EJack Calstack

可编程函数式高精度计算器

技术报告

狄志鹏 林隆中 王浩任* | C 程序设计专题 | 2019 年 3 月

*按姓氏首字母排序

操作说明与参考文档

本计算器一切符号均使用英文符号

变量的定义与使用

变量定义：let varName = exp　例如：let x = 1
注意：等号左右可以有空格，let 后至少有一个空格，varName 必须为全字母，变量名不能超过 16 个字母，最多不能定义超过 128 个变量。exp 可以为任意合法表达式（返回值为 Numeric 或者 Closure）

变量使用：varName　（返回 varName 对应的值）　例如：x　（返回 1 ）

函数（闭包）的定义与调用

1. 匿名函数：$varName{ exp }　例：$x{x+2}（输入 x，返回 x+2）
   VarName 是形参，exp 的结果为返回值，不要在 varName 附近加空格。
2. 使用 let：　let f = $varName{ exp }　例：let f = $x{x+2}
   注意事项同上
3. 匿名函数调用：$varName{ exp1 }[exp2]　例：$x{x+2}[10]（返回 12）
   可以嵌套
4. 有名字的函数调用：
   f[exp1][exp2][exp3]……[expn]　例：f[10] (其中 f = $x{x+2})　（返回 12）
   　　f[f[f[f[2]]]] = 10
   可以嵌套
5. let [varName = exp1] in exp2　例：let [x=1] in $x{x+2} (返回 3)

注意：变量具有局部作用域（函数级）。

if 语句、letrec 与递归

IF：　if exp1 boolSymbol exp2 then exp3 else exp4
例:　if 2>3 then 5 else 7 (返回 7)
递归：
　如果这个函数需要递归，我们必须要使用 letrec 声明（用法与 let 类似，但是我们不支持 letrec in）
　例：letrec f = $x{if x==1 then 1 else x*f[x-1] }
　　f[5] = 120

高精度科学计算

我们的计算器全部使用高精度计算，计算表达式中可以合理混用上述语句。
我们支持全部常见科学函数，小数精确到小数后 60 位，十进制整数位数 3000 万位以上。具体用法大概如下：
　　（2*sin(x)）^5-cos(7)+fact(4)-f[g[10]]+if 2>6 then 7 else y

使用 PI

我们内置了一个常数 PI，用这个常数代替圆周率，你可以自由使用它。
例：2*sin(PI/2)

高精度数据结构及基本运算

高精度数值类型 Numeric 的定义

计算机中实数的存储一般有两种方案：

- 浮点数，也就是 IEEE 754 标准定义的，以有效数字*指数的形式存储的方式，其有效位数是确定的。被目前几乎所有的计算机系统采用。
- 定点数，其中小数点的位置是确定的，即小数部分的位数是确定的。

在对精度要求较高的场合，浮点数的运算较慢而且容易损失精度，因此，IEEE 754 只规定了最多 128 位(sizeof = 16)的浮点数的存储形式，最高支持到约 31 位有效数字。而 EJack Calstack 计划精确到小数点后 60 位，因此我们采取了定点数的存储方式。

而高精度数据的存储策略也有多种。最平凡的如简单地用字符串存储，优点在于将 I/O 开销最小化，但空间浪费严重而且运算很慢；最精妙的如以类似于内置类型的二进制形式存储，优点在于（对计算机来说）运算自然、空间利用效率最大化，但 I/O 困难。故我们采取了以若干位十进制数为一个单位，存储在一个整数中，再将整数连接成组，赋予位权的方法存储高精度数值。换句话说，以 10^n 进制存储高精度数值。经过调整，n 被选取为 5。例如，整数 192608171989 在 data 中被存储为{71989,26081,19}。

考虑到需要处理的数字大小不定，整数部分采用了动态内存分配，data 保存指向首地址的指针，nbytes 保存长度。其中 data 动态数组按从低位到高位的顺序存储（即 data[0]是“个位”）。

fraction 数组保存了小数部分，越接近小数点的部分下标越小。

此外，还需要一个域来存储高精度数的符号。由于作为一个数值，符号的合法取值只有 POS 和 NEG（分别定义为 0 和 1），故 sign 的取值可以保留作为验证一个内存块是否为 Numeric 的判据。在实现中，我们将闭包的结构的最低的 4 个字节亦作为校验位，当*(int*)ptr 为 2 时，表示 ptr 所指的内存区域是一个闭包。

因此，高精度数类型的定义大致如下。

```
#define int64 long long
typedef unsigned int64 Udbyte;
typedef struct {
    int sign;
    Udbyte *data;
    size_t nbytes;
    Udbyte fraction[CALSTACK_FIXED_POINT];
} Numeric;
#define POS 0
#define NEG 1
```

此外，numeric_cfg.h 中还定义了若干宏常量，用来控制 Numeric 的内存布局。如 CALSTACK_FIXED_POINT 定义了 fraction 数组的长度

Numeric 对象的内存管理

Numeric 类型要求在定义对象后立即调用 Numeric_new()函数或 Numeric_new_len()函数，二者都初始化 Numeric 对象，后者能给 data 指向的动态数组指定长度。囿于 C 语言的限制，这一初始化过程只能显式地调用。如果允许使用 C++，则这一初始化可以优雅地用构造函数完成。在初始化时，data 被分配内存，nbytes 被设为 1，整个对象被置为 0.0。Numeric 要求不论整数部分是否为零，nbytes 都至少为 1。

Numeric 对象通过 Numeric_free()函数析构。该函数将 data 指针所指向的内存区域释放。另有 Numeric_free_ptr()函数将动态分配在堆上的 Numeric 对象释放。这一函数与 C++中的析构函数扮演着相同的角色。

类似地，Numeric_assign()和 Numeric_move()起到了类似 C++中复制构造函数和移动构造函数的作用。

Numeric 对象的输入输出

由于采用了 10^n 进制，输入输出时甚至不需要考虑进制转换的问题。我们通过字符串和 Numeric 之间的转换代替直接的 I/O 操作。

Numerictostr()函数将一个 Numeric 对象输出到字符串中，而 strtoNumeric()从字符串读取一个数将之转换为 Numeric 对象。前者只需按顺序分别输出整数部分和小数部分，并截去小数部分的尾部 0 即可，后者以'.'为界，前后分别 5 个数字一组，利用 sscanf()读入即可。

Numeric 对象的输入输出

由于采用了 10^n 进制，输入输出时甚至不需要考虑进制转换的问题。我们通过字符串和 Numeric 之间的转换代替直接的 I/O 操作。

Numerictostr()函数将一个 Numeric 对象输出到字符串中，而 strtoNumeric()从字符串读取一个数将之转换为 Numeric 对象。前者只需按顺序分别输出整数部分和小数部分，并截去小数部分的尾部 0 即可，后者以'.'为界，前后分别 5 个数字一组，利用 sscanf()读入即可。

Numeric 对象的比较操作

Numeric 对象之间的比较算法几乎与人比较两个数的过程完全相同。从高位顺次比较直至低位即可。此外，出于效率考虑，我们额外实现了一个 Numeric 对象与 0，1 和-1 之间的比较。

Numeric 对象的加减法

Numeric 对象的结构决定了其加减法与笔算过程几乎完全一致。从低位加/减至高位，并同时处理进位和退位的情况，算法的复杂度为 O(n)，且几乎没有任何优化的余地。

对于负数的处理以及小数减大数的情况，加减法函数之间可能互相委托，增大代码的复用性。

Numeric 对象的乘法

高精度乘法的平凡算法也是模拟笔算乘法，但这样将导致 O(nm)或者 $O(n^2)$的时间复杂度。这对于较长的高精度数将是不可接受的。我们在实现过程中，采用了快速傅里叶变换（Fast Fourier Transformation,FFT）算法，具体如下。

对于这个课题，我们先从多项式开始一步一步进行分析。一个度数为 n 的多项式 A(x)可以表示为：

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

一个多项式可以有不同的表示方式，一种是我们常见的系数表示，另一种是点值表示。例如上面的多项式就是以系数表示形式定义的。它展开之后可以写成下面这样：

$$A(x) = a_{n-1}x^{n-1} + \cdots + a_2x^2 + a_1x + a_0$$

这种方式的优点是很直观，也是我们最常用的表示多项式的方法，但是采用这种方式表示的多项式在进行大数值的乘法运算的时候，如果采用直接乘法方式，假设两个多项式的度数都为 n, 那么乘法计算的时间复杂度为 O(n2).

如果采用另一种不那么直观的点值表示法，对于多项式的度数比较大的情况，采用点值表示法进行乘法运算可以获得较大的性能提升。对于上面提到的多项式 A(x), 它的点值表示法大概像下面这样子：

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

其中 x_0~x_{n-1} 是 A(x)上面取的不同的点，而且

$$y_k = A(x_k)$$

可以看出，多项式从系数表示形式直接转化为点值表示形式需要选取 n（多项式的度数）个不同的值，进行 n 次的带入求值运算。那么转化为点值表示形式有什么好处呢？好处在于两个满足一定要求的点值表达式进行乘法运算的时间复杂度为 O(n). 相对于系数表示法直接进行乘法运算，点值表达式在计算乘法的性能是很理想的。

现在我们来看看点值表达式是如何快速地进行多项式乘法的。假设有两个多项式 A(x), B(x)，它们的乘积为一个新的多项式 C(x)。在计算乘法的时候，多项式的项数可能会增加，因此乘法运算得到的新的多项式 C(x)的度数是 degree(C) = degree(A) + degree(B). 因为这个原因，在计算乘法的时候，需要对 A(x)和 B(x)进行点值表达式的扩张。也就是说多项式 A(x), B(x)的点值表示形式要分别扩张为：

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$$

和

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\}$$

还有一点要注意的是上面两个式子中的 x_0~x_{2n-1} 是要一一对应的，否则不能直接进行乘法运算。这样 C(x) 就可以转化为 A(x), B(x)的点值表达式的对应项相乘的结果：

$$\{(x_0, y_0 y_0'), (x_1, y_1 y_1'), \ldots, (x_{2n-1}, y_{2n-1} y_{2n-1}')\}$$

这样，我们就能以 O(n)的时间复杂度（不考虑点值表达式的扩张运算，只需要计算 2n 次乘法）进行多项式 A(x)和 B(x)之间的乘法运算。

那么接下来，我们要讨论的是如何进行多项式的点值表示形式和系数表示形式之间的转换，这里涉及到一个问题，就是多项式的插值转换的唯一性。也就是说你要确定从点值形式转化为系数形式是唯一的，这个问题已经被数学证明，插值转换是唯一的，这个是利用线性代数的范德蒙德矩阵（下图左方的矩阵）的可逆性来证明的，具体的证明就不贴上来了。而系数形式转化为点值形式可以有很多种不同的方案，因此不存在唯一性。

对于系数表达式转化为点值表达式，称它为求值(evaluate)，我们可以利用秦九韶算法对给定的 n 个点进行代入

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

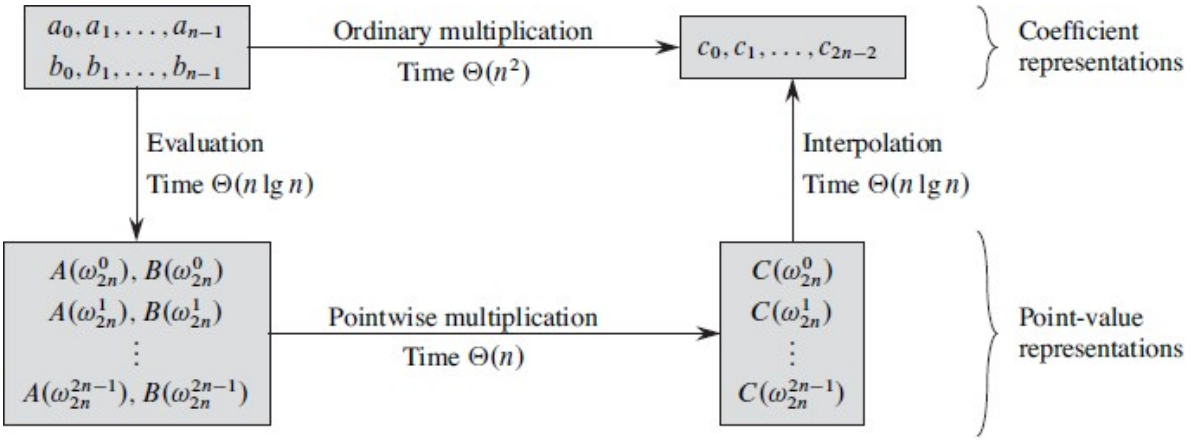
多项式 A(x)运算，这种方法的时间复杂度为 O(n2). 对于点值表达式转化为系数表达式，称它为插值(interpolate)，我们可以直接利用上图给出的线性代数方程，求出范德蒙德矩阵的逆矩阵，然后与向量上图右方的 y 向量相乘，计算出系数向量 a, 用这种方法计算的时间复杂度从上图就能看出来，为 O(n3). 也可以利用拉格朗日公式进行插值运算。

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}.$$

按拉格朗日公式计算系数表达式的时间复杂度为 O(n2).

采用以上提到的方法进行系数表示形式和点值表示形式之间的转化的效率并不理想，因此我们要考虑如何快速进行两者之间的转化，这就是 FFT 需要解决的问题。

以下这张图就是解决这个问题的总体概览，我们不妨一窥全豹：



DFT（**离散傅里叶变换**）

上图中的求值(evaluation)和插值(interpolation)是分别通过 DFT 和反向 DFT 实现的。DFT 是傅里叶分析的一种，它将一组有限的，均匀分布的函数样本值转化为一组复变正弦函数的系数。因为我看的资料大部分都是英文的，有些专业的词汇好像没有对应的翻译，为了严谨，这部分我就简单地说说 DFT 是怎么做的，尽力避免陈述其它的数学知识。

假定我们要进行对度数为 n 的多项式 A(x)进行 DFT，我们要将下列的值

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

代入 A(x)分别求值，得到一个 y 向量，这个向量就是原先多项式的系数向量 $a = (a_0, a_1, \ldots, a_{n-1})$.的 DFT，可以写 $y = \mathrm{DFT}_n(a)$。

上式出现的 ω_n 叫做 *principal* n*th root of unity*, 它的值为

$$\omega_n = e^{2\pi i/n}$$

这是最原始的计算 DFT 的方法，按这种方法计算的话，时间复杂度是 O(n2). 接下来我们讨论如何进行反向 DFT，其实反向 DFT 是很简单的，根据我们上面提到的插值转换的唯一性理论*(Uniqueness of an interpolating polynomial)*，我们进行 DFT 的运算实质是下面这样：

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

这样，我们只要求出中间的范德蒙德矩阵 V_n 的逆矩 V_n^{-1}，然后 V_n^{-1}与向量 $y = (y_0, y_1, \ldots, y_{n-1})$ 相乘，就能得出系数向 $a = (a_0, a_1, \ldots, a_{n-1})$，这就是反向 DFT，计算的时间复杂度为 O(n3)。能这样计算的前提是上图的范德蒙德矩阵 V_n 是可逆的，当然它的可逆性已经被证明了，我这里就不写了。从上面的讨论可以看到无论是 DFT 还是反向 DFT，直接计算的时间复杂度都不是很理想，接下来我们就来讨论一下 FFT 是如何降低计算 DFT 的时间复杂度的。

Cooley-Tukey **算法实现** FFT（**快速傅里叶变换**）

上文提到了 DFT 是什么以及如何进行 DFT 和反向 DFT 运算，现在我们来讨论 FFT 是如何快速进行 DFT 的。我们在上文提到过一个数值 ω_n ，这个数值有个有意思的特性，根据 *Halving lemma,* 下式是成立的：

$$(\omega_n^{k+n/2})^2 = (\omega_n^k)^2 .$$

其中 k 为非负整数。这个等式是 FFT 得以实现的关键。值得一提的是 FFT 的实现的算法有多种，我们这里讨论的 FFT 算法是基于的。这本书上提到的 FFT 算法叫做 Cooley-Tukey 算法，它是目前为止应用得最广泛的 FFT 算法。从现在开始，如果没有别的说明，我们就以 FFT 算法来称呼 Cooley-Tukey 算法。

假设我们有一个以系数形式表示的多项式 A(x)，它的度数为 n，n 为 2 的幂。那么它可以表示为

$$A(x) = a_{n-1}x^{n-1} + \cdots + a_2x^2 + a_1x + a_0$$

现在我们将它分成更小的两个多项式 $A^{[0]}(x)$和$A^{[1]}(x)$ ，它们分别满足如下关系：

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}. \end{aligned}$$

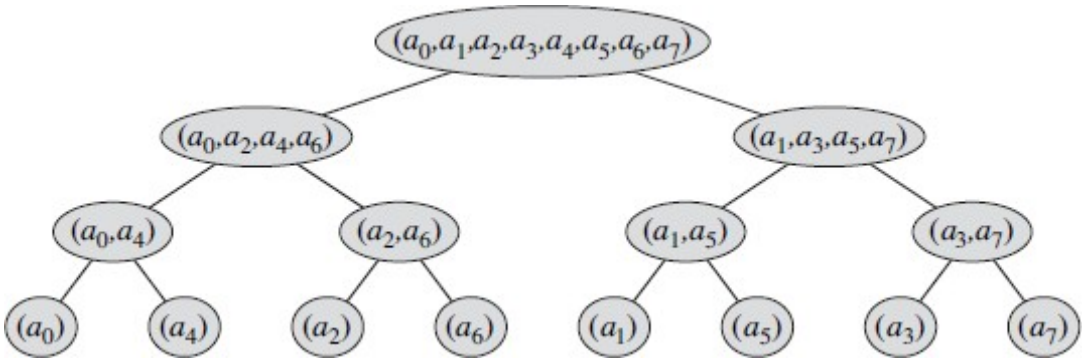
那么拆分出来的两个多项式和原来的多项 $A(x)$ 有如下关系：

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

这样的处理就是分治的基本步骤。

$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$ 并不是 n 个各不相同的值，它有一半是重复的。这样我们就能将问题的规模缩小成原来的二分之一，这是能够利用递归提升效率的关键。现在我们就能够根据上面的分析写出 FFT 的递归算法了，但递归算法的效率不高，接下来我们要讨论 FFT 算法的非递归实现。

我们来分析一下递归形式的 FFT 的步骤是怎么样的，假设我们要对元素个数为 8 的向量进行 FFT 运算，下面是自顶向下的递归的步骤：



运算的逻辑步骤是一颗树，而且我们猜想叶子结点的元素排列序号与根节点的元素排列序号应该有一定的联系。我们只要找出两者之间的联系，就可以通过它们之间的关系直接计算出递归最深层的元素排列序列，然后直接从最底层开始进行运算，用计算出的 DFT 替换掉上一层的节点，重复这个步骤直到根节点为止。这样我们就能将自顶向下的递归形式的 FFT 算法改进成自底向上的迭代形式的算法了。

为了分析它们之间的关系，我们打算从元素的下标着手，原始序列是 0, 1, 2, 3, 4, 5, 6, 7, 它们的二进制表示分别为 000, 001, 010, 011, 100, 101, 110, 111. 最底层的元素排列序列的下标是 0, 4, 2, 6, 1, 5, 3, 7. 它们的二进制表示分别为 000, 100, 010, 110, 101, 011, 111. 以下的表格可以清楚地显示对应位置的变化:

原始序列	000	001	010	011	100	101	110	111
最终序列	000	100	010	110	001	101	011	111

从表中可以看出原始的序列和最终的序列的对应位置的下标是互为相反位。也就是说原始序列的对应位置的下标进行位元反转运算就能得到最终序列。这样我们就能够利用这个关系着手实现 FFT 算法的迭代形式了。

算法实现的伪代码如下：

```
ITERATIVE-FFT(a)
 1  BIT-REVERSE-COPY(a, A)
 2  n = a.length                    // n is a power of 2
 3  for s = 1 to lg n
 4      m = 2^s
 5      ω_m = e^(2πi/m)
 6      for k = 0 to n − 1 by m
 7          ω = 1
 8          for j = 0 to m/2 − 1
 9              t = ω A[k + j + m/2]
10              u = A[k + j]
11              A[k + j] = u + t
12              A[k + j + m/2] = u − t
13              ω = ω ω_m
14  return A
```

其中 BIT-REVERSE-COPY 将我们要进行 FFT 的 a 向量按照位元反转位置复制到另一个向量 A，复制完成之后的 A 就是我们要进行递归运算的最终序列。它的伪代码如下：

```
BIT-REVERSE-COPY(a, A)
1  n = a.length
2  for k = 0 to n − 1
3      A[rev(k)] = a_k
```

其中 rev 实现位元反转，关于位元反转的具体问题我们在后面还会提到，现在我们来分析一下 FFT 非递归算法解决问题的步骤。在 ITERATIVE-FFT 中按照树形结构的运算逻辑从底层开始往上运算，第 3 行控制整个迭代的次数；第 4 行确定当前层数的元素对大小；第 5 行确定该层的螺旋因子；第 8 行~第 13 行是主要的运算逻辑，称为“蝶形运算”，在这里完成被分割为左右两部分的元素组的 DFT 运算。

以上就是 FFT 的大体叙述，FFT 在计算大整数乘法中是一个比较重要的部分，它为大整数乘法提供了优化方案，是目前实现大整数乘法的一种优秀算法。接下来我们要讨论在 C++下，利用 FFT 实现大整数乘法要考虑的细节问题。

位元反转（Bit reversal）

我们回顾一下之前在非递归形式的 FFT 中提到的位元反转问题，[1]中并没有给出位元反转的伪代码，但有提到我们可以实现时间复杂度为 O(log n)的位元反转算法。更进一步，如果我们采用表映射的方式，我们还能以近乎 O(1)的时间复杂度实现位元反转，只是我们可能需要比较多的内存空间，具体需要多少空间取决于计算机硬件和编译器。

进行位元反转的简单算法可以用伪代码描述如下：

```
BIT-REVERSAL(a)
1  A = a
2  n = a.bitLength
3  for i = 1 to n - 1
4      a >>= 1
5      A <<= 1
6      A += a & 1
7  a = A
```

这种通过移位来完成位元反转的时间复杂度为 O(k), k 为整数的位数。假如我们采用这种方法进行位元反转的话，那么在上面的 FFT 的非递归算法中 BIT-REVERSE-COPY 的时间复杂度就是 O(nk)。一般来说，在大整数乘法中，要进行位元反转的元素个数是很多的，因此 n 远大于 k，可以将 k 看成一个对效率影响不大的常数， BIT-REVERSE-COPY 的时间复杂度可以认为是 O(n).

我们在上面做的只是按照传入的数值进行一次位元反转，假如要进行反转的数有 n 个，那么就要调用 n 次上面的方法，不过我们在实现时可以将上述算法改写成内联函数以提高效率。如果不采用上面那种逐个计算的方法的话，还有下面这种方法。

```
BIT-REVERSAL(a)
1  n = a.length
2  a[0] = 0
3  a[1] = n / 2
4  for k = 2 to n - 1 by 2
5      a[k] = a[k / 2] >> 1
6      a[k + 1] = a[k] ^ a[1]
```

其中 a 是长度为 length 的数组，算法执行完后，从 a[0]到 a[lengt-1]都保存了与下标对应的位反转的值。这种方法计算的次数会减少很多，但是需要 O(n)的空间来保存计算出的数据。我们可以通过直接读取 a 数组中的数据来获得位反转的值，这就是表映射的基本思想。

这一算法的实际时间复杂度为 O(n logn loglogn)，在数值长度不太大时能获得满意的速度。

高精度基本初等函数的实现

我们实现的高精度科学函数有：绝对值函数，阶乘函数，平方根函数，指数函数，对数函数，双曲函数，三角函数，反三角函数。以下是我们实现科学函数的具体过程：

1、 绝对值函数

将数据的符号位改为正数值。

2、 阶乘函数

根据定义，利用循环乘法计算阶乘。

3、 平方根函数

一开始我们想用牛顿迭代法实现平方根函数，但发现其初始值的寻找有些困难，于是我们最后决定使用类似二分的方法实现了平方根函数。

4、 指数函数

（1） 快速幂：我们首先利用快速幂算法实现了整指数幂的计算

（2） 自然指数函数：由于我们数据定点数的特性，我们采用将数学分为整数和小数部分来计算，整数部分使用快速幂，小数部分利用 Taylor 展开，满足了我们对数据长度与精度的要求。

（3） 幂函数：整指数幂部分已使用快速幂完成，分数指数幂，我们通过换底实现。

5、 对数函数

（1） 自然对数函数：我们将数据用科学记数法以及多次开方缩小数据，利用 Taylor 展开实现对数。

（2） 对数函数：通过换底实现。

6、 双曲函数：根据定义及自然指数函数的计算实现。

7、 三角函数：通过诱导公式缩小数据，然后 Taylor 展开实现

8、 反三角函数：利用 CORDIC 算法及三角函数计算实现。

底层数据结构

栈类型的定义

Stack 和 SStack 结构分别被定义为数据栈和符号栈的类型。出于可扩展性和节约内存空间的考虑，我们为这两种栈动态分配内存空间，按需取用，避免了大量浪费内存空间。因此，我们的结构内只包含栈的首尾指针，而不是一个被定义得很大的数组。

在动态分配内存的情况下，如果每次将一个元素进栈，就重新分配一次内存并复制元素，将会导致巨大的时间开销，这是不适当的。因此，我们效仿 C++标准库中对 string 和 vector 这类动态增长的容器对象的实现，采用了逐级分配内存的方法。具体而言，就是当元素即将填满已分配内存时，一次性重新分配一块大小为原来的 n 倍的内存，直到入栈的元素再次将此内存填满为止。这种内存管理策略兼顾了时间和空间。因此，栈类型定义如下：

```
typedef struct {
    Numeric *head;
    Numeric *tail;
    size_t capacity;
} Stack;
```

```
typedef struct {
    Operation *head;
```

```
    Operation *tail;
    size_t capacity;
} SStack;
```

其中 head 是指向栈底元素的指针，tail 是指向栈顶元素的下一个元素（即一个逻辑上不存在的元素）的指针，capacity 则记录当前已经为该栈对象分配的空间。当元素占满这些空间后，Stack 类型将重新分配内存，并将元素迁移到新的地址。

Stack 和 SStack 类的操作

初始化时，栈对象的将被分配一个初始的内存。这段内存的大小被定义在宏 CALSTACK_INIT_SIZE 中。而每次内存扩张的比率则定义为 CALSTACK_RESIZE_FACTOR，参考 C++标准库的实现，这个宏常量的值被设为 2。

将一个元素入栈时，必须先判断是否达到了容量上限。如果达到了容量上限，就必须先将目前栈内的所有元素复制到新位置，再释放原来的内存。

解释器的构造与实现

解释器的构造与实现

EJack 计算器代码的主体部分本质上是一个图灵完备的纯函数式解释器。

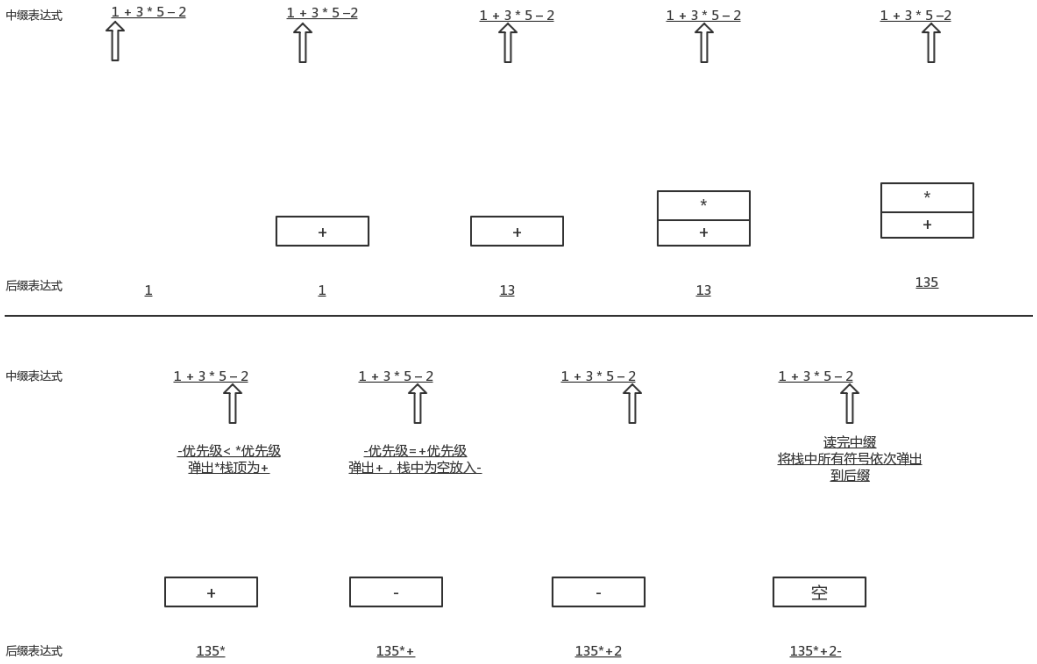
简单计算器的实现

实现加减乘除这样的基本功能并不复杂，我们首先需要解决的第一个问题是将“中缀表达式”转换为“后缀表达式”。

后缀表达式，又称逆波兰式，指的是不包含括号，运算符放在两个运算对象的后面，所有的计算按运算符出现的顺序，严格从左向右进行，所以不需要算符优先级，这对我们编写计算器来说很好实现。

中缀表达式：1 + 3 * 5 – (7 / 9)
后缀表达式：1 3 5 * + 7 9 / -

让我们来看一下中缀表达式转后缀表达式的具体步骤：



我们为数字制作一个数字栈 numericStack，有了后缀表达式和数字栈后，我们的计算器就可以开始计算了：

1. 从左到右读表达式
2. 遇到操作数压入栈中
3. 遇到操作符取并弹出栈顶 n 个元素，（n 取决于操作符是 n 元操作符），计算结果压入栈中
4. 后缀表达式读完，当前栈顶元素及为结果

复杂优先级计算器

这样我们就完成了简单的加减乘除运算，现在我们为这个计算器扩展功能：加入括号。有了括号后，我们就可以实现复杂的四则运算了。

但从读入中缀表达式字符串到获得后缀表达式，再通过后缀表达式计算结果这显然过于麻烦，而且后缀表达式的存放更是一个大问题，于是我们可以再优化一下算法，让我们的计算器在读入中缀表达式，转换成后缀表达式的过程中直接计算，当中缀表达式读取完毕的时候，我们也就获得了计算结果。

让我们看一下这样处理以后我们的计算逻辑：

首先我们需要两个栈：symbolStack 和 numericStack，分别用来存放操作符与数字，我们需要遵循这几条原则：

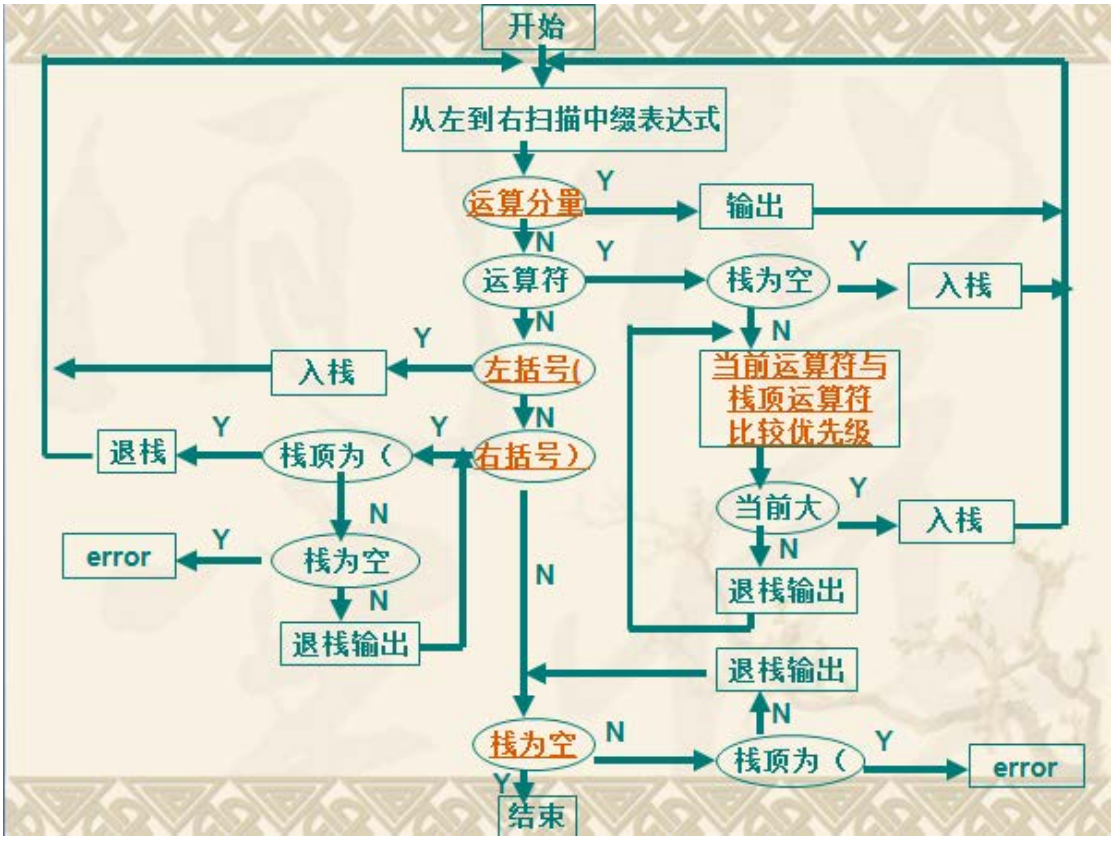
B. 读入一个数字的时候，直接压入 numericStack

C. 读入一个操作符时：
   1. 当 symbolStack 栈是空的，直接将当前读入的操作符压入 symbolStack
   2. 当 symbolStack 栈顶元素优先级小于当前读入的操作符，将当前读入的操作符压入栈内
   3. 当 symbolStack 栈顶元素的优先级大于等于当前读到的操作符，取出栈顶操作符，同时取出 numericStack 内相应数量的数字进行运算。重复此过程，直到 symbolStack 栈顶元素优先级小于当前读入的操作符或 symbolStack 为空。
   4. 遇到左括括号直接压入 symbolStack 内，将左括号优先级视作最低，同时不允许新的左括号将 symbolStack 内已有的左括号根据 B.3 原则弹出。
   5. 当遇到右括号的时候，右括号不入栈，将 symbolStack 内元素依次弹出，进行运算，直到碰到一个左括号时，将左括号弹出（左括号弹出不进行任何计算操作，直接舍弃左括号），停止操作，恢复字符串的正常读入。

D. 当前字符串读取结束，若 symbolStack 仍然不为空，则将栈内运算符依次弹出与 numericStack 内的数字进行运算。

E. 全部操作处理完毕后，返回存储在 numericStack 内的最终结果。

具体的流程可以参考下图：



现在，我们的计算器已经可以处理形如：2*3/(2-1)+3*(2+1) 这样具有复杂优先级的计算了，

科学计算器的实现

生活中总是要用到一些数学函数，比如：sin()，cos()，ln()，exp()，arctan()，fact()等等，现在让我们将这些函数加入计算器中。

科学函数的加入看似复杂，但实际上我们可以很轻松的解决这个问题，我们只需将科学函数名与左括号看做一体，将"sin("整体看做一个科学函数符号，问题就十分好办了。
我们规定：

所有单目科学函数（诸如：sin(，cos(，fact(）被读入的时候，都将其作为普通左括号进行操作，而当单目科学函数被弹出的时候，则调用相应的科学函数对 numericStack 内相应的元素进行计算。

而对于所有双目科学函数，则应该为其制定合适的语法。比如：以 n 为底 x 的对数，我们可以这样规定语法：nlog(x)。这样，我们就可以用与单目科学函数相同的方式处理这些双目科学函数。

解释器的理论基础

至此，一个具有基本运算能力的计算器已经完成了，但是，这样的计算器既不能声明变量，也不能自定义函数，显然还是不够方便，我们现在将它改造成一个解释器。
那么，什么是一个解释器？

作为一个初学者，我对“解释器”这三个字是十分敬畏的，在我的认知中，这三个字代表的应该是一个函数的名字：你为其输入一个表达式，它为你返回一个结果。

就如同图中所示，我们接受一个 1+2（图中是前缀表达式），返回一个 3，这就是一个最简单的解释器，某种意义上，计算器也是一种解释器。

从计算理论的角度来讲，每个程序都是一台机器的一种"描述"，而解释器就是在模拟这台机器的运转，进行“计算”。因此，在某种意义上，解释器就是计算的本质。当然，不同的解释器可以进行不同的运算。CPU 也可以看做是一种解释器，只不过它专门负责解释机器对应的机器语言。

词法分析

为了实现一个解释器，我们首先要完成词法分析的任务。所谓词法分析，就是将一个字符串了，也就是字符流，转换成解释器能够很方便处理的 Token 流形式。

所谓 Token，就是一个解释器能够接受的最小单位。在 C 语言中，他可以是这样的结构：

```
typedef struct token
{
    Tags tag;
    void *value;
}token;
```

其中，这里的 Tags 是一个枚举类型，包含解释器支持的全部 Token 种类。比如我们可以有 plusToken，numToken。如果是运算符号形式的 Token，那么 value 就是 NULL，如果是 num 或者 alpha 等等类型的 Token，则 value 会指向这个 Token 类型需要的数据内容。
比如我们有这样一个字符串：

$$1 + 3 + \mathrm{Sin}\left(\frac{\pi}{2}\right) - \mathrm{x}$$

现在我们将其解析成 Token 流，那么他就应该由三个 numToken（value 分别指向 1,3，π/2）、两个 plusToken、一个 sinToken、一个 subToken、一个 rBracketToken 和一个 alphaToken（value 指向 x）按照顺序组成。以此类推，我们就可以将字符串合理的转换为方便计算机处理的 Token 流

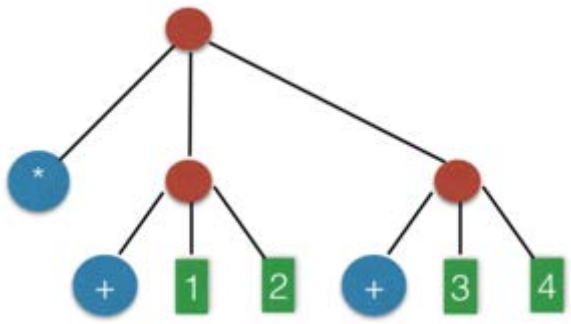
这样，我们用一个语法分析函数 Lexer，通过合理的方式，就能将字符串分析翻译成一个 Token 流。

语法分析与抽象语法树（AST）

字符串已经被变成了 Token 流，但这些 Token 并没有一个合理的结构来描述他们的关系。因此我们需要构建一个抽象语法树（AST）来完成这件事情。

我们输入到解释器的命令，虽然看似是一个线性结构，但各个 Token 之间关系复杂，用简单的线性结构难以描述他们。

实际上，这些有效命令是一个树状结构。比如：（1+2）*（3+4），他们之间的关系应该是这样：



（蓝色的操作符与绿色的数字都是叶子，而红色的节点都代表一个子树结构）

既然我们的计算器已经可以将（1+2）*（3+4）计算出结果，那么我们只需让其中的全部计算（如：1+2）产生的结果从数字改为一个子树，我们就可以获得一棵抽象语法树。

比如：1+2，3+4 改造后，我们计算出的结果不是 3 和 7，而是一个 Node，然后两个 Node 相乘后，我们又得到一个新的 Node，这些 Node 内不但可以包含操作符和数字，还可以包含其他 Node，实际上就已经形成了一棵树。

语法分析函数 Parser 就是为了将 Lexer 得到的 Token 流变成 AST 而生。

树遍历算法

在有了这棵树之后，我们需要把他对应的结果计算出来。这时，我们就需要一个树遍历操作来完成这件事情。

接受一个命令，返回对应结果，这是树遍历操作函数的目的，实际上，前文构造的数据结构都是为了给这个函数提供合法的、容易处理的输入，这个树遍历操作函数才是真正的解释器，因此我们把它命名为 interpret。

这个函数的构造非常巧妙，我们先用一个最简单的树遍历操作来举例。

比如：我们打算求一棵树的和。那我们可以这样做，对于属于这颗树的全部节点，如果这个节点是一个数，则做加法，返回他与之前接收全部数字的和，如果是一个子树，则递归调用求和函数本身。

这个树遍历函数其实就是由两个部分组成：一个部分用来处理数字，另一个部分用来处理子树。如果不把直接接受到一个数字，返回和这个操作看做一个递归函数的入口，实际上这个函数就是由一个出口，一个入口组成。而复杂的树遍历函数，实际上就是有着一个入口，多个出口的递归函数。

环境、变量、闭包、变量调用

解释器比计算器高明的一点是，解释器可以支持变量。为了实现这件事情，我们需要一个环境。

环境是一个数据结构，里面由变量值和变量名组成。环境有两个操作，一个是 lookup-env，它接受一个环境和一个变量名作为参数，返回一个变量值。这个操作会依照变量名在环境中查找对应的值，值得一提的是，lookup-env 总是会按照顺序，从最后加入环境的变量开始查找。

另一个操作则是 ext-env，它接受一个环境、一个变量名、一个变量值作为参数、返回一个新的环境。

而对于变量，我们只有变量绑定一种操作，即：我们不能通过不带初始化的声明获取一个变量。也就是说，我们只能这样：let x = 1。

对于变量调用，当我们接受一个字母输入的时候，我们会调用 lookup-env 函数，查找变量名对应的 value 值，从而实现变量的调用。

在我们的解释器中，函数是一等公民，也就是说，函数与变量具有平等的地位，我们允许一个函数作为参数传入，也允许它作为返回值。我们的解释器通过支持 lambda 表达式来实现函数。Lambda 表达式是一种匿名函数，我们实现的 lambda 表达式的具体形式：

$varName{exp}

其中，varName 代表这个函数的参数名字，exp 代表一个表达式。可以看出，lambda 表达式只有一个参数，但是我们可以通过嵌套的方式实现多参数函数，如：$x{ $y{ exp } }

因为我们的函数是一等公民，所以我们可以通过 let name = $varName{ exp }，这样我们就实现了有名字的函数。

对于函数的调用，我们可以直接使用$varName{exp}[value]，f[value]，let [varName = value] in exp (此处 exp 为 lambda 或者函数名)三种形式进行函数的调用。

为了实现 lambda 表达式，我们需要让我们的解释器支持闭包。闭包是一种强大的数据结构，当一个 lambda 表达式被定义的时候，我们会把当前的环境的拷贝与 lambda 表达式本身，参数一同封装进一个数据结构，就形成了一个闭包。如果使用 let f = exp 这样的形式定义有名字的函数，函数名与闭包会被一同添加进外部环境中。

我们可以发现，在这样的规定下，我们只能向环境添加变量，而不能“修改变量”。如果我们需要“修改变量”，我们只需向环境中添加一个新的变量，然后根据 looup-env 的从新到旧法则，原先的变量就会被“屏蔽”。这样的做法会为后来我们实现变量的作用域带来巨大的好处。

EJack 解释器的构造

我们现在已经了解了解释器需要的理论。现在就让我们把刚刚完成的计算器制作成一个解释器。

由于使用 AST 树作为命令的数据结构，会使得解释器变的非常复杂，鉴于时间问题，我考虑通过将字符串本身直接看做一棵树，把 Lexer、Parser 写在一起，成为一个表达式解析函数。（也就是我们最终采用了字符串解析的方式）

我们的主要函数是 interpret，他接受一个 env（环境）和一个 exp（表达式）作为输入。返回一个值（闭包或者 Numeric，这里是通过 void 型指针实现），这个函数有 5 个出口和一个入口组成，出口分别是：Let、变量调用、函数调用、数字、闭包，而一个入口则是处理复合表达式（exp）。

Let 的实现我们考虑这样做，我们检测到一个字符串的开头是 Let 时，自动将 let 后‘=’前的字母拷贝作为 varName（不含空格），将‘=’后的东西作为一个 exp，将这个 exp 与当前 env 作为参数，递归调用 interpret（对于所有可能是复合表达式 exp 形式的东西，我们都会采用类似的递归做法）得到 v1，把 varName 和 v1 作为键值对封入环境，我们就实现了一个函数绑定。为了方便嵌套，我们定义 let 的返回值是 v1。（如果是闭包则显示 Closure）

而对于 let [varName = exp1] in exp2 的形式，我们则同样对于 exp1 和 exp2 递归调用 interpret，获得 v1 和 v2，由于 v2 一定是闭包，因此我们继续解析这个闭包，我们从这个闭

包中拷贝出他的环境，将 varName 与 v1 组成键值对加入 v2 的环境中，使用这个环境与 v2 中封存的表达式递归调用 interpret，从而我们成功实现了一种 let in 形式的函数调用。Let in 允许我们多层嵌套，写成 let in let in let in 的形式。

变量的调用则更加容易，当我们读取到一个完全由字母组成的字符串的时候，由于变量调用在 let 后，因此这个字符绝对不是 let，所以他必然是一个变量，我们通过 lookup-env 获取 env 中对应的变量值 v，并将其返回。

处理数字是最简单的一个出口，当接受到一个数字的时候，我们只需要原样返回这个数字就可以了。

处理闭包的实现比较复杂，当读取到一个‘$’的时候，说明这是一个闭包的开始，我们开始将'$'与‘{’之间的全部字符（不含空格）存储为 varName，将'{'与‘}’之间的全部内容保存为 exp，之后直接将当前环境拷贝作为 lenv（local environment），与 varName、exp 一同封装成一个闭包，并返回这个闭包的指针。

函数的调用我们则是这样实现，当一个函数被调用的时候，如果它是一个“变量名+[exp]”的形式，那么我们就先对这个变量名进行 interpret，这样会为我们返回一个闭包，如果它本身就是一个闭包，那样更好，我们就省去了找到这个闭包的时间，然后，我们解析这个闭包，读取这个闭包内封存的环境，从中获取闭包的参数名，将实参值与参数名组成键值对添加进闭包的环境。从而我们完成了函数的调用。值得注意的一点是，我们在实现的时候，特别考虑了函数返回闭包的情况，因此实际上 f[ ][ ][ ]这样的式子是左结合的，也就是说，我们把 f[ ][ ]看做一个变量名字，他被 interpret 调用的时候，会返回一个闭包（实际上会递归调用很多次）。这样，我们的函数调用具有非常好的可嵌套性。你可以写出 f[g[f[ ]][ ]等等这样非常复杂的调用。

最后的这个入口，他负责处理一切非简单形式的表达式（简单形式就是指上面那些出口可以直接输出的形式）。这个入口，内部包含 numericStack 和 symbolStack，和一个无限循环的 parser 函数，每次 parser 函数会从输入的字符串中截取一个 token，如果这个 token 是常规运算符号或者数字，就进行常规运算；如果你如果这个 token 是变量绑定、变量调用、函数声明等等特殊情况、我们就递归调用 interpret 函数处理这个 token。

常规运算的原理是这样的，我们通过定义了一个枚举类型 operation，其中存放了所有的运算符，每当读取到运算符的时候，就调用 pushOperation 函数，这个函数会按照运算符优先级按照之前描述的流程将 operation 加入符号栈中，并根据情况调用 calculate 函数，当 calculate 函数被调用的时候，会将符号栈顶端的一个符号弹出，根据他的类型弹出对应数量的 Numeric 并调用合理的函数进行运算。常规计算就完成了。

这样，我们就完成了一个解释器的全部基本原理。

变量的作用域

我们实现闭包的时候，特殊强调了要把环境一同封装进闭包，这样的目的主要是为了实现变量的局部作用域，在这个闭包内部我们定义的变量不会影响外部（因为外部使用的环境是外部环境而不是闭包内的环境），而且更好的一点是，当这个闭包被定义后，以后再在外部定义任何变量都不会再影响闭包内部。闭包成为了一个封闭环境，因此他有了非常强大的功能。

而我们每次 let 的时候，都会直接加入新的变量，而不是修改旧的变量，通过变量屏蔽的方式来完成变量值的更新，这让我们非常简单的解决了定义局部变量后要屏蔽外部变量的问题。

闭包的作用域非常符合传统思想的作用域，我们可以方便的使用局部变量而不受外部全局变量的干扰。

为解释器添加更多强大功能

实现递归

我们的函数是一等公民，我们的 let 可以绑定一个 f 到闭包上，但是根据我们 let 的实现原理，这个闭包被定义的时候，他的环境里并没有 f 本身，因此我们并不能在函数内调用 f 本身。

为了补足这个缺陷，我们考虑实现一个 letrec 作为递归函数的绑定方式。通过 letrec 声明的函数可以正常递归。

Letrec 的实现我们可以考虑在 let 的基础上小改，当我们封装一个闭包的时候，我们通过直接操作闭包包含的环境内部的方式，将 f 强制加入环境内部，这样，我们就可以完成函数的递归。值得一提的是我们如何区分 letrec 定义的是普通变量还是闭包？我们通过在 Numeric 和 Closure 中最前面加入一个 int sign，然后通过*（int*）的强制类型转换方式，将万能指针 void*强制转换成 int*并读取他的第一个部分，从而获取 sign 的数值，这样就完成了一种多态性。(但不是很推荐这样做)

实现 IF 关键字

我们虽然支持了递归，但是没有 if 语句，我们没法定义一个出口，因此递归没法实际投入使用。现在我们为我们的解释器实现 if。

为了实现 if，我们干脆在 interpret 内加一个出口，专门用来处理 if。

我们规定语法为这样 if exp1 logicSymbol exp2 then exp3 else exp4 形式（logicSymbol 可以是“< > == >= <=”五种），依然是通过递归调用 interpret 解释 exp ，如果 condition 部分是真，那就返回 exp3 解释后的值，若假，则返回 exp4 解释后的值。
现在，我们可以写出各种复杂的递归函数了。

实现微分

我们在实现了普通科学函数后，考虑稍微做一下有趣的东西，于是我们选择了微分。
微分的语法为 diff function exp，然后我们会调用 interpret 解释 function 和 exp 两个部分，会得到 v1 与 v2，v1 必须为闭包，v2 必须为 Numeric，我们会根据微分定义计算出 function 在这个 v2 点的微分值。具体的方法是：

$$\frac{\mathrm{F(x+dx)-F(x-dx)}}{2dx}$$

这样我们就得到了一个点的微分值。

实现文件读入

作为一种编程语言,Ejack 自然还是要支持文件读入的，这个功能非常简单，我们只需要简单的吧字符串输入改成文件输入就可以了。

为解释器制作友好界面

Interpret 接受 env 和 exp 两个变量，然而用户只会输入 exp，显然这是不友好的。我们考虑做一个 Jack 函数，套在 interpret 外面，他只接受一个 exp 输入，初始环境 env0 在 main 中初始化，这样显然友好了很多。我们再让 Jack 循环，以及添加输入提示符，我们就可以友好的输入多条命令了。

总结

我的解释器是一个纯函数式的解释器，它每次只能接受一条指令，函数内部也只能具有一条指令，但是对于一个纯函数式语言，这一句命令已经足够他图灵完备了。作为我第一次写解释器，这个作品虽然非常强大，但我依然感到有很多不足，比如数据结构还是不是很完善，我的 Code 能力还不够强，逻辑不够清晰等等，但在未来，我相信 Jack 很快会被我完善成一门优秀的语言的。

这个解释器让我学到了非常非常非常多的知识，写完这个纯函数式解释器，我对“函数是一等公民”、“闭包”、“递归”、“lambda 演算”有了与之前完全不同的理解。我在短时间内学习了大量知识，从阅读入门的《计算机程序的构造和解释》，到后来的《计算的本质》、《EOPL》等等书籍，并加以运用，这让我感觉收获极大。希望未来，我能有机会参与更多这样具有挑战性的项目！