

燕山大学

编译原理课程设计报告

题目 基于编译原理的公式计算器设计与实现

目录

[基于编译原理的公式计算器设计与实现 1](#_Toc60301420)

[1 概述 1](#_Toc60301421)

[1.2课程设计要求 1](#_Toc60301422)

[1.3课程设计计划 1](#_Toc60301423)

[1.4使用的开发工具 1](#_Toc60301424)

[2 使用的基本概念和原理 2](#_Toc60301425)

[2.1 EBNF范式 2](#_Toc60301426)

[2.2确定的有穷自动机(DFA) 2](#_Toc60301427)

[2.3词法分析程序和语法分析程序的接口方式 2](#_Toc60301428)

[3 总体设计 3](#_Toc60301429)

[4 详细设计 5](#_Toc60301430)

[4.1异常类 5](#_Toc60301431)

[4.2计算器类 5](#_Toc60301432)

[4.2.1词法分析 5](#_Toc60301433)

[4.2.3语法分析 7](#_Toc60301434)

[4.2.4前缀表达式计算 7](#_Toc60301435)

[4.2.5后缀表达式计算 8](#_Toc60301436)

[4.3界面类 9](#_Toc60301437)

[4.4主界面类 10](#_Toc60301438)

[5 编码实现 11](#_Toc60301439)

[5.1开发环境 11](#_Toc60301440)

[5.2程序设计的注意事项 11](#_Toc60301441)

[5.3主要程序的代码设计及注释 11](#_Toc60301442)

[5.4解决的技术难点 21](#_Toc60301443)

[6 测试和试运行 22](#_Toc60301444)

[7 总结 23](#_Toc60301445)

[8 参考文献 24](#_Toc60301446)

基于编译原理的公式计算器设计与实现

# 1 概述

本文利用编译原理设计编译器的基本思想，基于python3.8以及pycharm集成开发环境，设计并实现一个简单的公式计算器。该公式计算器通过对输入公式的词法分析、语法分析以及中间代码生成各个阶段，实现了对输入公式的词法语法检查以及计算，同时将计算生成的值存在单词表中，支持赋值语句以及+、-、\*、/基本的四则运算以及(左括号、)右括号、%取模、//取整、^乘幂运算符。

## 1.2课程设计要求

（1）设计及实现计算表达式的公式计算器，表达式中至少包含＋、－、×、／、（、）等运算符。

（2）对于一个给定的表达式，能够检查有无词法错误和语法错误，若有错误能指定出错位置和错误类型。

（3）给出该表达式的中间表示形式，前缀和后缀表达式。

（4）输出运算结果，达到公式计算的目的。

（4）准备多组测试样例，对测试样例进行测试和验证，并对输出结果进行分析。

## 1.3课程设计计划

12月28日查阅相关资料，构建整体思路，设计整体流程；12月29日到30日上机实现，使用C++编程语言编写词法分析器并测试；12月31日撰写课程设计报告。

## 1.4使用的开发工具

编程环境：Pycharm 开发语言：Python

# 2 使用的基本概念和原理

## 2.1 EBNF范式

扩展巴科斯-瑙尔范式(Extended Backus–Naur Form,EBNF)是一种用于描述计算机编程语言等正式语言的与上下文无关语法的元语法(metasyntax)符号表示法。简而言之，它是一种描述语言的语言。它是基本巴科斯范式(BNF)元语法符号表示法的一种扩展。本文通过对公式的文法结构进行分析，设计出了响应的EBNF范式。

## 2.2确定的有穷自动机(DFA)

有穷自动机（也成有限自动机）作为一种识别装置，能够准确地识别正规集，即识别正规文法所定义的语言和正规式所表示的集合。有穷自动机分为两种：确定的有穷自动机（Deterministic Finite Automata，DFA）和不确定的有穷自动机（Nondeterministic Finite Automata，NFA），本次课程设计主要采用了确定的有穷自动机。

一个确定的有穷自动机包括三种状态：

1. 初始状态（initial state）：自动机的开始状态
2. 终止状态（final state）：一个DFA至少有一个终止状态
3. 中间状态

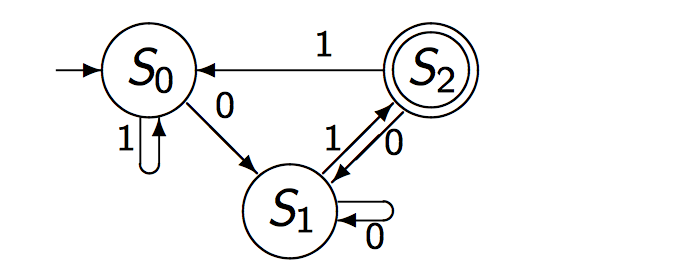


图2-1 DFA图

## 2.3词法分析程序和语法分析程序的接口方式

词法分析工作可以是独立的一遍,把字符流的源程序变为单词序列,输出到一个中间文件,这个文件作为语法分析程序的输入而继续编译过程。然而，更一般情况是将将词法分析程序设计成一个子程序,每当语法分析程序需要一个单词时,则调该子程序。词法分析程序每得到一次调用,便从源程序文件中读入一些字符,直到识别出一个单词,或说直到下一个单词的第一个字符为止。这种设计方案中,词法分析程序和语法分析程序放在同一遍里,省掉中间文件或存储区。

本文采用的第一种方式，即把字符流的源程序变为单词序列，存储在内存中。

# 3 总体设计

采用面向对象的设计方法，将编译执行过程封装为一个Calculate类，实现对输入字符串的词法分析、语法分析、前缀表达式和后缀表达式的计算以及出错处理；界面类Ui\_Form通过QtDesigner工具设计以及Pyuic自动生成，包括界面布局以及控件的基本属性。主界面类（界面逻辑）My\_ui通过继承Ui\_Form类，可直接使用其父类的控件，当界面布局发生变化，或者控件变动时，只需修改Ui\_Form类即可。主界面类实现了基本的按钮点击事件，通过调用计算器类Calculator，可以对用户输入的文本进行处理，实时显示在界面上。具体的UML类图如下：

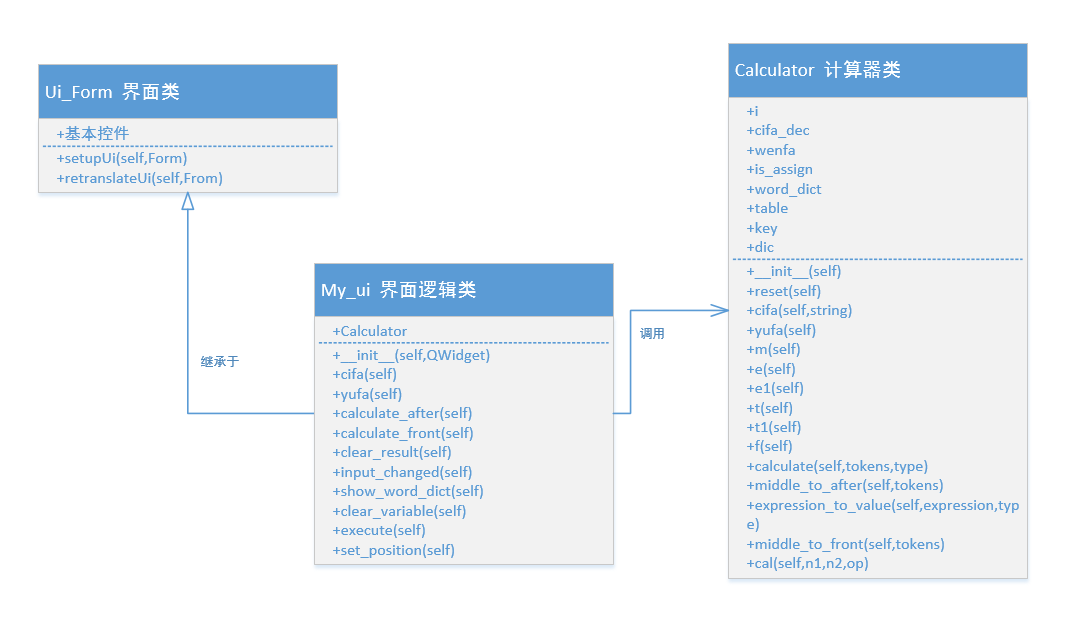


图3-1 UML类图

其中计算器类中进行词法、语法分析遇到错误时，会抛出自定义的异常，主界面类捕获异常，实时显示在界面上。其中异常种类如图3-2所示：



图3-2 异常类图



图3-3 计算器类数据流图

用户输出字符串的数据流图如图3.3所示，通过词法分析转换成tokens并存在table中，同时判断是否为赋值语句，并把每个单词类型存在dic中，tokens经过语法分析判断是否符合文法规则，如果不符合文法规则则抛出异常，符合即执行下一步：前缀、后缀表达式的转换，结合是否为赋值语句，把计算结果存在待赋值的标识符里，最终把计算结果输出到界面中。

# 4 详细设计

## 4.1异常类

定义异常基类(BaseError)，其中初始化参数为位置(position,默认为0)以及出错字符(char,默认为None)。其子类通过继承BaseError类，实现了这两个参数的传值，每个子类都有不同的错误声明，体现在\_\_str\_\_()函数中，具体的异常及声明如下表：

表4 1 异常类表

|  |  |  |
| --- | --- | --- |
| 异常类名 | 异常种类 | 错误声明 |
| PointError | 小数点出错 | 词法分析出错，位置x，小数点错误 |
| YufaError | 语法分析出错 | 语法分析出错，错误位置x,声明x |
| EOFError | 括号不匹配 | 语法分析出错，位置x,括号不匹配 |
| IdentifierDefineError | 标识符定义错误 | 词法分析出错，位置x，标识符中必须由数字和字母组成，且必须以字母开头 |
| NumberError | 数字串定义错误 | 词法分析出错，位置x，数字串中不能包含字母x |
| InvalidCharError | 非法符号错误 | 词法分析出错，位置x，检测到非法字符：x |
| EqualSignError | 等号错误 | 词法分析错误，‘=‘过多 |
| IndentiNotFountError | 未定义标识符 | 计算出错，标识符x没有在单词表中 |
| ZeroDivError | 除零错误 | 除数不能为零 |

## 4.2计算器类

计算器类(Calculator)实现了对输入字符串的词法、语法分析以及前缀、后缀表达式的计算。

### 4.2.1词法分析

词法分析通过对输入字符串的检查，将输入的字符串转化为标记(token)，符号种类包括：标识符、数字、运算符，具体的识别流程如下图。



图4 1 词法分析流程图

该部分通过对整个输入字符串处理，达到分割以及提取标识符的作用，对于标识符、数字以及运算符的识别，采用如图4-2的DFA进行识别。

整个过程在Calculator类中的cifa()函数中实现，函数的参数为待处理的字符串，返回值为处理之后的token，并存在table成员属性中，同时也会在Calculator中的cifa\_dec属性中生成对符号表的描述，is\_assign属性表示该字符串是否为赋值语句，如果为赋值语句，则将key属性设置为待赋值的标识符；dic为字典类型，存储着该字符串中的所有token及其类型。



图4 2词法分析DFA

### 4.2.3语法分析

定义表达式的基本文法为：

基本文法：

（1）M -> +E|-E|E

（2）E -> TE~

（3）E~ -> +TE~|-TE~|&

（4）T -> FT~

（5）T~ -> \*FT~|/FT~|&|^FT~|%FT~|//FT~

（6）F -> (E)|indentifier|digit

按照该文法设计递归下降子程序如图3-1中的Calculator类中的yufa、m、e、e1、t、t1、f函数。通过成员变量self.i，判断当前读取的token，在执行这些函数的过程中，若不符合文法规则，则抛出文法异常。

### 4.2.4前缀表达式计算

通过对输入字符串的词法、语法分析，得到了符合文法特点的tokens，将这些tokens进行前缀表达式的转换，实现了计算。算法具体流程如下：

(1) 初始化两个栈：运算符栈ops和储存中间结果的栈expression；

(2) 从右至左扫描中缀表达式；

(3) 遇到操作数时，将其压入expression；

(4) 遇到运算符时，比较其与ops栈顶运算符的优先级：

(4-1) 如果expression为空，或栈顶运算符为右括号“)”，则直接将此运算符入栈；

(4-2) 否则，若优先级比栈顶运算符的较高或相等，也将运算符压入OPS；

(4-3) 否则，将ops栈顶的运算符弹出并压入到expression中，再次转到(4-1)与ops中新的栈顶运算符相比较；

(5) 遇到括号时：

(5-1) 如果是右括号“)”，则直接压入ops；

(5-2) 如果是左括号“(”，则依次弹出ops栈顶的运算符，并压入expression，直到遇到右括号为止，此时将这一对括号丢弃；

(6) 重复步骤(2)至(5)，直到表达式的最左边；

(7) 将ops中剩余的运算符依次弹出并压入expression；

(8) 依次弹出expression中的元素并输出，结果即为中缀表达式对应的前缀表达式。

计算算法：

从右至左扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算(栈顶元素ops次顶元素)并将结果入栈；重复上述过程直到表达式最左端，最后运算得出的值即为表达式的结果。

### 4.2.5后缀表达式计算

后缀表达式类似于前缀表达式，具体算法流程如下：

(1) 初始化两个栈：运算符栈S1和储存中间结果的栈S2；

(2) 从左至右扫描中缀表达式；

(3) 遇到操作数时，将其压入S2；

(4) 遇到运算符时，比较其与S1栈顶运算符的优先级：

(4-1) 如果S1为空，或栈顶运算符为左括号“(”，则直接将此运算符入栈；

(4-2) 否则，若优先级比栈顶运算符的高，也将运算符压入S1（注意转换为前缀表达式时是优先级较高或相同，而这里则不包括相同的情况）；

(4-3) 否则，将S1栈顶的运算符弹出并压入到S2中，再次转到(4-1)与S1中新的栈顶运算符相比较；

(5) 遇到括号时：

(5-1) 如果是左括号“(”，则直接压入S1；

(5-2) 如果是右括号“)”，则依次弹出S1栈顶的运算符，并压入S2，直到遇到左括号为止，此时将这一对括号丢弃；

(6) 重复步骤(2)至(5)，直到表达式的最右边；

(7) 将S1中剩余的运算符依次弹出并压入S2；

(8) 依次弹出S2中的元素并输出，结果的逆序即为中缀表达式对应的后缀表达式（转换为前缀表达式时不用逆序）。

计算算法：

与前缀表达式类似，只是顺序是从左至右：

从左至右扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（次顶元素op栈顶元素），并将结果入栈；重复上述过程直到表达式最右端，最后运算得出的值即为表达式的结果。

## 4.3界面类

界面类主要通过QtDesinger设计，通过pyuic工具自动转为python代码。包括表达式输入区、变量显示区以及结果的显示，按钮有词法分析、语法分析、前缀表达式求值、后缀表达式求值、编译并执行、以及清空输出和变量按钮。采用Qt中的网格布局，在右下角显示当前在输入区光标的位置，具体界面如下图。

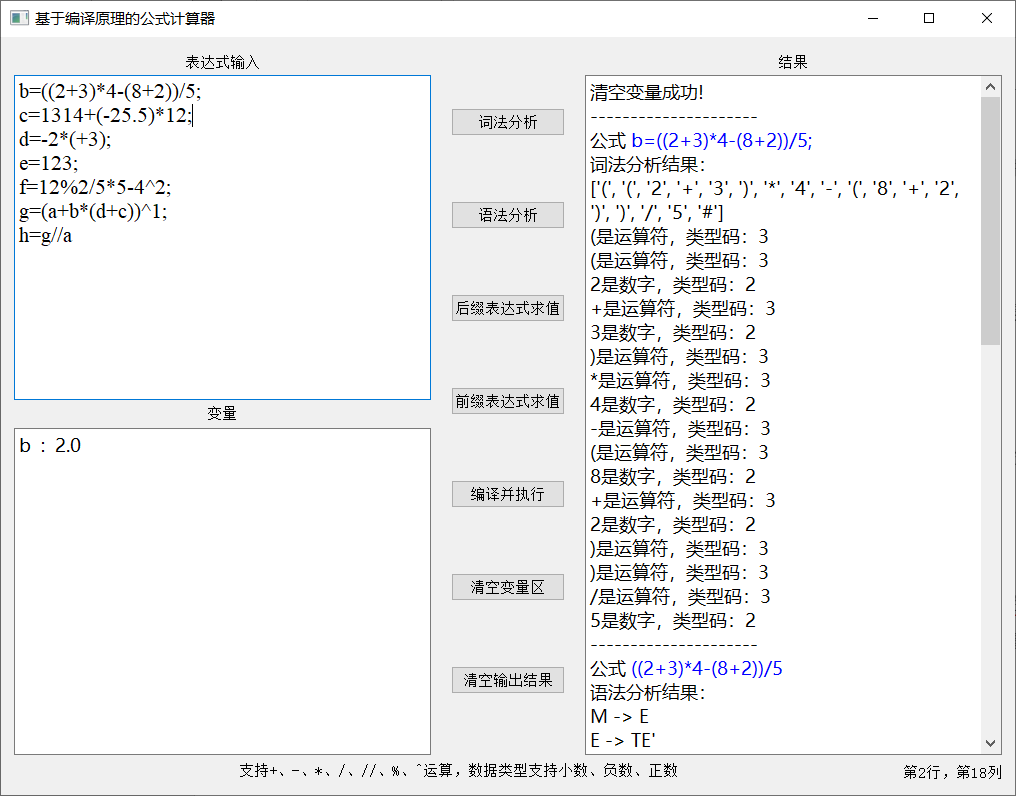


图4-3程序界面图

## 4.4主界面类

主界面类My\_ui实现了界面的逻辑，绑定按钮的点击事件。通过对成员属性cal进行操作，实现对公式的处理。在处理过程中利用异常判断，实时在界面上的结果显示区显示执行结果，操作友好。下面以语法分析的代码为例，解释该代码的执行逻辑：

def yufa(self):

try:

wenfa = self.cal.yufa()

self.show\_textBrowser.append("公式 <font color = blue> %s </font>" % "".join(self.cal.table[:-1]))

self.show\_textBrowser.append("语法分析结果：\n" + "\n".join(wenfa))

self.after\_btn.setEnabled(True)

self.front\_btn.setEnabled(True)

self.show\_textBrowser.append("---------------------")

except Exception as e:

self.show\_textBrowser.append("<font color = red>%s</font>" % str(e))

self.show\_textBrowser.append("语法分析结果：\n" + "\n".join(self.cal.wenfa))

self.show\_textBrowser.append("---------------------")

try中保护的代码为self.cal.yufa(),当该段代码抛出异常时，执行except块内的语句，通过qt中textBrowser控件对出错信息标红显示。

# 5 编码实现

## 5.1开发环境

Python版本为3.8、PyQt5，IDE: PyCharm2020.2.4(Professional Edition), windows10系统。

## 5.2程序设计的注意事项

（1）python环境中有内置的eval()函数，可直接计算字符串形式的表达式，但本次设计避免使用该函数，主要通过对编译原理中的词法分析、语法分析、以及中间代码生成的几个阶段，设计并实现了基于编译原理的公式计算器。

（2）在实现词法分析的过程中，避免使用正则表达式，通过自己手动实现逐个字符的检测，读取token并判断类型。

## 5.3主要程序的代码设计及注释

主要列举计算器类的代码。

class Calculator:

def \_\_init\_\_(self):

self.i = 0 # 栈指针

self.dic = {} # 符号表

self.table = [] # 单词栈

self.wenfa = [] # 字符串文法

self.cifa\_dec = [] # 词法分析的描述

self.is\_assign = False # 是否为赋值语句

self.key = None # 如果是赋值语句，单词是什么

self.word\_dict = {} # 单词表

def reset(self):

self.dic = {}

self.table = []

self.wenfa = [] # 字符串文法

self.cifa\_dec = [] # 词法分析的描述

self.is\_assign = False # 是否为赋值语句

self.key = None # 如果是赋值语句，单词是什么

################################### 词法分析 ######################################

def cifa(self, string): # 词法分析

self.reset()

string = string.replace(" ", "")

# 如果是赋值语句

if '=' in string:

try:

left\_str, right\_str = string.split("=") # 只能包含一个等号

except:

raise EqualSignError

# 开头必须为字母

if left\_str[0] not in letter: raise IdentifierDefineError

# 左侧必须为标识符

for i in range(len(left\_str)):

if left\_str[i] not in letter + number:

raise IdentifierDefineError(i, "标识符中必须由数字和字母组成")

string = right\_str

self.key = left\_str

self.is\_assign = True

self.table = []

print('')

m = 0 # 数字或标识符的开始位置

point = False # 是否允许出现小数点

state = None # 1：为标识符 2：为数字串 3：为运算符

next = False # 当检测到“//”时要前进两次

for i in range(len(string)):

# if string[i]!="-":

if next:

next = False

continue

if string[i] in operator\_list: # 如果是运算符

if state == identifier: # state=1表明其前面的为标识符

self.cifa\_dec.append(string[m:i] + '是标识符,类型码：1')

self.dic[string[m:i]] = identifier

self.table.append(string[m:i])

elif state == digit: # state=2表明其前面的为数字

self.cifa\_dec.append(string[m:i] + '是数字，类型码：2')

self.dic[string[m:i]] = digit

self.table.append(string[m:i])

point = False

if string[i] == '-': # 负数

if i > 0:

if string[i - 1] == "(":

point = True

state = digit

continue

elif state == None:

point = True

state = digit

continue

if string[i] == '+': # 正数

if i > 0:

if string[i - 1] == "(":

point = True

state = digit

continue

elif state == None:

point = True

state = digit

continue

if string[i:i + 2] == '//':

state = operator

self.cifa\_dec.append(string[i:i + 2] + '是运算符，类型码：3')

self.dic[string[i:i + 2]] = operator

self.table.append(string[i:i + 2])

m = i + 2

next = True

continue

m = i + 1

state = operator

self.cifa\_dec.append(string[i] + '是运算符，类型码：3')

self.dic[string[i]] = operator

self.table.append(string[i])

elif string[i] in number: # 如果是数字

if i == m: # 是数字的第一个字符

point = True

state = digit

elif string[i] in letter: # 如果是字母

if state == digit: # 判断此时的状态，若state=2表明状态为无符号整数，而整数内不能包含字母，故报错

print('词法分析检测到错误,数字串中不能包含字母')

raise NumberError(i, string[i])

if i == m: # 判断此时的字母是否为标识符的第一个字母，若是则改变状态为标识符

state = identifier

point = False

elif string[i] == '.':

if point == True:

continue

else:

print("小数点错误")

raise PointError(i)

else: # 当输入的字符均不符合以上判断，则说明为非法字符，故报错

print('词法分析检测到非法字符')

raise InvalidCharError(i, string[i])

if state == identifier: # 当字符串检查完后，若字符串最后部分为标识符，应将其print出来

self.cifa\_dec.append(string[m:] + '是标识符，类型码：3')

self.dic[string[m:]] = identifier

self.table.append(string[m:])

elif state == digit: # 若字符串最后部分为无符号整数，应将其print出来

self.cifa\_dec.append(string[m:] + '是数字，类型码：2')

self.dic[string[m:]] = digit

self.table.append(string[m:])

self.table.append('#')

print('字符栈:', self.table, '\n词法正确')

return self.table

################################### 语法分析 ######################################

def yufa(self):

self.i = 0 # 指针归零

self.wenfa = []

self.m()

return self.wenfa

def m(self): # PM程序

if (self.table[self.i] == '+'):

self.i += 1

self.wenfa.append('M -> +E')

self.e()

elif (self.table[self.i] == '-'):

self.i += 1

self.wenfa.append('M -> -E')

self.e()

else:

self.wenfa.append('M -> E')

self.e()

if (self.i is not len(self.table) - 1): # 语法分析结束时，若单词栈指针与单词表长度不相等，报错

print("\n语法分析程序检查到错误,'('前应该有运算符")

raise EOFError(self.i)

else:

print('\n字符串语法是：') # 若一切正确，则输出语法树文法

for i in self.wenfa:

print(i)

print('语法正确')

def e(self): # PE程序

self.wenfa.append('E -> TE\'')

self.t()

self.e1()

def e1(self): # PE1程序

if (self.table[self.i] == '+'):

self.i += 1

self.wenfa.append('E\' -> +TE\'')

self.t()

self.e1()

elif (self.table[self.i] == '-'):

self.i += 1

self.wenfa.append('E\' -> -TE\'')

self.t()

self.e1()

else:

self.wenfa.append('E\' -> &')

def t(self): # PT程序

self.wenfa.append('T -> FT\'')

self.f()

self.t1()

def t1(self): # PT1程序

if (self.table[self.i] == '\*'):

self.i += 1

self.wenfa.append('T\' -> \*FT\'')

self.f()

self.t1()

elif (self.table[self.i] == '/'):

self.i += 1

self.wenfa.append('T\' -> /FT\'')

self.f()

self.t1()

elif (self.table[self.i] == '^'):

self.i += 1

self.wenfa.append('T\' -> ^FT\'')

self.f()

self.t1()

elif (self.table[self.i] == '%'):

self.i += 1

self.wenfa.append('T\' -> %FT\'')

self.f()

self.t1()

elif (self.table[self.i] == '//'):

self.i += 1

self.wenfa.append('T\' -> //FT\'')

self.f()

self.t1()

else:

self.wenfa.append('T\' -> &')

def f(self): # PF程序

if (self.table[self.i] == '('):

self.wenfa.append('F -> (E)')

self.i += 1

self.e()

if (self.table[self.i] != ')'):

raise YufaError(position=self.i, dec='词法分析错误，该处应有“)”')

self.i += 1

elif (self.dic[self.table[self.i]] == identifier):

self.wenfa.append('F -> identifier ' + str(self.table[self.i]))

self.i += 1

elif (self.dic[self.table[self.i]] == digit):

self.wenfa.append('F -> Digit ' + str(self.table[self.i]))

self.i += 1

else:

raise YufaError(self.i, "因子只能由(E)或者标识符或者数字组成，出错字符：{}，字符类型：{}".format(self.table[self.i], self.dic[

self.table[self.i]])) # 若均不符合，则引出异常

def calculate(self, tokens=None, type='after'):

if tokens == None:

tokens = self.table

if type == 'after':

expression = self.middle\_to\_after(tokens[:-1])

elif type == 'front':

expression = self.middle\_to\_front(tokens[:-1])

else:

raise TypeError

result = self.expression\_to\_value(expression, type)

if self.is\_assign:

self.word\_dict[self.key] = result

return expression, result

def middle\_to\_after(self, tokens):

"""中缀表达式变为后缀表达式"""

expression = [] # s2

ops = [] # s1

for item in tokens:

# 当遇到运算符

if item in ['+', '-', '\*', '/', "^", '%', '//']:

while len(ops) >= 0:

# 如果栈中没有运算符，直接将运算符添加到后缀表达式

if len(ops) == 0:

ops.append(item)

break

# 如果栈中有运算符

op = ops.pop()

# 如果栈顶的运算符比当前运算符级别低，当前运算符入栈

if op == '(' or ops\_rule[item] > ops\_rule[op]:

ops.append(op)

ops.append(item)

break

else:

# 如果栈顶的运算符比当前运算符级别高，将栈顶运算符加入到表达式

# 当前运算符与栈中后面的运算符比较

expression.append(op)

# 遇到左括号入栈

elif item == '(':

ops.append(item)

# 遇到右括号，将栈中运算符加入到表达式直到遇到左括号

elif item == ')':

while len(ops) > 0:

op = ops.pop()

if op == '(':

break

else:

expression.append(op)

# 遇到运算数，添加到表达式

else:

expression.append(item)

# 最后将栈中全部运算符加到后缀表达式中

while len(ops) > 0:

expression.append(ops.pop())

return expression

def expression\_to\_value(self, expression, type='after'):

"""后缀表达式计算"""

if type == 'after':

stack\_value = []

for item in expression:

if item in ['+', '-', '\*', '/', '^', '%', '//']:

n2 = stack\_value.pop()

n1 = stack\_value.pop()

result = self.cal(n1, n2, item)

stack\_value.append(result)

else:

if self.dic[item] == 1: # 为标识符

try:

item = self.word\_dict[item]

except KeyError:

raise IndentiNotFountError(item)

stack\_value.append(float(item))

return stack\_value[0]

elif type == 'front':

# expression.reverse()

stack\_value = []

for item in expression[::-1]:

if item in ['+', '-', '\*', '/', '^', '%', '//']:

n1 = stack\_value.pop()

n2 = stack\_value.pop()

result = self.cal(n1, n2, item)

stack\_value.append(result)

else:

if self.dic[item] == 1: # 为标识符

try:

item = self.word\_dict[item]

except KeyError:

raise IndentiNotFountError(item)

stack\_value.append(float(item))

return stack\_value[0]

def middle\_to\_front(self, tokens):

"""中缀表达式变为前缀表达式"""

expression = [] # s2

ops = [] # s1

# tokens.reverse()

for item in tokens[::-1]:

# 当遇到运算符

if item in ['+', '-', '\*', '/', "^", '%', '//']:

while len(ops) >= 0:

# 如果栈中没有运算符，直接将运算符添加到前缀表达式

if len(ops) == 0:

ops.append(item)

break

# 如果栈中有运算符

op = ops.pop()

# 如果栈顶的运算符比当前运算符级别低，当前运算符入栈

if op == ')' or ops\_rule[item] >= ops\_rule[op]:

ops.append(op)

ops.append(item)

break

else:

# 如果栈顶的运算符比当前运算符级别高，将栈顶运算符加入到表达式

# 当前运算符与栈中后面的运算符比较

expression.append(op)

# 遇到左括号入栈

elif item == '(':

while len(ops) > 0:

op = ops.pop()

if op == ')':

break

else:

expression.append(op)

# 遇到右括号，将栈中运算符加入到表达式直到遇到左括号

elif item == ')':

ops.append(item)

# 遇到运算数，添加到表达式

else:

expression.append(item)

# 最后将栈中全部运算符加到后缀表达式中

while len(ops) > 0:

expression.append(ops.pop())

return expression[::-1]

# 计算函数

def cal(self, n1, n2, op):

if op == '+':

return n1 + n2

if op == '-':

return n1 - n2

if op == '\*':

return n1 \* n2

if op == '/':

return n1 / n2

if op == '^':

return n1 \*\* n2

if op == '%':

return n1 % n2

if op == '//':

return n1 // n2

## 5.4解决的技术难点

包括对表达式基本文法的归纳以及拓展，DFA的编程实现，中缀表达式转前缀、后缀并计算的算法。

# 6 测试和试运行

主要采用黑盒测试的方法，从软件的鲁棒性以及输入字符串的全面性进行测试，测试用例如下表。

表6-1测试用例

|  |  |  |
| --- | --- | --- |
| 测试说明 | 测试步骤 | 结果 |
| 界面逻辑测试 | 选中一行进行词法分析 | 成功 |
| 选中多行进行词法分析 | 输出：只能选择一行进行词法分析 |
| 选中一行编译并执行 | 成功 |
| 选中多行编译并执行 | 成功 |
| 基本运算符测试 | a=2+3\*(7-4)+8/4-(-4)^2+18//5-6%2 | 0.0 |
| 词法分析测试 | 1+2+@2 | 词法分析出错，位置5，检测到非法字符：@ |
| 1+2+.2 | 词法分析小数点错误 |
| 1+2+2a | 词法分析出错，位置6，数字串中不能包含字母 a |
| a=1+b= | 词法分析错误，‘=‘过多 |
| 2a=1+b | 词法分析出错，位置1，标识符中必须由数字和字母组成，且必须以字母开头 |
| 语法分析测试 | 1++2 | 语法分析出错，错误位置3,因子只能由(E)或者标识符或者数字组成，出错字符：+，字符类型：3 |
| 1+(2)) | 语法分析出错，位置6,括号不匹配 |
| 1+((2) | 语法分析出错，错误位置7,词法分析错误，该处应有“)” |
| 计算测试 | a=b+1 | 计算出错，标识符b没有在单词表中 |
| a=2/0 | 除数不能为零 |

试运行：

经测试，程序运行正常，基本功能使用正常。

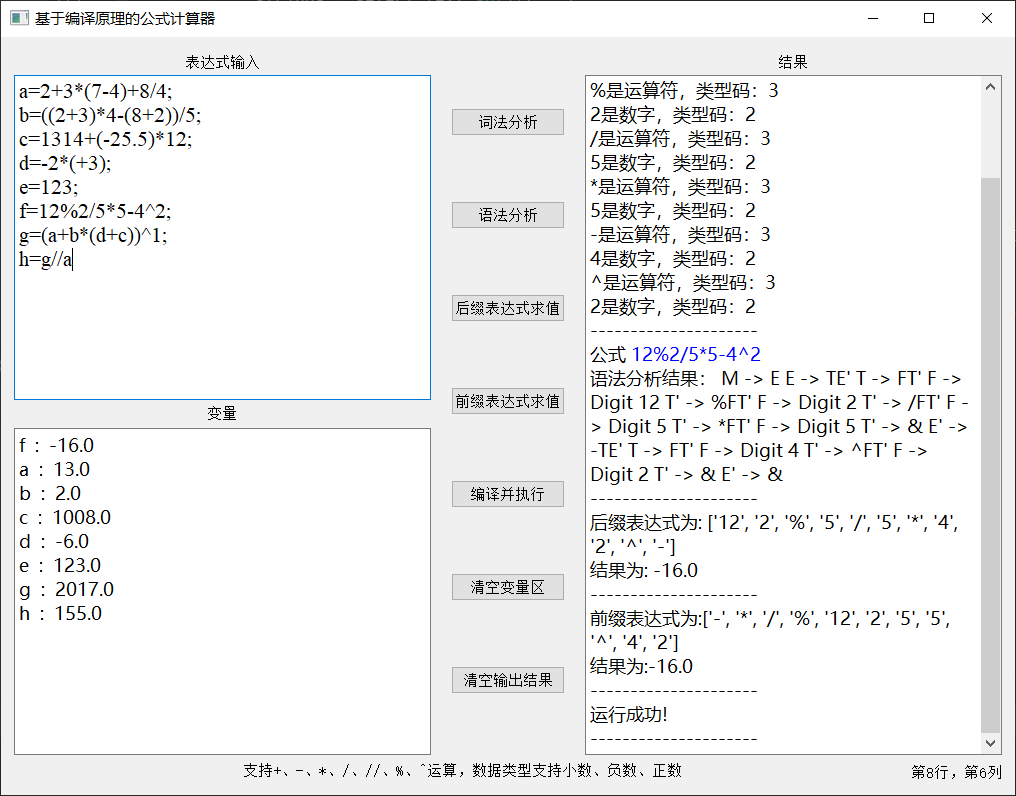


图6-1运行界面

# 7 总结

本次课程设计在实现了在基本四则运算的基础上，添加了取模，取整以及乘幂运算符，并将词法分析、语法分析以及计算过程封装为类，实现了简单的界面以及一些基本的功能。通过此次课程设计，深入理解了词法分析、语法分析对编译器的作用，以及词法分析、语法分析的设计过程，包括DFA、EBNF范式的设计，同时学习了python的编码规范。

但由于时间的限制，本次课程设计拓展的功能不够全面，像类似于单目运算符、正弦余弦函数以及求对数等运算没有实现，这也是本次课程设计的不足之处。

# 8 参考文献

[1]算法之前缀、中缀、后缀表达式转换研究，https://www.sanlt.com/post/software/algorithm/qianzhui\_houzhui\_zhongzhui/

[2]王生原、董渊、张素琴等，《编译原理（第三版）》，北京，清华大学出版社，44-52

编译原理课程设计成绩单

|  |  |
| --- | --- |
| 姓名 |  |
| 班级 |  |
| 学号 |  |
| 分析设计  （满分30分） |  |
| 程序开发  （满分40分） |  |
| 汇报答辩  （满分30分） |  |
| 总成绩 |  |

评阅教师签字：

验收日期： 年 月 日