**武汉大学计算机学院**

**本科生课程设计报告**

**计算器系统总体设计与实现**

专 业 名 称 ：软件工程

课 程 名 称 ：《Windows编程实验》

指 导 教 师 ：胡继承 职称：教授

学 生 学 号 ：2017302580198

学 生 姓 名 ：邹鑫

二○一九年十二月

**郑 重 声 明**

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名： 邹鑫 日期： 2019.12.20

摘□□要

（黑体小2）

XXXX实验的实验目的是XXXX。

实验设计主要遵循XXXX。

实验内容主要包括：

实验结论为XXXX

……

……

……

**关键词：**关键词1；关键词2；关键词3

（黑体小4） （宋体小4）

**目□□录**（黑体小2）

**1**□**实验目的和意义**

1.1□ 实验目的 ………………………………………………………………………1

1.2□ 实验意义………………………………………………1

(各章的名称黑体4号，其余宋体小4)

**……**

**……**

**……**

**2**□**实验设计**

2.1□概述……………………………………………………………………………… 35

2.2□实验原理………………………………………………………………………… 37

2.3□实验方案………………………………………………………………………… 39

**……**

**……**

**……**

**结论** …………………………………………………………………………………… 57

**参考文献 ………………………………………………………………………………** 59

**附录 ……………**………………………………………………………………………… 72

(结论、参考文献、致谢及附录黑体4号)

# 实验要求与背景

## 实验背景

为了进一步了解Windows程序设计在.Net平台上的各种应用，加强对理论

知识的掌握以及实践的能力，本文选择制作一个有比较多的功能的计算器。

## 1.2 实验要求

●实现加、减、乘、除四种基本运算功能

●上述四种运算均要求使用dll实现，其中dll使用c++或c#编程语言来创建

●上述四种运算输入、操作和运算结果均要求直接在界面中进行显示，其中界面可以选用MFC、Winform、WPF技术实现

●上述四种运算的实现均要求输入数据兼容数据类型int、double，输出结果也兼容数据类型int、double

●\*对于输入的非法数据或无效数据要进行相关提示；

●\*对于输入的数据有回退删除功能

## 1.3 实现的功能

为了更好地考察自己的能力，本文除了完成要求中的6个功能之外，还进行了一定的扩展，为计算器添加了更多的功能，具体如下：

●除了加、减、乘、除外，本文还为计算器添加了一些新的运算符，比如：乘方，阶乘等。

●本计算器不是那种很简单的一次只能执行一个运算符的计算器，它可以执行表达式，比如(3+2)×5÷2^3+sin(20-3÷2)。

●为了进一步增强计算器的功能，本文还在初始的传统界面的基础上，实现了扩展界面，扩展界面占空间更大、有更多功能。

●扩展界面中，增加了一些常用得函数界面，比如sin、cos、tan、exp、log、ln等，方便用户进行复杂一点的函数运算。

●扩展界面中，增加了进制转换的功能，支持2进制、8进制、10进制以及16进制的相互转换。

●扩展界面中，增加了函数自定义功能，用户可以自定义函数f，比如f(x) = x^2 + 2x + 1，然后在计算器中输入f(5)就能得到结果5^2 + 2\*5 + 1 = 36

。

●扩展界面中，可以实现函数的嵌套，如sin(f(cos(5)+3)\*6)。

# 2 项目架构简要分析

## 2.1 项目结构图

此项目的总体结构图如图2.1所示。

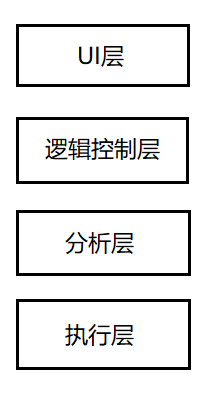


图2.1 项目总体结构图

其中，UI层负责显示UI界面，捕捉用户与系统的交互动作，并将之传递给逻辑控制层进行处理。

逻辑控制层负责整个系统的逻辑控制，它从UI层接收交互时间，并进行相应的处理。在需要进行算术表达式求解时，它调用分析层和执行层的功能得出结果。

分析层主要负责对传入的算术表达式的语法规范性进行分析，如果表达式的语法不规范，比如：1+2+-3，则会弹出警告框告知用户输入不规范。如果输入符合语法，则生成相应的语法执行树，该执行树中包含了运算符的优先级和结合性信息，用于执行层进行执行。图2.2展示了表达式3^4 × sin(5+7)的语法执行树。

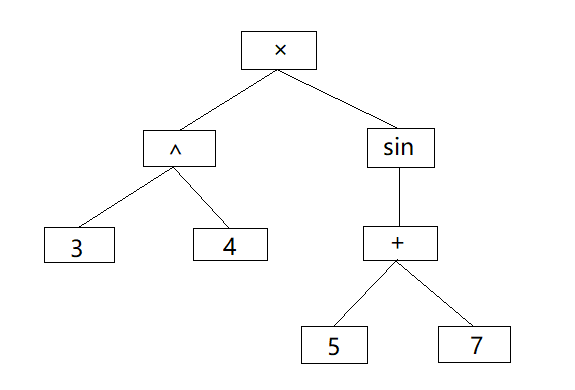


图2.2 表达式3^4 × sin(5+7)的语法执行树

执行层则是接收分析层传入的语法执行树，从树的根节点开始，递归地执行，获得每一个节点的值，最终可以获得根节点的值，也就是表达式的值。

## 2.2 项目进程视图

本项目的进程视图如图2.3所示。

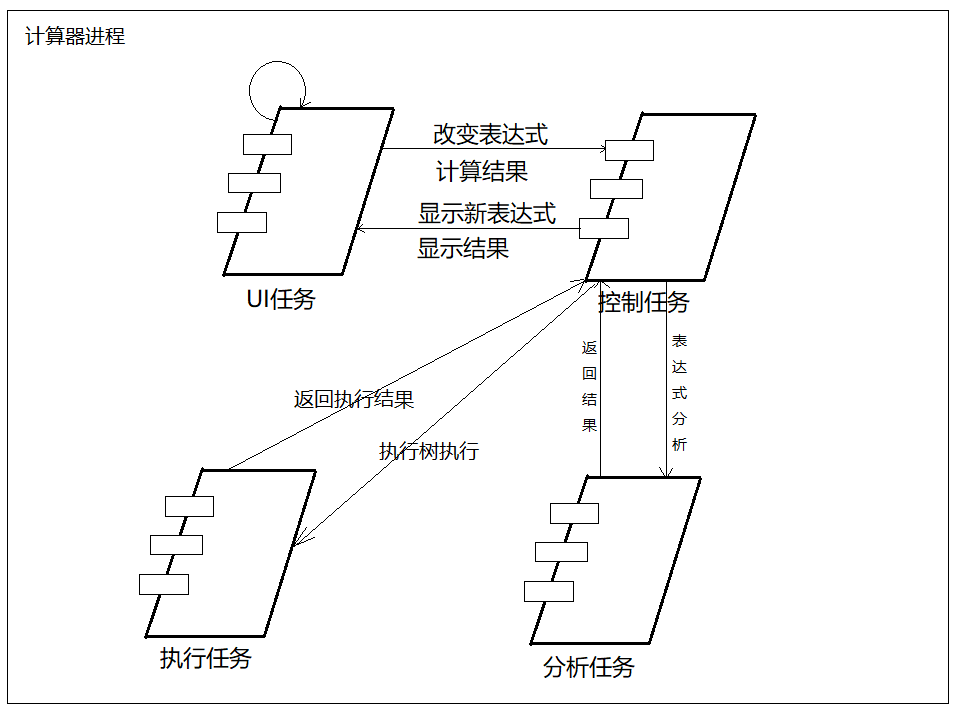


图2.3 项目进程视图

从图中可以看到本项目是单进程的，在计算器进程中，UI任务显示UI界面，当用户想要改变表达式，则将信息传递给控制任务，控制任务得到相应的信息之后，更改表达式字符串，并显示在UI界面上。当UI界面传递要计算结果的信息之后，控制任务则调用分析任务，传入表达式的字符串，分析任务分析获得计算执行树或者得到语法错误的结论，返回给控制任务，如果语法错误，控制任务直接将信息传递给UI任务，显示出来提示用户。如果语法正确，控制任务则根据正确的语法执行树调用执行任务，获得执行结果，并将其发送给UI任务进行显示。

# 3 UI界面

本文为了将功能更好地展现，尽可能不占用过多的桌面空间，将项目界面分

为了3个部分——基础功能部分、扩展功能部分以及进制转换部分。

基础功能部分主要实现基本计算功能——加减乘除、乘方运算，基础功能界面占用空间比较小，适合进行简单的运算，如图3.1所示。



图3.1 基础界面图

扩展功能界面实现了更多的功能，比如常见函数（sin、cos、tan等），阶乘，以及自定义函数功能。在遇到基础功能界面解决不了或者解决起来非常麻烦的问题时，就可以考虑使用扩展功能界面。扩展功能界面图3.2所示。

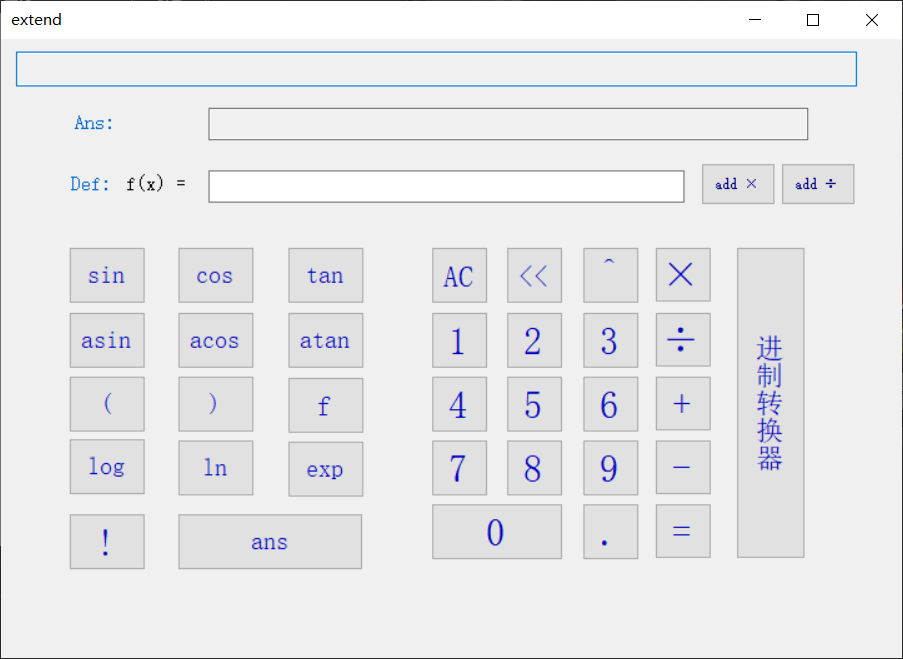


图3.2 扩展功能界面图

进制转换界面主要用于处理进制转换的问题，可以在2进制、8进制、10进制以及16进制之间互相转换，如图3.3所示。

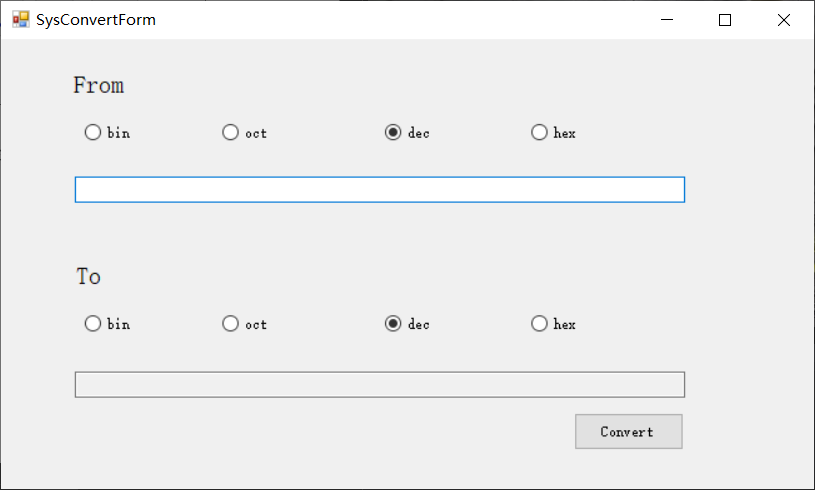


图3.3 进制转换界面图

其中，bin表示二进制、oct表示8进制、dec表示10进制、hex表示16进制。

# 4 功能实现

为了实现上述功能，下面我将一一介绍编码过程中的一些比较重要的过程。

## 4.1 加、减、乘、除基本运算的动态链接库创建

根据要求，为了考察我们是否能创建动态链接库并进行调用，我们必须为基本的加、减、乘、除运算创建相应的动态链接库，以供执行时执行器Excutor调用。

首先，创建一个新项目，选择“类库”项目，如图3.1.1所示。

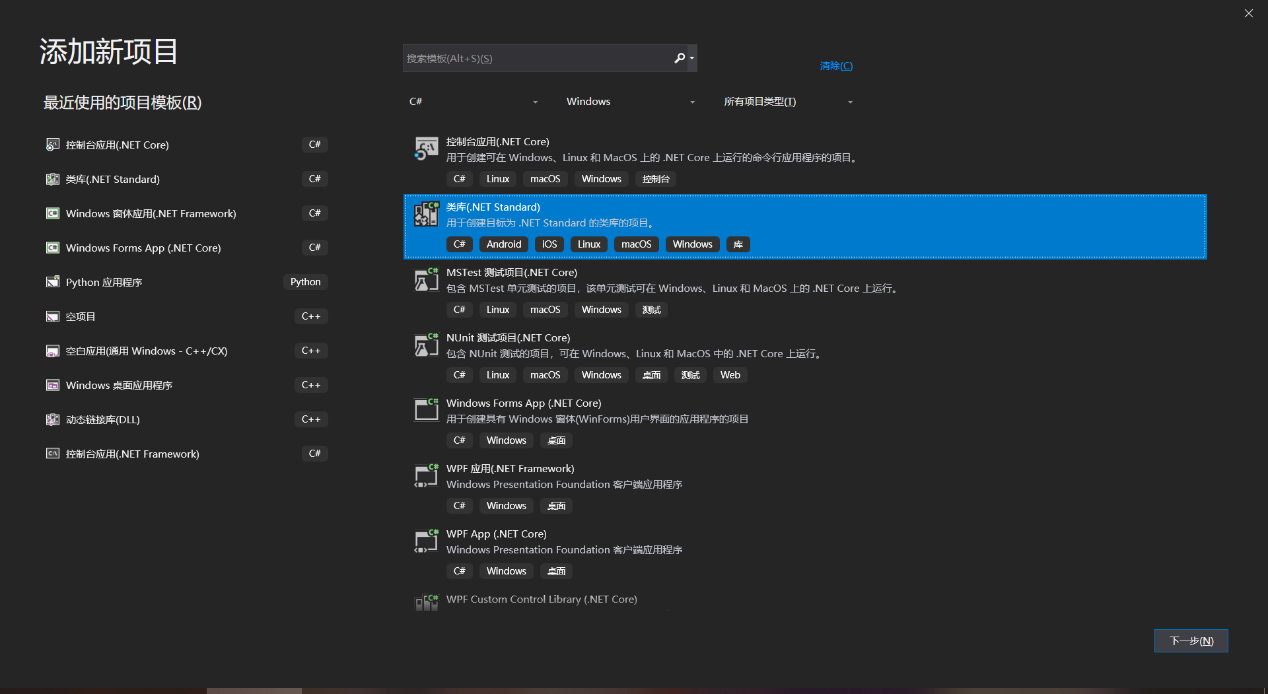


图3.1.1 新建类库项目

然后，为项目命名，选择存储路径，我为项目命名为“OPS”。

然后，新建一个静态类，我将之命名为Operations，并且在其中建了许多静态函数用于实现基本运算功能，该类的代码如下：

public static class Operations

{

public static double add(double a,double b)

{

return a + b;

}

public static double minus(double a, double b)

{

return a - b;

}

public static double multiply(double a, double b)

{

return a \* b;

}

public static double divide(double a, double b)

{

return a / b;

}

}

我针对加、减、乘、除各写了一个函数，传入的参数是double类型，返回值也是double类型。我为什么不根据用户输入的数字决定到底是传int还是double，然后为每个运算符写4个重载函数呢？因为我个人觉得这不是非常有必要，绝大多数运算都是带有浮点数的，运算中全是整数的情况很少，既然几乎所有运算都含有浮点数，那又何必到时候再将整数转为浮点数呢？于是，我选择将所有输入的数字都当作浮点数使用。

然后，重新生成该项目，生成完毕后，进入该项目的Debug文件夹下，就会发现有一个名为OPS.dll的文件，这就是我们要使用的动态链接库。

使用时，只需要在项目的引用中添加该动态链接库，然后就可以将其当作正常的静态类、静态函数使用了。

## 4.2 回退的实现

针对回退，本文对不同功能的界面采取了不同的措施。

### 4.2.1 基本功能界面的回退实现

对于基本功能界面，由于用户输入的只有加减乘除运算符、乘方运算符、左右括号以及数字，所以针对此界面，本文决定直接采用去掉输入字符串的最后一个字符的方法来进行回退。过程如图4.2.1所示。

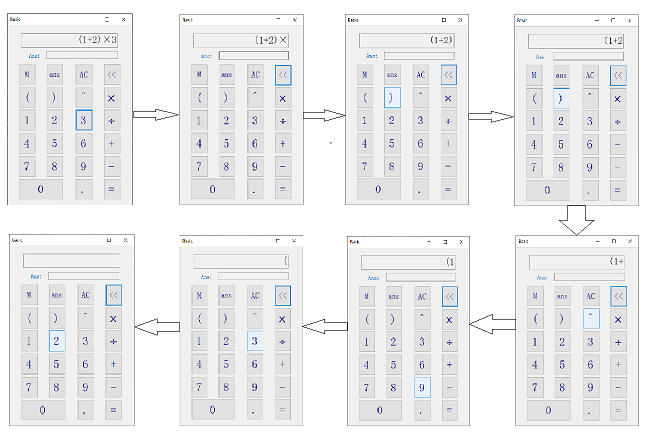


图4.2.1 基本功能界面连续回退图

从图中可以清晰地看到，随着回退按钮不断被点击，输入框每次少一个字符。

### 4.2.2 扩展功能界面的回退功能的实现

针对扩展功能界面，由于扩展界面实现了许多函数功能，故不能再像基础功能界面那样一个字符一个字符地删除，而应该以每次的输入为单位进行删除（每次的输入不一定是一个字符，比如sin就是3个字符）。为此，我建了一个栈，用于存放字符串，每次添加内容，则将新内容入栈，每次需要回退，则将字符串出栈，并且从输入框的尾部删掉该字符串。

过程如图4.2.2所示。

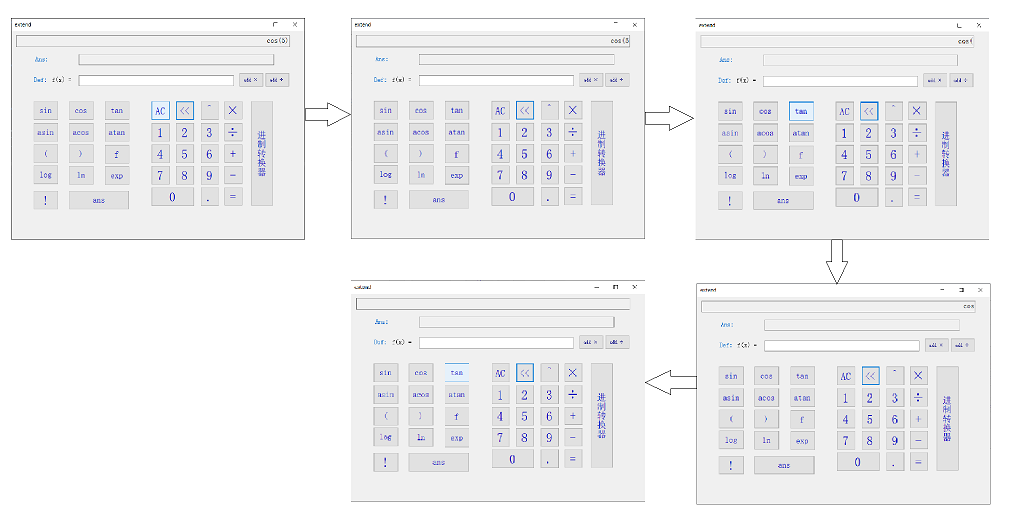


图 4.2.2 扩展功能界面连续回退图

从图中可以看到，前几步是和基本功能界面相同，都是回退了一个字符，这是因为输入的时候就只输入了那个字符。但在最后一步，整个“cos”都消失了，因为cos被看作整体，一次回退就会将其整体删除。

## 4.3 单词元素获取

因为从文本输入框获得的是字符串类型的数据，所以需要对其进行一定的处理。处理文本有很多方式，比如正则表达式等，但要不打乱元素之间的位置关系，我个人认为最方便快捷的方式是借鉴设计模式中的解释器模式。解释器模式中对一个要处理的问题规定了一系列的上下文无关文法，然后根据文法对串进行分析，不满足文法的串则是不和规范的。

### 4.3.1 各个元素的文法

为了逐个提取出正确的元素，我为各个元素定义的文法产生式如下所示（准确来说这并不算文法，只算是文法的产生式中的一部分，因为它没有文法所要求的开始符号）：

<OP> → +

| -

| \*

| /

| ^

| !

<NUMBER> → <PURE\_NUMBER>

| -< PURE\_NUMBER >

<FUNC> → <STRING>

<OTHER> → (

| )

<PURE\_NUMBER> → <INT\_PART>

| <INT\_PART>.<FLOAT\_PART>

<INT\_PART> → 0

| <DIGIT>

| <NONE\_ZERO\_DIGIT> { <DIGIT> }

<FLOAT\_PART> → { <DIGIT> } <DIGIT>

<NONE\_ZERO\_DIGIT> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<DIGIT> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<STRING> → { <CHAR> } <CHAR>

<CHAR> → a|b|c|…|x|y|z|A|B|C|…|X|Y|Z

其中，{}中的元素可以出现0到任意次。

根据上述文法产生式，我实现了相应的算法，将每个元素合理地分开，当字

符串不满足文法产生式时，抛出异常。具体见4.3.2和4.3.3节。

### 4.3.2 Item类的实现

Item类的实现非常简单，主要是用来储存元素的，具体实现如下：

enum Token { NUMBER,OP,FUNC,OTHER};

class Item

{

private Token token;

private String value;

public Item(Token token,String val)

{

this.token = token;

this.value = val;

}

public Token GetToken()

{

return this.token;

}

public void setNeg()

{

if(this.token == Token.NUMBER && this.value.Length > 0 && this.value[0] != '-')

{

this.value = '-' + this.value;

}

}

public String GetValue()

{

return this.value;

}

}

Item类主要记录了2个信息，一个是元素的所属种类token，即OP(运算符)、FUNC(函数)、NUMBER(数字)、OTHER(左右括号)这四类以及一个String类型的值value来存储元素的字符串信息，如加号的token=OP，value = “+”。

### 4.3.3 Lexer类的实现

Lexer类沿用了解释器中对词法分析器的命名，类的主要成员变量是一个从文本框获得的String：souceStr，以及解析获得的一个元素为Item的动态数组list。

主要的函数就是Lex函数，对传入的字符串进行分析。具体代码如图4.3.1。

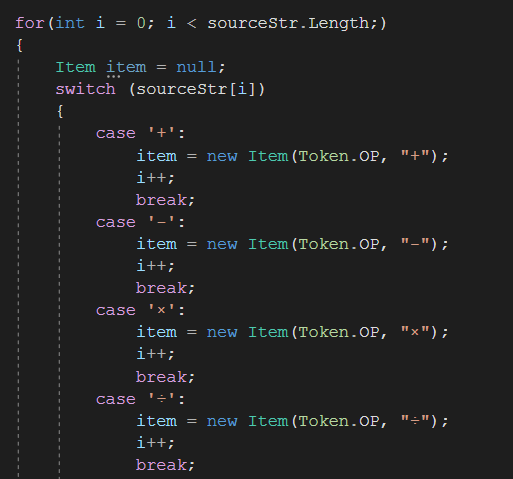


图4.3.1 Lex函数循环部分和简单运算符部分

函数对sourceStr中的字符进行遍历，如果字符是“+”，则创建Item对象，token设置为OP，value设置为“+”，运算符-、×、÷、^、!都是类似的。关于运算符的部分和左右括号的部分都比较简单，下面来看看关于函数部分，如图4.3.2。

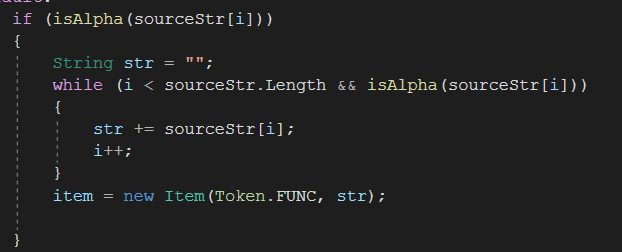


图4.3.2 函数的识别

可以看到，从第一个是字母的字符开始，一直到最后一个是字符的字母，他们组成了函数名（isAlpha()函数是我自己写的一个判断字符是不是a-z或A-Z的函数，比较简单，没必要过多介绍），然后通过函数名创建Item，token为FUNC，value为函数名。接下来看看数字的识别，如图4.3.3所示。

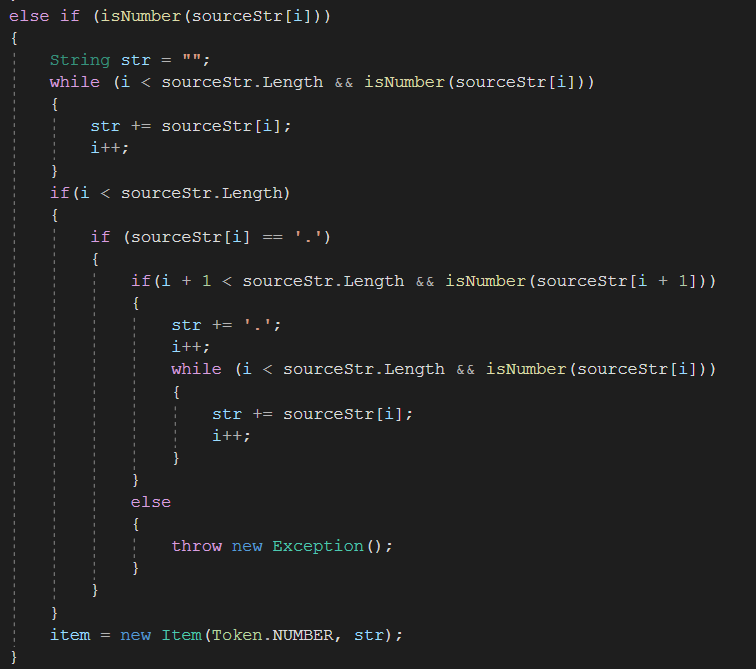


图4.3.3 数字的识别代码

首先，从第一个数字开始识别，直到第一个非数字的字符（isNumber()也是我自己写的判断字符是否在0-9之间的函数），如果这个字符是小数点，那么接着识别后面的一串数字，如果不是小数点，那么输入的就是整数型的数字，就直接将当前字符串当作数字的value即可。

至此，就完成了对每个元素的识别，接下来要介绍的是对元素的分析过程。

## 4.4 生成语法分析树

要解决这个问题，有很多种思路，其中一种思路就是延续解释器模式的做法，对算术表达式再规定一个文法，然后根据编译原理中的分析方法对其进行分析，生成相应的语法分析树。还有一种思路是根据数据结构教程中的经典例子，使用栈对元素串进行分析，完成中缀表达式到前缀表达式或者后缀表达式的转换，从而对其进行分析。

针对第一种方案，由于计算器的算术表达式规则实在过于简单，要为如此简单的功能如此兴师动众去做很多词法分析的准备工作，如：求解非终结符的First集Follow集，求解项目集规范族，生成语法分析表等，实在是不值得，于是我舍弃了第一种方案。

所以本人采用的是第二种方案，但是在原有的算法基础上进行了一些更改，本人的解决方案仍然是利用栈进行分析，但是并没有生成前缀表达式或者后缀表达式，而是直接生成分析树。

### 4.4.1 算法说明

为了得到分析树，首先要解决的是运算符优先级和结合性的问题，为了解决这个问题，我还另外设置了一个栈levelStack，每当将一个运算符入栈，则同时将其优先级也入栈，其中+和-的优先级为0，×和÷的优先级为1，^的优先级为2。而!是单目运算符，是优先级最高的，所以暂时不用写入。

然后我设置了两个类，一个是Operator，一个是Node，Operator是一个运算符，而Node则是分析树的一个节点。Node种含有四个属性：Node类型的left，Node类型的right，Operator类型的op以及double类型的value。表示此node的值等于left op right；如果left为空，则值为op(right)，此时op记录的是函数名字；如果right的值为空，此时op一定为阶乘符号，所以node.value = left op。

分析函数为analyse，调用函数analysePart，分析时的总体思路是：

●当遇到NUMBER，则生成一个node，其value则为该NUMBER表示的值，left，right均为null，op为空运算符。

●当遇到FUNC，则由于FUNC后面必定紧跟左括号，则程序的整体思路是可以通过括号对隔离，遇到括号则先分析括号内的，将括号内的作为子树插入当前树中作为子节点，所以在遇到FUNC时递归地调用analysePart函数先对后面括号中的内容进行分析，得到子树node1，然后创建一个新的Node：newNode，其left为null，op为函数操作符，right为node1，将newNode入栈。

●当遇到左右括号，则进行一些控制。第一次遇到左括号时，由于该括号本身便是此子树的括号，不做处理。第二次遇到左括号时，则调用analysePart函数对子树先进行分析，得到一个node对象。

至于优先级的问题：

当遇到一个符号时，先比较它和levelStack里的优先级的值，如果当前符号的优先级大于levelStack（levelStack中的优先级是递增的，往后读就能知道），则表示它本就应当比栈中的其他元素先运算，不确定会不会有比它优先级更高的运算在后文出现，所以先将其入栈，再将其优先级入levelStack栈。如果它的优先级等于栈顶运算符优先级，则表示他们是同级运算，由于加减乘除都是左结合的运算，所以此时应当先执行栈中运算；如果它的优先级小于栈顶运算符优先级，此时也应当先执行栈顶运算。在这两种情况(即当前运算符优先级<=栈顶运算符优先级)下，应当依次取出栈顶元素：Node类型的right，Operator类型的op，Node类型的left（若这三个元素类型不如上所说，那么则不满足语法规则，应当抛异常），创建新的Node：newNode，newNode.left = left，newNode.op = op，newNode.right = right，然后将newNode入栈，再循环判断当前运算符的优先级和栈顶运算符的优先级，直到当前运算符优先级大于栈顶运算符优先级或者栈中无运算符，再将当前运算符入栈，然后将当前运算符的优先级入levelStack栈。

当遇到阶乘运算符!时，由于此运算符优先级最高，所以可以直接生成子树——取出栈顶的Node对象node（若对象不是node类型而是Operator类型，则表示语法不和规范，应当抛出异常），然后创建新的Node对象newNode，其left为node，op为阶乘运算符，right为null，然后将newNode入栈。

### 4.4.2 遇到的问题和解决思路

在实施算法的过程中，我发现我忽略了一个问题：

负号“-”有两种意思，一种是减、一种是取负。对于一般的数字前的负号，在词法分析的过程种就已经将之识别出来了，但是有一种情况的负号被我忽略了，那就是括号前的负号，也代表取负。于是，我使用了我的第一种解决方案——当出现这种情况时，直接递归调用analysePart分析括号内的内容，然后再在前面加一个0，即0-a=-a。

但在后续的测试中，我又发现了另一个问题——当出现-sin(1)的时候，上述的方法不再适用，而程序提示“输入不和规范”，但众所周知，这样的输入其实是符合规范的。为了解决该问题，我又采取了我的第二种对负号的解决方案——如果负号后是括号或者是函数，那么就在负号前先加入一个value为-1的Node，然后加入“×”号，这样就把-a变成了-1×a，并且适用于负号后是括号和函数的两种情况。

于是问题就得到了比较好的解决。

### 4.4.3 关键代码

此功能实现的关键代码其实就是函数analysePart，analyse函数十分简单，如图4.3.4所示。

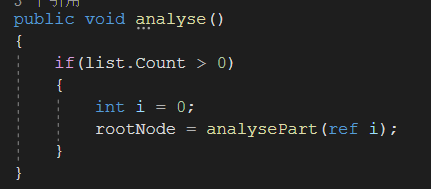


图4.3.4 analyse函数

该函数提供一个能在函数内部改变的循环变量i，i的初值为0，调用analysePart函数。

主要的分析功能在analysePart函数中，由于代码较长，下面分开介绍各个部分的代码。

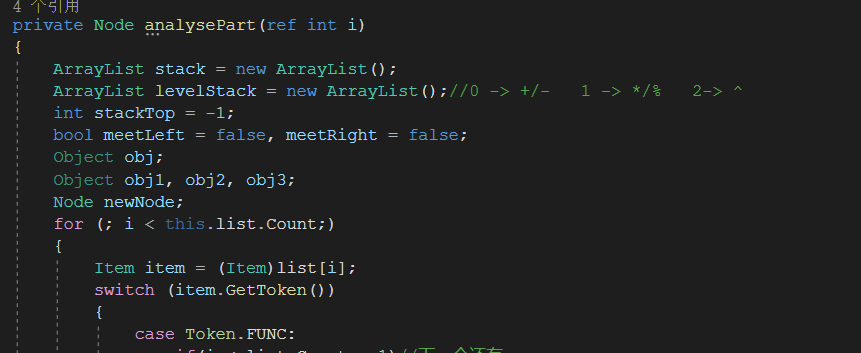


图4.3.5 主控部分

此部分主要初始化需要用到的栈，定义一些后文会循环利用的变量，然后对i进行循环，当i不超过元素列表总数时一直执行，判断第i个元素的token，根据不同的token值进行不同的处理。

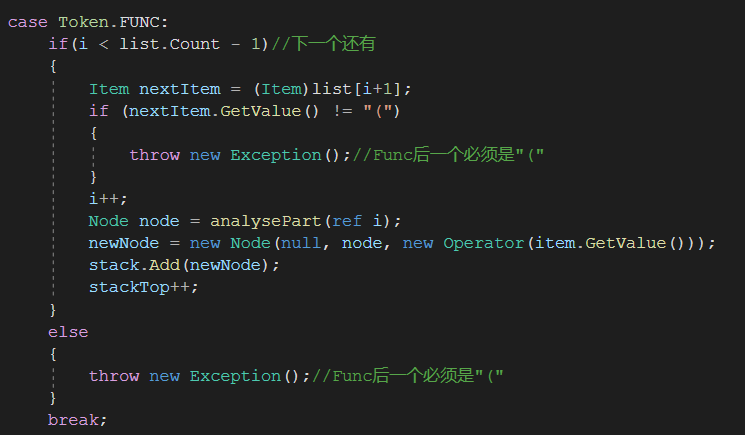


图4.3.6 函数的处理

当token为函数时，对函数后括号的内容先进行分析，然后将分析得到的node和函数一起生成新的Node对象并将之入栈。

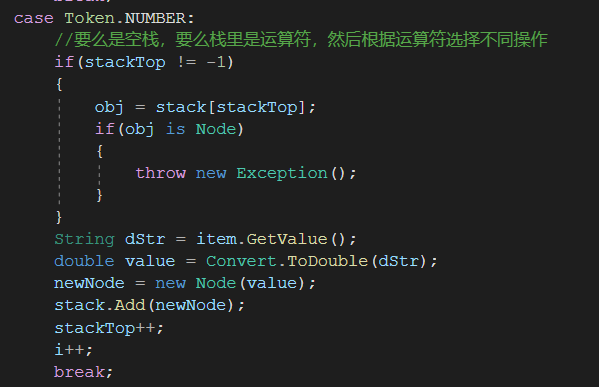


图4.3.7 数字的处理

当token为数字时，直接利用数字创建node，将node入栈即可，当然，栈顶不能是node对象，否则违反规则，抛出异常。

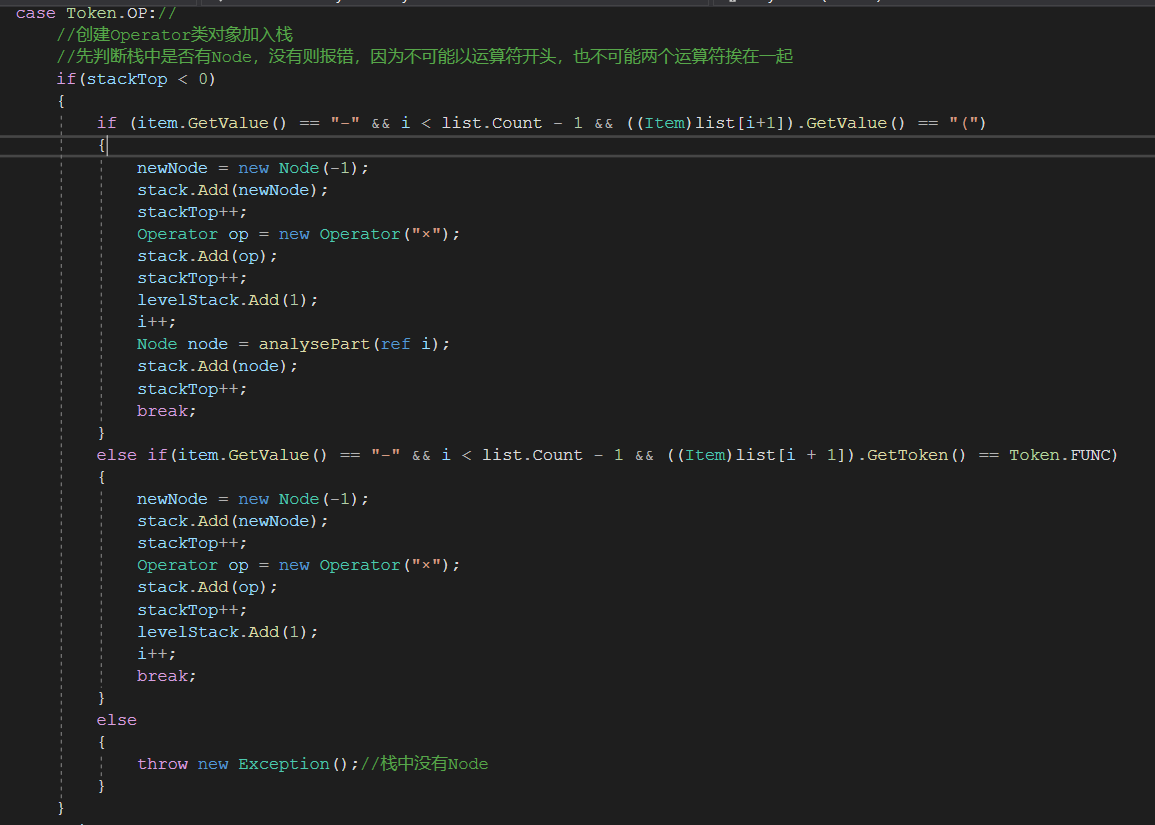


图4.3.8 运算符的处理（一）

当是负号时，则进行判断，如果满足4.4.2中所描述的情况，则进行相应的处理。

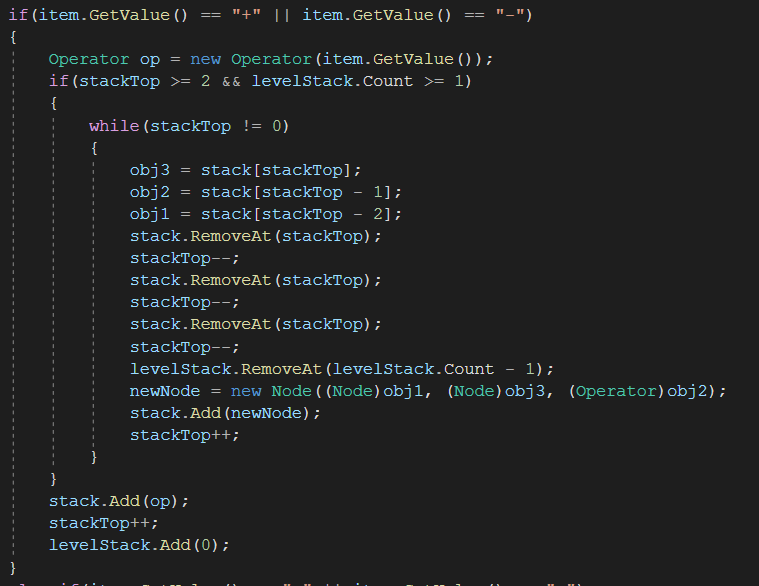


图4.3.9 运算符的处理（二）

当遇到加减符号时，如4.4.1中所说，由于加减法优先级最低，所以一直取出栈顶运算符进行运算，直到栈中无运算符，再将此加减符号及其优先级入栈。

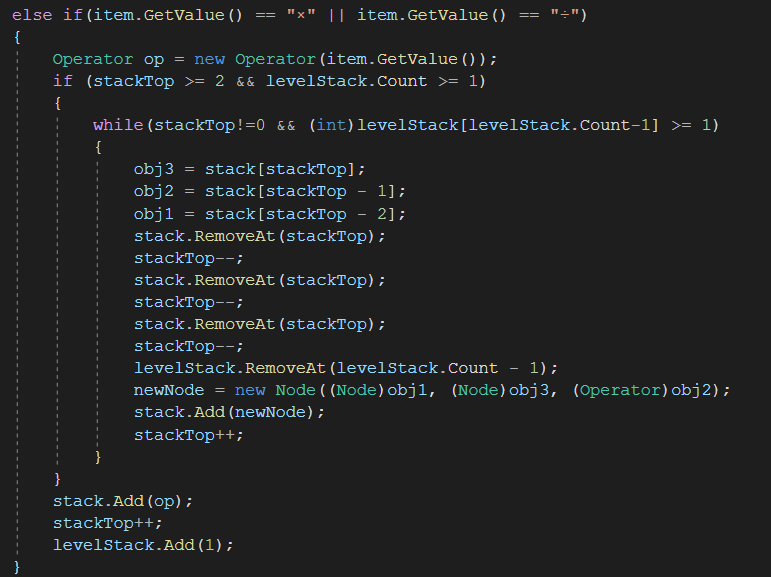


图4.9.10 运算符的处理（三）

当遇到乘除法时，对栈顶优先级大于等于1的运算符进行运算，直到优先级小于1为止，然后将此运算符及其优先级入栈。遇到“^”运算符时也类似，此处不再介绍。

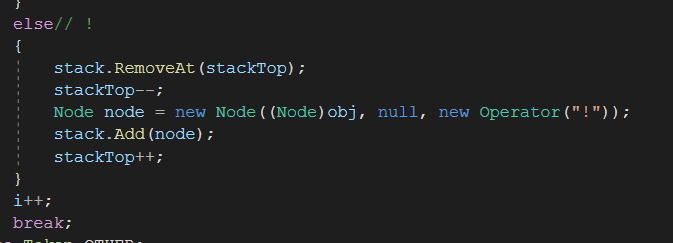


图4.9.11 运算符的处理（四）

当遇到阶乘运算符时，直接将栈顶元素出栈，生成阶乘运算子树，再将子树node入栈。

此算法的主要部分就是以上内容。

## 4.5 自定义函数功能的实现

本计算器最大的特色之一便是自定义函数功能的实现。在运算的时候，为了

# 5 反思与改进

F实现时的思路转变（正则处理->执行时处理）