

一. 上机目的

1. 会用正规式的产生式设计简单的语言语法
2. 会用递归下降子程序编写编译器或解释器
3. 会写上机报告

二. 上机题目：简单函数绘图语言的解释器

2.1 题目简述

<1>实现简单函数绘图的语句

- 循环绘图 (FOR-DRAW)
- 比例设置 (SCALE)
- 角度旋转 (ROT)
- 坐标平移 (ORIGIN)
- 注释 (-- 或 //)

<2> 屏幕（窗口）的坐标系

- 左上角为原点
- x方向从左向右增长
- y方向从上到下增长(与一般的坐标系方向相反)

<3> 函数绘图源程序举例

```
----- 函数f(t)=t的图形
origin is (100, 300);  -- 设置原点的偏移量
rot is 0;             -- 设置旋转角度(不旋转)
scale is (1, 1);      -- 设置横坐标和纵坐标的比例
for T from 0 to 200 step 1 draw (t, 0);
                        -- 横坐标的轨迹(纵坐标为0)
for T from 0 to 150 step 1 draw (0, -t);
                        -- 纵坐标的轨迹(横坐标为0)
for T from 0 to 120 step 1 draw (t, -t);
                        -- 函数f(t)=t的轨迹
```

默认值:

```
origin is (0, 0)
rot is 0;
scale is (1, 1)
```

2.2 语句的语法和语义 (syntax & semantics)

语句满足下述规定(原则):

<1> 各类语句可以按任意次序书写, 且语句以分号结尾。源程序中的语句以它们出现的先后顺序处理。

<2> ORIGIN、ROT和SCALE 语句只影响其后的绘图语句, 且遵循最后出现的语句有效的原则。例如, 若有下述ROT语句序列:

```
ROT IS 0.7 ;  
ROT IS 1.57 ;
```

则随后的绘图语句将按1.57而不是0.7弧度旋转。

<3> 无论ORIGIN、ROT和SCALE语句的出现顺序如何，图形的变换顺序总是：比例变换→旋转变换→平移变换

<4> 语言对大小写不敏感，例如for、For、FOR等，均被认为是同一个保留字。

<5> 语句中表达式的值均为双精度类型，旋转角度单位为弧度且为逆时针旋转，平移单位为点。

2.2.1 循环绘图（FOR-DRAW）语句

语法

```
FOR T FROM 起点 TO 终点 STEP 步长 DRAW(横坐标, 纵坐标);
```

语义

令T从起点到终点、每次改变一个步长，绘制出由(横坐标, 纵坐标)所规定的点的轨迹。

举例

```
FOR T FROM 0 TO 2*PI STEP PI/50 DRAW (cos(T), sin(T));
```

该语句的作用是令T从0到2*PI、步长 PI/50，绘制出各个点的坐标(cos(T), sin(T))，即一个单位圆。

注意

由于绘图系统的默认值是

```
ORIGIN IS (0,0);  
ROT IS 0;  
SCALE IS (1, 1);
```

所以实际绘制出的图形是在屏幕左上角的一个点。

2.2.2 比例设置(SCALE)语句

语法

```
SCALE IS (横坐标比例因子, 纵坐标比例因子);
```

语义

设置横坐标和纵坐标的比例，并分别按照比例因子进行缩放。

举例

```
SCALE IS (100, 100);
```

将横坐标和纵坐标的比例设置为1:1，且放大100倍。

若：SCALE IS (100, 100/3);

则：横坐标和纵坐标的比例为3:1。

2.2.3 坐标平移(ORIGIN)语句

语法

```
ORIGIN IS (横坐标, 纵坐标);
```

语义

将坐标系的原点平移到横坐标和纵坐标规定的点处。

举例

```
ORIGIN IS (360, 240);
```

将原点从(0, 0)平移到(360, 240)处。

2.2.4 角度旋转(ROT)语句

语法

```
ROT IS 角度;
```

语义

逆时针旋转角度所规定的弧度值。具体计算公式：

- 旋转后X=旋转前XCOS(角度)+旋转前YSIN(角度)
- 旋转后Y=旋转前YCOS(角度)-旋转前XSIN(角度)

公式的推导可参阅辅助教材。

举例

```
ROT IS PI/2;
```

逆时针旋转PI/2，即逆时针旋转90度。

2.3 记号的语法和语义

记号的种类：**常数、参数、函数、保留字、运算符、分隔符**

<1> 常数

常数字面量和标识符形式的常量名均称为常数。字面量的形式为普通的数值，如果没有小数部分，可以省略小数点。例如2、2.、2.0都是合法的常数。标识符PI、E也是常数，它们分别代表圆周率和自然对数的底。常数不能有符号位，如-1和+2不是常数而是（一元运算的）表达式。

<2> 参数

本作图语言中唯一的、已经被定义好的变量名T被称为参数，它也是一个表达式。

除了预留参数T之外本作图语言还支持自定义参数。

<3> 函数（调用）

为简单起见，当前函数调用仅支持Sin、Cos、Tan、Sqrt以及Exp和Ln。

<4> 保留字

语句中具有固定含义的标识符，包括：ORIGIN, SCALE, ROT, IS, TO, STEP, DRAW, FOR, FROM

<5> 运算符

PLUS, MINUS, MUL, DIV, POWER
即：+ - * / **

<6> 分隔符

SEMICO, L_BRACKET, R_BRACKET, COMMA
即：; () ,

三、 题目与要求

题目：为函数绘图语言编写一个解释器

解释器接受用绘图语言编写的源程序，经语法和语义分析之后，将源程序所规定的图形显示在显示屏（或窗口）中。

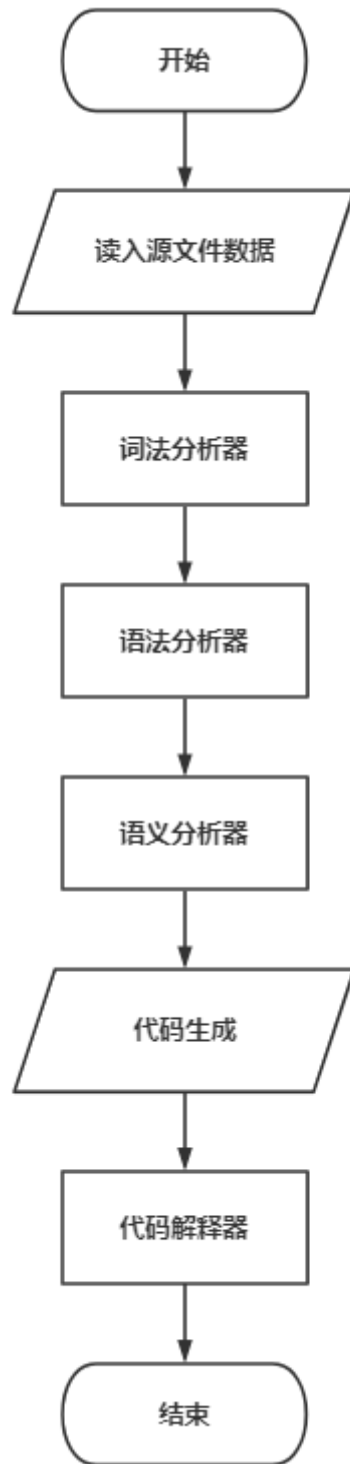
目的：通过自己动手编写解释器，掌握语言翻译特别是语言识别的基本方法。

四、 实验过程

1.结构设计

本编译器采用一趟扫描，以语法分析器为核心，语法分析器通过词法分析器从代码源文件中读入与分析词素，通过递归下降分析生成对应的抽象语法树（Abstract Syntax Tree），并检测代码中的语法错误。

语法分析器在生成抽象语法树后将其传递给语义分析器，由语义分析器进行语义分析，生成最终的可执行代码，此处为了保证代码跨平台的兼容性，笔者选择使用 cmake 构建整个项目，整个项目中使用平台无关的标准库函数，最终生成 python 代码使用 tkinter 库进行绘图。



2.词法分析器 (Lexical Analyzer, aka Scanner)

(1) 词法分析器的主要功能

词法分析器的主要功能为识别输入序列，并为语法分析器提供记号

- 过滤掉源程序中的空白字符、注释等编译无关字符
- 输出**记号 (token)** 供语法分析器使用
- 识别非法输出，将其标记为错误记号

(2) 记号定义

笔者将记号定义为如下形式：

```
struct token
{
    enum token_type type;
    char *lexeme; // original string of the token
    double val; // for if the token is a value
    double (*func_ptr)(double); // for function calls
};
```

其中的 `token_type` 为枚举类型，定义了记号的种类，定义如下：

```
enum token_type
{
    // reserved word
    ORIGIN, SCALE, ROT, IS, TO, STEP, DRAW, FOR, FROM,
    // param
    T,
    // separator ; ( ) ,
    SEMICO, L_BRACKET, R_BRACKET, COMMA,
    // operator + - * / **
    PLUS, MINUS, MUL, DIV, POWER,
    // function
    FUNC,
    // const value
    CONST_ID,
    // nullptr token, which means that the source code come to an end
    NONTOKEN,
    // invalid token type
    ERRORTOKEN
};
```

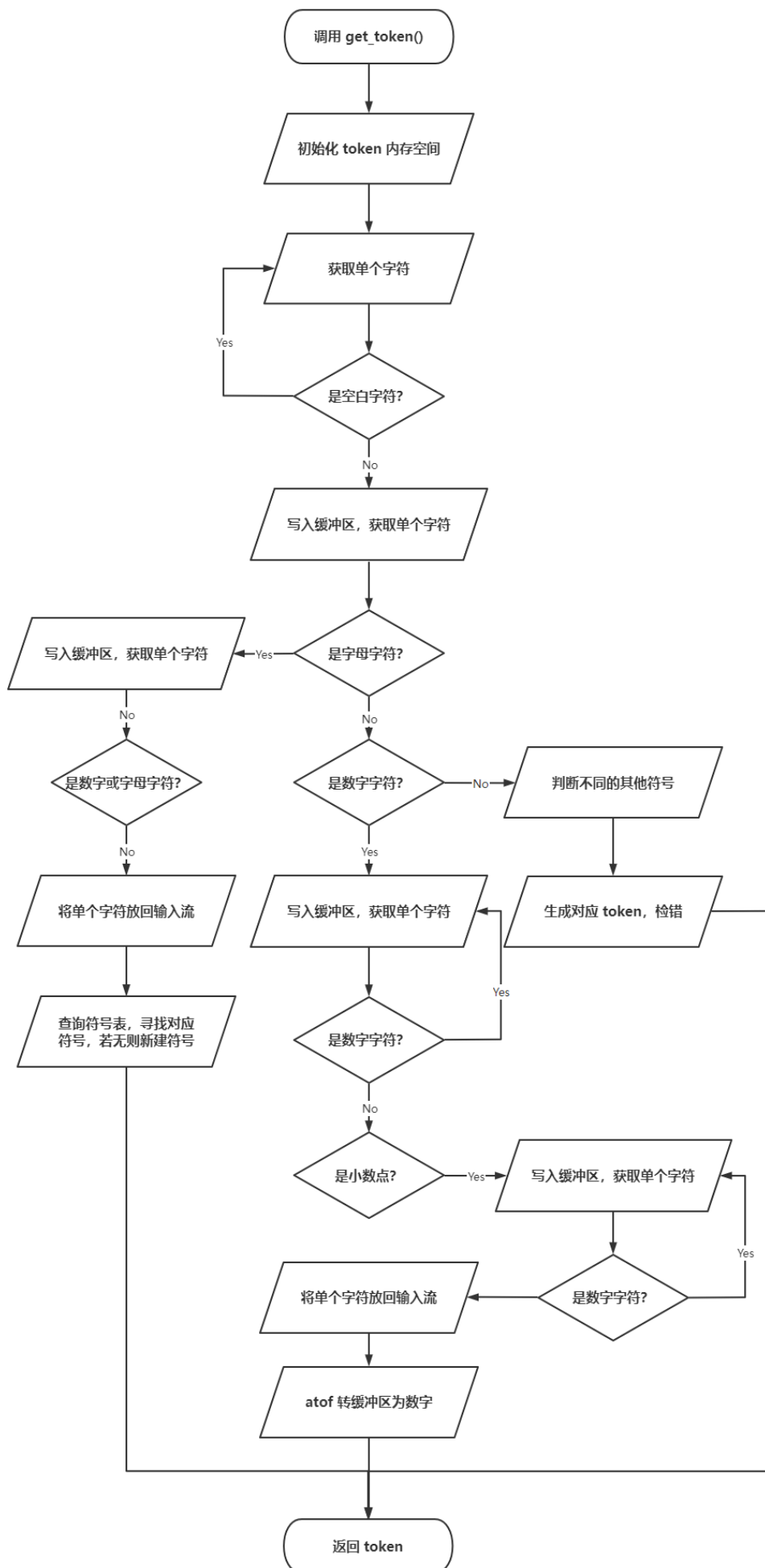
为了达成对预设关键字的解析，笔者预定义了一个内置的符号表 `bulit_in_token_table`，为了支持自定义变量，笔者还定义了一个追加符号表 `append_token_table`，由两个符号表组成整个编译器的符号表。

(3) 词法分析的主要流程

笔者选择使用有限状态自动机来实现词法分析器，其从文件中读取单个记号的流程如下：

- 清除前导空白字符
- 获取第一个非空白字符进行判断
- 若第一个非空白字符为字母则不断读取单个字符到缓冲区直到遇到非数字或非字母字符，将之放回输入流，缓冲区内内存留字符作为一个记号，查询符号表并返回对应的记号（若符号表中不存在则视为新的变量，追加到符号表中）
- 若第一个非空白字符为数字则不断读取单个字符到缓冲区直到遇到非数字字符（若遇到小数点符号，则重复读取单个字符到缓冲区直到遇到非数字字符），将之放回输入流，通过 `atof()` 将缓冲区内字符转换为浮点值赋给一个新的数字记号并返回
- 若第一个非空白字符为其他分隔字符则进行针对性的模式匹配
- 其余则为非法字符，进行报错

整体流程图如下所示



3.语法分析器 (Syntactic Analyzer, aka Parser)

(1) 语法分析器的主要功能

语法分析器根据记号流识别句子，并为表达式构造语法树，主要有：

- 通过输入构造抽象语法树 (Abstract Syntax Tree)
- 检测输入序列的合法性

(2) 文法

笔者设计的适合递归下降分析算法的文法如下：

```
Expression → Term { (PLUS | MINUS) Term }
Term       → Factor { (MUL | DIV) Factor }
Factor     → PLUS Factor | MINUS Factor | Component
Component  → Atom [POWER Component]
Atom       → CONST_ID
           | T
           | FUNC L_BRACKET Expression R_BRACKET
           | L_BRACKET Expression R_BRACKET
```

(3) 表达式的语法树

<1> 语法树的节点

表达式语法树的节点可以分为以下三类：

- 叶子节点：常数、变量等
- 两个孩子的内部节点：二元运算如Plus、Mul等
 - 一元加：+5 转化为 0+5
 - 一元减：-5 转化为 0-5
- 一个孩子的内部节点：函数调用，如cos(t)等

<2> 语法树节点的数据结构

笔者将语法树的节点使用如下数据结构进行描述：

```
/*
 * definition of a node of the abstract syntax tree
 */
struct expr_node
{
    enum token_type opcode;
    union
    {
        struct                                // tree node
        {
            struct expr_node *left;
            struct expr_node *right;
        }case_operator;                    // the node is a operator
        struct
        {
            struct expr_node *child;
            double (*func_ptr)(double);
        }case_func;                       // the node is a call of function
    }
};
```



```

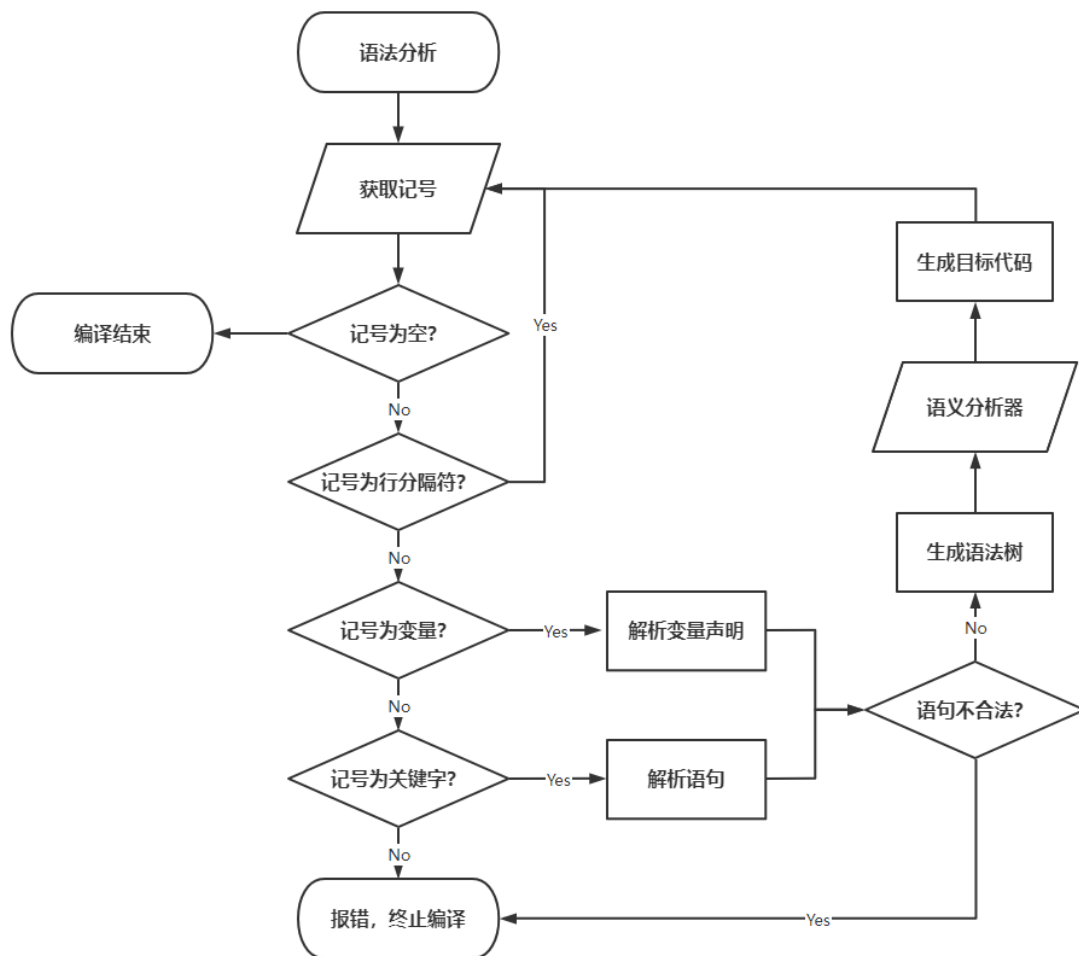
double case_const;           // the node is a const value   (right
val)
double *case_param;         // the node is a param         (left
val)
}content;
};

```

(4) 语法分析器的实现

笔者选择使用递归下降分析算法来完成语法分析器的工作。

整体流程图如下所示



4.语义分析器

(1) 语义分析器的主要功能

语义分析器的主要功能为根据语言结构，处理函数绘图语言程序的语义，主要有以下两点

- 深度优先后序遍历语法分析器生成的语法树，进行表达式的值的计算
- 生成绘图代码（python 代码，使用 tkinter 库进行绘图）

(2) 语义分析器的工作流程

- 从 origin、rot、scale 语句中获得坐标变换所需信息
- 从变量声明语句中记录新的变量
- for-draw 绘图语句根据变量的每一个值进行如下处理：
 - 计算被绘制点的横、纵坐标值

- 根据坐标变换信息进行坐标变换，得到实际坐标
- 根据点的实际坐标生成画出该点的代码（python）

(3) 辅助语义函数

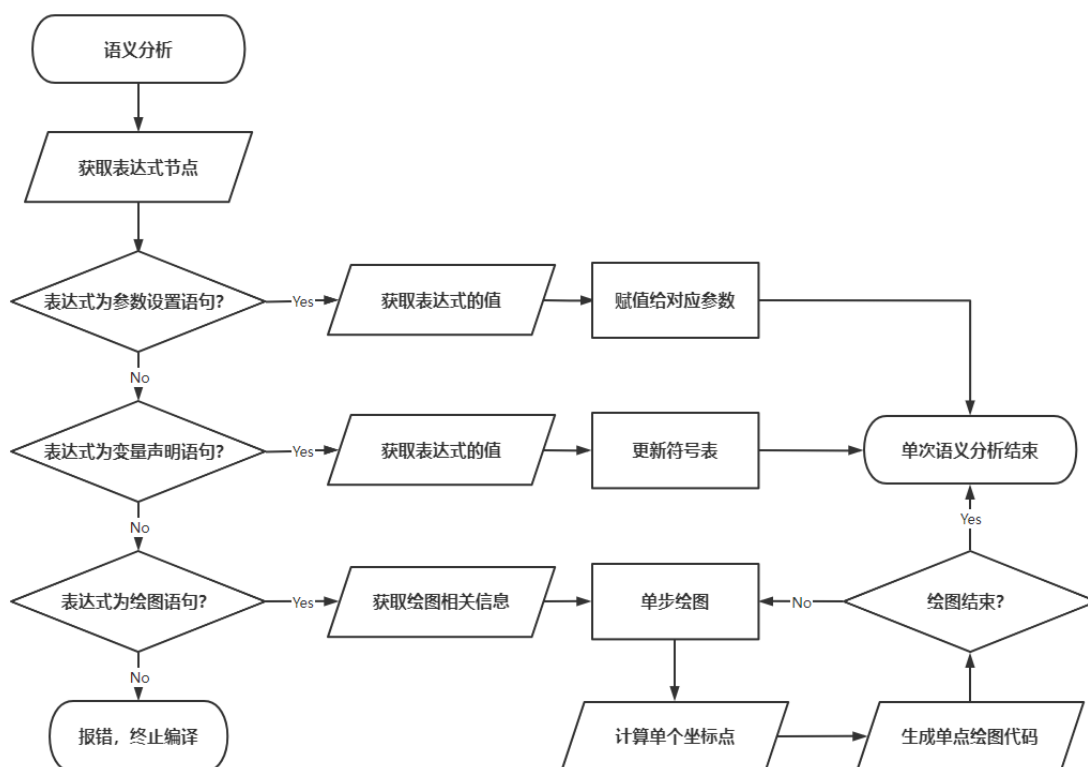
- `double get_expr_val(struct expr_node *tree)`: 计算表达式的值
- `void get_coordinate(struct expr_node *exp_x, struct expr_node *exp_y, double &x, double &y)`: 获得一个点的坐标
- `void draw_loop(struct token *param, double start, double end, double step, struct expr_node *exp_x, struct expr_node *exp_y)`: 由语法树绘制该表达式中所有的点
- `void draw_pixel(double x, double y, double step)`: 核心绘图函数，生成绘制一个点的python 代码

(4) 语义分析器的实现

为了设计方便，笔者将语义分析作为语法分析的中间流程，与语法分析并行工作（可以参见语法分析的流程图）。

- 对于参数设置语句（origin、rot、scale），在每次调用 `expression()` 之后就使用 `gext_expr_val()` 获取对应表达式的值，赋值给全局变量
- 对于变量声明语句，先对符号表进行查询，若不存在该符号则新建符号，之后调用 `gext_expr_val()` 来获取变量的值，记录到符号表中
- 对于绘图语句，则调用 `draw_loop()` 函数来生成绘图代码，并写入到文件中

语义分析的整体流程图如下所示：



五、心得体会

本次实验虽然难度并不算高，但独立完成整个编译器的设置让笔者受益匪浅

六、实验结果

笔者本次实验代码同步上传 GitHub，地址：https://github.com/arttnba3/compiler_principles

我们的源码是使用 cmake 进行部署的，因此可以很方便地跨平台编译运行

首先创建一个新文件夹 `build`，笔者将在这里面通过 cmake 完成项目构建

```
mkdir build
cd build
```















然后使用 cmake 自动进行构建

```
cmake ..
```

Windows 环境下编译

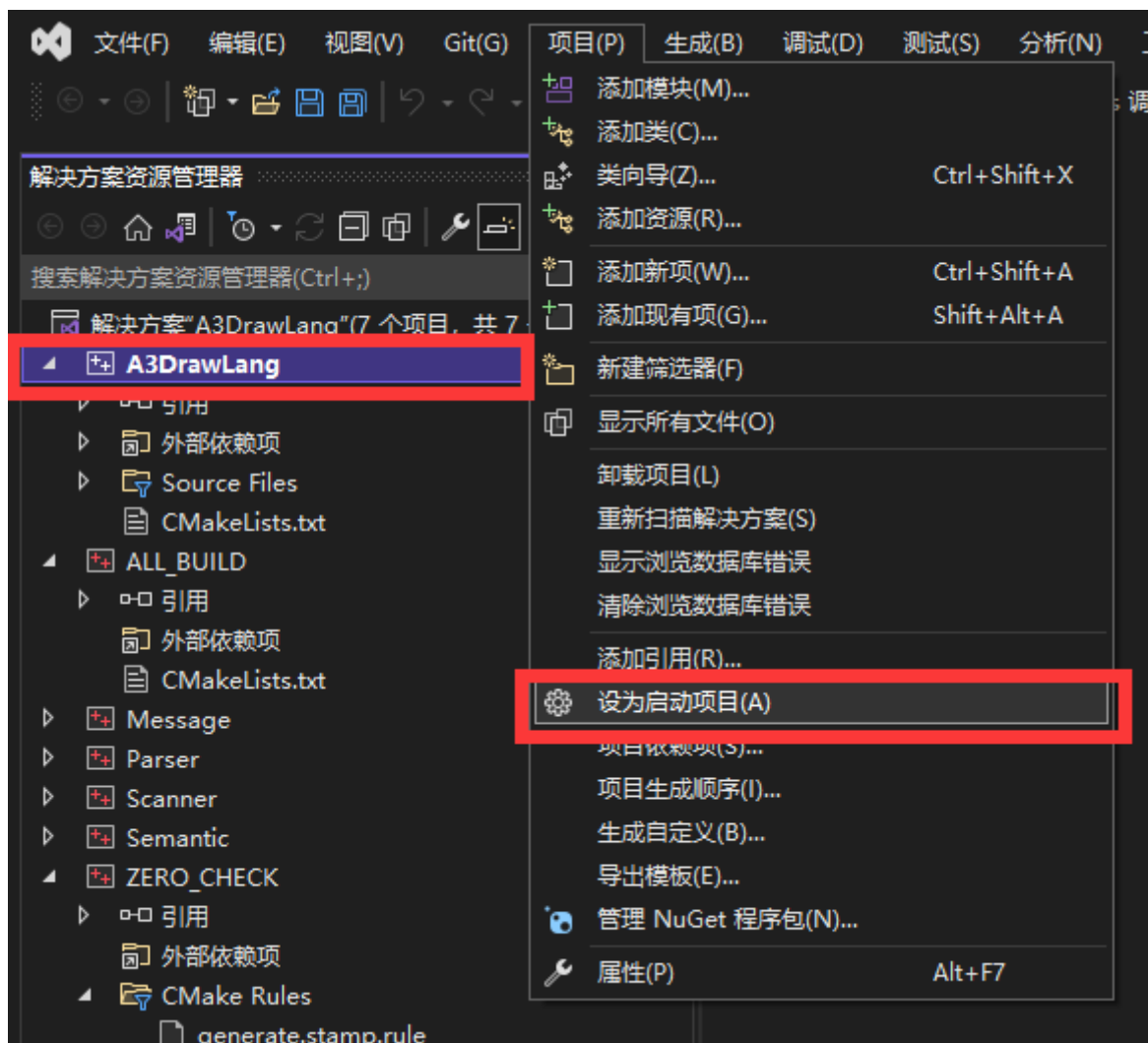
对于 Windows 系统，需要提前安装好 visual studio

cmake 构建完成之后会在我们创建的 `build` 目录下生成一个 visual studio project，如下：

	CMakeFiles	2021/11/19 10:48	文件夹	
	message	2021/11/19 10:48	文件夹	
	parser	2021/11/19 10:48	文件夹	
	scanner	2021/11/19 10:48	文件夹	
	semantic	2021/11/19 10:48	文件夹	
	A3DrawLang.sln	2021/11/19 10:48	Microsoft Visual...	8 KB
	A3DrawLang.vcxproj	2021/11/19 10:48	VC++ Project	54 KB
	A3DrawLang.vcxproj.filters	2021/11/19 10:48	VC++ Project Fil...	1 KB
	ALL_BUILD.vcxproj	2021/11/19 10:48	VC++ Project	43 KB
	ALL_BUILD.vcxproj.filters	2021/11/19 10:48	VC++ Project Fil...	1 KB
	cmake_install.cmake	2021/11/19 10:48	CMake 源文件	3 KB
	CMakeCache.txt	2021/11/19 10:48	文本文档	14 KB
	ZERO_CHECK.vcxproj	2021/11/19 10:48	VC++ Project	44 KB
	ZERO_CHECK.vcxproj.filters	2021/11/19 10:48	VC++ Project Fil...	1 KB

然后直接双击 `A3DrawLang.sln`，用 visual studio 打开即可

需要注意的是这里我们需要**手动设置启动项为 A3DrawLang**，在左边的解决方案管理器中选中 `A3DrawLang`，然后点击 `项目`，点击 `设为启动项目`



选中 生成，点击 生成解决方案

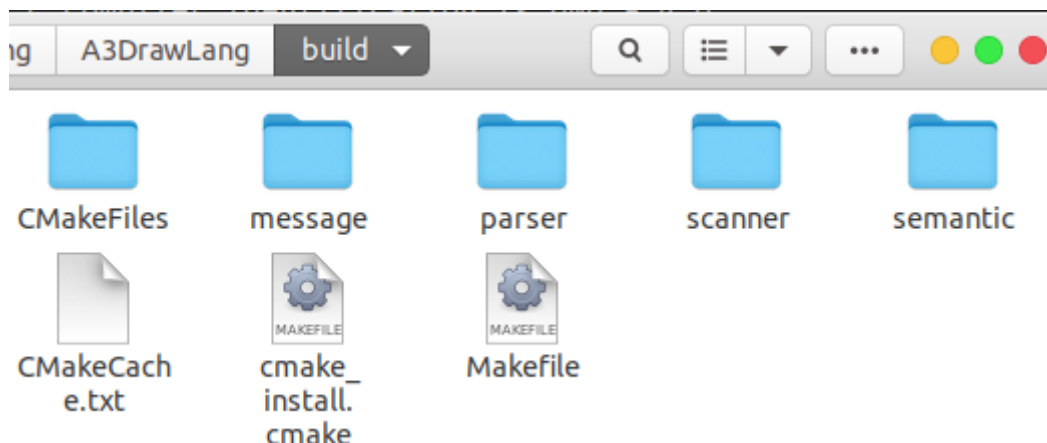


在当前目录下会生成一个 `Debug` 文件夹，其中会有一个可执行文件 `A3DrawLang.exe`，这便是我们的编译器了

Linux 环境下编译

对于 Linux 系统，需要提前安装好 gcc 组件及 makefile

make 构建完成之后会在我们创建的 build 目录下复制一份源码并生成对应的 makefile 文件，如下：



直接在当前目录下打开 shell 输入：

```
make
```

之后一个可执行文件 `A3DrawLang` 就会出现在当前目录下，这便是我们的编译器了

Running

命令执行格式如下，我们需要手动指定代码源文件路径作为第一个参数：

```
./A3compiler [object file] [opt]
```

对我们输入的绘图代码源文件编译成功之后会在当前目录下生成 `draw.py` 文件，你可以选择直接运行也可以不运行，默认是直接运行

```
./A3DrawLang example.a3lang
Start to parse...
Done.
Interpretation done. run it now? [Y/n]
n
Result saved in ./drawer.py
```

当你指定 `opt` 参数为 `test` 时，会逐个输出文件中读到的 token

```
./A3DrawLang ./example.a3lang test
Testing Token...
token 0: letme: ROT type: 2 val: 0 func ptr: 0
token 1: letme: IS type: 3 val: 0 func ptr: 0
token 2: letme: 0 type: 20 val: 0 func ptr: 0
token 3: letme: NULL type: 10 val: 0 func ptr: 0
...
token 174: letme: NULL type: 12 val: 0 func ptr: 0
token 175: letme: NULL type: 10 val: 0 func ptr: 0
```

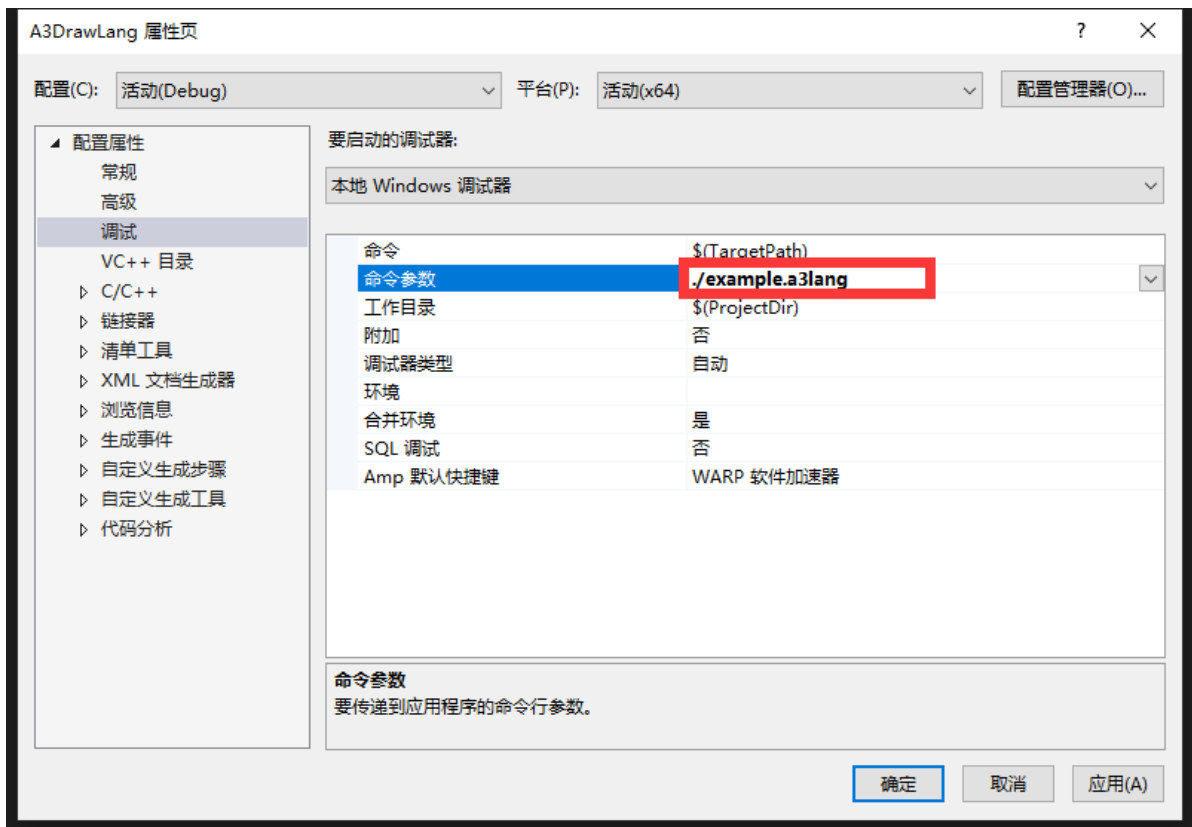
**for visual studio*

在 Windows 下除了通过命令行运行之外，也可以直接用 visual studio 来运行，这里我们需要额外指定源文件路径作为参数

选中 项目->属性



在 配置属性->调试 中设置 命令参数，输入的代码源文件应当同样放在 build 目录下



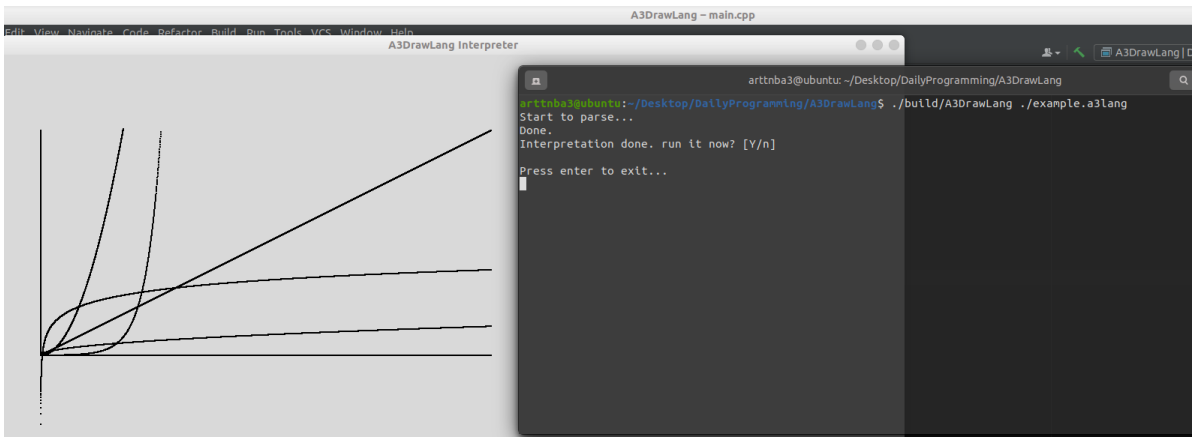
测试例程

测试例程如下：

```
rot is 0;
origin is (50, 400);
scale is (2, 1);
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 300 step 0.01 draw (t, -t);
scale is (2, 0.1);
for T from 0 to 55 step 0.01 draw (t, -(t*t));
scale is (10, 5);
for T from 0 to 60 step 0.01 draw (t, -sqrt(t));
scale is (20, 0.1);
for T from 0 to 8 step 0.01 draw (t, -exp(t));
scale is (2, 20);
for T from 0 to 300 step 0.01 draw (t, -ln(t));
```

运行效果如下图所示，演示视频见邮件附件

Linux



Windows

