计算器（Calculator）实验指南

>

0. 前置知识

如果你不了解 C 语言中的 `struct, enum, union, typedef` 的用法，请你首先查阅课本或其他资料再来阅读本实验指南。

如果你的目标为 100 分，建议不要急于开始，首先阅读实验指南的全文以进行合理的顶层设计，这可能比在原来的基础上加东西更加节省时间。

1. 总览

在读入了每一行语句之后，我们可能需要依次进行以下步骤。

1. 词法分析
2. 语法分析 & 表达式求值
3. 变量赋值

大家会在《编译原理》课程中学会真正的词法分析和语法分析。这里，我们大致描述一下这些步骤分别都是在做什么。

1.1 词法分析

当我们得到一个表达式，

```
1 + ( 33 + 2 ) * 4 - 5
```

我们要做的第一步就是将这个输入的字符串转化为计算机内部的数据表示，具体来说，我们需要按照分隔符（一个空格 `' '`）将每个字符串分段，并求出每一段（称为 token）的类型。

比如，上面的表达式可以分成以下几个 token 。

```
number    "1"
operator  "+"
operator  "("
number    "33"
...
```

你可以阅读 “语言规约” 部分来了解需要支持的不同 token 类型。

在词法分析结束之后，你可以将所有的 token 储存在一个连续的数组中，你可以使用一个 `tokens[]` 数组来连续储存所有 token 。

```
typedef struct token {
  int type;
  char str[32];
} Token;

Token tokens[...];
```

1.2 语法分析 & 表达式求值

1.2.0 概述

这里的语法分析主要是分析表达式逻辑的正确性，比如 `1 + + 3` 和 `1 ( 9 )` 就不是逻辑上正确的表达式，而这个步骤通常可以和表达式求值一起完成。

当词法分析结束后，你应该得到所有的 token 的值，举个例子，

```
4 + 3 * ( 2 - 1 )
```

的 token 表达式为

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| NUM | '+' | NUM | '*' | '(' | NUM | '-' | NUM | ')' |
| "4" |     | "3" |     |     | "2" |     | "1" |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

1.2.1 递归求值

根据表达式的归纳定义特性，我们可以很方便地使用递归来进行求值。

首先我们给出算数表达式的归纳定义：

```
<expr> ::=
  <integer>           # 一个数是表达式
| <variable>          # 一个变量也是表达式
| "(" <expr> ")"      # 在表达式两边加个括号也是表达式
| <expr> "+" <expr>   # 两个表达式相加也是表达式
| <expr> "-" <expr>   # 接下来你全懂了
| <expr> "*" <expr>
| <expr> "/" <expr>
```

上面这种表示方法就是大名鼎鼎的 BNF ，任何一本正规的程序设计语言教程都会使用 BNF 来给出这种程序设计语言的语法.

为了在 token 表达式中指示一个子表达式，我们可以使用两个整数 `l` 和 `r` 来指示这个子表达式的开始位置和结束位置。这样我们就可以很容易把求值函数的框架写出来了：

```
eval(l, r) {
  if (l > r) {
    /* Wrong expression */
  }
```

```
  else if (l == r) {
    /* Single token.
     * For now this token should be a number or variable.
     * Return the value of the number or variable.
     */
  }
  else if (check_parentheses(l, r) == true) {
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(l + 1, r - 1);
  }
  else {
    /* We should do more things here. */
  }
}
```

其中 `check_parentheses()` 函数用于判断

1. 表达式是否被一对匹配的括号包围着
2. 表达式的左右括号是否匹配

如果不匹配， 这个表达式肯定是不符合语法的，也就不需要继续进行求值了。

我们举一些例子来说明 `check_parentheses()` 函数的功能：

```
"( 2 - 1 )"              // true
"( 4 + 3 * ( 2 - 1 ) )"  // true
"4 + 3 * ( 2 - 1 )"      // false, the whole expression is not surrounded by a matched
// pair of parentheses
"( 4 + 3 ) ) * ( ( 2 - 1 )" // false, bad expression
"( 4 + 3 ) * ( 2 - 1 )"  // false, the leftmost '(' and the rightmost ')' are not matched
```

上面的框架已经考虑了 BNF 中算术表达式的开头三种定义，接下来我们来考虑剩下的情况(即上述伪代码中最后一个 `else` 中的内容)。 一个问题是，给出一个最左边和最右边不同时是括号的长表达式，我们要怎么正确地将它分裂成两个子表达式?

我们定义"主运算符"为表达式人工求值时，最后一步进行运行的运算符，它指示了表达式的类型(例如当一个表达式的最后一步是减法运算时，它本质上是一个减法表达式).

要正确地对一个长表达式进行分裂，就是要找到它的主运算符。我们继续使用上面的例子来探讨这个问题:

```
"4 + 3 * ( 2 - 1 )"
/*********************/
case 1:
      "+"
   /      \
"4"     "3 * ( 2 - 1 )"

case 2:
```

```
        "*"
      /     \
"4 + 3"     "( 2 - 1 )"

case 3:
             "-"
           /     \
"4 + 3 * ( 2"     "1 )"
```

上面列出了 3 种可能的分裂，注意到我们不可能在非运算符的 token 处进行分裂，否则分裂得到的结果均不是合法的表达式。

根据主运算符的定义，我们很容易发现，只有第一种分裂才是正确的。这其实也符合我们人工求值的过程：先算 `4` 和 `3 * ( 2 - 1 )`，最后把它们的结果相加。第二种分裂违反了算术运算的优先级，它会导致加法比乘法更早进行。第三种分裂破坏了括号的平衡，分裂得到的结果均不是合法的表达式。

通过上面这个简单的例子，我们就可以总结出如何在一个 token 表达式中寻找主运算符了：

1. 非运算符的 token 不是主运算符.
2. 出现在一对括号中的 token 不是主运算符。注意到这里不会出现有括号包围整个表达式的情况，因为这种情况已经在 `check_parentheses()` 相应的 if 块中被处理了。
3. 主运算符的优先级在表达式中是最低的。这是因为主运算符是最后一步才进行的运算符。

当有多个运算符的优先级都是最低时，根据结合性，最后被结合的运算符才是主运算符。一个例子是 `1 + 2 + 3`，它的主运算符应该是右边的 `+`。

要找出主运算符，只需要将 `tokens` 全部扫描一遍，就可以按照上述方法唯一确定主运算符。

找到了正确的主运算符之后，事情就变得很简单了：先对分裂出来的两个子表达式进行递归求值，然后再根据主运算符的类型对两个子表达式的值进行运算即可。于是完整的求值函数如下：

```
eval(l, r) {
  if (l > r) {
    /* Wrong expression */
  }
  else if (l == r) {
    /* Single token.
     * For now this token should be a number or variable.
     * Return the value of the number or variable.
     */
  }
  else if (check_parentheses(l, r) == 1) {
     /* The expression is surrounded by a matched pair of parentheses.
      * If that is the case, just throw away the parentheses.
      */
    return eval(l + 1, r - 1);
  }
  else {
    op = the position of 主运算符 in the token expression;
    val1 = eval(l, op - 1);
    val2 = eval(op + 1, r);

    switch (op_type) {
```

```
        case '+': return val1 + val2;
        case '-': /* ... */
        case '*': /* ... */
        case '/': /* ... */
        default : assert(0);
      }
    }
}
```

1.3 变量赋值

我们称变量赋值语句为 `<assignment>` ，那么

```
<assignment> ::=
  <variable> "=" <expr>
```

首先，我们需要检查 `<variable>` 即变量名的合法性，如果不合法需要报错。

然后我们可以用一个结构体，记录下变量和对应的值，从而来记录现在有的所有的赋值，如（仅作建议，不强制）

```
typedef struct assignment {
  char name[32];
  int val;
} Assignment;
```

那么你就可以开一个 `Assignment` 类型的数组，把所有的赋值关系存下来。

当你在 `eval` 中遇到需要查询一个变量的值的时候，就只需要在这个数组里找到符合的，然后返回对应的值。

你也可以尝试使用不同的数据结构（比如 哈希表）来提高查询的效率，但是在本项目中，数组（线性表）已经足够。

2. 选做部分

2.1 Bonus #1 实现带有负数的算术表达式的求值

在上述实现中，我们并没有考虑负数的问题，例如

```
"1 + - 1"
"- - 1"     /* 我们不实现自减运算，这里应该解释成 -(-1) = 1 */
```

它们会被判定为不合法的表达式。为了实现负数的功能，你需要考虑两个问题：

1. 负号和减号都是 `-` ， 如何区分它们?
2. 负号是个单目运算符，分裂的时候需要注意什么?

2.2 Bonus #2 浮点数支持

如果要支持浮点数，你无法简单地用一个 `int eval(int l, int r)` 来进行求值，因为你不知道当前是整数还是浮点数，那么你可能需要(仅作建议，不强制)

```
typedef struct value {
  union {
    double a;
    int b;
  } val;
  enum NumberType {
    DOUBLE, INT
  } type;
} Value;

Value eval(int l, int r);
```

2.3 Bonus #3 连续变量赋值

我们可以发现上边的赋值语句并不支持形如 `a = b = c = 233` 的赋值语句，我们称这种赋值语句为连续赋值语句。

那么，

```
<assignment> ::=
  <variable> "=" <expr>
| <variable> "=" <assignment>
```

实现连续变量赋值的方法有很多，相信你能自己实现捏。

3. 语言规约

3.1 字符集

定义本项目的所要支持的表达式求值语言定义在以下字符集上，也就是说，除去用于区分 TOKEN 之间的空格 `' '(ASCII = 32)` ，在输入中仅可能出现以下字符。

$$\Sigma = \{=, +, -, *, /, (,), \_, [0-9], [a-z], [A-Z]\}$$

如果你选做了 Bonus #2 浮点数支持，那么 $\Sigma' = \Sigma \cup \{.\}$ 。

3.2 token

本项目中会出现的所有 token 类型和其合法性定义如下。

1. 变量 (variable)：由字母、数字、下划线组成，但不能以数字开头。
2. 整数 (integer) ：由数字组成，不能含有前导 0 (比如 `002`) 。
3. 运算符 (operator) ： `+, -, *, /, (, ), =` 。
4. * 浮点数 (float) ：由数字、小数点 `'.'` 和数字连接而成，其中前后的数字必须出现且合法，例如 `3.2` 和 `0.03` 都是合法的浮点数，而 `00.1` 和 `.233333` 和 `22.` 均不是合法的浮点数。**请注意这个定义，这和 C 语言中不同。**

上面的定义已经保证，一个字符串最多可以被解释为一种合法的 token 。

3.3 语法和计算规则

我们将 <integer> 和 <float> 统称为 <number> ，重新给出表达式和赋值语句的形式化描述。

```
<expr> ::=
  <number>
| <variable>
| "(" <expr> ")"
| <expr> "+" <expr>
| <expr> "-" <expr>
| <expr> "*" <expr>
| <expr> "/" <expr>
```

在这里，整数和整数、浮点数和浮点数之间的 +-*/ 操作都和 C 语言中的定义相同，即整数除法默认向下取整，而整数与浮点数之间的运算将会通过“类型提升”规则，首先将该整数“提升”为等值的浮点数，然后再作为两个浮点数之间的运算处理。

如果你选做了 Bonus #1 ，那么需要加一条规则。

```
<expr> ::= "-" <expr>
```

请注意：这只是文法的展开方式的定义，而不能作为运算优先级的顺序定义。

举个例子，加入了上述这条规则后 - 3 + 5 这个表达式可以通过以下两种方式展开得到，后者才是我们期望的结果。

```
- (3 + 5) = -8
(-3) + 5 = 2
```

关于运算的优先级问题，与 C 语言以及大家的常识完全相同， +- 的优先级相同，且低于 */ ，而括号 () 可以提升它包裹的表达式的优先级。

```
<assignment> ::=
  <variable> "=" <expr>
```

如果你选做了 Bonus #3 ，那么

```
<assignment> ::=
  <variable> "=" <expr>
| <variable> "=" <assignment>
```

在本项目中，我们用变量名作为区分不同变量的唯一标准。

赋值操作的语义是：若 <expr> 为合法的表达式，则将（所有）涉及到的变量的值覆盖为新值，如果你选做了 Bonus #2，那么变量的类型可能会发生改变。

4. 输入输出、错误处理和数据范围

4.1 输入

输入包含若干行字符串，每一行中只包含字符集中的字符以及空格 `' '` 。保证不同的字符集中的字母组成的字符串（$\sum^*$）之间被恰好一个空格隔开。保证每一行都以 `\n` 结尾。

4.2 输出

对于每一行输入，首先需要检查是否合法，若不合法，则输出一行 `Error` ；否则：

- 对于表达式求值语句，输出表达式求值的值，若结果为浮点数，保留 6 位小数。
- 对于赋值语句，输出赋的值（即 `<expr>` 的计算结果）。

你可以会疑惑，对于赋值语句为什么要输出计算结果，不妨尝试运行以下代码片段。

```
printf("%d\n", a = b = c = 10);
```

你一定会看到输出。事实上这就是赋值语句的返回值，而连续赋值也依赖于这个返回值（ `a = (b = (c = 10))` ）。

4.3 错误处理

在本项目中，你只需要处理以下两种错误。

4.3.1 词法错误

若有 token 在解析过程中无法与 3.2 节的任何一种类型匹配，则认为发生了词法错误。

比如 `123_abc` 和 `*1` 就会触发词法错误。

词法错误的分析应该在词法分析阶段完成。

4.3.2 语法错误

在无词法错误的情况下，若某个字符串无法与 BNF 范式的任何一种展开相匹配，则认为发生语法错误。

比如 `1 + + 2` 和 `1 ( 38 )` 和 `a = b + c = 10` 就会触发语法错误。

需要强调的是，在表达式求值的过程中调用了当前未定义的变量也被认为是语法错误。

语法错误的分析应该在语法分析 & 表达式求值阶段完成，且给出的 `eval()` 函数中已经包含足量的提示，仅借助该函数的返回值已经足够处理语法错误的所有情况。

4.4 数据范围

保证每行的长度 $<$ 1024，保证每个输入 token 的长度（不论合不合法）都不超过 30 ，保证最多使用不超过 128 个变量。

保证所有的运算即中间结果都不会超过 `int` 或是 `double` 的范围。

尽管上面的数据范围看上去很大，但是实际测试数据远远没有这么大，为了让大家多得点分，必做部分的测试数据甚至可以称得上有点水。事实上为了保证测试数据的正确性，实际数据都是手造的。大家在实现时应该不需要过多关注效率。

5. 一个可行的设计和实现（基于所有 Bonus）*

> 这一部分为助教在重构 100 分标程时的一些思路，不做强制要求，事实上上面的提示已经足够，如果你想要挑战自己，那么可以直接略过这一部分。
>
> 我们不鼓励大家写差的代码，如果你思考较长时间仍然没有什么头猪，那么可以参考我的设计。
>
> 这个设计不一定好，它可能还有很多的可改进之处，与大家共同交流探讨。
>
> — czh

5.1 数据类型

本项目的总体思路很简单，与大家常见的 OJ 题目类似，无非就是输入 → 处理 → 输出结果，而处理的过程已经在“总览”中基本给出。

事实上，由于任何输入都有对应着一个输出，那么我们其实可以将三种不同的类型的值封装起来。

```
typedef struct value {
  enum {
    INT,
    FLOAT,
    ERROR
  } type;
  union {
    int iVal;
    double fVal;
  } val;
} Value;
```

这样，我们在使用 `eval()` 函数的过程中就可以直接返回一个 `type == ERROR` 的 `Value`，比如

```
Value eval(l, r) {
  if (l > r) return error; // error.type == ERROR
  ......
}
```

我们也可以封装一个输出函数 `printValue()`，这样我们就可以方便地用 `printValue(eval(...))` 来统一处理输出。

```
void printValue(Value v) {
  switch (v.type) {
    case INT: ...
    case FLOAT: ...
    ...
  }
}
```

5.2 文法

看上去我们需要处理两种语句，事实上两种语句的处理过程非常类似，我们可以稍稍改写 `<assignment>` 的展开式。

```
<assignment> ::=
  <expr>
| <variable> "=" <assignment>
```

这个文法和原来的文法几乎等价，但它支持将所有的表达式语句视作一个 `<assignment>` 来统一处理（视为对 0 个变量赋值）。

我们可以设计一个函数 `evalAssign(l, r)` 来统一处理两种语句。

```
Value evalAssign(int l, int r) {
  if (there exists "=") {
    Variable var = ...
    Value val = evalAssign(...)
    load(var, val)
  } else {
    return eval(...)
  }
}
```

如果你看懂了上面的文法，那么也应该能读懂 `evalAssign()` 的伪代码。请一定要理解之后再动手，切忌生搬硬套。

事实上，这个 `evalAssign()` 函数的设计已经可以处理连续赋值的情况。

5.3 再设计 `eval` 函数

为了支持负号（minus），我们需要稍微修改一下 `eval(l, r)` 这个函数。

我们首先需要思考负号的优先级问题。给出如下例子，请同学们自己找出以下运算中的”主运算符“，思考这些运算递归的顺序。

```
- 1 + 2
2 + - 1
( 1 + 2 ) - - ( 3 * 4 )
```

我们发现，负号（minus）的优先级非常非常低，和表达式最外层包裹的配对括号类似，只有当找不到主运算符的时候，我们才会考虑”解引用“这个负号。

类似的，我们可以修改一下 `eval()` 函数。上面的思考也应该对你寻找“主运算符”的过程有所启发。

```
Value eval(l, r) {
  if (l > r) {
    /* Wrong expression */
  } else if (l == r) {
    /* Single token.
     * For now this token should be a number or variable.
     * Return the value of the number or variable.
     */
  } else if (check_parentheses(l, r) == true) {
```

```
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(l + 1, r - 1);
  } else if (check_minus(l, r) == true) {
    /* do something here. */
  } else {
    /* We should do more things here. */
  }
}
```

5.4 为不同 `type` 的数据类型设计统一的处理函数

在 `eval` 中，很容易涉及到两个不同变量的 `Value` 在某一个二元运算符下“合流”的结果，我们可以设计统一的“合流”函数 `meetValue()` 来处理这个过程（这个函数的原型来自于《软件分析》的课程作业）。

考虑两个不同 `Value` 的类型相交的情况，共有 $3 \times 3 = 9$ 种，你可能需要绘制一张 3×3 的表格来枚举并整理所有相交的情况，你可以在 `meetValue()` 中处理类型提升的过程。

```
Value meetValue(Value v1, Value v2, Op op) {
  if (v1.type == ERROR || v2.type == ERROR) {
    return ...
  }
  // 现在 v1 和 v2 都不为 ERROR 了
  if (v1.type != v2.type) {
    // 类型提升
  }
  switch (op) {
    case '+': ...;
    ...
  }
}
```

如果你这样实现的话， `eval` 函数的最后一部分也就非常简单了。

```
Value eval(l, r) {
  ...
  else {
    Value v1 = ?, v2 = ?;
    Op op = ?;
    return meetValue(v1, v2, op);
  }
}
```