关于一个C语言计算器的逻辑分析与高级功能实现

本题中已知的参考源代码只能实现加减乘除的基本运算,然而想要快速实现增加高级算术运算函数的添加在大致观察函数形式后是可以做到的，这会在后面提到。但是透彻的理解该程序的逻辑还是最根本的方法。

首先需要大致了解主函数结构，不复杂:先初始化存放表达式的缓冲字符数组，屏幕打印一些欢迎信息后使用for制作了一个无尽循环（功能是使计算器可在进行一次计算后进行下一次运算，直至用户主动退出为止）。在一次运算循环过程中，将用户输入的字符串一次写入到缓冲字符数组内直至换行符为止，并将数组中最后一个字符的后一位写为结束符；此时调用eatspaces函数对字符串进行格式化，去除可能输入的空格，根据格式化后的数组是否无有效内容来选择是结束程序还是调用expr函数来计算并输出结果。

此外，被调用的函数还有double term(char \* str, int \* pindex)用来计算多项式中一项的值，double number(char \* str, int \* pindex)用来识别多项式某一项中单独的一个数，以及char \* extract(char \* str, int \* index)遇到括号则优先处理括号内的表达式。首先介绍如何快速分析解决该问题。

**高级运算功能的快速实现**

注意到总体上的调用关系是expr→term→number→extract，其对应的优先级也从低到高：加减→乘除→算术因子→→（括号内的）算术因子，。这里extract（对应括号内的）算术因子，与前者之间并不是完全的从属关系，是因为extract函数涉及对主计算函数expr的调用，是递归结构。所以从快速做题的角度来看要加入高级函数结构中应该加一环：加减→乘除→算术函数→算术因子→→（括号内的）算术因子。所以定位至number函数中判断数字起始点的isdigit函数前加入对新定义的mfunc函数的调用。当然需要一个判断条件用islower或isupper函数判断是否可能是函数名，可能是则转入mfunc函数进行判断及选择。

在mfunc函数中同样需要初始化缓冲字符串数组buffer[MAX]与存储初始工作位置至bufindex，没有使用指针所以只是个局部变量与前面的buffer不冲突；初始化fname以存储函数名，一行存储一个函数名；接着初始化行数i、列数j、匹配字符数k以及满足条件时的行数t；将接下来用islower函数判断取出的字符串与数组的每行进行逐位比较（首先转化所有大写字母为小写），完全匹配则将该函数的当前行数存储至变量t。根据t值在switch语句中选择调用对应的算术函数计算并返回至number函数即可。

值得一提的是，在调用算术函数时，是对函数后的一个算术因子进行计算，所以需要递归调用number函数。同时其形式为mathfunction(number(str,pindex))；是单目运算，此时当前工作位置pindex已经移到mathfunction函数后的第一个算术因子处，想要进行双目运算的话则需要重新取前面的number，比较繁琐，比较简洁的方法是在term函数计算乘除之前同样用if判断是否需要进行双目运算如乘方取余等，双目分别是value和number（str,index），根据对应法则计算即可。这也保证了双目运算函数的优先级仍然是在乘除运算之前的。这样就基本上实现了功能的添加。

**前后计算器的逻辑分析**

虽然前面介绍了如何快速解决问题，但是我们有理由相信，算法的逻辑永远是我们应该关注的核心点。严谨分析其逻辑过程也有助于我们深化对程序乃至C语言的理解。文章开头已经分析过主函数的结构，接下来按函数来分别分析每个函数的原理。（附上的思维导图是按照执行的顺序来制作的，建议用来辅助理解函数间关系。）

**一、eatspace函数**

// Function to eliminate blanks from a string

void eatspaces(char \* str)

{

int i=0; // 'Copy to' index to string

int j=0; // 'Copy from' index to string

while((\*(str+i) = \*(str+j++)) != '\0') // Loop while character copied is not \0

if(\*(str+i) != ' ') // Increment i as long as

i++; // character is not a blank

return;

}

在主函数调用eatspace函数时传入的参数是含有空格的字符串，这里首先初始化会用到的移动字符所需的用于确定地址的增量i、j；下一步通过while和if语句实现通过指针对buffer的操作（while判断条件已经包含了自增与赋值操作）。如果遇到字符则不改变，遇到空格则用后面的字符替换空格。这就实现了“吃空格的功能”，即对输入字符串的格式化。如果用户输入的只有回车或者空格，那么操作后的buffer[0]为结束符，返回主函数时下一行的if(!buffer[0])成立，主函数直接返回0，程序结束。

**二、expr函数**

// Function to evaluate an arithmetic expression

double expr(char \* str)

{

double value = 0; // Store result here

int index = 0; // Keeps track of current character position

value = term(str, &index); // Get first term

for(;;) // Infinite loop, all exits inside

{

switch(\*(str+index++)) // Choose action based on current character

{

case '\0': // We're at the end of the string

return value; // so return what we have got

case '+': // + found so add in the

value += term(str, &index); // next term

break;

case '-': // - found so subtract

value -= term(str, &index); // the next term

break;

default: // If we reach here the string

printf("Arrrgh!\*#!! There's an error.\n");

exit(1);

}

}

}

本函数是用来负责处理表达式的计算主函数，在这里初始化的value和index将被后面的所有函数通过指针来进行处理，这也实现了跨函数的操作。首先调用term函数得到第一项的值，在使用一个无尽循环进行项与项之间的叠加。在该循环中，如果到了字符串尾部则返回已计算出的值，如果是加（或减）则与后一项的值相加（或相减）成为新的Value，是其它的以外情况则报错。

**三、term函数**

// Function to get the value of a term

double term(char \* str, int \* pindex)

{

double value = 0; // Somewhere to accumulate the result

value = number(str, pindex); // Get the first number in the term

if(islower(\*(str+(\*pindex)))||isupper((\*(str+(\*pindex)))))

value\*=mfunc(str,pindex);

// Loop as long as we have a good operator

while((\*(str+(\*pindex))=='\*')||(\*(str+(\*pindex))=='/')||(\*(str+(\*pindex))=='^')||(\*(str+(\*pindex))=='%'))

{

if(\*(str+(\*pindex))=='!')

{

++(\*pindex);

int t=floor(value),i=1;

for(;i<t;i++)

{

value\*=i;

}

}

if(\*(str+(\*pindex))=='^')

{

++(\*pindex);

value=pow(value,number(str, pindex));

}

if(\*(str+(\*pindex))=='%')

{

++(\*pindex);

value=(int)value%(int) number(str, pindex);

}

if(\*(str+(\*pindex))=='\*') // If it's multiply,

{

++(\*pindex);

value \*= number(str, pindex); // multiply by next number

}

if(\*(str+(\*pindex))=='/') // If it's divide,

{

++(\*pindex);

value /= number(str, pindex); // divide by next number

}

}

return value; // We've finished, so return what we've got

}

Term函数：用于计算多项表达式中的单项。首先通过调用number函数获取这一项中第一个数的值，然后根据数字后的运算符进行双目函数运算及乘除。最后返回value的值。请注意，我在原版的基础上加了乘方、取余函数，以及：

if(islower(\*(str+(\*pindex)))||isupper((\*(str+(\*pindex)))))

value\*=mfunc(str,pindex);

这是为了保证输入如3sin(X)这样的表述形式等价于3\*sin(X)。

这就是term函数的思路，不复杂。在知道了各函数间的逻辑关系的前提下，当你抛开复杂的符号去分析函数的时候，你会发现expr→term→number→mfunc→extract的**结构原理都是统一的**，此时按照自己的想法添加功能甚至是改造原有功能就变得很简单。

**四，number函数**

// Function to recognize a number in a string

double number(char \* str, int \* pindex)

{

double value = 0.0; // Store the resulting value

char \* psubstr; // Pointer for substring

if(\*(str + (\*pindex)) == '(') // Start of parentheses

{

++(\*pindex);

psubstr = extract(str, pindex); // Extract substring in brackets

value = expr(psubstr); // Get the value of the substring

return value; // Return substring value

}

if(islower(\*(str+(\*pindex)))||isupper((\*(str+(\*pindex)))))

value+=mfunc(str,pindex);

while(isdigit(\*(str+(\*pindex)))) // Loop accumulating leading digits

value=10\*value + (\*(str+(\*pindex)++) - 48);

// Not a digit when we get to here

if(\*(str+(\*pindex))!='.') // so check for decimal point

return value; // and if not, return value

double factor = 1.0; // Factor for decimal places

while(isdigit(\*(str+(++(\*pindex))))) // Loop as long as we have digits

{

factor \*= 0.1; // Decrease factor by factor of 10

value=value + (\*(str+(\*pindex))-48)\*factor; // Add decimal place

}

return value; // On loop exit we are done

}

同上，在取数字的时候判断是否有括号，有的话调用extract函数取出括号内的内容。值得注意的是，此处递归调用了主运算函数expr（）计算括号内子表达式的值。若非括号而的确是数字，则根据位数及是否有小数点将这个以字符串形式存储的数转化为双精度浮点型值，并返回。此时我们得到了真正可以用于计算的数，而不是用户输入的字符。

在判断时，于括号的检查之后、小数点检查之前我加入高级算术运算函数的函数名检查，如果匹配则调用增加的mfunc函数计算函数值。

当然，理解结构之后，**细节上算法的处理**也是得仔细琢磨的。接下来以extract函数和mfunc函数为例来阐述。

**五、extract函数**

// Function to extract a substring between parentheses

// (requires string.h)

char \* extract(char \* str, int \* pindex)

{

char buffer[MAX]; // Temporary space for substring

char \* pstr = NULL; // Pointer to new string for return

int numL = 0; // Count of left parentheses found

int bufindex = \*pindex; // Save starting value for index

do

{

buffer[(\*pindex) - bufindex] = \*(str + (\*pindex));

switch(buffer[(\*pindex) - bufindex])

{

case ')':

if(numL == 0)

{

buffer[(\*pindex) - bufindex] = '\0'; // Replace ')' with '\0'

++(\*pindex);

pstr = (char \*) malloc((\*pindex) - bufindex + 1);

if (!pstr)

{

printf("Memory allocation failed, program terminated.") ;

exit(1);

}

strcpy(pstr, buffer); // Copy substring to new memory

return pstr; // Return substring in new memory

}

else

numL--; // Reduce count of '(' to be matched

break;

case '(':

numL++; // Increase count of '(' to be // matched

break;

}

} while(\*(str + (\*pindex)++) != '\0'); // Loop - don't overrun end of string

printf("Ran off the end of the expression, must be bad input.\n");

exit(1);

}

1. 初始化函数内缓冲字符串存放数组。
2. 初始化一个指针，指向用于返回的子表达式的首字符地址，置空。
3. 初始化整形变量numl存储发现的左括号数。
4. 初始化bufindex变量记录被调用时传入的表达式在内存中的地址（首地址）。
5. 执行循环：

(1). 将传入的字符串在当前检查到的位置处存放的内容存储在缓冲区的第个区域，其中是当前检查到的地址与首地址的差；

(2). 根据缓冲区当前最后一个字符进行选择：

(2.1)如果是右括号：

(2.1.1)如果此时记录到的左括号只有一个，那么：

{

将最后的右括号替换成结束符；

将检查地址向后移一位；

动态分配与最小表达式字符数相同的字节的空间并将空间的首地址返回给字符型的pstr指针；

如果pstr中由于某些原因为存入有效地址，那么：

报错，直接退出整个程序；

将buffer中存储的字符串复制到为其分配的pstr指向的内存空间中；（此时buffer中已经存储了全部的子表达式）

返回pstr；（使得number函数中psubstring也指向该地址）

}

(2.1.2)如果此时记录到的左括号不止一个，那么：

减少一个需要匹配的左括号；

结束是右括号的case；

(2.2)如果是左括号，那么：

增加待解决的左括号数；

结束时左括号的case；

整个do循环的条件是while(\*(str + (\*pindex)++) != '\0')，条件中已经实现了每检查一个字符，则移动至下一位置进行检查。直到结束符为止。

而如果到结束符为止extract函数仍旧没能匹配所有括号并返回最小括号内的子表达式的话，则报错，终止程序；

以上是extract函数找出最小子表达式的过程。值得强调的是，在所有的函数中，都存在着逐个识别字符串中的字符并将有意义的几个字符转化为可计算的量的过程。简单来说，**逐个检查+字符串转化**的过程是贯穿整个程序的，这就是前面提到的各函数间的统一性。所以，“逐个”就表现在所有函数中的++（\*pindex），而字符串的转化过程才是这些函数的真正功能区别所在。

也就是说，接下来我们自己设计的mfunc函数也应该遵循这样的原则。于是：

**六、mfunc函数**

double mfunc(char \* str, int \* pindex)

{

char buffer[MAX];

char fname[16][5] = {{'s','i','n'},{'c','o','s'},{'t','a','n'},{'a','s','i','n'},{'a','c','o','s'},{'a','t','a','n'},{'s','q','r','t'},{'a','b','s'},{'l','o','g'},{'l','n'},{'l','g'},{'s','i','n','h'},{'c','o','s','h'},{'t','a','n','h'},{'c','e','i','l'},{'f','l','o','o','r'}};

int bufindex = \*pindex;

int i=0,j=0,k=0,t;

while(islower(\*(str+(\*pindex)))||isupper(\*(str+(\*pindex))))

{

if(isupper(\*(str+(\*pindex))))

(\*(str+(\*pindex)))+=32;

buffer[(\*pindex) - bufindex] = \*(str + (\*pindex));

++(\*pindex);

}

buffer[(\*pindex) - bufindex]='\0';

for (i = 0; i <= 15; i++)

{

for(j=0;j<=2;j++)

{

if (buffer[j] == fname[i][j])

{

k++;

if(k==3)

{

t=i;

break;

}

}

}

k=0;

}

switch(t)

{

case 0 :return sin(number(str,pindex));

case 1 :return cos(number(str,pindex));

case 2 :return tan(number(str,pindex));

case 3 :return asin(number(str,pindex));

case 4 :return acos(number(str,pindex));

case 5 :return atan(number(str,pindex));

case 6 :return sqrt(number(str,pindex));

case 7 :return fabs(number(str,pindex));

case 8 :return mlog(str,pindex);

case 9 :return log(number(str,pindex));

case 10 :return log10(number(str,pindex));

case 11 :return sinh(number(str,pindex));

case 12 :return cosh(number(str,pindex));

case 13 :return tanh(number(str,pindex));

case 14 :return ceil(number(str,pindex));

case 15 :return floor(number(str,pindex));

default:

{

printf("Syntax error.\n");

exit(1);

}

}

printf("Syntax error\n");

exit(1);

}

}

在高级算术运算函数中：

同样地，初始化缓冲区；

定义一个多行五列的字符型二位数组fname来存放各函数名；（我试过用一维数组，例如fname[3]={‘sin’,’cos’,’tan’}，但是事实是每个位置由于字符串压栈的顺序存储的都是函数名中最后一个字母，故只能使用二维数组）；

使用局部变量bufindex存储起始检查位置；

定义i、j控制行列，k存储字符匹配数目，t转储匹配成功时的行数（也就是函数序号）；

While循环控制对字母的读取直到数字或括号等其它字符

在循环内，转换大写为小写（格式化，允许用户输入大写字母）；

并将传入的表达式存至缓冲区

当前检查到的字符位置后移，继续循环检查；

退出while循环后，此时已经经过++(\*pindex)处理，(\*pindex)-bufindex对应函数最后一个字母存储地址的后一位，设为结束符；

进入for循环检查函数名的匹配：

此时buffer这个字符型数组中存储的是函数名称，那么如果每换一列都有buffer[j] == fname[i][j]，则说明函数名匹配；

这时候k存储的字符匹配数目加一；

等到k==3即三个字母都匹配时，可以确定输入的函数了，将行数（也就是函数序号）转储至t并结束循环；

当然如果某行函数名只匹配一个字母，例如asin，acos，atan的首字母就相同，为了不互相干扰，在一行循环过后，匹配数目要归零；

现在开始根据t的值调用math.h库中对应的函数：（在此处我一开始试图直接用switch（buffer），但是switch（）结构中条件表达式的类型必须是整数，故能作用在int和比int范围下的数字型数据类型:int,short,char,byte。 long和String类型是不行的。所以才使用了前面使用二维数组检查匹配的方法）

此处根据行数调用对应函数即可；

默认出现其他情况时（如没有函数匹配）报语法错误并退出程序；

另外，前面提到过，在进行算术函数运算时，此时当前工作位置pindex已经移到mathfunction函数后的第一个算术因子处，想要进行双目运算的话则需要重新取前面的number，比较繁琐。但是对于即任意底数的对数函数来说:1.它是一个双目运算；2.math.h头文件中也没有现成的函数可用，该头文件中的log函数其实对应数学运算中的ln函数，即a为自然 e。那么当用户输入log(a,b)的时候只能用换底公式处理。3.对该双目函数的输入格式控制也比较麻烦。4.函数名为字符串，不像乘方、取余那样可以用单个字符代替并可置于term函数中。这样就不得不另辟方法了。

我是这样解决的：

//The log(a,b) function in mathmatics

double mlog(char \* str, int \* pindex)

{

double a,b;

if(\*(str + (\*pindex))=='(')

{

++(\*pindex);

a=number(str,pindex);

if(\*(str + (\*pindex))==',')

{

++(\*pindex);

b=number(str,pindex);

if(\*(str + (\*pindex))==')')

++(\*pindex);

else

{

printf("Syntax error.\n");

exit(1);

}

return log(b)/log(a);

}

else

{

printf("Syntax error.\n");

exit(1);

}

}

else

{

printf("Syntax error.\n");

exit(1);

}

}

在mfunc函数中遇到对于log的选择时调用mlog函数，该函数的特别作用就是进行格式输入控制，当且仅当用户输入格式为log(double a, double b)时再能用换底公式计算出该值。

当然，当前的程序中可使用的函数有连乘、乘方、取余、三角函数、反三角函数、梯度函数、对数函数、向上（下）取整函数、绝对值、开方。不过还不是很全，之后我会继续完善。

到这里，整篇文章关于逻辑的分析就结束了。说到底，想要彻底理解这样一个看似简单的计算器，最重要的还是逻辑、逻辑、逻辑。