值表达式，在T的结束值表达式后紧随的是 **draw token**

可以看到测试结果中输出的信息表示原本期望读到的step token没有读到，而是读到了一个unexpected token draw，其所在行为line 1，由此可以反映出程序能够发现缺少结构的语法错误。

## 测试样例4



```
E:\Project\Cpp\ParserTest\parserMain.exe
Please input the filename you wanna compile:
4.txt
enter in program
enter in statement
enter from for_statement
  MatchToken  FOR
  MatchToken  T
  MatchToken  FROM
  enter in expression
  MatchToken  0
  exit from expression
  0.000000
  MatchToken  TO
  enter in expression
  MatchToken  60
  exit from expression
  60.000000
  MatchToken  STEP
  enter in expression
  MatchToken  0.01
  exit from expression
  0.010000
  MatchToken  DRAW
Error token "%&#" at line 1!
请按任意键继续...
```

该用例中含有一个ERRORTOKEN "%&#"

可以看到测试结果中输出了该ERRORTOKEN "%&#" 并且输出了其所在行line 1，由此可以反映出程序能够发现词法错误。

# Semantics

Semantics的测试程序即为解释器完整程序，存放在目录 ./MyInterpreter。

测试主程序源代码为 main.cpp

解释器各模块的编译、链接的Windows脚本已写入到批处理文件 compile.bat 中，打开即可生成解释器的可执行程序 MyInterpreter.exe。

测试结果将输出一个和绘图源程序相对应的python脚本文件和一张图像

## 测试用例设计

该测试用例绘制的图形应当是一个x-y坐标系和以下函数的部分图像：

$$y = x^2$$

$$y = -\sqrt{3}x$$

$$y = -\frac{1}{\sqrt{3}}x$$

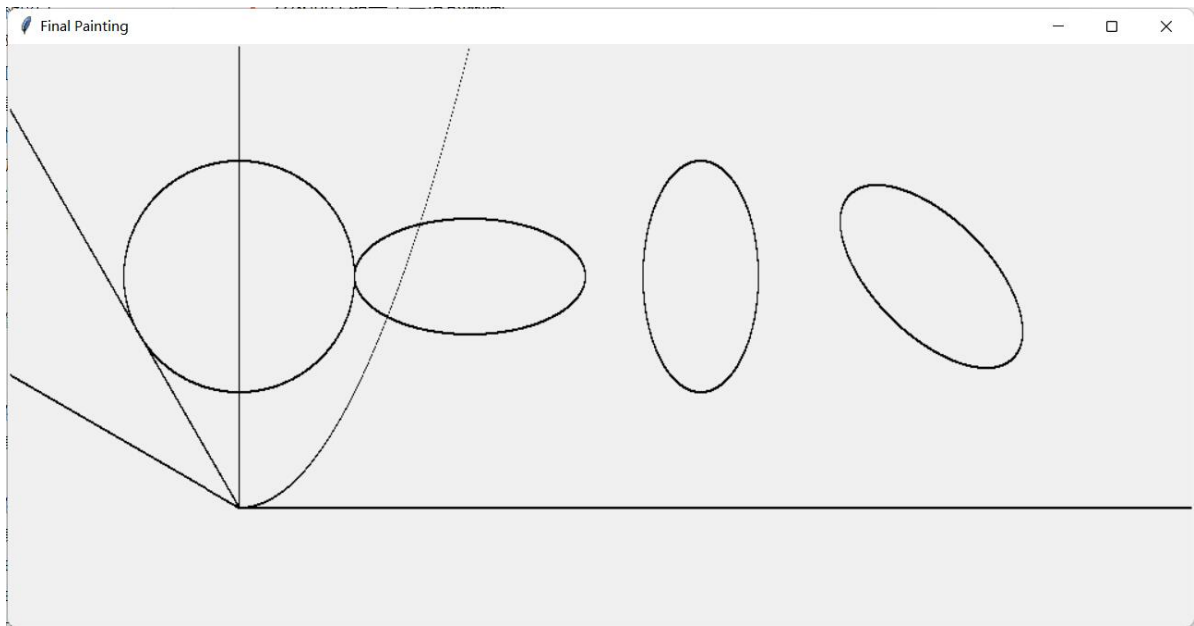
并且包含一个圆，一个纵向压缩一半高度的椭圆、一个横向压缩一半宽度的椭圆、一个横向压缩一半宽度并且逆时针旋转45°的椭圆。

测试用例源程序如下：

```
rot is 0;
origin is (200, 400);
scale is (100, 100);
for T from 0 to 300 step 0.01 draw (t, 0);
for T from 0 to 300 step 0.01 draw (0, -t);
for T from 0 to 55 step 0.01 draw (t, -(t*t));
rot is PI/6;
for T from 0 to 300 step 0.01 draw (0, -t);
rot is PI/3;
for T from 0 to 300 step 0.01 draw (0, -t);
//圆
rot is 0;
origin is (200,200);
scale is (100, 100);
for T from 0 to 60 step 0.01 draw (cos(t), sin(t));
//纵向压缩一半高度的椭圆
origin is (400,200);
scale is (100, 50);
for T from 0 to 60 step 0.01 draw (cos(t), sin(t));
//横向压缩一半宽度的椭圆
origin is (600,200);
scale is (50, 100);
for T from 0 to 60 step 0.01 draw (cos(t), sin(t));
```

```
//横向压缩一半宽度并且逆时针旋转45°的椭圆  
rot is PI/4;  
origin is (800,200);  
scale is (50, 100);  
for T from 0 to 60 step 0.01 draw (cos(t), sin(t));
```

## 测试结果及分析



由于Python绘图脚本文件过大，就不在报告中展示了，可以打开附件中的 [./MyInterpreter/drawit.py](#) 查看。

可以看到，测试主程序能够根据绘图源程序进行绘图脚本的生成和绘制：

- 能够绘制不同 旋转角度、偏移位置、纵横缩放比例 的线条和图形
- 能够绘制坐标表达式较为复杂且含有函数调用（如  $\sin$  ,  $\cos$  ）的点

## # 总结

---

- 通过一步步构建函数绘图语言的解释器，理解了文件中的字符流被解释器通过词法分析生成记号流，再通过语法分析器按照递归下降文法使用递归下降子程序进行处理并得到表达式语法树，然后由语义分析器后序遍历语法树，将statement处理成一条条Python脚本代码再绘制成图像的过程。
- 在团队协作编写项目的过程中掌握了C++的.h文件和.cpp文件编译链接的原理和过程，学会了使用 `g++ -c` 命令编译生成.o文件并使用 `g++ -o` 命令将它们链接起来生成可执行程序
- 团队协作过程中应当一开始就统一文件编码，在本实验中词法分析器的编写使用的是UTF-8编码，而语法分析器和语义分析器均使用的是GB2312编码，导致用中文书写的注释出现了乱码，后续我们统一为了UTF-8编码才使乱码正常显示。
- 词法分析器的构建要点在于设计token和构造出能够识别token的最小DFA，构建完成后根据初态和读入的字符逐步进行状态转移并根据到达的终态返回相应的token即可。
- 语法分析器的构建要点有两个：
  1. 是利用词法分析器读取到的记号对每个产生式进行匹配
  2. 是为语义分析的计算构建语法树

遇到的主要难点及解决方式：

1. 构建语法树要考虑计算的优先级。解决该难点的方式是在程序中设置产生式的产生顺序可以确保符号的优先级，将优先级高的运算符放在后面的产生式中，在创建树的过程中会先递归调用底层的产生式函数，可以确保运算的优先级
  2. 构建语法树的节点时要根据记号的类型进行构建，不同的记号类型有不同的创造方法，要设计合适的节点数据类型。将语法树的节点设置成一个嵌套的结构体，里面包含所有记号类型所需要的数据结构，在使用的时候，通过一个节点操作类型opcode来判断要使用哪一部分的数据结构
- 语义分析部分较为简单，主要完成两个任务：计算表达式的值和绘图。

在编写过程中遇到主要的难点在于绘图工具/语言的选择，一开始想要采用WinGUI来实现UI和绘图，但知识储备有限，遇到了问题始终无法解决，最终选择采用Python的tkinter库来进行绘图解决。