汇编大作业说明文档

开发环境

- 操作系统：Windows 10
- IDE：Visual Studio
- 汇编器：MASM

难点与创新点

这里首先陈述难点与创新点，之后的实现原理部分也会按此处的逻辑进行讲述。我们项目的难点/创新点有如下若干项：

1. 图形界面：在界面上管理多个输入框，维护计算输入/计算结果的显示，提供滚动条等；
2. 表达式处理：正确解析用户输入的任意算式，并能处理算符的优先级等关系，为下一步计算打下基础；
3. 计算逻辑：处理各种参数与算符，在得出正确结果的同时保证程序的鲁棒性；
4. 变量处理：支持变量的定义与使用，在算式中考虑变量并正确计算结果，以丰富用户的使用体验。
5. 内置函数计算：支持内置函数的使用，以丰富用户的使用体验。

实现原理

下面分别展开讲述各难点与创新点的原理。鉴于这些点本身也都比较复杂，我们将重点描述实现思路与实现过程中的关键步骤。

图形界面

图形界面的实现基于 Win32 API，原理较为简单，其主要结构如下。
整个页面由可变数量的子窗口组成，该子窗口为自定义窗口，命名为 RowBow(以下称为 Box)，每个 Box 又是由两个子控件组成，一个文本编辑框用于交互，以及一个静态文本框用于显示计算历史。这部分窗口的逻辑都通过调用 win32 API 实现。
程序初始会包含一个 Box 用于第一次输入，之后的新 Box 产生逻辑为：无论用户在哪一个 Box 中进行了计算，都需要保证其下方还有一个 Box(用于放置输出结果)，以及最下方仍有一个空白的 Box(用于空白的新输入)。
因此，若计算发生在最后一个 Box，则在末尾产生两个新的 Box；若计算发生在倒数第二个 Box，则在

其下方产生一个新的 Box。其余情况则无需生成新 Box。
还有一点值得提及的是，生成 Box 时程序会始终维护最后一个 Box 的高度为占满屏幕，这样用户可以直接点击空白区域并进行输入，其余的输入框的高度则均为固定值。
在用户按下 Enter 键进行计算时，图形部分的程序会将当前输入框中的文本传递给负责计算逻辑的函数，而计算完成后计算部分会更新一个指定的字符串，图形部分程序再从这段字符串获取计算结果并更新下方输入框与计算历史文本的显示。
在菜单栏中包含了两个子项，分别会跳转至我们的项目的 github 主页以及一个支持的操作符说明文档，我们认为这还是很有必要的，因为我们的程序支持的操作符种类较多，用户可能需要进行查阅才能较好的使用。
右侧的滚动栏长度也会根据 Box 的数量对应的总高度进行动态调整，从而支持几乎任意数量的 Box 显示。这部分由于需要调用的接口较多，实现的过程略为复杂，我们花费了较多精力，好在最终的 ui 还算令人满意。

表达式处理

表达式处理的核心是将正常的数学算式转化为逆波兰表达式，从而让之后的计算能通过栈操作进行。
在参考了这篇文章，我们得知了逆波兰表达式的转化方法如下：

1. 所有算术步骤加上括号，表明优先级。这一步去除了符号与符号间的优先级关系，计算先后顺序依照括号进行。如：乘除要优先于加减。
2. 将括号内的符号移动到被包裹的最内层的右括号之后。
3. 去掉括号即可。

下面分析一下这三步的实现。

前戏：算符表

在加括号前便会显现的一个问题是，我们要先找到算符，为此需要开设一张表记录运算符的情况。这张表应该包含所有算符，同时表现出算符的优先级。
首先我们对算符进行了分类：按算符需要的操作数可以分为一元与二元运算符，按形式则可以分为函数与算符。形式上算符为特殊字符，函数则为大写字母串。我们认为函数也是一种算符，这样能大大增多我们程序支持的功能，也使得之后的处理更加统一。
算符表被组织为一个二维 BYTE 数组，其实质为一定长度的一维 BYTE 字串，每行 (这里的行即一定长度的字符串) 存储若干算符，这些算符具有相同的优先级；整个字串则包含若干行，越靠前的行优先级越高。
除此之外相应地设置了一个辅助表，用于记录上面描述到的算符的类型。
我们还设置了一个功能性函数用于查找并匹配算符，该函数在之后的处理中被广泛使用。

加括号

通过上面的算符识别函数，我们可以在算式中找到所有的算符，然后根据算符表中的优先级，将算符加上括号。
具体而言，我们逐行对算符表进行处理，每次处理一行，为该行中的算符加上括号。通过从算符所在的位置向左/右查找，我们可以确定括号应该出现的位置。以二元算符的右半括号为例，初始指针在算符右侧：

- 若此时指针指向的值已经是一个左括号：说明左侧是一个被括号包裹的区域，此时可以采用遍历并统计左/右括号数量的方式，找到原本左括号对应的右括号位置，此处即是正确的插入右括号位置。
- 若此时指针指向的是一个正常的值，则说明左侧是一个正常的操作数，在经过这个操作数之后加上括号即可。

对于左半括号以及其他种类的算符格式操作类似。在完成了加括号功能后，逐行处理算符添加括号就能达成考虑算符优先级的目的。

移动算符与去括号

上面提到的加括号步骤已经包括了一个简单的思想：向某一方向遍历时统计两个方向的括号即可找到括号的位置。此处的移动算符也是类似的思路，由于算符本身已经在一对括号之中，所以统计二者数量差值时的初值应为 1。
此处的移动还有一点细节：我们需要在合适的位置插入空格，以方便后续栈处理时的读取。最简单的想法是每次移动算符就在原本的位置加入一个空格，但这样会导致算符之间的空格数量不一致，因此我们按算符的类型分别进行处理。
对于一元算符/函数，我们在将其移动到右侧后，再在其左侧加入一个空格即可。对于二元运算符我们采用上面的空格插入方式，二元函数的处理较为特殊，我们将其移动至最后的同时将中间的分隔符逗号替换为空格。
至此，完成的表达式已经是规整且已处理的。作为示例，考虑算式 `1+2*3-4` ，其处理流程如下：

1. 逐行加括号 `1+2*3-4 -> 1+(2*3)-4 -> ((1+(2*3))-4)`
2. 移动算符并生成空格 `((1+(2*3))-4) -> 1 2 3 * + 4 -`

计算逻辑

得到逆波兰表达式后就可以开展计算了，这里又要分为几个步骤：

1. 维护计算用的栈，该栈为自定义栈并存储计算需要的一切信息；
2. 读取逆波兰表达式，将其转化为栈上操作；
3. 根据不同的算符进行不同的计算逻辑；
4. 解析结果并输出。

我们仍然逐步分析。

维护栈与栈上协议

有关栈的实现在文件 `mathStack.asm` 中，首先我们创建一个大数组，模拟了一个栈的空间，同时设立两个值作为栈顶与栈底的指针，这部分的思路与程序运行栈类似。
我们接受的输入可能是多种多样的，为了支持后续的操作，我们规定了一个栈上的协议来实现信息的分类区分。
协议规定一个数据由三部分组成：数据类型，数据大小，数据体。其中数据类型为一个 BYTE，数据大小为一个 WORD，数据体为一个不定长的数据段，根据数据类型和大小的不同而决定长度。三者在栈上紧密排列，数据体在栈靠近底部的位置，数据大小在中间，数据类型在顶部，这样排列才能支持栈的弹出操作。
有了协议之后我们封装了计算栈上的 push 和 pop 操作，统一完成了元素的压栈与弹栈。我们同时提供了用于读取顶部元素的函数，可以在不弹出元素的条件下获知元素情况，方便后续的计算过程实现。
作为具体实现的示例，考虑读取顶部元素值的函数，我们首先读取数据类型和大小，然后从栈顶的位置向下移动 n+3 个字节 (此处的 n 为数据大小)，此处便是数据段开始的地址，再利用 strncpy 等函数就能将目标数据拷贝到传入的地址，实现数据的读取。

读取表达式并操作

有了栈的支持，我们就可以开始读取表达式并进行操作了。读取较为简单，只要按照逆波兰表达式中的顺序正向遍历检测，在碰到一个元素/算符时调用相应的函数即可。首先判断碰到的是元素还是算符，我们先来讨论前者。
此处的细节问题在于如何处理遇到的元素。我们按如下方法检测元素类型：

- 若元素中包含小写字母，则为变量；
- 若元素中包含小数点.，则为浮点数；
- 否则，为整数。

根据上面的分类逻辑，压入栈时会压入不同的数据类型。数据段则是由原本的字符串转化而来，我们使用 sprintf 与 sscanf 函数完成在字符串与数值之间的转换，这样就能获取字符串对应的数值并压入栈中。
对于算符，我们的处理逻辑为首先获取栈顶的一个元素，并尝试匹配算符表中的一元算符，若匹配成功则进行一元运算，否则再获取一个栈顶元素并尝试匹配二元算符，若匹配成功则进行二元运算，否则进行错误处理。这是一个大致的思路，具体处理时还有些细节问题。
无论一元还是二元，匹配成功的同时也拿到了需要的元素，这里调用算符对应的计算函数即可。我们会针对不同的数据类型支持不同的计算，同时类型之间还能完成隐式的类型转换，让计算过程尽量合理，因此在获取元素的时候，其实会根据拿到的元素类型再进行分类处理。
此外，计算过程中发生的错误也应得到处理。栈上的数据类型包含了一个对应错误的类型，当计算过程中发生错误时，我们会将带有着错误类型的元素压入栈中，并且无论何时读取到一个错误类型时，程序都会将这个错误直接压回栈中，保留该错误信息到最后。

计算过程正常完成后，得到的结果压入栈中即可，在所有的算符/元素都被处理完后，正确的表达式应该使得栈中仅剩一个元素。

算符计算逻辑

上面也提到了算符实际执行的操作将与数据类型有关，数据类型之间也能进行转换，我们共有三种较为基本的数据类型：整数、浮点数与布尔值。在发生计算时，算符会有不同的需求，例如：

- '+'算符需要两个整数/浮点数，返回一个整数/浮点数，若两个操作数中有一个为浮点数，则应将二者都转化为浮点数并使用浮点数加法计算；
- '&&'算符需要两个布尔值，返回一个布尔值，传入的两个值均应转化为布尔值；
- '^'算符需要两个整数/一个整数一个浮点数，返回一个整数/浮点数，传入的两个值中第一个值类型不定，第二个值则不能为浮点数，需要转化为整数。

上面列举的几个例子足以展示问题的复杂性。我们在实现时采用了一种较为简单的思路：布尔值可以任意转换，当值为真时转化为整数 1/浮点数 1.0，否则转化为 0；整数可以转化为浮点数，但浮点数不能隐式转化为整数，这样做是为了保证数值计算时的精度不被舍入影响。
在保证计算函数都能拿到需要的数据类型后就可以进行计算了，每个算符对应的计算函数不同，这些计算函数大多在 `longInt.asm` ， `double.asm` 和 `boolean.asm` 中实现，三者分别处理与三种元素类型相关的计算函数实现。

解析结果与输出

经过上面的流程，我们已经得到了计算结果，这个结果本身仍然是一个与上面的数据相同格式的元素，我们需要将其解析并输出。对于不同种类的数据，我们设置了不同的输出函数，分别在不同数据类型的文件中。
作为示例，假设栈顶最终剩余的值是一个长整数，我们会调用 `longInt.asm` 中的 `strToLong` 函数，将其转化为字符串，之后将这个字符串放在最终的结果 BYTE 数组中，上面提到的图形界面部分就会从这里读取并进行输出。
输出时还有一个细节处理：由于我们实现的算符包含了 In/Out 算符，因此我们需要记录每次得到的结果。我们采用了一种较为“巧妙”的方式：在最终获取到结果时不再弹出这一个值，而是提升我们的计算栈的底部，这样就可以将结果保留在栈中，从而方便了之后的实现。

变量处理

变量处理的内容都包括在 `variables.asm` 中。

为了实现表达式的赋值与利用值，我们利用了一个类似哈希表的结构来管理各个变量的值。

我们定义一个数组 `variableHashTable BYTE MaxVariableHashTableSize*MaxVariableHashTableElemSize` 用以存取所有的变量。这个数组由 `MaxVariableHashTableSize` 个长度为 `MaxVariableHashTableElemSize` 的数据段组成。

我们规定变量的存取规则为：

```
; LOW:|<--VAR NAME SIZE-->|<--VAR NAME-->|<--VAR TYPE-->|<--VAR SIZE-->|<--VAR VALUE-->|:HIGH
```

其中：

VAR NAME SIZE: 变量名的长度，2 BYTEs

VAR NAME: 变量名，VAR NAME SIZE BYTEs

VAR TYPE: 变量类型，1 BYTE

VAR SIZE: 变量值的长度，2 BYTEs

VAR VALUE: 变量值，VAR SIZE BYTEs

当想要存入一个变量时：我们首先求指定变量名的哈希值，并且根据哈希值找到对应的数据段。如果这个数据段的 VAR NAME SIZE 为 0，说明这个数据段还没有被使用。于是我们就找到了存入的目的地。然后，按照传入的数据与存取的规定，即可完成存储。另一方面，如果数据段被使用了，那么如果是同名变量的话，就在此处覆盖变量的值; 而如果不是，就代表出现了哈希冲突，那就找下一个元素，直到出现上述两种我们可以插入数据的位置为止。

当想要取一个变量时，和存入时的思路大致类似。先通过哈希值计算出应当在那个位置寻找变量。如果当前数据段还未使用，这就表明没有找到，返回一个空地址; 而如果当前数据段的变量正是我们寻找的变量，则返回指向当前数据段的指针。如果当前数据段存在着一个变量，但不是我们寻找的变量，那么就说明出现了哈希冲突，那么就找下一个元素，直到出现上述两种可以返回指针的情况为止。

我们使用形如算符 `x := 1` 的形式为变量赋值，一经赋值，x 与 1 就完全等价，直到 x 被重新赋值为止。

内置函数计算

这一部分的内容主要在 `numasm.asm` 中实现。

为了更好地服务于科学计算，我们内置了一些给定的科学常数和函数，以供用户使用。

内置科学常数 (以变量的形式呈现，后续也可以根据需要赋成其他值) 包括：

- e: 自然常数
- pi: 圆周率
- ln10: 以 e 为底的 10 的对数
- ln2: 以 e 为底的 2 的对数
- c0: 真空中的光速

内置函数主要包括：

- Fact: 阶乘
- Sqrt: 平方根
- Sin: 正弦
- Cos: 余弦
- Tan: 正切
- Exp: 指数
- Ln: 自然对数

内置函数的具体实现方法如下：

1. Fact

我们首先管理一个数组 `factTable` ，用以存储 0 到 12 的阶乘，用于加速计算。然后，我们利用递归的思想，每次递归调用时，将当前的数值减一，直到减到 12 为止。递归调用的过程中，我们将每次的结果乘到一起，最后返回结果即可。

2. Sqrt

我们利用牛顿法，得到迭代公式：

$$t_{n+1} = (t_n + x/t_n)/2$$

随便设定一个初值，重复计算，直到

$$|t_{n+1} - t_n| < 1 \times 10^{-6}$$

为止。最后返回 t_{n+1} 即可。

3. Sin, Cos, Tan

我们利用泰勒展开，得到：

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

我们找到一个θ，使得 $x = \theta + 2k\pi$，其中 k 为整数，且 $|\theta| < \pi$。然后，我们利用上述公式计算 $sin(\theta)$ 和 $cos(\theta)$，因为级数收敛颇快，因此直接计算 20 阶左右的级数即可。

对于 Tan 的情形。利用 $tan(x) = sin(x)/cos(x)$ 即可。

4. Exp

我们基于浮点数的整数次方的实现来实现这个函数。

浮点数的整数次方：用递归的思想。我们每次递归调用这个过程。把这个整数次方拆成两个具一半指数的浮点数相乘，直到指数为 0 为止。最后返回结果即可。

我们首先求 $e^{floor(x)}$, 然后在$floor(x)$处将指数函数展开：

$$e^x = e^{floor(x)}(1 + dx + dx^2/2! + dx^3/3! + ...)$$

重复求和，直到 $|sumNext - sum| < 1 \times 10^{-6}$ 为止。最后返回结果即可。

5. Ln

我们利用牛顿法，得到递推式：

$$t_{n+1} = t_n - 1 + x/e^{t_n}$$

重复这个迭代，直到 $|t_{n+1} - t_n| < 1 \times 10^{-6}$ 为止。最后返回 t_{n+1} 即可。

与中期相比

与中期相比我们的项目取得了相当大的进展 (虽然细想下来中期之前搭建了最为困难也最为关键的表达式处理，由于这部分的撰写和 debug 要花费大量时间，因此剩余留给实际的功能细节的时间较少)。
我们仍然按照难点与创新点的顺序进行讲述。

图形界面

中期时图形界面大体完成，但我们新处理了两个很重要的点：一是滚动栏的完整实现让计算不局限于屏幕大小，二是菜单栏的添加让用户能够更好地使用我们的程序。

表达式处理

中期的表达式处理基本成型，但仅仅支持二元算符与基本的四则运算 (实际上受限于时间不足，当时只支持了其中的三个)，这与我们现在的项目可谓天壤之别。

我们设立了算符类型表并添加了对于一元算符乃至函数的支持，这部分的进展也同步带来了支持算符数量的增长。

计算逻辑

算符的数量已经由当时的 3 个增长到了现在的二十余个，在这二十余个算符的背后是大量的计算函数实现，我们以数据类型为切入点，按照建立数据类型支持->实现计算函数->支持算符的顺序进行开发。事实证明这样的开发流程相当合理。
在引入更多的数据类型的同时，几乎所有相关的逻辑处理部分的代码都得到了重构，我们需要应对不同数据类型的转化、计算、输入与输出，因此这里是一个重大的革新点。

变量处理与内置函数

我们使用哈希表，实现了变量的值的快速管理，这符合我们科学计算的初衷.
另一方面，我们实现了一些内置函数，以供用户使用。这些内置函数的实现也是我们项目的一个亮点.

小组分工

钟健坤：图形界面实现，表达式处理实现，长整数、浮点数与布尔值计算与支持，报告撰写
陈者霖：变量处理实现，内置函数支持实现，栈协议与栈实现，报告撰写