CS205 · C/C++ Program Design

A Better Calculator

Monad

October 16, 2022

目录

1 功能展示

1. **交互式输入输出**：当需要用户输入时，会打印 > （如下所示）来提示用户输入。
2. **任意精度计算**

   ```
   > 9999999999999999999.2222222 + 1.0
   10000000000000000000.2222222
   ```

   对于除法，对于一些无限小数（如 1 / 3），会根据设定的 scale 保留相应的位数。
3. **支持定义变量**

   ```
   > x = 3
   3
   > y = 6
   6
   > x + 2 * y
   15
   ```

   赋值甚至能嵌套

   ```
   > y = (x = 3) * 2
   6
   > x + 2 * y
   15
   ```
4. **支持定义函数**

   ```
   > f[x] = x * (x + 1)
   > f[5]
   30
   ```
5. **内置数学函数**

   ```
   > sqrt[3]
   1.7320508075688772935 2
   ```

   数学函数支持的精度可以看作是无限的，它的精度可以通过设置 scale 来调整。

   ```
   $ ./calc --scale 50
   > sqrt[3]
   1.73205080756887729352744634150587236694280525381039
   ```
6. **友善的错误提示**：当你的输入的某个地方有错误时，它会⋯⋯

   它会很温馨地把错误的地方标出来，告诉你具体是哪个地方寄了。

   ```
   > 1 + sqrt[5, 6]
         ~~~~~~~~~~
   Error: expected 1 arguments, but got 2
   ```
7. **列出所有变量**

   注：下面的输出忽略了一些内建函数，感兴趣的可以继续阅读下去。

```
> env
(variable) y = 6
(function) f = (x * (x + 1))
(function) sqrt = <built-in function>
(variable) x = 3
...
```

以上就是本计算器的主要功能。如果你想进一步了解“这个计算器能做什么炫酷的事情”，可以直接跳到“还能这么干？”部分。

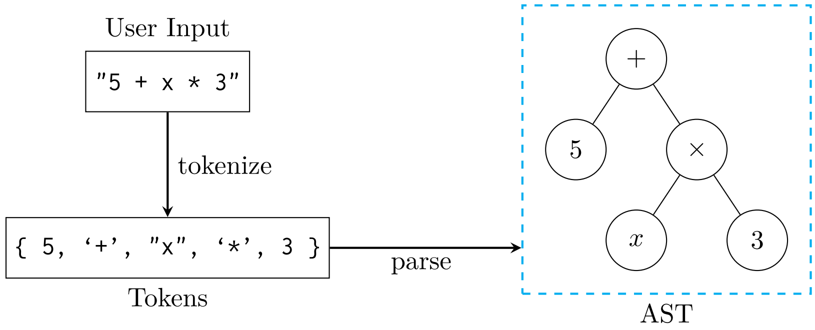
2 功能实现

2.1 输入解析

因为之前接触过一点点编译相关的东西（真的只有一点点），所以为了方便起见，输入的解析主要分为两步：

1. **tokenize**：将输入字符串初步解析为一个 token 列表，里面只包含对解析有用的语义元素（分为数字、符号、identifier 三种），空格在这一步就可以扔掉了。
2. **parse**：将上述的 token 列表解析成一颗 AST 树。

以输入`"5 + x * 3"`为例，具体过程就是这样子的：



把输入转成 AST 之后，我们就只需要简单地遍历一遍这棵树，就可以算出结果了。

那，这样子做有什么好处呢，或者换句话说，为什么不在解析的时候，就顺手算出结果，而是要特意构建这棵树呢？

这样做主要有两个好处：

- 首先，这样做可以降低代码耦合度，使代码模块化。让解析部分的代码只负责解析，让运算部分的代码只负责运算，将两者的逻辑分离，无疑可以使代码更加清晰易懂。

  如果把功能不同的代码都堆在一起，无疑会使代码非常乱，而且也较难维护，也很容易出 bug。

- 其次，因为我们后面还有定义函数的部分。对于函数而言，只有在调用的时候才知道参数的值，就没有办法在解析的时候就算出结果。

  如果不构建 AST，那么只能在每次调用函数的时候，都重新解析一次来运算，除了降低性能不说，储存 unstructured data 也是一个非常不好的 practice [1]，也非常容易出 bug。

  但是如果解析和储存 AST，那对于每次函数调用，都只需要带着参数的值遍历一遍 AST，就十分地方便。

对于 tokenize 和 parse 这两个部分具体怎么操作，下面会慢慢道来。

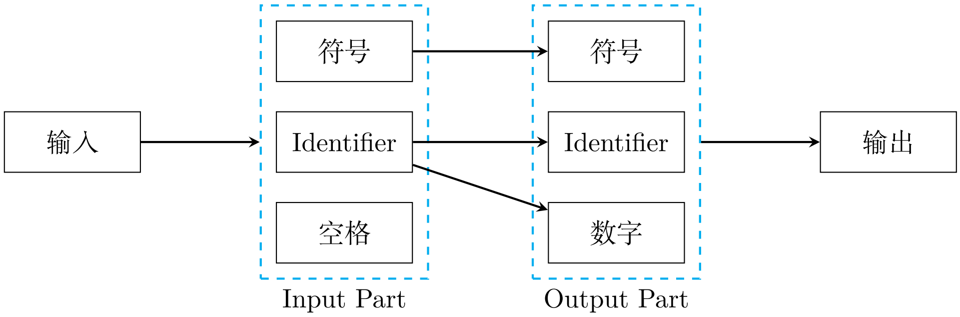
[1] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html#p7-catch-run-time-errors-early

2.1.1 Tokenize

前面说到，token 元素有三种类型，为数字、符号（punctuator）和 identifier。这里需要补充一点，就是这三种类型是对于 tokenize 的结果而言的。

这是因为在 tokenize 的内部过程中，对输入也分为三种类型，不过与上面的输出的不太一样：符号、identifier 和空格。其中空格会在 tokenize 的过程中丢弃，不会返回。

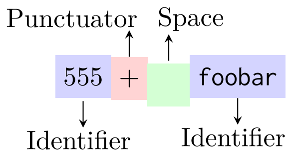
它们之间的关系可以用这么一张图来解释：



输入 → Input Part

对于输入的三种类型，符号只包含 `+`, `-` 等会用到的运算符和括号，identifier 只包含数字、字母和下划线（`[0-9a-zA-Z_]`）还有小数点 `.`，空格只有空格和 `\t` 两种。然后不难发现，这么一分之后，三种类型就是正交的了，即可以直接由单个字符判断类型，与上下文无关。

然后接下来的事情，就只需要对每个字符判断其类型，然后按照它们的类型划分成若干个块（如下所示）。具体操作就是，当遍历的时候，如果发现字符的类型发生了变化，那就可以确定前面一个块的范围了。然后当一个块被发现之后，就可以拿去解析了（下一部分）。特别的，对于符号类型，因为本项目没有两个符号的操作符，所以每个符号都是一个块，不需要连在一起。



代码如下：

```
enum TokenType { WhitespaceType, PunctuatorType, ValueType } type = WhitespaceType;

for (size_t i = 0, j = 0; i < input.length(); i++) {
    char ch = input[i];
    TokenType next = is_space(ch) ? WhitespaceType : is_punctuator(ch) ? PunctuatorType : ValueType;
    if (type != next || type == PunctuatorType) {
        if (type != WhitespaceType)
            // we get (input[i:j], type) as Input Part here
        j = i;
        type = next;
    }
}
```

Input Part → Output Part

这部分讲的是将上文所说的一个一个块解析成 token 的过程。

首先，对于 Input Part 的 Punctuator，直接原样转成 Output Part 即可。

对于 Input Part 的 Identifier，我们需要判断它是不是

- 真正的 Identifier：即仅包含数字字母和下划线，并且不以数字开头

- 数字：扔给 BigDecimal 解析并成功

如果这两者都不能成功解析，那就说明遇到了非法输入，就可以报错了。

输出的表示

因为最终 token 有三种类型，而且每一种的类型对应的 C++ 类型都不一样（分别为 std::string、char 和 BigDecimal）。所以这里用了 C++17 的 std::variant 来表示 token。这玩意与 union 差不多，只不过有比较完善的模板支持和检查，不容易出错。而且后面 parse 的时候，它支持根据类型做分发，也挺方便的。

具体类型定义如下：

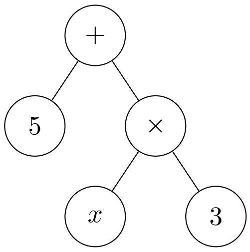
```
using Punctuator = char;
using Identifier = std::string;
using TokenContent = std::variant<BigDecimal, Identifier, Punctuator>;
```

2.1.2 解析

这里的解析，是指从 token 列表生成 AST 树的过程。

然后由于我确实不太会编译原理，不会手写 LL、LR，所以就只能用 DSAA 里面讲过的、用栈来解析表达式的方法了。

AST 树的表示



如上所示就是一颗典型的 AST 树。虽然每个节点都有不同的意义，但是它们都有一个共同的特征（或者说可以用一个接口来表示）：可以计算（evaluate）。即对于每个节点，它都可以代表以它为根的子树，而每棵子树，都可以计算出其下表达式的值。并且在后面计算的时候，我们并不关心每个节点具体是什么，我们只关心它们算出来的结果。

所以我们可以用多态来实现。它们共有的特征可以用这样一个接口表示：

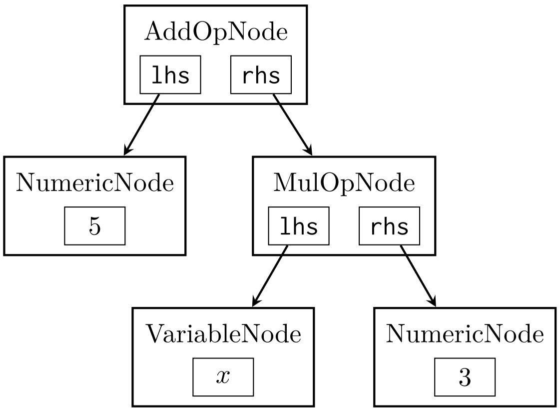
```
class Expression {
 public:
    virtual ~Expression() = default;

    virtual BigDecimal eval(Context &context) = 0;
    virtual void print(std::ostream &stream) const = 0;
};
```

其中 Context 是储存了一些变量信息的上下文，在后面的变量部分会详解介绍。然后 print 是用来打印这个表达式。然后其它的节点只需要继承 Expression，添加自己的属性，实现这些方法就行。调用方只需要调用 eval，而无需关心具体细节。

在 C++ 里的多态，需要涉及指针的操作。然后为了方便（不泄漏内存），这里不直接使用 new 和 delete 申请和释放内存，而是使用 C++ 的智能指针 unique_ptr 和 shared_ptr 来管理。

所以上面那棵 AST 树，储存起来大概是这样子的：



基础解析

这里的话，先假设所有运算符都是二元运算符（然后再逐步加 feature）。二元运算符就是就是像 `+`, `-` 之类的、左右各有一个 operand 的运算符。

然后在这种情况下，用栈来解析表达式就是一种经典做法，非常简单。就是扫描 token 的时候，遇到操作数就扔进栈里，如果遇到操作符，就把前面的具有更高优先级的运算符先合并了，然后再把自己压进栈里。

伪代码如下：

```
stack ← {}                                              ▷ 运算符栈
values ← {}                                             ▷ operand 栈
function MergeTopOperation                              ▷ 合并栈顶运算符
    lhs ← Pop(values)
    rhs ← Pop(values)
    op ← Pop(stack)
    Push(values, lhs <op> rhs)
end function

for each token ∈ tokens do
    if token is operand then                            ▷ 运算数就直接丢栈里
        Push(values, token)
    else                                 ▷ 运算符就先尝试合并前面的，再丢进栈里
        while stack is not empty and Precedence(Top(stack)) ≥ token's do
            MergeTopOperation
        end while
        Push(stack, token)
    end if
end for

while stack is empty do                  ▷ 解析完成之后，把剩下的东西都合并了
    MergeTopOperation
end while
result ← Pop(values)
```

这样我们就完成了最基本的表达式解析。

错误处理

为什么这么快讲错误处理？因为这里面会引入一些新的、后面会用到的东西。

上面的解析虽然非常简单，但是对于一些错误的处理会比较的无力。举个例子，由于上面把操作符和操作数分开处理，所以 `"5 + 2 * 3"` 和 `"5 2 3 + *"` 会被认为是等价的。

为了检测和处理这种错误，这里引入一个类似于状态机的东西。它有两种状态：Value（记为 V）和 Punctuator（记为 P），分别表示当前解析到的位置的前一个元素，是数字类（包括变量）的还是 Punctuator 类的。

举个例子，

- `"5"` $\to V$（5 是 Value）
- `"5+"` $\to P$（`+` 是 Punctuator）
- `"5+2"` $\to V$（2 是 Value）

然后对于解析中出现的 token，可以看作是转移条件，从一个状态转移到另一个状态。不过这里的转移与一般的自动机不同，在一般的自动机中，一般是 $q \leftarrow \delta(q, \mathsf{token})$，即 "原状态 + 转移条件 → 下一状态"。而这里的转移条件，会同时约束原状态并规定下一状态。

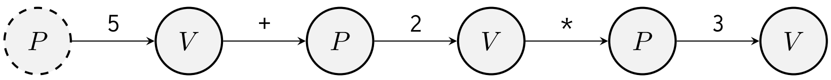
例如，对于 `+` 而言，它期望原状态是 V（即前面是数字），并且转移到 P 去。为了方便起见，这里记为 `+` 的转移为 $V \to P$。

类似的，在这个简单表达式下，不同 token 的转移如下：

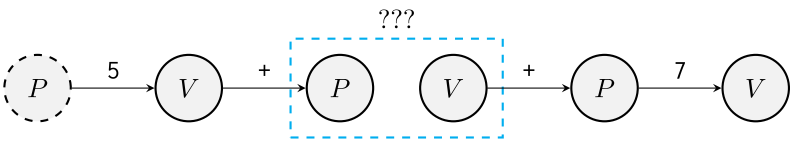
- 操作符（`+`, `-` 等）：$V \to P$。
- 数字类（`5`, `y` 等）：$P \to V$。

然后对于一个合法输入，不难发现，把它的所有 token 的转移连接起来，会形成一条路径，并且开头一定是 P，结尾一定是 V。

还是以 `"5 + 2 * 3"` 为例，它显然可以连成一条合法路径：



如果是一个不合法的输入，比如 `5 + + 7`，就会在两个 `+` 之间寄掉：



然后用这种方法就可以在处理 token 的时候，顺便看看当前状态是否转移条件，如果不符合，就说明输入不合法。

括号与函数

处理完了简单的表达式之后，还有更复杂的带括号 `"2 * (3 + 4)"` 和带函数调用 `"2 * sqrt[5]"` 的表达式需要处理。

这里为了方便处理，用来规定优先级的圆括号就用了圆括号本身，函数调用就用了方括号表示。~~没事，反正 Mathematica 也是这样的。~~

对于括号来说，遇到 '`(`' 就直接把它压到栈里面（不进行 MergeTopOperation 操作）。并且要把括号的优先级设为最低，以免被后面的运算符 MergeTopOperation 了。

然后遇到 '`)`'，就一直 MergeTopOperation，直到遇到 '`(`' 即可。即：

```
function FoldParentheses
    while stack is not empty and Top(stack) ≠ '(' do
        MergeTopOperation
    end while
    if stack is empty then
        Error: no matching '(' was found
    end if
    Pop(stack)                                    ▷ pop '('
end function
```

对于方括号，其实也与圆括号类似。对于 ‘[’ 就直接压到栈上。对于 ‘]’ 在没有遇到 ‘[’ 之前就一直 MergeTopOperation 即可。

不同之处在于，方括号（即函数调用）里面可能会有一个或多个 ‘,’（当然也有可能没有），用来分割不同参数。而且参数是要收集起来，变成一个函数调用节点，而不是像圆括号那样继续堆在栈上。

这里为了方便起见，我们规定函数至少有一个参数，即 [] 内不为空。

通过分析，不难发现，如果遇到 ‘,’ 或 ‘[’ 时，就要从 values 栈中弹出一个元素（Pop(values)），这个元素就是此函数调用中的一个参数，然后把它加入到 args 参数列表中。

然后函数调用除了参数，还得有函数名字，这个也不难实现。如果 ‘[’ 前面有函数名字的话，那么它（函数名字）一定会

1. 被看作变量放在 stack 中
2. 不会被合并，因为 ‘[’ 不会合并前面的运算符，也会阻止后面的运算符合并到它前面

这样的话，我们就可以在弹出 ‘[’ 之后，再从 values 弹出一个元素。如果输入是合法的话，那么它一定是一个变量节点，那我们就可以从它里面提取出名字，就 ok 了。（如果这个元素不是变量的话，那就说明输入不合法。）

写成伪代码，就是这个样子：

```
function FoldSquareBrackets
    args ← {}
    while stack is not empty and Top(stack) ≠ ‘[’ do
        if Top(stack) = ‘,’ then
            Prepend(args, Pop(values))                ▷ 倒序插入
        else
            MergeTopOperation
        end if
    end while
    if stack is empty then
        Error: no matching ‘[’ was found
    end if
    Prepend(args, Pop(values))
    Pop(stack)                                        ▷ pop ‘[’
    return args
end function

function HandleSquareBrackets
    args ← FoldSquareBrackets()
    name ← Pop(values) as Identifier
    Push(values, function call name(args))
end function
```

然后分析一下，就可以得出这些操作符的状态转移：

- ‘(’：$P \to P$
- ‘)’：$V \to V$
- ‘[’：$V \to P$（前面的函数名（变量）是 V）
- ‘]’：$V \to V$
- ‘,’：$V \to P$

其它

除了上面讲到的一些比较基本的运算符，这里还实现了一些其它的运算符，这样解析和使用起来会更方便：

‘=’: ‘=’ 运算符主要用在变量或函数的定义中。赋值运算会把 ‘=’ 右边的表达式，赋值给 ‘=’ 的变量。

‘=’ 本质上也是二元运算，所以它的解析也与二元运算符相似，即把右边的 Expression 赋值给左边的 Expression。不同之处在于，‘=’ 左边的 Expression 需要是一个合法的变量名或一个合法的函数声明（例如 "f[x, y]"）。这一点的处理，与前面如何在函数调用中取得函数名字的处理相似，这里就不展开了。

‘;’: ‘;’ 主要用于分割语句。在函数中或一些其它的场合，我们往往会希望执行几条语句（如 "x=1; x+2"），而不仅仅是一条，所以就引入了 ‘;’ 来分割。

‘;’ 本质上也是一个二元运算，它会执行左、右两边的 Expression，并返回右边的结果。

运算符优先级

到这里，已经把这次 Project 中会用到的或者会实现的运算符，全部都介绍完了。然后通过综合分析，可以得到这些运算符的优先级：

Operator	Precedence
([,	1
;	2
=	3
< >	4
+ -	5
* / %	6

注：Precedence 越大，则它的运算优先级更高。

然后没有) 和] 的原因是，这两个符号在遇到的时候，会直接折叠前面的内容，直到遇到相应的 (或 [，并不会用到它们两个的优先级。

2.2 运算

本 Project 的高精度数字的存储沿用了上次 Project 的方案。

即先用 BigInteger 储存一个高精度整数，然后 BigDecimal 用 $m \times 10^n$ 的方式来存储，其中 m 为 BigInteger，n 为一个有符号整数。

这次 Project 的储存部分的一些细节，相较于上次 Project 有所调整。

- 本次 Project 的数字存储不压位。

  因为如果压位的话，在两个数字的 n 不相同的情况下，做加减法时，要把两个数的小数点对齐会比较麻烦，很容易会产生 bug。

- 本次 Project 中 BigInteger 储存数字的数组是反序存放的。即上次 Project 的储存方式是

$$
\begin{array}{rccccccccccccc}
\text{number} = \text{"} & 9 & 7 & 8 & 1 & 9 & 7 & 5 & 3 & 3 & 2 & 9 & 5 & 2\text{"} \\
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\texttt{char[] = \{} & 9, & 7, & 8, & 1, & 9, & 7, & 5, & 3, & 3, & 2, & 9, & 5, & 2\ \}
\end{array}
$$

  而这次的是

$$
\begin{array}{rccccccccccccc}
\text{number} = \text{"} & 9 & 7 & 8 & 1 & 9 & 7 & 5 & 3 & 3 & 2 & 9 & 5 & 2\text{"} \\
\texttt{char[] = \{} & 2, & 5, & 9, & 2, & 3, & 3, & 5, & 7, & 9, & 1, & 8, & 7, & 9\ \} \\
\text{indices:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array}
$$

  这样做有两个好处，首先是我们算加法的时候，一般是从低位到高位遍历的，刚好就是从下标 0 开始递增遍历，更符合日常的编写习惯。再者，如果两个整数的长度不一样，做加减法的时候，它们之间就是第 i 位对第 i 位（而不是 $(\text{length}(s_1) - i)$ 位对应第 $(\text{length}(s_2) - i)$ 位），同样也更自然更方便处理。

- 正负号的 flag 从 BigInteger 提升到了 BigDecimal，现在 BigInteger 不储存正负号。

2.2.1 加法与减法

把加法与减法放在一起，是因为它们两个比较相似，并且由于我们这里有正负数之分，所以它们两个在某种程度上是相互依赖的，所以一起讲会比较合适。

`BigInteger` 加减法

为了简单起见，我们先从符号相同（因为 `BigInteger` 是无符号的）的加减法开始考虑。

indices	3	2	1	0
			2	3
+		5	9	1
		6	1	4

这样加减法很简单，先把两个数对齐（~~其实本来就是对齐的，这就是反序储存的好处~~），然后直接按位相加，并且处理一下进位即可。

然后减法也类似，`BigInteger` 的减法只考虑大数减小数（靠调用方 `BigDecimal` 保证），也是直接按位相减，并且处理一下借位即可。

`BigDecimal` 加减法

对于 `BigDecimal`，相比 `BigInteger` 而言，它多了一个正负号，还多了一个 n（指数）。

如果指数不相同，我们需要把其中一个数的 m（底数）末尾补相应的 0，使两个数的 n 一样，再对底数部分相加减即可。比如说，我们有两个数 $a_1 = m_1 \times 10^{n_1}$，$a_2 = m_2 \times 10^{n_2}$（设 $n_1 < n_2$），则

$$
\begin{aligned}
a_1 \pm a_2 &= m_1 \times 10^{n_1} \pm m_2 \times 10^{n_2} \\
&= m_1 \times 10^{n_1} \pm (m_2 \cdot 10^{n_2 - n_1}) \times 10^{n_1} \\
&= \big(m_1 \pm (m_2 \cdot 10^{n_2 - n_1})\big) \times 10^{n_1}
\end{aligned}
$$

把指数部分的差异抹除之后，我们就只需要关注 $m_1 \pm (m_2 \cdot 10^{n_2 - n_1})$ 了。

这里令 $v_1 = m_1, v_2 = m_2 \cdot 10^{n_2 - n_1}$。若 v_1 与 v_2 同号，那很好，只需要直接做加减法就行了（需要注意的是，减法需要保证大减小，如果 $v_2 > v_1$，只需要交换一下，并且在结果中加一个负号就行了）。如果异号，那也很好，直接加法变减法，减法变加法，变完之后和上面的同号加减法就一样了。

总的来说，v_1 和 v_2 相加减的伪代码（逻辑）如下：

```
function ADD(v1, v2)
    if SIGN(v1) = SIGN(v2) then
        return v1 + v2
    else if |v1| ≥ |v2| then                ▷ 确保是大数减小数
        return v1 − v2 (sign of v1)
    else
        return v2 − v1 (sign of v2)
    end if
end function

function SUB(v1, v2)
    if SIGN(v1) ≠ SIGN(v2) then
        return v1 + v2 (sign of v1)          ▷ 异号就是相加
    else if |v1| ≥ |v2| then
        return v1 − v2 (sign of v1)
    else
        return v2 − v1 (sign of −v1)
    end if
end function
```

然后拼在一起，就是 `BigDecimal` 的加减法了。

2.2.2 乘法

`BigDecimal` 的乘法就直接沿用上次 Project 的乘法了。主要还是用多项式 FFT 来加速乘法。

2.2.3 除法

计算方式

假设我们有两个数 v_1 和 v_2，现在要求 $v_1 \div v_2$，并且要求保留到 s 位小数。

普通的高精度除法，即使加上商的估算，时间复杂度也还是高达 $O(n^2)$，太暴力了感觉有点不优雅。

所以下面介绍一种时间复杂度约为 $O(n \log^2 n)$ 的算法 [1]。

要求 $v_1 \div v_2$，我们可以转换成计算

$$v_1 \cdot v_2^{-1}$$

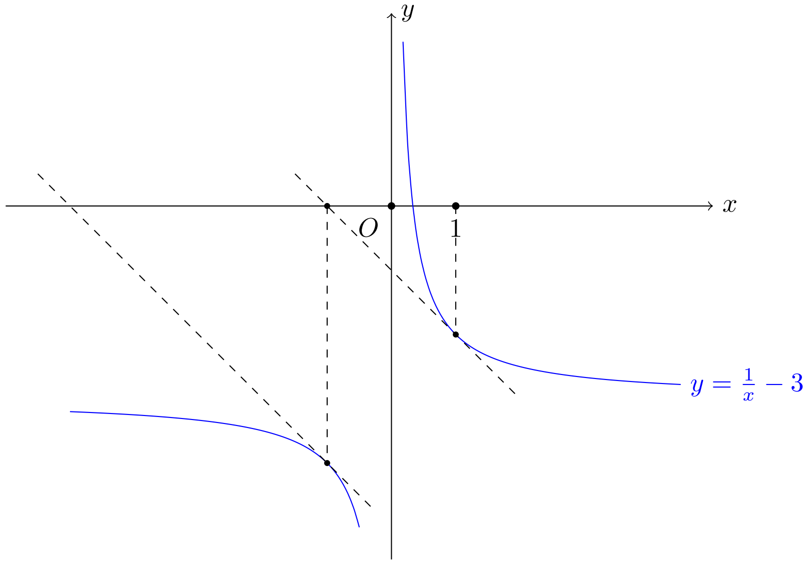
其中 v_2^{-1} 为 v_2 的逆元。要求这个逆元，我们可以令

$$f(x) = \frac{1}{x} - v_2$$

则 $f(x) = 0$ 的解即为 v_2 的逆元。要解出此方程，可以用牛顿迭代法

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{\frac{1}{x_n} - v_2}{-\frac{1}{x_n^2}} = (2 - v_2 x_n)x_n$$

由于 $y = \frac{1}{x}$ 图形的特殊性，牛顿迭代法的初值并不能随便设置，否则有极大概率不会收敛。例如，如果我们取 $x = 1$ 作为初值，那么如下图所示，我们下一步就会取到 $x = -1$，然后下一步会继续往 x 轴负方向跑，丝毫不会收敛。而如果取 $x = 0$，也只会停留在 $x = 0$。



所以为了使其更快更好地收敛，我们需要给它一个好的初值（最好在零点的附近）。因为我们储存高精度数的方式是 $v_2 = m \times 10^n$，所以我们可以取出它的 n，让 $x = 1 \times 10^{-n}$ 作为迭代的初值，就非常地合适。

此时，对于除数 6537，其收敛速度如下

```
0.0001
0.00013463
0.00015077532638 47
0.00015294373060299944813645983367
0.00015297536442100201550474073060405979470959761374847954272224007
0.0001529753709652743108274920292486741909751274244875910898055264372192722179
0.00015297537096527459079088266788995726759672352041890593267790481798242096 27
0.000152975370965274590790882667890469634388863392993728009790422025686100 8421
```

可见这个收敛速度还是非常不错的，准确的数字数量几乎以两倍的速度增加。

计算完 v_2^{-1} 之后，我们就只用把它和 v_1 乘一下，就可以得到除法的结果了。

精度控制

对于精度要求，我们的目标是运算的结果精确到 s 位小数，那么做牛顿迭代的时候，应该迭代到多少位小数合适呢?

不妨设 v_1 的数量级为 n_1（即 v_1 的整数部分有 n_1 位），v_2 的数量级为 n_2。那么它们相除的结果 $r = \frac{v_1}{v_2} = 10^{n_1 - n_2}$ 的数量级为 $(n_1 - n_2)$，则 r 会有 $(n_1 - n_2 + s)$ 位有效数字。

记 $k = n_1 - n_2 + s$ 为 r 的数量级，由于 $r = v_1 \cdot v_2^{-1}$，如果要让 r 的 k 位有效数字都是准确的，那么就要求 v_2^{-1} 也要有 k 位有效数字。又因为 v_2^{-1} 的数量级为 $-n_2$，那么 v_2^{-1} 的小数位数需要有 $k - (-n_2) = n_1 + s$ 位。

所以做迭代的时候，v_2^{-1} 保留到 $(n_1 + s)$ 位小数就够了。如果为了四舍五入或者冗余，也可以保留到 $(n_1 + s + 7)$ 位。

2.2.4 取模运算

一般来说，$a \bmod b$ 定义为 a 除以 b 得到的余数，其中 a 和 b 都是整数。但是本 Project 绝大部分数字都是小数，只支持整数多少有点别扭。所以我这里把取模运算推广了一下，使其也可以支持小数。

这里定义

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

值得注意的是，当 a, b 为整数时，其结果与普通的取模的结果是相同的。

这样定义除了可以方便实现之外，还可以支持一些炫酷的操作（见“还能这么干？”的“工具函数”部分），属于是一举两得了。

2.2.5 数学函数

开方运算

开方运算当然也可以用牛顿迭代法做。

令 v 为需要开方的数，则可以转化为求解 $f(x) = x^2 - v$ 的零点，$x_{n+1} = \frac{1}{2}\left(x_n + \frac{v}{x_n}\right)$，虽然也“能用”，但是速度不会很快：因为每一次迭代都要做一次除法运算，而除法运算是非常昂贵的，大量进行除法操作会拖慢开方的速度。

受到 Fast inverse square root[2] 和上面除法的神奇操作的启发，我们可以先计算 $\frac{1}{\sqrt{x}}$，然后再取倒数。

计算 $\frac{1}{\sqrt{x}}$ 也是用牛顿迭代法，令

$$f(x) = \frac{1}{x^2} - v$$

[2] https://en.wikipedia.org/wiki/Fast_inverse_square_root

则

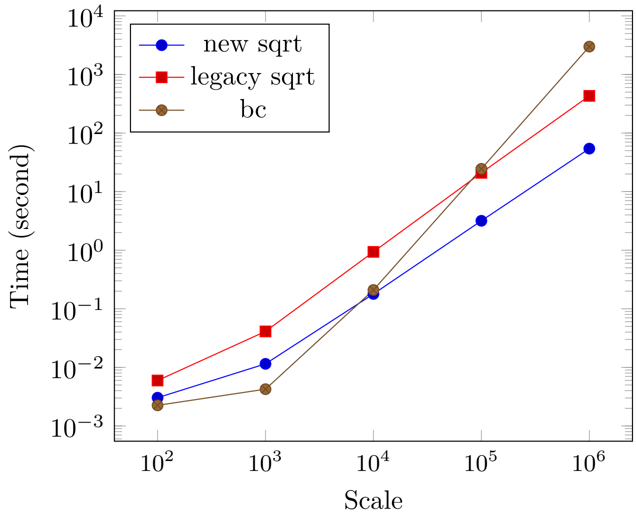
$$x_{n+1} = x_n - \frac{f(x)}{f'(x)} = \left(\frac{3}{2} - \frac{v}{2}x_n^2\right)x_n$$

虽然同样是牛顿迭代法，但是与上一种方法相比，可以发现，它没有用到任何除法。

由于 $y = \frac{1}{x^2}$ 的图像也很特殊，所以也需要挑一个合适的初值来迭代。因为 $v = m \times 10^n$，那我们令初值为 $1 \times 10^{-\frac{n+1}{2}}$ 即可。

把 $r = \frac{1}{\sqrt{x}}$ 算出来之后，再用 r 去除一下 1，即可得到 $\sqrt{x}$。如果担心取倒数会影响精度，可以在最后加一次牛顿迭代 $x_{n+1} = \frac{1}{2}\left(x_n + \frac{v}{x_n}\right)$。

算上最后的修正，整个开方过程只用到了两次除法，比上面那种方法快七八倍是没有问题的。[3]



三角函数

三角函数主要有两个：sin 和 cos。因为有泰勒展开的存在，将一些数学函数的计算变成多项式的计算之后，就会十分方便。

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \cdots = \sum_{k=0}^{\infty} \frac{(-x^2)^k x}{(1+2k)!}$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \cdots = \sum_{k=0}^{\infty} \frac{(-x^2)^k}{(2k)!}$$

根据泰勒展开的性质，如果我们取前 n 项的话，那么其误差不会超过第 $n-1$ 项的大小。所以我们可以从第一项一直累加，直到某一项小于指定误差，就说明 ok 了。

function $\textsc{Sin}(x)$
    $y \leftarrow 0$
    $k \leftarrow 2$
    $t \leftarrow x$
    while $t \geq EPS$ **do**
        $y \leftarrow y + t$
        $t \leftarrow t \times \frac{-x^2}{k(k+1)}$
        $k \leftarrow k + 2$
    end while
    return y
end function

[3]在 MacBook Air, M1 chip, 16G memory 机器上，计算 $\sqrt{2}$，重复 8 次取平均时间

其中 $(-x^2)$ 可以提取到循环的外部，就不用每次都乘一遍，可以加快运算速度。对于 cos 函数，其泰勒展开形式与 sin 类似，故计算方法也与 sin 类似，这里就不展开了。

反三角函数

对于反三角函数，这里实现 $\tan^{-1}$。对于 $\tan^{-1}$，也有泰勒展开

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots = \sum_{k=0}^{\infty} \frac{(-x^2)^k x}{1+2k}$$

但是这个泰勒展开是只在 $(-1, 1)$ 上收敛，而且 x 越接近 0，则收敛速度就越快。

但是当 x 比较大时，比如 $x = 1$，这个公式收敛的速度就会非常慢，如果要计算精确到小数点后 n 位的话，就要算到第 $O(n)$ 项，十分地“可观”。

从 bc 的计算脚本 [2] 中得到启发，幸运的是，对于比较大的 x，可以通过一个公式把 x 拆成比较小的数。由等式 [4]

$$\tan^{-1}(x) + \tan^{-1}(y) = \tan^{-1}\left(\frac{x+y}{1-xy}\right)$$

可得

$$\tan^{-1}(x) = \tan^{-1}(c) + \tan^{-1}\left(\frac{x-c}{1+cx}\right)$$

这里我们令 c 是一个比较小的数，这里取 $c = 0.2$。

当 $x > 0.2$ 时，我们就可以把它拆成 c 和 $\frac{x-c}{1+cx}$ 两部分分别求解。经过这样处理，代入到泰勒展开的 x 最大就只有 0.2，$\frac{(-x^2)^k x}{1+2k}$ 会比较快地收敛。

具体的实现方式也与 sin 函数类似。

指数函数

对于指数函数 $y = e^x$，有泰勒展开：

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

而且它的泰勒展开在 $\mathbb{R}$ 上都是收敛的，所以直接像 sin 那样实现一个泰勒展开即可。

对数函数

同样，对数函数（即 ln）也有一个非常常用的泰勒展开

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots = -\sum_{k=1}^{\infty} \frac{(-x)^k}{k}$$

且只有当 $-1 < x \le 1$ 时收敛，且越接近 0 收敛速度越快。所以和上面的 $\tan^{-1}$ 类似，求解 $\ln x$ 的时候最好让 x 在 1 的附近。

如果输入的数比较大，我们可以用 $\ln(xy) = \ln x + \ln y$ 来拆开。这里可以令 $y = e$，这样 $\ln y = \ln e = 1$ 就会比较方便。对于比较小的数也类似，可以乘上 e 让它变大。其中 e 的值可以用上面的指数函数求得。

有了指数函数 e^x 之后，我们也可以以此实现一个数的任意次幂了。

[4] 证明：`https://math.stackexchange.com/questions/502189/why-is-arctan-fracxy1-xy-arctan-x-arctan-y`

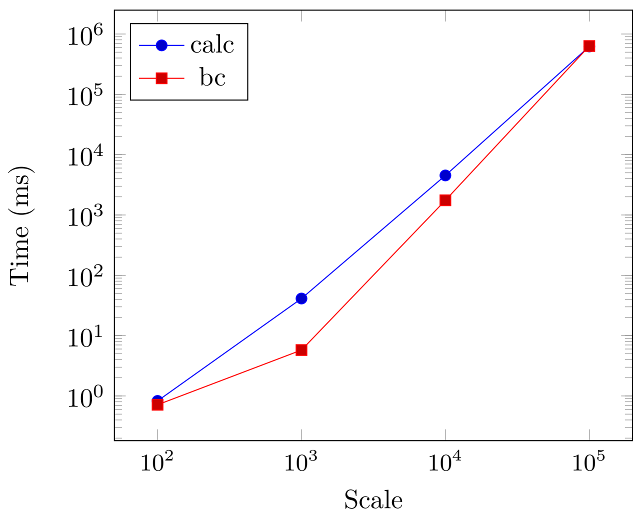
$$x^y = e^{y \ln x} \quad (x > 0,\ y \in \mathbb{R})$$

π

根据 John Machin's formula [5]，有

$$\frac{\pi}{4} = 4 \tan^{-1} \frac{1}{5} - \tan^{-1} \frac{1}{239}$$

用这个公式计算 π 即可。与 bc 计算 π 的速度的比较如下 [6]。相信到了 10^6 应该会反超的（可惜有点跑不动了）。



标准正态分布的累积分布函数

本来呢，是没想写这个函数的。但是在我弄完 Project 之后，在写概统作业的时候，我发现计算标准正态分布的累积分布函数需要查表，上网查感觉很不方便的样子，于是就搜了一下它的泰勒展开（虽然它没有解析表达式）[7]，顺手写了一个，~~还蛮好用的~~。

$$\Phi(x) = \frac{1}{2} + \frac{x}{\sqrt{2\pi}} \sum_{k=0}^{\infty} \frac{(-x^2)^k}{2^k\, k!\,(2k+1)}$$

2.3 变量

对于定义变量和函数部分，我们显然需要一个 key-value map 来储存这些变量。

为了方便处理，我选择新建一个 `Context` 类，用来存储包括但不限于变量之类的上下文信息。

```
class Variable { ... };
class Function { ... };
class BuiltinFunction { ... };
class LazyVariable { ... };

class Entry {
    std::variant<Variable, Function, BuiltinFunction, LazyVariable> content_;
    std::string name_;

    BigDecimal get_variable(Context &context, TokenRange caller) const;
    BigDecimal invoke_function(const std::vector<Expression*> &arguments,
```

[5] `https://en.wikipedia.org/wiki/John_Machin#Formula`
[6] 在 MacBook Air, M1 chip, 16G memory 机器上，重复 8 次取平均 user 时间；bc 使用 `16*a(1/5) - 4*a(1/239)` 进行计算
[7] `https://en.wikipedia.org/wiki/Normal_distribution#Cumulative_distribution_functions`

```
                                      Context &context, TokenRange caller) const;
};

class Context {
    std::unordered_map<std::string, Entry> map_{};
    size_t scale_ = 20;
    int64_t depth_ = 0;  // 递归深度，下面的错误处理会用到
    bool disabled_divergent_check_ = false;

    const Entry& get(const std::string &key, TokenRange caller) const;
    void insert(std::string key, Entry value);
    bool remove(const std::string &key);  // return true if success, false if no such key
    void print(std::ostream &stream) const;
};
```

并且“变量”有很多种，这里分成了 4 种：

1. `Variable`：就是普通的变量，里面储存一个 `BigDecimal`。
2. `Function`：函数，里面储存着形式参数的名字，以及函数体（`Expression`）。

   因为 `Context` 可以被 clone（有多个实例），所以 `Expression` 要用 `std::shared_ptr` 存储。
3. `BuiltinFunction`：builtin 函数，例如上面的 sin、cos 之类的函数。

   `BuiltinFunction` 里面存储参数的个数（用来检查），和函数体。这个函数体采用 `std::function` 类型，这样它就可以支持 Lambda 函数，就会比较方便实现。
4. `LazyVariable`：“懒”变量。例如像 π 和 e 的变量，虽然计算器能实现，但是用户不一定要用。如果每次计算器启动的时候，都计算一遍，就会有点耗费计算资源（毕竟当 `scale` 非常大的时候，确实会很慢）。所以就引入了 `LazyVariable`，只有当用到的时候才进行计算，然后把值存起来，下次再用到的时候就不用再计算一次了。

因为实际使用只分为普通变量和函数两种，所以如果把 4 种类型都暴露给调用方，就多少都有点繁琐。于是就弄了一个 `Entry` 类型，用来稍微封装一下他们的调用，并且做一些错误检查。然后 `Entry` 里面就用 `std::variant` 来储存 4 种类型。

然后在 `Context` 中，就加 `insert` 和 `remove` 两个接口，来新增或删除变量。同时加一个 `print`，支持用户输入 `env` 的时候可以打印所有变量。

除此之外，`Context` 还储存了一些额外的信息，如 `depth_` 和 `disabled_divergent_check_`，这两者会在下面的错误处理中用到。

2.4 错误处理

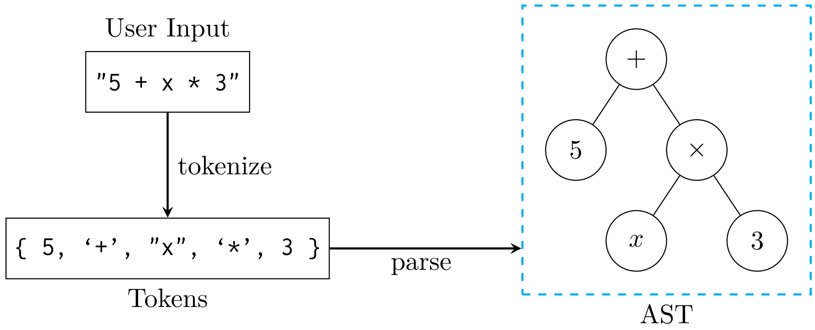
2.4.1 错误定位

首先呢，在实现错误处理之前，我设想对于输入的错误检测和显示可以做成这个样子：

```
> f[x] = 3 * PI * x / 2
> g[x] = f[x] + x
> g[1]
Input #0:
  f[x] = 3 * PI * x / 2
             ~~
Error: no such variable or function: PI
```

这种错误消息最大的一个特点是，它可以精确地标出错误的地方。例如上面变量 `PI` 找不到，它就把输入中的 `PI` 特别标注了出来，这样用户就可以一眼知道是哪里炸了。

至于如何实现它，我们先来回顾前面讲到的解析流程，

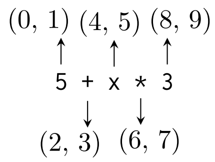


理论上在 tokenize 这一步，我们就已经把输入中的位置信息丢弃了，只剩下有语义的 token。为了能继续携带这些位置信息，我们在每个 token 中多储存一个 TokenRange，表示该 token 对应的是原输入中的哪段位置。

```
struct TokenRange {
    size_t begin_, end_;
    size_t frame_id_;
}
```

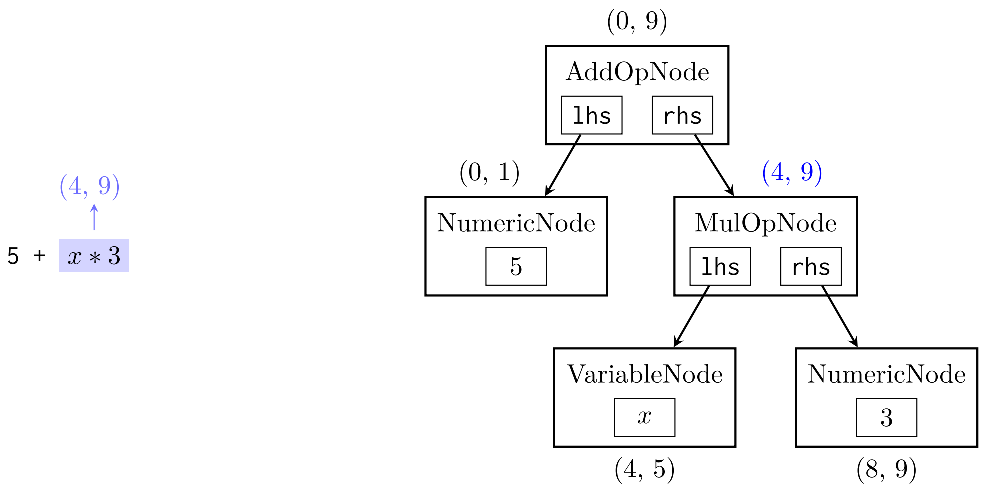
```
class ranged_error : public std::exception {
    TokenRange range_;
    std::string message_;
};
```

还是以 `5 + x * 3`（中间有一些空格）为例：



这样，只要解析的时候，如果发现哪个 token 出错了，就把这个 token 的 TokenRange 放在异常中，一起抛出。然后调用者捕获到错误之后，就可以很快地定位到错误的位置，并输出错误信息。

同样地，我们也可以在 Expression 节点中储存这一 TokenRange 信息，表示这个节点对应的子树所对应的输入的位置。



这样，即使在计算中，如果发现哪个 Expression 节点出现错误（比如变量找不到了），也可以携带 TokenRange 信息抛出异常。

对于跨行错误（例如上面例子中展示的那样），我们可以给每行输入安排一个序号，这个序号也储存在 TokenRange 里面（即上面的 frame_id_）。同时也把每一行输入都保存下来。当错误抛出的时候，如果错误

是属于刚刚输入的那一行，那就直接标出来。如果是属于之前的输入的话，那就从保存的输入中找到相应的输入，然后把它与错误信息一起输出即可。

2.4.2 其它错误检测

因为这个计算器功能有点过于强大，导致它很容易被完坏，所以就需要加一些奇奇怪怪的错误检测。

1. **递归深度检测**

   如果有小天才输入 `f[x] = f[x - 1]`，然后运行 `f[1]` 的话，由于递归没有终止条件，所以很容易就会 segmentation fault。想要避免这个丑陋的错误发生，我们可以在调用函数的时候，检查一下当前的递归深度，如果超过一定的层数（比如 5000 层）就报错。

   同时也许要考虑到，用户有可能确实会有这种需求。所以这里也加一个选项，可以禁用掉上面这个检查。如果用户的确需要递归 5000 层以上，就可以用 `-no_depth_check` 把这个检查禁用掉。

   测试例子如下：

   ```
   > f[x] = f[x - 1]
   > f[1]
   Input #0:
     f[x] = f[x - 1]
            ~~~~~~~~
   Error: Your recursion depth is exceeded 5000
   If you are sure what to do, you disable the check with "--no_depth_check"
   >
   ```

2. **发散检测**

   在我开发测试的过程中，经常会遇到写一个牛顿迭代或者泰勒展开的时候，一不小心把里面的某一项写错了，导致函数无法收敛，甚至会发散，越来越大。这种情况下，如果不加控制，程序基本无法正常停止，并且这个越来越大的数也会占用越来越大的内存。如果只是因为这点手误就引起程序崩溃，那么用起来会很难受（特别是如果强行停止，前面设置的所有变量和函数都要重新再设置一遍）。

   所以我们可以加入一个发散检测，当检测到运算中某一个数字非常大，它的位数超过了一定程度（比如 100000 位）之后，就报错。当然了，和上一个检测一样，如果用户真的有这个需求，也可以通过选项禁用该检查。

   测试例子如下：

   ```
   > f[x] = f[x * x]
   > f[2]
   Input #0:
     f[x] = f[x * x]
              ~~~~~
   Error: One decimal is exceeded 500000 digits, maybe is divergent
   If you are sure what to do, you can disable it with "--no_divergent_check"
   >
   ```

通过这两个小小的检测，就可以避免大部分因手误而引起的程序运行错误，还是蛮不错的。

3 结果与验证

3.1 源码

源代码会在 ddl 截止之后放到 `https://github.com/YanWQ-monad/SUSTech_CS205_Projects` 的第二次 Project 文件夹下。由于这个 repository 是镜像性质的，所以没有很详细的 commit 记录。

目录结构如下：

```
/
├── CMakeLists.txt
├── src
│   ├── CMakeLists.txt
│   ├── constant.h    constant.cpp    定义一些常数
│   ├── context.h     context.cpp     变量储存
│   ├── error.h                       自定义异常
│   ├── eval.h        eval.cpp        一些数学函数和数学方法（如牛顿法）
│   ├── main.cpp                      主程序入口点
│   ├── node.h        node.cpp        AST 节点
│   ├── number.h      number.cpp      高精度数字
│   ├── parse.h       parse.cpp       解析
│   └── token.h       token.cpp       tokenize
└── test
    ├── CMakeLists.txt
    ├── number_test.cpp   （等）测试文件
    └── test.hpp
```

然后按照传统，加上 -Wall -Wextra 编译（除此之外，实际上还加了很多），本代码不会产生任何编译警告。

代码风格遵循 Google C++ Style Guides，可以通过 cpplint.py 检验（关闭了 -build/include_subdir，-build/header_guard，-legal/copyright 三个警告类型，因为我认为这里 include 不需要写上 src 文件夹，header guard 也不需要从系统根目录开始算）。

3.2 测试

基本的运行测试已经在上面的"功能展示"中展示过了，这里就不重复了。

本项目继续沿用 Google Test 进行单元测试，非常地方便。中途有几次重构了一些核心部份的计算逻辑，然后想看看自己改得对不对，直接跑一下单元测试，看到全绿了就很开心。

这里贴出一部分测试，更多测试可以翻阅源代码。

```
TEST(DecimalTest, AddTest) {
    EXPECT_EQ(BigDecimal("123") + BigDecimal("456"), BigDecimal("579"));
    EXPECT_EQ(BigDecimal("123.456") + BigDecimal("0.00001"), BigDecimal("123.45601"));
    EXPECT_EQ(BigDecimal("-0.1") + BigDecimal("0.1"), BigDecimal("0"));
    EXPECT_EQ(BigDecimal("943047228") + BigDecimal("-373.97859"), BigDecimal("943046854.02141"));
}

TEST(DecimalTest, DivideWithIndivisibleTest) {
    EXPECT_EQ(BigDecimal("2").div_with_scale(BigDecimal("3"), 5), BigDecimal("0.66667"));
    EXPECT_EQ(BigDecimal("4").div_with_scale(BigDecimal("3"), 5), BigDecimal("1.33333"));
    EXPECT_EQ(BigDecimal("2").div_with_scale(BigDecimal("7"), 10), BigDecimal("0.2857142857"));
}

TEST(ParsingTest, InvalidTest) {
    EXPECT_THROW(parse("1+", 0), ranged_error);
    EXPECT_THROW(parse("f[3][3]", 0), ranged_error);
    EXPECT_THROW(parse("1+1[3]", 0), ranged_error);
    EXPECT_THROW(parse("1+(1", 0), ranged_error);
}
```

3.3 还能这么干？

俗话说的好，一个工具写得好不好，不在于这个工具实现了什么，而在于这个工具能创造什么。在这个部分，我只会使用 **if 函数**（内建），以及 **5 个基本运算**（加减乘除模），来看看我们能创造什么好玩的东西。

在开始之前，请允许我先介绍一下 if 函数。if[cond, a, b] 函数接受三个参数，都是表达式类型。如果 cond 条件不为 0（即为真），则返回 a，否则返回 b。这个 if 函数具有延迟执行的特性，即假如 cond 为假，

那么 a 中的内容不会被执行，反之亦然。

然后为什么要这么突兀的加一个 if 呢，因为在一般的传统编程语言中，一般有三种执行流程：顺序、选择、循环。在这个计算器中，顺序很简单，循环可以用函数实现（下面会提到），还剩下一个选择确实没办法，只能靠计算器本身实现。把“选择”补充完成之后，我们就有了顺序、选择、循环三板斧，就可以为所欲为了。

并且由于这部分的某些运算的运算量确实很大，做高级运算时也会调用很多次这些基本运算，相当于某种压力测试，所以我们也可以通过观察它的运行和输出是否正确，来判断本 Project 的基本功能是否正常工作。当时写的时候重构过一些核心部份，想知道程序运行是否正确，直接跑一下 pi，然后看一下输出对不对，对了就稳了（毕竟整个 π 的运算涉及到 $\tan^{-1}$、除法、乘法、加减法、四舍五入等等的运算，可谓是牵一发而动全身了）。虽然 pi 属于内建变量，但是对于自己写的脚本也差不多，甚至能测到更多方面。

用这种方式测试，相比枯燥地造数据，用这种方法不但能很方便地进行测试，而且过程也非常有意思，~~刚写完的时候还自己玩了一个晚上~~。

工具函数

由于下面的例子中，会用到一些工具函数。所以先在这里进行介绍。

- **向下取整**：`floor[x] = x - x % 1`。根据上面定义的推广取模，`x % 1` 实际上就是取 x 的小数部分，然后把它减去就可以砍掉小数部分。
- **绝对值**：`abs[x] = if[ x < 0, 0 - x, x ]`
- **四舍五入**到指定精度：`round[x, p] = x - x % p + if[x % p < p/2, 0, p]`（精度为 p）。其实现思路与向下取整相似，除此之外，还需要判断指定精度的下一位是否大于 5，如果是的话还需要进位。

斐波那契数列

按照数学中斐波那契数列的定义

$$f(n) = \begin{cases} f(n-1) + f(n-2) & ,n > 2 \\ 1 & ,n = 1, 2 \end{cases}$$

那么我们就可以用 if 函数复刻这个数列：`f[n] = if[n < 3, 1, f[n - 1] + f[n - 2]]`。

运行如下：

```
> f[n] = if[n < 3, 1, f[n - 1] + f[n - 2]]
> f[18]
2584
> f[30]
832040
```

对照标准斐波那契数列的值，可以验算是正确的。

当然，标准递归的斐波那契数列计算速度有点慢，我们可以把它改成“迭代”的。斐波那契数列的迭代伪代码如下：

function FIBONACCI(N)
  $a, b \leftarrow 0, 1$ ▷ f(0) 和 f(1)
  for $i \leftarrow 2$ to N **do**
    $a, b \leftarrow b, a + b$
  end for
  return b
end function

由于循环变量 i 没有用到，所以我们让它从 N 到 2 也是可以的。所以我们就有了第二代代码（这里用 n 表示循环变量，a, b 的意义和上面一样）：

```
> g[a, b, n] = if[ n < 2, b, g[b, a + b, n - 1] ]
> f[n] = g[0, 1, n]
> f[200]
280571172992510140037611932413038677189525
```

最大公约数

最大公约数可以用辗转相除法得出：

$$\gcd(a,\,b) = \begin{cases} \gcd(b,\,a \bmod b) & ,a \bmod b \neq 0 \\ b & ,a \bmod b = 0 \end{cases}$$

则写成脚本也很简单：

```
> gcd[a, b] = if[ a % b, gcd[b, a % b], b ]
> gcd[5, 6]
1
> gcd[24, 90]
6
```

快速幂

当计算 a^b 的时候（其中 b 是整数且不为负），我们可以对 b 进行二进制分解，将幂运算变成 $O(\log n)$ 次乘法运算来达到优化的目的，就是所谓的快速幂。

由于这个技巧在 OI 以及算法课上被广泛使用，这里就不多介绍了，对于更多信息，可以查阅 OI Wiki [8]。

先给出计算脚本：`pow[a, b] = if[ b < 1, 1, x = pow[a, floor[b/2] ]; x * x * if[b % 2, a, 1] ]`

1. `pow[a, b] =` **`if[ b < 1, 1,`** `x = pow[a, floor[b/2] ]; x * x * if[b % 2, a, 1] ]`

   首先，先判断若 $b = 0$，则返回 1，为边界条件。
2. `pow[a, b] = if[ b < 1, 1,` **`x = pow[a, floor[b/2] ]`**`; x * x * if[b % 2, a, 1] ]`

   然后，计算出 $a^{\lfloor \frac{b}{2} \rfloor}$。
3. `pow[a, b] = if[ b < 1, 1, x = pow[a, floor[b/2] ];` **`x * x * if[b % 2, a, 1]`** `]`

   最后，根据 b 的奇偶，返回 x^2a 或 x^2。

执行结果为：（结果太长，在这里就省略掉了）

```
> pow[a, b] = if[ b < 1, 1, x = pow[a, floor[b/2] ]; x * x * if[b % 2, a, 1] ]
> pow[2, 100]
1267650600228229401496703205376
> pow[2, 1000000]
990065622929589825069792361630190325073362424178756733286639611453170948330...
```

其中计算 2^{10^6} 只需要花费不到 1 秒，甚至比 Python 的 `2 ** 1000000` 快。并且经过验证，结果是正确的。

解方程

假设我们有一个方程 $x^3 - 7 = 0$，如果我们要解它，我们可以令 $f(x) = x^3 - 7$，然后用牛顿迭代法解决：

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)} = \frac{2}{3}x + \frac{7}{3x^2}$$

[8] `https://oi-wiki.org/math/binary-exponentiation/`

要在这个计算器完成牛顿迭代法，我们可以从 x_1 开始迭代，然后直到 x_n 与 x_{n+1} 的差值小到一定程度的时候，我们就可以认为它收敛到了一个解。

则写成脚本，就是

```
> EPS = 1e-18
> f[x] = x * x * x - 7
> g[x1, x2] = if[ abs[x1 - x2] < EPS, x1, g[2/3 * x1 + 7 / (3 * x1 * x1), x1] ]
> g[1, 0]
1.912931182772389101206223486815887580614254389214558156505535344465805 0394...
```

其中在 g[x1, x2] 中，x1 表示 x_{n+1}，x2 表示 x_n（即最后）

由于加减法和乘法不受精度的约束，所以位数可能会很长，这里省略掉了一部分。经过验证，该结果的立方确实等于 3。

啊什么，你说你连求导都懒得求？那…好吧，其实也不是不行。回归到导数的定义，我们有

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

只要 Δx 足够小，就有

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

我们就可以用这个方法来计算 $f'(x)$。

计算和求解脚本如下：

```
> EPS = 1e-18
> f[x] = x * x * x - 7
> df[x] = (f[x + EPS] - f[x]) / EPS
> g[x1, x2] = if[ abs[x1 - x2] < EPS, x1, g[x1 - f[x1] / df[x1], x1] ]
> g[1, 0]
1.912931182772389101
```

这时，如果我们想要求另一个方程 $3x^3 - 2x + 7 = 0$，我们就只需要修改函数 f[x] 即可，其它什么都不用动。

```
(接 上一个运行记录)
> f[x] = 3 * x * x * x - 2 * x + 7
> g[1, 0]
-1.493115680093464320 92
```

经过验算，这个结果也是正确的。

数学函数

和内建函数中的数学函数的原理一致（除了 $\sqrt{x}$ 还是在土法炼钢），只不过也可以用纯脚本实现：

```
> EPS = 1e-18
0.000000000000000001
>
> f1[x2, term, k] = if[ abs[term] < EPS, 0, term + f1[x2, term * x2 / (k * (k+1)), k+2 ]]
> sin[x] = f1[0 - x * x, x, 2]
> cos[x] = f1[0 - x * x, 1, 1]
>
> f7[mul, term, k] = if[ abs[term] < EPS, 0, term / (1 + k) + f7[ mul, term * mul, k + 2 ] ]
> arctan_c = f7[0 - 0.04, 0.2, 0]
0.19739555984988075837
> arctan[x] = if[x < 0.2, f7[0 - x * x, x, 0], arctan_c + arctan[ (x - 0.2) / (1 + 0.2 * x) ] ]
>
> f2[term, x, k] = if[ term < EPS, 0, term + f2[ term * x / k, x, k + 1 ] ]
> exp[x] = f2[1, x, 1]
>
> e = exp[1]
2.71828182845904523493
> f4[x, term, k] = if[ abs[term] < EPS, 0, term / k + f4[ x, 0 - round[ term * x, EPS ], k + 1 ] ]
> ln[x] = if[x < 1.5, if[x < 0.5, ln[x * e] - 1, f4[ x-1, x-1, 1 ]], 1 + ln[x / e]]
>
> pow[x, y] = exp[y * ln[x]]
>
> f5[b, x, y] = if[abs[x-y] < EPS, x, f5[b, 0.5 * (x + b / x), x] ]
> sqrt[x] = f5[x, x, 0]
>
> sin[1]
0.84147098480789650664
> cos[1]
0.54030230586813971699
> exp[1.5]
4.48168907033806482216
> ln[3]
1.0986122886681096909
> sqrt[3]
1.7320508075688772935290625
> pow[2, e]
6.58088599101792097452
>
> pi = 16 * arctan[1/5] - 4 * arctan[1/239]
3.14159265358979323848
```

经过验算，这些结果都是正确的。当然这也可以从侧面证明，这个计算器的基础功能是可以正常工作的。

可见，该计算器的功能十分强大，仅仅凭借 `if` 函数和五则基本运算，就可以玩出如此多的花样。

4 总结

总体来说，这个 Project 的感觉与上个 Project 差不多——基本需求比较简单，但是可以自由发挥的空间非常大。所以我就按照我直觉走，

- 对于**解析**写了一个相对完善的引擎，可以处理比较复杂的输入，也能很好地判错。
- 对于如何执行**计算**，我选择在解析的时候，将表达式解析成 AST 树，虽然代码量看起来的增加了，但是带来的好处有，模块化使代码更好维护了、解析成 AST 树可以更舒服地存储和调用了、也做到了表达式与数据（变量）分离。
- 然后对于**变量和函数**部分，因为只有变量确实很难玩出花样（我可不希望它仅仅是个玩具），所以就多弄了一个可以定义函数。弄完函数之后发现无意中支持了函数中调用其它函数，但是递归执行缺少终止条件，于是就加了一个 **`if` 函数**。

然后就发现凭借着上面一些非常基础的功能，居然能用脚本实现算斐波那契数列、开方、解方程、计算 sin cos 等高级运算，玩了几乎一个晚上，也属于是意外之喜了。

前面说到，我不希望它仅仅是一个玩具，实际上，在我写完这个 Project 之后在做概统作业的时候，我就直接用这个计算器算 $\Phi(x)$（标准正态分布的累积分布函数）以及解 $\Phi(x) = c$ 的方程，确实给我带来了实质性的帮助。个人还是觉得非常地不错和好玩的。

这次 Project 能整出完成度比较高的计算器确实有点出乎我的意料，如果我之后有时间的话，我还会想继续维护这个计算器（~~如果有人用的话~~）。如果老师或同学们对这个 Project 还有什么新的想法，也欢迎提出（~~不保证有时间实现~~）。

最后按照惯例，这个项目也会放到 GitHub 开源，会放在 `https://github.com/YanWQ-monad/SUSTech_CS205_Projects` 仓库的 Project2 的子目录下。欢迎大家 star，谢谢！

References

[1] Morris. 關於高效大數除法的那些事. `https://morris821028.github.io/2017/04/09/big-integer-division/`, 2017.

[2] GNU. bc builtin script. `https://ftp.gnu.org/gnu/bc/bc-1.07.1.tar.gz`, path: `/bc/libmath.b`, 2017.