实验 05 实验报告

题目：设计一个交互式的计算器，用户可以提出不同的数据处理或计算要求

姓名：余宏昊　学号：2015200975　完成日期：2018.11.25

目录

一、 需求分析

模块 I 用顺序表来完成任意同维度向量的计算，包括加法、减法、夹角余弦值

1. 向量存放在顺序表中，维度不限，组成向量的元素为浮点数。各元素可以为任意实数。
2. 演示程序以用户和计算机的对话方式执行，即在计算机终端上显示相关提示信息之后，由用户在键盘上输入演示程序中规定的运算命令；相应的输入数据和运算结果显示在后。
3. 向量的形式为$(X_1, X_2, X_3, \dots, X_n)$，其中 n 为向量的维数，$X_i (i < n)$为浮点数。
4. 程序执行的命令包括：

1）构造向量 1；2）构造向量 2；3）作加法运算；4）作减法运算；5）求向量 1 和 2 的夹角余弦值；6）结束。

“构造向量 1”和“构造向量 2”时，需由用户先输入整数以定义向量维数，然后再依次输入各元素以定义向量。

5. 测试数据

Vec1=(2.0, -45.0, 32.0, -245.0, 0.0, 3442.0, 5.0, 0.0, 25.0, 56.0, -23.0),

Vec2=(8.0, 90.0, 55.0, -3.0, 0.0, 67.0, 790.0, 234.0, 804.0, 0.0, -687.0),

Vec1+Vec2=(10.0, 45.0, 87.0, -248.0, 0.0, 3509.0, 795.0, 234.0, 829.0, 56.0, -710.0),

Vec1-Vec2=(-6.0, -135.0, -23.0, -242.0, 0.0, 3375.0, -785.0, -234.0, -779.0, 56.0, 664.0)

Vec1*Vec2=268926.0

Cos(Vec1, Vec2)=0.0579

模块 II 用顺序表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 一元多项式存放在顺序表中，项数不限，每项形如aX^b，其中 a 和 b 均可以为任意实数。
2. 演示程序以用户和计算机的对话方式执行，即在计算机终端上显示相关提示信息之后，由用户在键盘上输入演示程序中规定的运算命令；相应的输入数据和运算结果显示在后。
3. 一元多项式的形式$a_1X^{b_1} + a_2X^{b_2} + \cdots + a_nX^{b_n}$为，其中 n 为一元多项式的项数，$a, b \in \mathbb{R}$。
4. 程序执行的命令包括：

1）构造一元多项式 1；2）构造一元多项式 2；3）作加法运算；4）作减法运算；5）作乘法运算；6）作求导运算；7）结束。

“构造一元多项式 1”和“构造一元多项式 2”时，需由用户先输入整数以定义向一元多项式项数，然后再依次输入各元素以定义一元多项式。

5. 测试数据

$$\text{Polynomial1} = 5X^{1000} - 1X^{828} + 7X^{7} - 89X^{5} - 23X^{3} + 90X^{2} - 235X^{1} + 754X^{0}$$

$$\text{Polynomial2} = -34X^{643} + 554X^{103} - 7X^{7} - 89X^{5} + 243X^{4} - 222X^{2} + 1X^{0}$$

$$\begin{aligned}\text{Polynomial1} + \text{Polynomial2} &= 755X^{0} - 235X^{1} - 132X^{2} - 23X^{3} + 243X^{4} - 178X^{5} + 0X^{7} \\ &+ 554X^{103} - 34X^{643} - 1X^{828} + 5X^{1000}\end{aligned}$$

$$\begin{aligned}\text{Polynomial1} - \text{Polynomial2} &= 753X^{0} - 235X^{1} + 312X^{2} - 23X^{3} - 243X^{4} + 0X^{5} + 14X^{7} \\ &- 554X^{103} + 34X^{643} - 1X^{828} + 5X^{1000}\end{aligned}$$

$$\begin{aligned}\text{Polynomial1} * \text{Polynomial2} &= 754X^{0} - 235X^{1} - 167298X^{2} + 52147X^{3} + 163242X^{4} \\ &- 119194X^{5} + 42785X^{6} + 888X^{7} + 3692X^{8} - 23811X^{9} \\ &+ 8082X^{10} + 1701X^{11} + 0X^{12} - 49X^{14} + 417716X^{103} \\ &- 130190X^{104} + 49860X^{105} - 12742X^{106} - 49306X^{108} \\ &+ 3878X^{110} - 25636X^{643} + 7990X^{644} - 3060X^{645} + 782X^{646} \\ &+ 3026X^{648} - 238X^{650} - 1X^{828} + 222X^{830} - 243X^{832} + 89X^{833} \\ &+ 7X^{835} - 554X^{931} + 5X^{1000} - 1110X^{1002} + 1215X^{1004} \\ &- 445X^{1005} - 35X^{1007} + 2770X^{1103} + 34X^{1471} - 170X^{1643}\end{aligned}$$

$$\text{DER Polynomial1} = 180X^{1} - 69X^{2} - 445X^{4} + 49X^{6} - 828X^{827} + 5000X^{999}$$

$$\begin{aligned}\text{DER Polynomial2} &= -444X^{1} + 972X^{3} - 445X^{4} - 49X^{6} + 57062X^{102} - 21862X^{642}\end{aligned}$$

模块 III 用单链表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 一元多项式存放在单链表中，项数不限，每项形如aX^b，其中 a 和 b 均可以为任意实数。
2. 其他的同模块 II。

模块 IV ⑴四则运算表达式求值。操作符包括加(‘+’)、减(‘-’)、乘(‘*’)、除(‘/’)、幂(‘^’)、左括号(‘(’)、右括号(‘)’)，而操作数则包括整数、浮点数等不同类型的数值。比如“30+4*2.5”，得到 40 或 40.0 等形式的结果。⑵含单变量的表达式求

值。变量可以是 C/C++的标识符。比如“3+4*X2”，需要输入变量 X2 的值，然后计算结果。数字形式包括用科学计数法表示的，如 1.14e2，以及负数等。

1. 运算表达式由字符串表示，长度不限。每个表达式有三种类型的元素，界限符、数值串以及未知变量串。
2. 表达式若以负数开头，则必须用括号括起来，如(-1)、(-2.0)。
3. 演示程序以用户和计算机的对话方式执行，即在计算机终端上显示相关提示信息之后，由用户在键盘上输入演示程序中规定的运算命令；相应的输入数据和运算结果显示在后。
4. 程序执行的命令包括：

1）接收用户输入构建表达式字符串；2）解析字符串，利用栈进行运算；3）若遇未知数，则要求用户输入数字进行赋值；4）返回运算结果；5）结束。

5. 测试用例

1) (-1.8e2)*((-5.2)+(2*3-1))^3+8/2.3=4.918

2) (-1.8e2)*(X1+(2*3-1))^3+8/X2(其中 X1=-5.2, X2=2.3)=4.918

模块 V ⑴定义并运行简单函数。比如定义：f(X2)=3+4*X2，然后执行 f(5)，则得到结果 23。⑵保留函数定义历史，并可以运行历史函数。⑶函数的调用：比如已经定义了函数 f(X)，新定义函数 g(X)中调用了 f。例如：

```
DEF f(X)=3+4*X;
DEF g(X)=3+4*f(X);
RUN g(5)
95
```

1. 简单函数的定义和运行与模块 IV 中含未知变量的表达式的定义和运行一致。只是简单函数可以被存储。
2. 在定义复合函数时，被调用的函数的自变量应与复合函数的自变量保持一致。
3. 演示程序以用户和计算机的对话方式执行，即在计算机终端上显示相关提示信息之后，由用户在键盘上输入演示程序中规定的运算命令；相应的输入数据和运算结果显示在后。
4. 程序执行的命令包括：

1）接收用户输入构建函数解析式字符串；2）选择运算或存储；3）若选择运算，则要求用户对自变量赋值，并对函数解析式字符串进行解析，利用栈进行运算；4）返回运算结果；5）结束。

5. 测试用例

1) DEF f(X)=6-X*2+1/X

   RUN f(2)

   2.50

   SAVE f(2)

2) DEF g(X)=2*f(X^2)-1/f(1/X)

   RUN g(2)

   -9.88

模块 VI 矩阵运算。实现矩阵的加、减、乘、转置、行列式求值运算。

1. 矩阵以三元组的形式存储在顺序表中，每个三元组定义了一个元素的行列位置和数值。矩阵的行列数可以为任意正整数。矩阵的元素可以为任意实数。
2. 演示程序以用户和计算机的对话方式执行，即在计算机终端上显示相关提示信息之后，由用户在键盘上输入演示程序中规定的运算命令；相应的输入数据和运算结果显示在后。
3. 矩阵形如：

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

   其中$m, n \in \mathbb{Z}^+$.

4. 程序执行的命令包括：

1）构造稀疏矩阵 1；2）构造稀疏矩阵 2；3）作加法运算；4）作减法运算；5）作乘法运算；6）行列式求值；7）结束。

“构造稀疏矩阵 1”和“构造稀疏矩阵 2”时，需由用户先输入整数以定义矩阵的行列维数，再依次输入三元组以构建矩阵。

5. 测试用例

$M_{4*5}=\{(2,4,7), (1,5,2), (4,5,4)\}$

$N_{4*5}=\{(1,1,1), (2,2,2), (3,3,3), (4,4,4), (4,5,5)\}$

$P_{5*2}=\{(1,2,3), (3,1,5), (4,2,6)\}$

$Q_{3*3}=\{(1,1,1), (2,2,2), (3,3,3)\}$

如下图：

```
0    0    0    0    2
0    0    0    7    0
0    0    0    0    0
0    0    0    0    4
```

M_{4*5}

1	0	0	0	0
0	2	0	0	0
0	0	3	0	0
0	0	0	4	5

N_{4*5}

0	3
0	0
5	0
0	6
0	0

P_{5*2}

1	0	0
0	2	0
0	0	3

Q_{3*3}

$M_{4*5}+ N_{4*5}=$

1	0	0	0	2
0	2	0	7	0
0	0	3	0	0
0	0	0	4	9

$M_{4*5}- N_{4*5}=$

-1	0	0	0	2
0	-2	0	7	0
0	0	-3	0	0
0	0	0	-4	-1

$Q_{4*2}=M_{4*5}* P_{5*2}=$

0	0
0	42
0	0
0	0

Transpose(Q_{4*2})=

0	0	0	0
0	42	0	0

DetVal(Q_{3*3})=6.00

二、 概要设计

模块 I 用顺序表来完成任意同维度向量的计算，包括加法、减法、夹角余弦值

用顺序表来表示向量，需要两个抽象数据类型：顺序表和向量。

1. 顺序表的抽象数据类型定义为：

ADT SqList {

数据对象：$D = \{a_i | a_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R_1 = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D, i = 1,2,\dots,n\}$

基本操作：

InitSqList(&L)

操作结果：构造一个空的顺序表 L。

DestorySqList(&L)

初始条件：顺序表 L 已存在。

操作结果：销毁顺序表 L。

ClearSqList(&L)

初始条件：顺序表 L 已存在。

操作结果：将 L 重置为空表。

ExtendSqList(&L, n)

初始条件：顺序表 L 已存在。

操作结果：对 L 进行扩容，增加 n 个单位的存储容量。

GetElemSq(L, i, &e)

初始条件：顺序表 L 已存在，$1 \leq i \leq L.length$。

操作结果：用 e 返回 L 中第 i 个数据元素的值。

SqListInsert(&L, i, e)

初始条件：顺序表 L 已存在，$1 \leq i \leq L.length + 1$。

操作结果：在 L 的第 i 个位置插入新的数据元素 e，L 的长度加 1。

SqListDelete(&L, i, &e)

初始条件：顺序表 L 已存在，$1 \leq i \leq L.length$。

操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。

SqListTraverse(L, visit())

初始条件：顺序表 L 已存在。

操作结果：依次对 L 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败

} **ADT** SqList

2. 向量的抽象数据类型定义为：

ADT Vector {

数据对象：$D = \{a_i | a_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R_1 = \{< a_1, a_2, \dots, a_n > | a_i \in D, i = 1,2,\dots,n\}$

基本操作：

CreateVector(&V, Dim)

初始条件：Dim 为整数

操作结果：生成一个维数为 Dim 的元素为浮点数的向量 V。

DestoryVector(&V)

初始条件：向量 V 已存在。

操作结果：销毁向量 V 的结构。

DimensionVector(&V)

初始条件：向量 V 已存在。

操作结果：返回 V 的维度。

AddVectors(V_1, V_2, &V)

初始条件：向量 V_1 和 V_2 已存在。

操作结果：将两个向量相加，并将结果返回到 V。

SubtractVectors(V_1, V_2, &V)

初始条件：向量 V_1 和 V_2 已存在。

操作结果：将两个向量相减，并将结果返回到 V。

MultiplyVectors(V_1, V_2)

初始条件：向量 V_1 和 V_2 已存在。

操作结果：将两个向量相乘，并将乘积输出。

CosineVectors(V_1, V_2)

初始条件：向量 V_1 和 V_2 已存在。

操作结果：求出两个向量的夹角余弦值，并将结果输出。

PrintVector(V)

初始条件：向量 V 已存在。

操作结果：按照$(X_1, X_2, X_3, \dots, X_n)$的格式打印出向量 V。

} **ADT** Vector

3. 本程序包含四个模块：

1) 主程序模块：

void main() {

初始化；

构建向量 1；

构建向量 2；

打印向量；

向量相加，输出结果；

向量相减，输出结果；

向量相乘，输出结果；

求夹角余弦值，输出结果；

销毁向量；

退出程序；

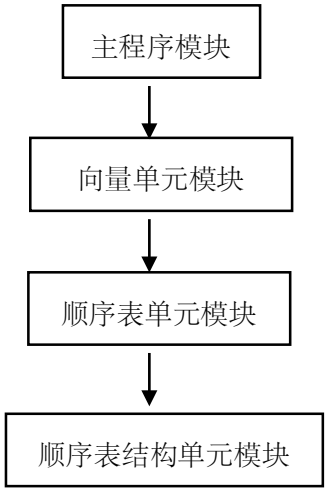
}

2) 向量单元模块——实现向量的抽象数据类型；

3) 顺序表单元模块——实现顺序表的抽象数据类型；

4) 顺序表结构单元模块——定义顺序表结构。

各模块之间的调用关系如下：



模块 II 用顺序表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

用顺序表来表示一元多项式，需要两个抽象数据类型：顺序表和一元多项式。

1. 顺序表的抽象数据类型定义为：

ADT SqList {

数据对象：$D = \{a_i | a_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R_1 = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D, i = 1,2,\dots,n\}$

基本操作：

InitSqList(&L)

操作结果：构造一个空的顺序表 L。

DestorySqList(&L)

初始条件：顺序表 L 已存在。

操作结果：销毁顺序表 L。

ClearSqList(&L)

初始条件：顺序表 L 已存在。

操作结果：将 L 重置为空表。

ExtendSqList(&L, n)

初始条件：顺序表 L 已存在。

操作结果：对 L 进行扩容，增加 n 个单位的存储容量。

GetElemSq(L, i, &e)

初始条件：顺序表 L 已存在，$1 \leq i \leq \mathrm{L.length}$。

操作结果：用 e 返回 L 中第 i 个数据元素的值。

SqListInsert(&L, i, e)

初始条件：顺序表 L 已存在，$1 \leq i \leq \mathrm{L.length} + 1$。

操作结果：在 L 的第 i 个位置插入新的数据元素 e，L 的长度加 1。

SqListInsertOrder(&L, e)

初始条件：顺序表 L 已存在。

操作结果：按升序插入新的数据元素 e，L 的长度加 1。

SqListDelete(&L, i, &e)

初始条件：顺序表 L 已存在，$1 \leq i \leq \mathrm{L.length}$。

操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。

SqListTraverse(L, visit())

初始条件：顺序表 L 已存在。

操作结果：依次对 L 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败

} **ADT** SqList

2. 一元多项式的抽象数据类型定义为：

ADT Polynomial {

数据对象：$\mathrm{D} = \{(a_i, b_i) | a_i, b_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$\mathrm{R_1} = \{< (a_1, b_1), (a_2, b_2), \dots, (a_n, b_n) > | (a_i, b_i) \in \mathrm{D}, i = 1,2,\dots,n\}$

基本操作：

CreatePolynomial(&P, numTerm)

初始条件：numTerm 为整数

操作结果：生成一个项数为 numTerm、系数和指数都为浮点数的一元多项式 P。

DestoryPolynomial(&P)

初始条件：一元多项式 P 已存在。

操作结果：销毁一元多项式的结构。

LengthPolynomial(&P)

初始条件：一元多项式 P 已存在。

操作结果：返回 P 的项数。

AddPolynomials(P_1, P_2, &P)

初始条件：一元多项式 P_1 和 P_2 已存在。

操作结果：将两个一元多项式相加，并将结果返回到 P。

SubtractPolynomials(P_1, P_2, &P)

初始条件：一元多项式 P_1 和 P_2 已存在。

操作结果：将两个一元多项式相减，并将结果返回到 P。

MultiplyPolynomials(P_1, P_2, &P)

初始条件：一元多项式 P_1 和 P_2 已存在。

操作结果：将两个一元多项式相乘，并将结果返回到 P。

DerivativePolynomial(P, &P_1)

初始条件：一元多项式 P 和 P_1 已存在。

操作结果：对一元多项式 P 求导，并将结果返回到 P_1。

PrintPolynomial(P)

初始条件：一元多项式 P 已存在。

操作结果：按$a_1X^{b_1} + a_2X^{b_2} + \cdots + a_nX^{b_n}$的格式打印输出一元多项式 P。

} **ADT** Polynomial

3. 本程序包含四个模块：

1) 主程序模块：

```
void main() {
    初始化；
    构建一元多项式 1；
    构建一元多项式 2；
    打印一元多项式；
    一元多项式相加，输出结果；
```

一元多项式相减，输出结果；

一元多项式相乘，输出结果；

一元多项式求导，输出结果；

销毁一元多项式；

退出程序；

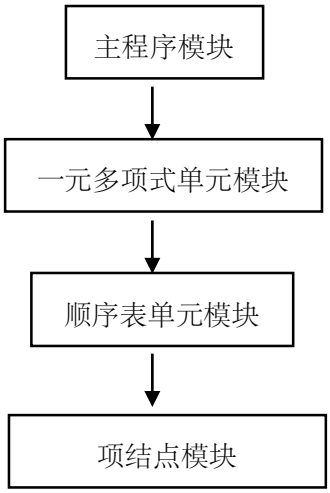
}

2) 一元多项式单元模块——实现一元多项式的抽象数据类型；

3) 顺序表单元模块——实现顺序表的抽象数据类型；

4) 多项式项的结点模块——定义项的结构。

各模块之间的调用关系如下：



模块 III 用单链表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

用单链表来表示一元多项式，需要两个抽象数据类型：单链表和一元多项式。

1. 单链表的抽象数据类型定义为：

ADT LkList {

数据对象：$D = \{a_i | a_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R_1 = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D, i = 2,\dots,n\}$

基本操作：

InitLkList(&L)

操作结果：构造一个空的单链表 L。

DestoryLkList(&L)

初始条件：单链表 L 已存在。

操作结果：销毁单链表 L。

ClearLkList(&L)

初始条件：单链表 L 已存在。

操作结果：将 L 重置为空表。

GetElemLk(L, i, &e)

初始条件：单链表 L 已存在，$1 \leq i \leq L.length$。

操作结果：用 e 返回 L 中第 i 个数据元素的值。

CreateNode(coef, expo)

操作结果：创建结点，结点数据域为 coef 和 expo。

LkListInsert(&L, i, e)

初始条件：单链表 L 已存在，$1 \leq i \leq L.length + 1$。

操作结果：在 L 的第 i 个位置插入新的数据元素 e，L 的长度加 1。

LkListInsertOrder(&L, e)

初始条件：单链表 L 已存在。

操作结果：按升序插入新的数据元素 e，L 的长度加 1。

LkListDelete(&L, i, &e)

初始条件：单链表 L 已存在，$1 \leq i \leq L.length$。

操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。

LkListTraverse(L, visit())

初始条件：单链表 L 已存在。

操作结果：依次对 L 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败

} **ADT** LkList

2. 一元多项式的抽象数据类型与模块 II 一致：

3. 本程序包含四个模块：

1) 主程序模块：

```
void main() {
    初始化;
    构建一元多项式 1;
    构建一元多项式 2;
    打印一元多项式;
    一元多项式相加，输出结果;
    一元多项式相减，输出结果;
    一元多项式相乘，输出结果;
    一元多项式求导，输出结果;
```

销毁一元多项式；

退出程序；

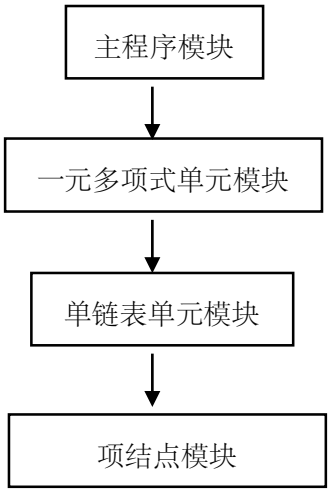
}

2) 一元多项式单元模块——实现一元多项式的抽象数据类型；

3) 单链表单元模块——实现单链表的抽象数据类型；

4) 多项式项的结点模块——定义项的结构。

各模块之间的调用关系如下：



模块 IV ⑴四则运算表达式求值。操作符包括加(‘+’)、减(‘-’)、乘(‘*’)、除(‘/’)、幂(‘^’)、左括号(‘(’)、右括号(‘)’)，而操作数则包括整数、浮点数等不同类型的数值。比如“30+4*2.5”，得到 40 或 40.0 等形式的结果。⑵含单变量的表达式求值。变量可以是 C/C++的标识符。比如“3+4*X2”，需要输入变量 X2 的值，然后计算结果。数字形式包括用科学计数法表示的，如 1.14e2，以及负数等。

要进行表达式运算，需要一个抽象数据类型：栈。

1. 栈的抽象数据类型定义为：

ADT Stack {

数据对象：$D = \{a_i | a_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R_1 = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D, i = 2,\dots,n\}$

约定a_n端为栈顶，a_1端为栈底。

基本操作：

InitStack(&S)

操作结果：构造一个空栈 L。

DestoryStack(&S)

初始条件：栈 S 已存在。

操作结果：销毁栈 S。

ClearStack(&S)

初始条件：栈 S 已存在。

操作结果：将 S 重置为空栈。

StackEmpty(S)

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 TRUE，否则 FALSE。

StackLength(S)

初始条件：栈 S 已存在。

操作结果：返回 S 的元素个数，即栈的长度。

GetTop(S, &e)

初始条件：栈 S 已存在且非空。

操作结果：用 e 返回 S 的栈顶元素。

Push(&S, e)

初始条件：栈 S 已存在。

操作结果：插入元素 e 为新的栈顶元素。

Pop(&S, &e)

初始条件：栈 S 已存在且非空。

操作结果：删除 S 的栈顶元素，并用 e 返回其值。

StackTraverse(S, visit())

初始条件：栈 S 已存在且非空。

操作结果：从栈底到栈顶依次对 S 的每个数据元素调用函数 visit()。一旦 visit()失败，则操作失败

} **ADT** Stack

2. 本程序包含六个模块：

1) 主程序模块：

```
void main() {
    键入表达式 1（不含未知变量）;
    对表达式 1 进行标准化处理;
    解析表达式 1，输出结果;
    键入表达式 2（含未知变量）;
    对表达式 2 进行标准化处理;
    解析表达式 2;
```

遇到未知变量要求用户键入数字进行赋值；

表达式 2 求值，输出结果；

退出程序；

}

2) 表达式单元模块——对表达式进行逐项解析；

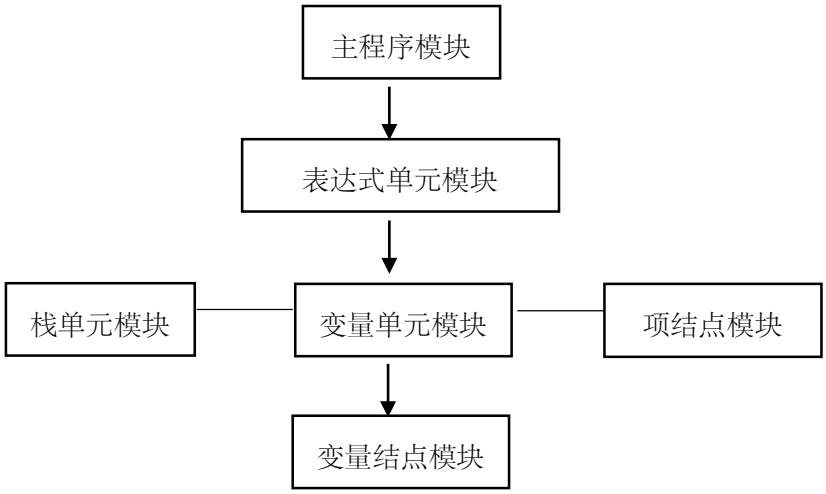
3) 栈单元模块——实现栈的抽象数据类型以及基于栈的各项运算操作；

4) 变量单元模块——对各变量进行处理。

5) 变量结点模块——定义变量结构。

6) 项结点模块——定义项结构。

各模块之间的调用关系如下：



模块 V ⑴定义并运行简单函数。比如定义：f(X2)=3+4*X2，然后执行 f(5)，则得到结果 23。⑵保留函数定义历史，并可以运行历史函数。⑶函数的调用：比如已经定义了函数 f(X)，新定义函数 g(X)中调用了 f。

实现上述功能需要两种抽象数据类型：栈（用于运算）和单链表（用于存储函数）。

1. 栈的抽象数据类型与模块 IV 中栈的抽象数据类型一致。
2. 单链表的抽象数据类型与模块 III 中单链表的抽象数据类型一致。
3. 本程序包含八个模块：

1) 主程序模块：

```
void main() {
    键入未知变量名和含未知变量的表达式 1 以构建简单函数 1;
    运行简单函数 1，用户键入未知变量的值;
    解析简单函数 1，输出结果;
    键入调用函数名、未知变量名和含被调函数的表达式 2 以构建复合函数 2;
```

运行复合函数 2，用户键入未知变量的值；

解析复合函数 2，输出结果；

退出程序；

}

2) 表达式单元模块——对函数解析式进行逐项解析；

3) 函数单元模块——实现函数定义、存储与运行；

4) 函数结点模块——定义函数结构

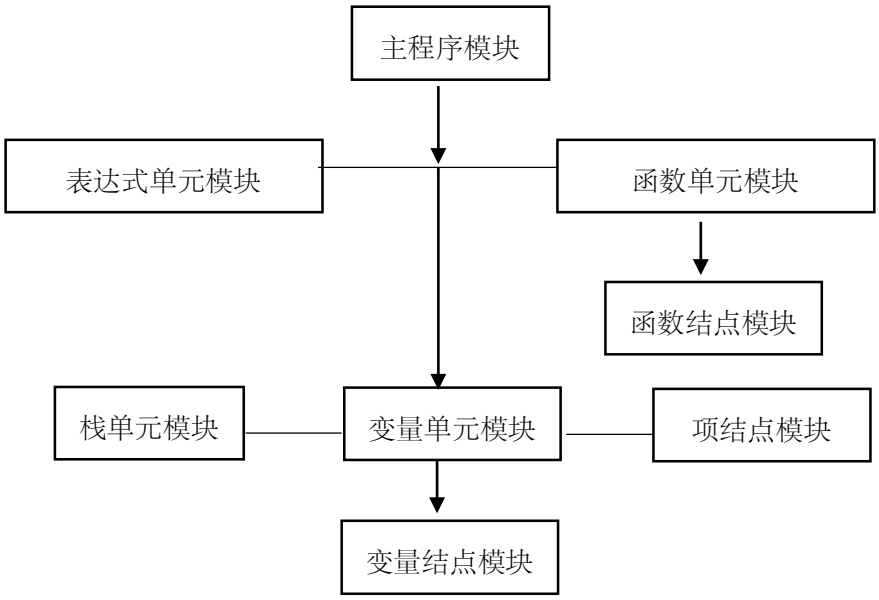
5) 栈单元模块——实现栈的抽象数据类型以及基于栈的各项运算操作；

6) 变量单元模块——对各变量进行处理。

7) 变量结点模块——定义变量结构。

8) 项结点模块——定义项结构。

各模块之间的调用关系如下：



模块 VI 矩阵运算。实现矩阵的加、减、乘、转置、行列式求值运算。

要进行表达式运算，需要一个抽象数据类型：稀疏矩阵。

1. 稀疏矩阵的抽象数据类型定义为：

ADT TSMatrix {

数据对象：$D = \{a_i | a_i \in ElemSet, i = 1,2,\dots,n, n \geq 0\}$

数据关系：$R_1 = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D, i = 2,\dots,n\}$

约定a_n端为栈顶，a_1端为栈底。

基本操作：

InitTSMatrix(&M)

操作结果：初始化一个稀疏矩阵 M。

CreateTSMatrix(&M)

操作结果：构造一个稀疏矩阵 M。

DestoryTSMatrix(&M)

初始条件：稀疏矩阵 M 已存在。

操作结果：销毁稀疏矩阵 M。

ResetTSMatrix(&M)

初始条件：稀疏矩阵 M 已存在。

操作结果：将 M 置空。

InsertTSMatrix(&M, i, j, e)

初始条件：稀疏矩阵 M 已存在。

操作结果：将元素 e 插入到稀疏矩阵 M 第 i 行第 j 列的位置上。

GetElemTSMatrix(M, i, j, &e)

初始条件：稀疏矩阵 M 已存在。

操作结果：用 e 返回稀疏矩阵第 i 行第 j 列的元素。

PrintTSMatrix(M)

初始条件：稀疏矩阵 M 已存在。

操作结果：按矩阵格式打印出稀疏矩阵 M。

CopyTSMatrix(M, &T)

初始条件：稀疏矩阵 M 已存在。

操作结果：将稀疏矩阵 M 复制到 T 中。

AddTSMatrix(M, N, &T)

初始条件：稀疏矩阵 M 和 N 已存在。

操作结果：将稀疏矩阵 M 和 N 相加，存入到 T 中。

SubtractTSMatrix(M, N, &T)

初始条件：稀疏矩阵 M 和 N 已存在。

操作结果：将稀疏矩阵 M 和 N 相减，存入到 T 中。

MultiplyTSMatrix(M, N, &T)

初始条件：稀疏矩阵 M 和 N 已存在，且 M 的列数与 N 的行数相等。

操作结果：将稀疏矩阵 M 和 N 相加，存入到 T 中。

TransposeTSMatrix(M, &T)

初始条件：稀疏矩阵 M 已存在。

操作结果：将稀疏矩阵 M 进行转置，存入到 T 中。

DetVal(M)

初始条件：稀疏矩阵 M 已存在，且 M 为方阵。

操作结果：对 M 所对应的行列式进行求解。

} **ADT** TSMatrix

2. 本程序包含三个模块：

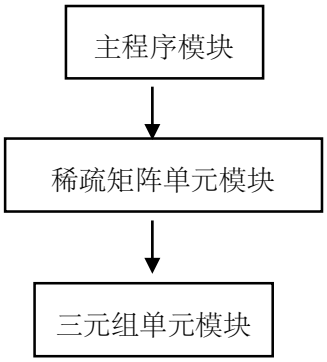
1) 主程序模块：

```
void main() {
    初始化;
    构建稀疏矩阵 M;
    构建稀疏矩阵 N;
    构建稀疏矩阵 P;
    构建稀疏矩阵 Q;
    将 M 与 N 相加，存入 Q;
    打印 Q；重置 Q;
    将 M 与 N 相减，存入 Q;
    打印 Q；重置 Q;
    将 M 与 P 相乘，存入 Q;
    对 Q 进行转置;
    构建 n 阶矩阵 S;
    求解 S 对应的行列式;
    退出程序;
}
```

2) 稀疏矩阵单元模块——实现稀疏矩阵的抽象数据类型；

3) 三元组单元模块——实现三元组的结构；

各模块之间的调用关系如下：



三、 详细设计

模块 I 用顺序表来完成任意同维度向量的计算，包括加法、减法、夹角余弦值

1. 元素类型、顺序表结构

```
typedef float ElemTypeVc;  //元素类型
typedef struct {
    ElemTypeVc * elem;   //存储空间基址
    int        length;   //线性表长度（元素个数）
    int      listsize;   //线性表当前分配的存储容量
} Vector;   //定义向量类型
```

2. 向量的基本操作设置如下：

```
StatusVc InitVector(Vector * pVec);
    //构建一个空的向量
StatusVc ExtendVector(Vector * pVec, int n);
    //扩充存放向量的线性表的容量
StatusVc InsertVector(Vector * pVec, int i, ElemTypeVc e);
    //在向量的特定位置插入元素
StatusVc CreateVector(Vector * pVec);
    //根据用户输入创建向量
StatusVc DeleteVector(Vector * pVec, int i, ElemTypeVc * e);
    //根据用户输入删除指定位置的元素，同时维数减 1
StatusVc ReplaceVector(Vector * pVec, int i, ElemTypeVc e);
    //根据用户输入替换向量中特定位置的元素
StatusVc PrintVector(Vector * pVec);
    //打印向量
StatusVc DestroyVector(Vector * pVec);
    //销毁向量
StatusVc ClearVector(Vector * pVec);
    //清空向量
StatusVc EmptyVector(Vector Vec);
    //判断向量是否为空，如果是返回 TRUE，如果否返回 FALSE
int DimensionVector(Vector Vec);
    //返回向量维数
StatusVc GetElemVector(Vector Vec, int i, ElemTypeVc * e);
```

```
    //用基本元素 e 返回向量中第 i 个元素的值
StatusVc AddVectors(Vector VecA, Vector VecB, Vector * Vec);
    //将向量 VecA 和向量 VecB 相加，并将结果存放在新的向量 Vec 中
StatusVc SubtractVectors(Vector VecA, Vector VecB, Vector * Vec);
    //将向量 VecA 和向量 VecB 相减，并将结果存放在新的向量 Vec 中
double MultiplyVectors(Vector VecA, Vector VecB);
    //求向量 VecA 和向量 VecB 的乘积
double LengthVector(Vector Vec);
    //求向量的模长

double CosineVectors(Vector VecA, Vector VecB);
    //求向量 VecA 和向量 vecB 的夹角余弦值
```

其中部分操作的伪码算法如下：

```
Status InitVector(Vector &Vec)
{   //构建一个空的向量
    pVec->elem=(ElemTypeVc *)malloc(sizeof(ElemTypeVc)*LIST_INIT_SIZE);
    if(pVec != NULL)
    {
        pVec->length=0;                 //空表长度（元素个数）为零
        pVec->listsize=LIST_INIT_SIZE;  //将表的初始容量设定为 LIST_INIT_LIST
        return OK;
    }
    else return ERROR;
} //InitVector

Status InsertVector(Vector &Vec, int i, ElemType e)
{   //根据用户输入向向量中插入元素
    if(pVec->length==pVec->listsize)  //如果表已满则进行扩容
    {ExtendVector(pVec,1);}
    if(i<1||i>pVec->length+1)     //判断插入位置是否正确
        return ERROR;
    else
    {
```

```
        for(j=pVec->length-1;j>=i-1;j--)        //向右挪动元素
            pVec->elem[j+1]=pVec->elem[j];
        pVec->elem[i-1]=e;
        pVec->length++;
        return OK;
    }
} //InsertVector

Status CreateVector(Vector &Vec)
{   //根据用户输入的维数创建向量
    scanf("%d",&dim);
    for(i=0;i<dim;i++)
    {
        scanf("%f",&input);
        InsertVector(pVec, i+1, input);
    }
    return OK;
} //CreateVector

Status ClearVector(Vector &Vec)
{   //清空一个向量
    len=pVec->length;
    for(i=1;i<len+1;i++)
        DeleteVector(&Vec, i, &e);
    return OK;
} //ClearVector

Status AddVectors(Vector VecA, Vector VecB, Vector &Vec)
{   //将向量 VecA 与向量 VecB 相加，并将结果存放在向量 Vec 中
    if(VecA.length!=VecB.length)
        return ERROR;
    len=VecA.length;
    for(i=0;i<len;i++)
```

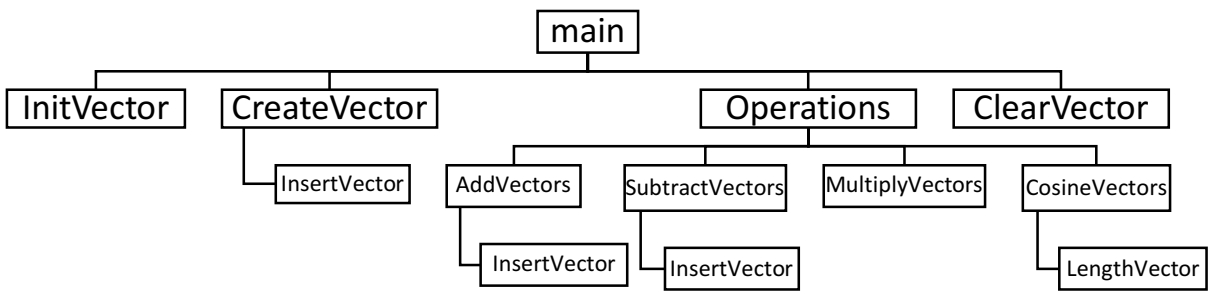
```
    {
        sum=VecA.elem[i]+VecB.elem[i];   //将对应位置的元素相加，并插入向量Vec
        InsertVector(Vec, i+1, sum);
    }
    return OK;
} //AddVectors

Status SubtractVectors(Vector VecA, Vector VecB, Vector &Vec)
{   //将向量VecA与向量VecB相减，并将结果存放在向量Vec中
    if(VecA.length!=VecB.length)
        return ERROR;
    len=VecA.length;
    for(i=0;i<len;i++)
    {
        sum=VecA.elem[i]-VecB.elem[i];   //将对应位置的元素相减，并插入向量Vec
        InsertVector(Vec, i+1, sum);
    }
    return OK;
} //AddVectors

double MultiplyVectors(Vector VecA, Vector VecB)
{   //求两个向量的乘积
    if(VecA.length!=VecB.length)
        return ERROR;
    else
    {
        for(int i=0;i<VecLength(VecA);i++)
            sum+=VecA.elem[i]*VecB.elem[i];
        return sum;
    }
} //MultiplyVectors

double CosineVectors(Vector VecA, Vector VecB)
```

```
{   //求两个向量的夹角余弦值
    if(VecA.length!=VecB.length)
        return ERROR;
    else
    {
        lengthA = LengthVector(VecA);
        lengthB = LengthVector(VecB);
        product = MultiplyVectors(VecA, VecB);
        cosine = product/(lengthA*lengthB);
        return cosine;
    }
}
```

3. 主函数的伪码算法：

```
void main()
{  //主函数
    InitVector(&Vec);InitVector(&VecA);InitVector(&VecB); //初始化
    CreateVector(&VecA);  //根据用户输入创建向量A
    CreateVector(&VecB);  //根据用户输入创建向量B
    AddVectors(VecA, VecB, &Vec);  //向量加法
    ClearVector(&Vec);    //清除向量Vec
    SubtractVectors(VecA, VecB, &Vec);  //向量减法
    ClearVector(&Vec);    //清除向量Vec
    MultiplyVectors(VecA, VecB);    //向量乘法
    CosineVectors(VecA, VecB);      //向量夹角余弦值
} //main
```

4. 函数的调用关系图反映了演示程序的层次结构：



模块 II 用顺序表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 元素类型、结点类型、多项式类型

```
typedef float ElemTypeSq;
typedef struct {
    ElemTypeSq  coefSq;
    ElemTypeSq  expoSq;
} TermSq;
typedef struct {
    TermSq          * termSq;
    int             lengthSq;  //一元多项式的项数
    int           listsizeSq;  //当前分配的存储容量
} PolynomialSq;
```

2. 一元多项式的基本操作设置如下：

```
Status InitPolynomialSq(PolynomialSq * pPolySq);
    //构建一个空的用于存放一元多项式的顺序表
Status ExtendPolynomialSq(PolynomialSq * pPolySq, int n);
    //扩充存放一元多项式的顺序表的容量
Status InsertPolynomialSq(PolynomialSq * pPolySq, int i, ElemTypeSq coefSq,
ElemTypeSq expoSq);
    //根据用户输入向顺序表中插入多项式的项
Status InsertPolynomialOrderSq(PolynomialSq * pPolySq, ElemTypeSq coefSq,
ElemTypeSq expoSq);
    //按指数升序插入新的项，如果指数相同，则合并同类项
Status DeletePolynomialSq(PolynomialSq * pPolySq, int i, ElemTypeSq *
coefSq, ElemTypeSq * expoSq);
    //根据用户输入删除一元多项式中特定的项
Status ReplacePolynomialSq(PolynomialSq * pPolySq, int i, ElemTypeSq coefSq,
ElemTypeSq expoSq);
    //根据用户输入替换一元多项式中特定的项
Status PrintPolynomialSq(PolynomialSq * pPolySq);
    //输出一元多项式
Status DestroyPolynomialSq(PolynomialSq * pPolySq);
```

```
    //销毁一个存放一元多项式的顺序表
Status ClearPolynomialSq(PolynomialSq * pPolySq);
    //清空一个存放一元多项式的顺序表
Status EmptyPolynomialSq(PolynomialSq PolySq);
    //判别一元多项式是否含有项
Status LengthPolynomialSq(PolynomialSq PolySq);
    //返回一元多项式的项数
Status GetTermPolynomialSq(PolynomialSq PolySq, int i, ElemTypeSq * coefSq,
ElemTypeSq * expoSq);
    //用基本元素 coef 和 expo 返回一元多项式中第 i 项的值
Status AddPolynomialsSq(PolynomialSq * pPolySqA, PolynomialSq * pPolySqB,
PolynomialSq * pPolySq);
    //将多项式 A 和多项式 B 相加，把结果存放在另一个多项式中
Status SubtractPolynomialsSq(PolynomialSq * pPolySqA, PolynomialSq *
pPolySqB, PolynomialSq * pPolySq);
    //将多项式 A 和多项式 B 相减，把结果存放在另一个多项式中
Status MultiplyPolynomialsSq(PolynomialSq * pPolySqA, PolynomialSq *
pPolySqB, PolynomialSq * pPolySq);
    //将多项式 A 和多项式 B 相乘，把结果存放在一个新的多项式中
Status DerivativePolynomialSq(PolynomialSq * pPolySq, PolynomialSq *
pPolySq1);
    //多项式求导，结果存放在顺序表 PolySq1 中
```

其中部分操作的伪码算法如下：

```
Status InitPolynomialSq(PolynomialSq * pPolySq)
{   //构建一个空的用于存放一元多项式的顺序表
    pPolySq->termSq=(TermSq *)malloc(sizeof(TermSq)*LIST_INIT_SIZE);
    if(pPolySq != NULL)
    {
        pPolySq->lengthSq=0;                //空表长度（元素个数）为零
        pPolySq->listsizeSq=LIST_INIT_SIZE;  //将表的初始容量设定为
LIST_INIT_LIST
        return OK;
    }
```

```
    else
        exit(OVERFLOW);
} //InitPolynomialSq

Status InsertPolynomialOrderSq(PolynomialSq * pPolySq, ElemTypeSq coefSq,
ElemTypeSq expoSq)
{   //按指数升序插入新的项，如果指数相同，则合并同类项
    if(pPolySq->lengthSq==0)   //如果还没有项，则把新项插在第一个元素位置
        InsertPolynomialSq(pPolySq, 1, coefSq, expoSq);
    else
    {
        for(int j=1;j<pPolySq->lengthSq+1;j++)  //遍历现有的一元多项式
            if(expoSq<pPolySq->termSq[j-1].expoSq)
            //如果指数小于正在比较的项的指数，则把新项插在正在比较的项的位置上
            {
                InsertPolynomialSq(pPolySq, j, coefSq, expoSq);
                break;
            }
            else if(expoSq==pPolySq->termSq[j-1].expoSq)
            //如果指数相同，则合并同类项
            {
                pPolySq->termSq[j-1].coefSq+=coefSq;
                break;
            }
            else if(expoSq>pPolySq->termSq[j-1].expoSq)
            //如果指数大于正在比较的项的指数
            {
                if(j==pPolySq->lengthSq)   //如果已经到了最后一项，则把新项插在表尾
                {InsertPolynomialSq(pPolySq, j+1, coefSq, expoSq);break;}
                else   //如果不是最后一项，则与下一项进行比较
                    continue;
            }
    }
```

```
    return OK;
} //InsertPolynomialOrderSq

Status AddPolynomialsSq(PolynomialSq * pPolySqA, PolynomialSq * pPolySqB,
PolynomialSq * pPolySq)
{   //将多项式A和多项式B相加，把结果存放在多项式A的顺序表中，并输出新的多项式A
    lenA=pPolySqA->lengthSq;
    lenB=pPolySqB->lengthSq;
    for(int i=0;i<lenA;i++)
    {
        coefSq=pPolySqA->termSq[i].coefSq;
        expoSq=pPolySqA->termSq[i].expoSq;
        InsertPolynomialOrderSq(pPolySq, coefSq, expoSq);
    }
    for(int j=0;j<lenB;j++)
    {
        coefSq=pPolySqB->termSq[j].coefSq;
        expoSq=pPolySqB->termSq[j].expoSq;
        InsertPolynomialOrderSq(pPolySq, coefSq, expoSq);
    }
    return OK;
} //AddPolynomialsSq

Status SubtractPolynomialsSq(PolynomialSq * pPolySqA, PolynomialSq *
pPolySqB, PolynomialSq * pPolySq)
{   //将多项式A和多项式B相减，把结果存放在多项式A的顺序表中，并输出新的多项式A
    lenA=pPolySqA->lengthSq;
    lenB=pPolySqB->lengthSq;
    for(i=0;i<lenA;i++)
    {
        coefSq=pPolySqA->termSq[i].coefSq;
        expoSq=pPolySqA->termSq[i].expoSq;
        InsertPolynomialOrderSq(pPolySq, coefSq, expoSq);
```

```
    }
    for(j=0;j<lenB;j++)
    {
        coefSq=pPolySqB->termSq[j].coefSq;
        coefSq=-coefSq;
        expoSq=pPolySqB->termSq[j].expoSq;
        InsertPolynomialOrderSq(pPolySq, coefSq, expoSq);
    }
    return OK;
} //SubtractPolynomialsSq

Status MultiplyPolynomialsSq(PolynomialSq * pPolySqA, PolynomialSq *
pPolySqB, PolynomialSq * pPolySq)
{   //将多项式A 和多项式B 相乘，把结果存放在一个新的多项式中
    lenA=pPolySqA->lengthSq;
    lenB=pPolySqB->lengthSq;
    for(i=0;i<lenA;i++)
    //将多项式A 的每一项与多项式B 的每一项分别相乘，并按升序插入多项式C 中
    {
        coefSqA = pPolySqA->termSq[i].coefSq;
        expoSqA = pPolySqA->termSq[i].expoSq;
        for(j=0;j<lenB;j++)
        {
            coefSqB = pPolySqB->termSq[j].coefSq;
            expoSqB = pPolySqB->termSq[j].expoSq;
            coefSqC = coefSqA*coefSqB;
            expoSqC = expoSqA+expoSqB;
            InsertPolynomialOrderSq(pPolySq, coefSqC, expoSqC);
        }
    }
    return OK;
} //MultiplyPolynomials
```

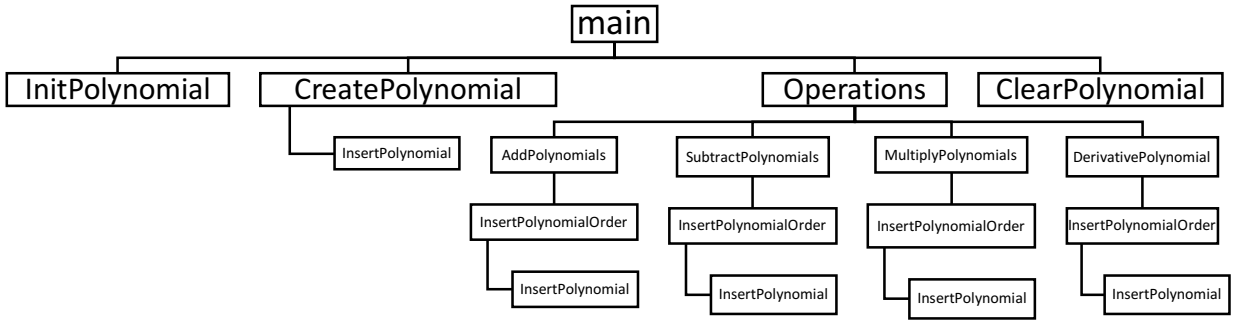
```
Status DerivativePolynomialSq(PolynomialSq * pPolySq, PolynomialSq *
pPolySq1)
{   //多项式求导，结果存放在顺序表 PolySq1 中
    numTerm = pPolySq->lengthSq;
    for(i=0;i<numTerm;i++)
    {
        if(pPolySq->termSq[i].expoSq==0)
            i++;
        else
        {
            coefSq=pPolySq->termSq[i].coefSq*pPolySq->termSq[i].expoSq;
            expoSq=pPolySq->termSq[i].expoSq-1;
            InsertPolynomialOrderSq(pPolySq1, coefSq, expoSq);
        }
    }
    return OK;
} //DerivativePolynomialSq
```

3. 主函数的伪码算法：

```
void main()
{  //主函数
    //初始化
    InitPolynomial(&Poly);InitPolynomial(&PolyA);InitPolynomial(&PolyB);
    CreatePolynomial(&PolyA);  //根据用户输入创建一元多项式 A
    CreatePolynomial(&PolyB);  //根据用户输入创建一元多项式 B
    AddPolynomials(PolyA, PolyB, &Poly);  //一元多项式加法
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    SubtractPolynomials(PolyA, PolyB, &Poly);  //一元多项式减法
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    MultiplyPolynomials(PolyA, PolyB, &Poly);    //一元多项式乘法
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    DerivativePolynomial(PolyA, &Poly);    //一元多项式 A 求导
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    DerivativePolynomial(PolyA, &Poly);    //一元多项式 B 求导
```

```
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
} //main
```

4. 函数的调用关系图反映了演示程序的层次结构：



模块 III 用单链表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 元素类型、顺序表结构

```
typedef float ElemTypeLk;  //元素类型
typedef struct TermLk {
    ElemTypeLk        coefLk;   //系数
    ElemTypeLk        expoLk;   //指数
    struct TermLk    *next;   //指针域
} TermLk;    //多项式的项
typedef struct {
    int           lengthLk;   //项数
    TermLk          * head;   //头结点
} PolynomialLk;
```

2. 单链表一元多项式的基本操作设置如下：

```
Status InitPolynomialLk(PolynomialLk * pPolyLk);
    //初始化用于存放一元多项式的链表
Term * CreateTermLk(ElemTypeLk coef,ElemTypeLk expo);
    //创建新的项，并返回项的存储地址
Status PushHeadLk(PolynomialLk * pPolyLk, ElemTypeLk coefLk, ElemTypeLk
expoLk);
    //在单链表头部插入项
```

```
Status PushTailLk(PolynomialLk * pPolyLk, ElemTypeLk coefLk, ElemTypeLk
expoLk);
    //在单链表尾部插入项
Status DeleteHeadLk(PolynomialLk * pPolyLk);
    //删除单链表的头结点
Status DeleteTailLk(PolynomialLk * pPolyLk);
    //删除单链表的尾结点
Status InsertPolynomialLk(PolynomialLk * pPolyLk, int i, ElemTypeLk coefLk,
ElemTypeLk expoLk);
    //在任意位置插入项
Status InsertPolynomialOrderLk(PolynomialLk * pPolyLk, ElemTypeLk coefLk,
ElemType expoLk);
    //根据项的指数大小进行升序插入，如果指数相同则进行同类项合并
Status DeletePolynomialLk(PolynomialLk * pPolyLk, int i);
    //删除一元多项式的任意项
Status PrintPolynomialLk(PolynomialLk * pPolyLk);
    //打印输出一元多项式
int LengthPolynomialLk(PolynomialLk * pPolyLk);
    //返回多项式项数
Status ClearPolynomialLk(PolynomialLk * pPolyLk);
    //清空存放一元多项式的链表
Status DestroyPolynomialLk(PolynomialLk * pPolyLk);
    //销毁存放一元多项式的链表
Status ReplacePolynomialLk(PolynomialLk * pPolyLk, int i, ElemTypeLk coefLk,
ElemTypeLk expoLk);
    //根据用户输入替换一元多项式中的某一项
Status EmptyPolynomialLk(PolynomialLk PolyLk);
    //判别一个一元多项式是否为空，返回 TRUE 或者 FALSE
Status GetTermPolynomialLk(PolynomialLk * pPolyLk, int i, ElemTypeLk *
coefLk, ElemTypeLk * expoLk);
    //根据用户输入获取一元多项式中的第 i 项，并将其返回到(coef,expo)
Status AddPolynomialsLk(PolynomialLk * pPolyLkA, PolynomialLk * pPolyLkB,
PolynomialLk * pPolyLk);
```

```
    //将多项式A与多项式B相加，将结果存放在另一个多项式中
Status SubtractPolynomialsLk(PolynomialLk * pPolyLkA, PolynomialLk *
pPolyLkB, PolynomialLk * pPolyLk);
    //将多项式A与多项式B相减，将结果存放在另一个多项式中
Status MultiplyPolynomialsLk(PolynomialLk * pPolyLkA, PolynomialLk *
pPolyLkB, PolynomialLk * pPolyLk);
    //将多项式A和多项式B相乘，将结果存放在另一个多项式中
Status DerivativePolynomialLk(PolynomialLk * pPolyLk, PolynomialLk *
pPolyLk1);
    //多项式求导，将结果存放在另一个多项式中
```

其中部分操作的伪码算法如下：

```
Status InitPolynomialLk(PolynomialLk * pPolyLk)
{   //初始化用于存放一元多项式的链表
    pPolyLk->head = (TermLk *)malloc(sizeof(TermLk));
    pPolyLk->head->next = NULL;
    pPolyLk->lengthLk = 0;
    return OK;
} //InitPolynomialLk

TermLk * CreateTermLk(ElemTypeLk coef,ElemTypeLk expo)
{   //创建新的项，并返回项的存储地址
    TermLk *pNew=(TermLk *)malloc(sizeof(TermLk));   //为新结点开辟空间
    pNew->coefLk=coef;
    pNew->expoLk=expo;
    pNew->next=NULL;
    return pNew;
} //CreateTermLk

Status InsertPolynomialLk(PolynomialLk * pPolyLk, int i, ElemTypeLk coefLk,
ElemTypeLk expoLk)
{   //在任意位置插入项
    //判断插入位置是否合法
    if (i<1 || i>pPolyLk->lengthLk)  //若插入位置不合法，则插入到尾部
```

```
    {
        PushTailLk(pPolyLk, coefLk, expoLk);
        return OK;
    }
    //头插
    else if (i == 1)
    {
        TermLk *pn = CreateTermLk(coefLk, expoLk);
        pn->next = pPolyLk->head->next;
        pPolyLk->head->next = pn;
        pPolyLk->lengthLk++;
        return OK;
    }
    //中间任意位置插入
    else
    {
        TermLk *pn = CreateTermLk(coefLk, expoLk);
        TermLk *pm = pPolyLk->head->next;
        for (int j=1; j<i-1; j++)
        {
            pm = pm->next;
        }
        pn->next = pm->next;
        pm->next = pn;
        pPolyLk->lengthLk++;
        return OK;
    }
} //InsertPolynomialLk

Status DeletePolynomialLk(PolynomialLk * pPolyLk, int i)
{   //删除一元多项式的任意项
    //判断删除位置是否合法
    if (i<1 || i>pPolyLk->lengthLk)       //如果位置不合法，则删除最后一项
```

```
    {
        DeleteTailLk(pPolyLk);
        return OK;
    }
    TermLk *pn = pPolyLk->head->next;
    if (i == 1)     //头删
    {
        pPolyLk->head->next = pPolyLk->head->next->next;
        free(pn);
        pPolyLk->lengthLk--;
        return OK;
    }
    //任意位置删除
    for (int j = 1; j < i-1; j++)
    {
        pn = pn->next;
    }
    TermLk *pm = pn->next;
    pn->next = pn->next->next;
    free(pm);
    pm = NULL;
    pPolyLk->lengthLk--;
    return OK;
} //DeletePolynomialLk

Status InsertPolynomialOrderLk(PolynomialLk * pPolyLk, ElemTypeLk coefLk,
ElemTypeLk expoLk)
{   //根据输入的指数大小进行元素插入，如果指数相同则进行同类项合并
    if(pPolyLk->lengthLk==0)
        PushHeadLk(pPolyLk, coefLk, expoLk);
    else
    {
        TermLk *pm = pPolyLk->head->next;
```

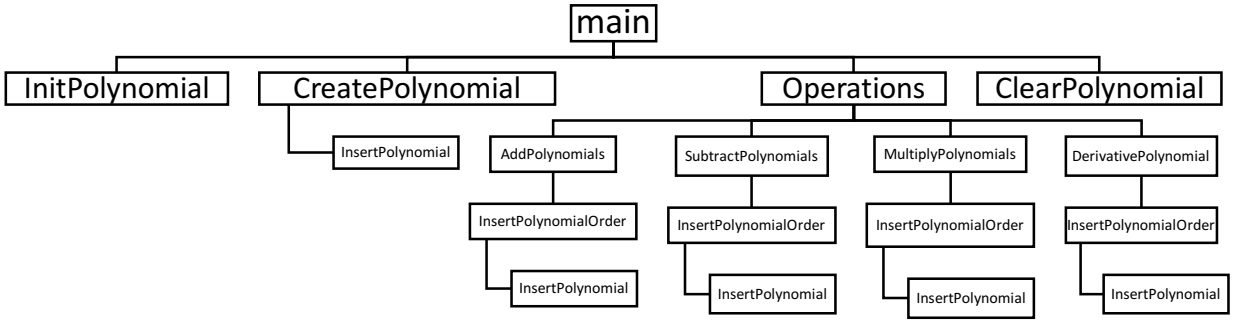
```
        for(int i=1;i<pPolyLk->lengthLk+1;i++)
        {
            if(expoLk<pm->expoLk)
            {
                InsertPolynomialLk(pPolyLk, i, coefLk, expoLk);
                break;
            }
            else if(expoLk==pm->expoLk)
            {
                pm->coefLk+=coefLk;
                break;
            }
            else if(expoLk>pm->expoLk)
            {
                if(pm->next==NULL)
                {
                    PushTailLk(pPolyLk, coefLk, expoLk);
                }
                else
                    pm=pm->next;
            }
        }
    }
    return OK;
} //InsertPolynomialOrderLk
```

3. 主函数的伪码算法：

```
void main()
{  //主函数
    //初始化
    InitPolynomial(&Poly);InitPolynomial(&PolyA);InitPolynomial(&PolyB);
    CreatePolynomial(&PolyA);  //根据用户输入创建一元多项式A
    CreatePolynomial(&PolyB);  //根据用户输入创建一元多项式B
    AddPolynomials(PolyA, PolyB, &Poly);  //一元多项式加法
```

```
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    SubtractPolynomials(PolyA, PolyB, &Poly);  //一元多项式减法
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    MultiplyPolynomials(PolyA, PolyB, &Poly);    //一元多项式乘法
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    DerivativePolynomial(PolyA, &Poly);    //一元多项式 A 求导
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
    DerivativePolynomial(PolyA, &Poly);    //一元多项式 B 求导
    ClearPolynomial(&Poly);    //清除一元多项式 Poly
} //main
```

4. 函数的调用关系图反映了演示程序的层次结构：



模块 IV ⑴四则运算表达式求值。操作符包括加(‘+’)、减(‘-’)、乘(‘*’)、除(‘/’)、幂(‘^’)、左括号(‘(’)、右括号(‘)’)，而操作数则包括整数、浮点数等不同类型的数值。比如“30+4*2.5”，得到 40 或 40.0 等形式的结果。⑵含单变量的表达式求值。变量可以是 C/C++的标识符。比如“3+4*X2”，需要输入变量 X2 的值，然后计算结果。数字形式包括用科学计数法表示的，如 1.14e2，以及负数等。

要进行表达式运算，需要一个抽象数据类型：栈。

1. 栈的元素及数据类型：

```
typedef float OpndType;     //操作数元素类型
typedef struct {
    OpndType    * base;   //在栈构造之前和销毁之后，base 的值为 NULL
    OpndType    *  top;   //栈顶指针
    int      stacksize;   //当前已分配的存储空间，以元素为单位
} OpndStack;              //操作数栈
```

```
typedef char OptrType;    //操作符元素类型
typedef struct {
    OptrType    * base;   //在栈构造之前和销毁之后，base 的值为 NULL
    OptrType    *  top;   //栈顶指针
    int      stacksize;   //当前已分配的存储空间，以元素为单位
} OptrStack;              //操作符栈
```

2. 项的结构及数据类型：

```
typedef struct {
    char    term[30];
    int     termType;    //
} Term;
typedef struct {
    Term   *termList[TERMNUM];
    int             termNum;
} Terms;
```

3. 未知变量的结构及数据类型：

```
typedef struct {
    char   VarName[20];   //变量名
    int     IsAssigned;   //是否已经被赋值
    float       VarVal;   //函数值
} Var;                    //变量
typedef struct {
    Var    *VarList[VarListNum];   //用一组连续的存储单元存储一组 Var
    int                NumVar;   //变量个数
} Vars;                           //变量库
```

4. 表达式的基本操作设置如下：

```
void ConvertStandard(char *expression);
    //将用户输入的表达式转化为标准的算术式
Status In(char c,char OP[]);
    //判断输入的 c 是否属于某一字符数组
Status InitTerm(Term * Term);
    //对项进行初始化操作
```

```
Status InitTerms(Terms * Terms);
    //对项库进行初始化操作
int TermType(char * Term);
    //返回项的类别，1-运算数，2-未知数，3-运算符
Status Fragmentation(char *expression, Terms * Terms);
    //对表达式进行分割处理，将各项存入项库
float ConvertNum(char * Num);
    //将字符串转化为数字
OptrType Precede(OptrType a, OptrType b);
    //比较 a 和 b 的优先级
OpndType Operate(OpndType a, OptrType theta, OpndType b);
    //基本的加减乘除幂操作
OpndType GetVal(char *c, Vars * Vars);
    //获取用户输入的数字
OpndType EvaluateExpression(char *expression);
    //解析表达式，进行运算
```

一些伪码算法如下：

```
void ConvertStandard(char *expression)
{
    //将字符串表达式标准化为算术表达式
    *p=expression;
    while(*p!='\0')
    {   //遍历字符串
        if(*p==' ')
        {   //如果是空格，删除
            q=p;
            for(;*q!='\0';q++)
                *q=*(q+1);
        }
        p++;
    }
    *p++='#';  //在表达式末尾加上'#'以与最先压入算符栈的'#'匹配
    *p='\0';
```

```
} //ConvertStandard


float ConvertNum(char * Num)
{   //将字符串转化为数字
    len=strlen(Num);
    temp[30]={0};    //用于存放数字部分
    //对数字字符串进行预处理，得到IsMinus/MinusPos,IsDot/DotPos,IsE/EPos,digit
    for(int i=1;i<len+1;++i)
    {
        if(*c=='-')
            {IsMinus+=1;MinusPos=i;}
        else if(*c=='.')
            {IsDot+=1;DotPos=i;}
        else if(*c=='e'||*c=='E')
            {IsE+=1;EPos=i;}
        else if(isdigit(*c))
            {b=*c-'0';temp[temp_pos]=b;++temp_pos;}
        ++c;
    }
    //转化为数字
    if((IsMinus==1&&MinusPos==1)||IsMinus==2) //转换正负
    {IsNegative=-1;}
    if(IsE==0) //非科学计数法表示的数
    {
        for(pos=0;pos<len-IsDot-IsMinus;++pos)
            {sum*=10.0;sum+=temp[pos];}
        if(IsDot==1) TenPower=DotPos-len;
    }
    else if(IsE==1) //科学计数法表示的数
    {   //
        for(pos=0;pos<EPos-IsDot-ceil(IsMinus/2.0)-1;++pos)
            {sum*=10.0;sum+=temp[pos];}
        if(IsDot==1) TenPower+=DotPos-EPos+1;
```

```
        for(pos=len-IsMinus-IsDot-IsE-1;pos>EPos-IsDot-ceil(IsMinus/2.0)-2;--
pos)
        {EPower*=10;EPower+=temp[pos];}
        if(MinusPos<=1)
        {TenPower+=EPower;}
        else if(MinusPos>1)
        {TenPower-=EPower;}
    }
    sum=IsNegative*sum*pow(10, TenPower);
    return sum;
} //ConvertNum

Status Fragmentation(char *expression, Terms *Terms)
{   //对表达式进行分割处理，将各项存入项库
    *c=expression;
    //处理首元素，构建一个项
    Terms->termList[0]->term[0]=*c;
    Terms->termNum+=1;
    ++c;
    //处理后面的元素
    for(;*c!='\0';++c)
    {
        c_pre=c-1;c_post=c+1;
        if(*c=='-')  //如果当前元素为'-'
        {
            if(*c_pre!='('&&*c_pre!='e'&&*c_pre!='E')
            //如果前一个元素不为'('且不为e或E，则分割为减号项
            {
                Terms->termList[Terms->termNum]->term[strlen(Terms->termList[
Terms->termNum]->term)]='-';
                Terms->termNum+=1;
            }
            else if(*c_pre=='(')
```

```
            //如果前一个元素为'('，则分割为负数项，并且将括号删除
            {
                Terms->termList[Terms->termNum-1]->term[0]='-';
                ++c;
                for(;*c!=')';++c)
                {
                    Terms->termList[Terms->termNum-
1]->term[strlen(Terms->termList[Terms->termNum-1]->term)]=*c;
                }
            }
            else if(*c_pre=='e'||*c_pre=='E')     //如果前一个元素为 e 或 E
            {
                * c_prepre=c_pre-1;
                if(In(*c_prepre, digit))
                 //如果前前元素为数字，则分割为用科学计数法表示的常数
                {
                    Terms->termList[Terms->termNum-
1]->term[strlen(Terms->termList[Terms->termNum-1]->term)]=*c;
                    ++c;
                    for(;!In(*c, delimiter);++c)
                    {
                        Terms->termList[Terms->termNum-
1]->term[strlen(Terms->termList[Terms->termNum-1]->term)]=*c;
                    }
                    --c; //回溯指针
                }
                else                        //如果前前元素不是数字，则分割为减号项
                {
                    Terms->termList[Terms->termNum]->term[strlen(Terms->termLi
st[Terms->termNum]->term)]=*c;
                    Terms->termNum+=1;
                }
            }
```

```
        }
        //-------------------------------------------------------------//
        else if((In(*c, delimiter)&&*c!='-'))   //如果为界限符
        {
            if(*c=='('&&(In(*c_pre,digit)||In(*c_pre,letter)))  //函数体
            {
                for(;*c!=')';++c)
                {
                    Terms->termList[Terms->termNum-
1]->term[strlen(Terms->termList[Terms->termNum-1]->term)]=*c;
                }
                Terms->termList[Terms->termNum-
1]->term[strlen(Terms->termList[Terms->termNum-1]->term)]=*c;
                //--c;      //指针回溯一个元素
            }
            else
            {
                Terms->termList[Terms->termNum]->term[strlen(Terms->termList[
Terms->termNum]->term)]=*c;
                Terms->termNum+=1;
            }
        }
        //-------------------------------------------------------------//
        else if(In(*c,digit)||In(*c,letter))              //如果当前字符为标识符
        {
            if(In(*c_pre, delimiter))       //如果前一个字符为界限符
            {
                Terms->termList[Terms->termNum]->term[0]=*c;
                Terms->termNum+=1;
            }
            else if(In(*c_pre, digit)||In(*c_pre,letter))
            //如果前一个字符为标识符或数字或小数点
            {
```

```
                Terms->termList[Terms->termNum-
1]->term[strlen(Terms->termList[Terms->termNum-1]->term)]=*c;
            }
        }
    }
    return OK;
} //Fragmentation

OptrType Precede(OptrType a, OptrType b)
{   //判断运算符之间的优先级关系
    pre[8][8]={
        /*运算符之间的优先级制作成一张表格*/
             /*  +   -   *   /   ^   (   )   #  */
        /* + */{'>','>','<','<','<','<','>','>'},
        /* - */{'>','>','<','<','<','<','>','>'},
        /* * */{'>','>','>','>','<','<','>','>'},
        /* / */{'>','>','>','>','<','<','>','>'},
        /* ^ */{'>','>','>','>','>','<','>','>'},
        /* ( */{'<','<','<','<','<','<','=','0'},
        /* ) */{'>','>','>','>','>','0','>','>'},
        /* # */{'<','<','<','<','<','<','0','='}};
    switch(a){
        case '+': i=0; break;
        case '-': i=1; break;
        case '*': i=2; break;
        case '/': i=3; break;
        case '^': i=4; break;
        case '(': i=5; break;
        case ')': i=6; break;
        case '#': i=7; break;
    }
    switch(b){
        case '+': j=0; break;
```

```
        case '-': j=1; break;
        case '*': j=2; break;
        case '/': j=3; break;
        case '^': j=4; break;
        case '(': j=5; break;
        case ')': j=6; break;
        case '#': j=7; break;
    }
    return pre[i][j];      //运用在矩阵中的位置带回优先级关系'<','>'或'='
} //Precede

OpndType EvaluateExpression(char *expression)
{   //操作数为1，操作符为2，未知变量为3
    Push_OPTR(&OPTR, '#'); //向运算符栈推入'#'
    Fragmentation(expression, &Terms); //分割表达式，将各项存入项库
    for(i=0;i<Terms.termNum;i++)
        Terms.termList[i]->termType=TermType(Terms.termList[i]->term);
        //设置各项类别

    //处理第一项
    if(Terms.termList[0]->termType==3)  //运算符，压入运算符栈
        Push_OPTR(&OPTR, *Terms.termList[0]->term);
    else if(Terms.termList[0]->termType==1)  //常数，压入运算数栈
        {num=ConvertNum(Terms.termList[0]->term);Push_OPND(&OPND, num);}
    else if(Terms.termList[0]->termType==2)  //未知数，赋值并压入运算数栈
            num=AssignVar(&Vars, Terms.termList[0]->term);

    //处理后面的项
    for(i=1;i<Terms.termNum;i++)
    {
        if(Terms.termList[i]->termType==1)        //常数
            num=ConvertNum(Terms.termList[i]->term);Push_OPND(&OPND, num);
        else if(Terms.termList[i]->termType==2)    //未知数
```

```
            num=AssignVar(&Vars, Terms.termList[i]->term);
        else                                    //运算符
        {
            switch(Precede(GetTop_OPTR(OPTR,&e1),*Terms.termList[i]->term))
            //根据 c 与运算符栈栈顶元素的优先级关系进行操作
            {
                case '<':  //如果 c 优先于栈顶元素，则将 c 压入运算符栈
                    Push_OPTR(&OPTR,*Terms.termList[i]->term);
                    break;
                case '=':  //如果 c 与栈顶元素平级，则用 x 带回运算符栈栈顶元素
                    if(*Terms.termList[i]->term=='#')
                        Push_OPTR(&OPTR, *Terms.termList[i]->term);
                    Pop_OPTR(&OPTR,&x);
                    break;
                case '>':  //如果栈顶元素优先于 c
                    Pop_OPTR(&OPTR,&theta);  //用 theta 带回运算符栈顶元素
                    Pop_OPND(&OPND,&b);      //用 a,b 带回运算数栈栈顶的两个元素
                    Pop_OPND(&OPND,&a);
                    result=Operate(a,theta,b); //进行运算
                    Push_OPND(&OPND,result);   //将运算结果压入运算数栈
                    i--; //回溯一个项
                    break;
                default:
                    break;
            }//switch
        }//else
    }//for loop
    GetTop_OPND(OPND,&result);    //用 result 带回运算数栈栈顶元素，即运算结果
    return result;
}
```

5. 运算数、运算符栈的基本操作设置如下：

```
Status InitStack(Stack *S);
    //构造空栈
```

```
Status DestroyStack(Stack *S);
    //销毁栈
Status ClearStack(Stack *S);
    //清空栈中元素
Status StackEmpty(Stack S);
    //判定栈是否为空，返回布尔值
int StackLength(Stack S);
    //返回栈中元素个数
ElemType GetTop(Stack S, ElemType *e);
    //若栈不为空，则用 e 返回 S 的栈顶元素，并返回 OK；否则返回 FALSE
Status Push(Stack *S, ElemType e);
    //插入元素 e 为新的栈顶元素
Status Pop(Stack *S, ElemType *e);
    //若栈 S 不为空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK,否则返回 ERROR
Status Traverse(Stack *S);
    //遍历栈，对从栈顶到栈底的元素逐一调用 visit()函数
```

一些伪码算法如下：

```
Status InitStack(Opnd *S)
{
//构建空栈，先分配内存，把首元素地址赋给栈底指针变量，并将栈底指针变量的值赋给栈顶指针变量
    S->base = (ElemType *)malloc(STACK_INIT_SIZE*sizeof(ElemType));
    if(! S->base) exit(-2);
    S->top=S->base;
    S->stacksize=STACK_INIT_SIZE;
    return OK;
} //InitStack

ElemType GetTop(Stack S, ElemType *e)
{   //获取栈顶元素
    if(S.top == S.base) return ERROR;
    *e=*(S.top-1);
    return *e;
} //GetPop
```

```
Status Push(Stack *S,ElemType e)
{   //压栈
    if((S->top-S->base)>=S->stacksize)   //如果栈满，追加存储空间
    {
        S->base=(ElemType *)realloc(S->base,(S->stacksize +
STACKINCREMENT)*sizeof(ElemType));
        if(!S->base) exit(-2);
        S->top=S->base+S->stacksize;
        S->stacksize += STACKINCREMENT;
    }
    *(S->top) = e;     //将栈顶元素设为 e
    S->top++;          //将栈顶指针向上挪动
    return OK;
} //Push

Status Pop(Stack *S,ElemType *e)
{   //弹栈
    if(S->top==S->base) return ERROR;
    S->top--;          //将栈顶指针向下挪动
    *e = *(S->top);
    return OK;
} //Pop
```

6. 未知变量的基本操作设定如下：

```
Status InitVars(Vars * Vars);
    //初始化变量数组
Status InitVar(Var * Var);
    //初始化变量
float AssignVar(Vars * Vars, char * var);
    //为给定变量分配值
```

其中赋值的伪码算法如下：

```
float AssignVar(Vars * Vars, char * var)
{   //为给定变量分配值
```

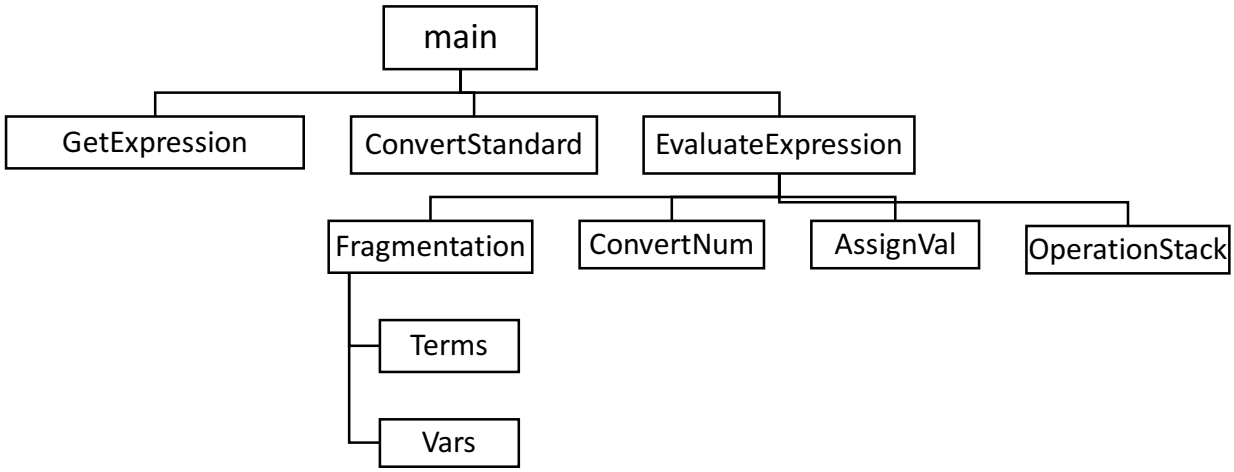
```
if(Vars->NumVar==0)   //变量库为空
{
    strcpy(Vars->VarList[0]->VarName, var);
    scanf("%s",input);
    val=ConvertNum(input);
    Vars->VarList[0]->VarVal=val;
    Vars->VarList[0]->IsAssigned=1;
    Vars->NumVar+=1;
    output=val;
}
else
{
    for(pos=0;pos<Vars->NumVar;++pos)
    {
        if(strcmp(var, Vars->VarList[pos]->VarName)==0)
        //如果该变量已经出现过
        {
            output=Vars->VarList[pos]->VarVal;
            IsAssigned=1;
        }
        else IsAssigned=0;
    }
    if(IsAssigned==0)
    {
        strcpy(Vars->VarList[Vars->NumVar]->VarName, var);
        scanf("%s",input);
        val=ConvertNum(input);
        Vars->VarList[Vars->NumVar]->VarVal=val;
        Vars->VarList[Vars->NumVar]->IsAssigned=1;
        Vars->NumVar+=1;
        output=val;
    }
}
```

```
    return output;
} //AssignVar
```

7. 主函数的伪码算法：

```
void main()
{  //主函数
    //初始化
    scanf("%s",&expression);
    ConvertStandard(&expression);
    result=EvaluateExpression(&expression);
    printf("%f\n",result);
} //main
```

8. 函数的调用关系图反映了演示程序的层次结构：



模块 V ⑴定义并运行简单函数。比如定义：f(X2)=3+4*X2，然后执行 f(5)，则得到结果 23。⑵保留函数定义历史，并可以运行历史函数。⑶函数的调用：比如已经定义了函数 f(X)，新定义函数 g(X)中调用了 f。

1. 运算数/符栈、项、未知变量和表达式的结构及实现与模块 IV 基本一致。但是如果运行复合函数，则需要进行特殊处理。假如复合函数形如$Comp(X)=m(f(X))$，例如$g(X)=7*f(X)-1/f(X)$，则先把用户键入的值代入$f(X)$进行求值，得出结果以后将其视作常数，因而整个复合函数转化为数值表达式求值；假如复合函数形如$Comp(X)=m(f(n(X)))$，例如$g(X)=7*f(X^2)-1/f(\frac{1}{X})$，则采用递归的思路，先把用户键入的值分别代入最内层函数进行求值，把结果视作常数，从而将外一层函数转化为数值表达式求值，

如此求得最外层函数的值。

具体的伪码算法如下：

```
OpndType EvaluateExpression(char * Func, char *expression, int IsFunc, float
AssignedVal)
{   //操作数为1，操作符为2，未知变量为3，函数体为4
    //IsFunc 为0 表示算术表达式，为1 表示简单函数，为2 表示复合函数
    //Func 为复合函数中被调用的函数
    //OpndStack 为运算数栈，OptrStack 为运算符栈
    Fragmentation(expression, &Terms);
    //判别项的类别
    for(i=0;i<Terms.termNum;i++)
        Terms.termList[i]->termType=TermType(Terms.termList[i]->term);
    //处理第一项
    if(Terms.termList[0]->termType==3)  //运算符，压入运算符栈
        Push_OPTR(&OPTR, *Terms.termList[0]->term);
    else if(Terms.termList[0]->termType==1)  //常数，压入运算数栈
        {num=ConvertNum(Terms.termList[0]->term);Push_OPND(&OPND, num);}
    else if(Terms.termList[0]->termType==2)  //未知数，赋值并压入运算数栈
    {
        if(IsFunc==0)          //算术表达式
            num=AssignVar(&Vars, Terms.termList[0]->term);
        else if(IsFunc==1||IsFunc==2)     //函数解析式
            num=AssignedVal;
        Push_OPND(&OPND, num);
    }
    else if(Terms.termList[0]->termType==4)   //如果是函数体
    {
        GetSubTerm(Terms.termList[0]->term, subfuncname, subfuncbody);
        ConvertStandard(subfuncbody);
        num=EvaluateExpression(Func, subfuncbody, IsFunc, AssignedVal);
        //递归调用
        strcpy(chosenfuncbody, Func);
        ConvertStandard(chosenfuncbody);
```

```
        num=EvaluateExpression(Func, chosenfuncbody, IsFunc, num);
        //递归调用
        Push_OPND(&OPND, num);
    }
    //处理后面的项
    for(i=1;i<Terms.termNum;i++)
    {
        if(Terms.termList[i]->termType==1)          //常数
            {num=ConvertNum(Terms.termList[i]->term);Push_OPND(&OPND, num);}
        else if(Terms.termList[i]->termType==2)     //未知数
        {
            if(IsFunc==0)
                num=AssignVar(&Vars, Terms.termList[i]->term);
            else if(IsFunc==1||IsFunc==2)
                num=AssignedVal;
            Push_OPND(&OPND, num);
        }
        else if(Terms.termList[i]->termType==4)   //如果是函数体
        {
            GetSubTerm(Terms.termList[i]->term, subfuncname, subfuncbody);
            ConvertStandard(subfuncbody);
            num=EvaluateExpression(Func, subfuncbody, IsFunc, AssignedVal);
            //递归调用
            strcpy(chosenfuncbody, Func);
            ConvertStandard(chosenfuncbody);
            num=EvaluateExpression(Func, chosenfuncbody, IsFunc, num);
            //递归调用
            Push_OPND(&OPND, num);
        }
        else                                        //运算符
        {
            switch(Precede(GetTop_OPTR(OPTR,&e1),*Terms.termList[i]->term))
            //根据 c 与运算符栈栈顶元素的优先级关系进行操作
```

```
            {
                case '<':  //如果 c 优先于栈顶元素，则将 c 压入运算符栈
                    Push_OPTR(&OPTR,*Terms.termList[i]->term);
                    break;
                case '=':  //如果 c 与栈顶元素平级，则用 x 带回运算符栈栈顶元素
                    if(*Terms.termList[i]->term=='#')
                        Push_OPTR(&OPTR, *Terms.termList[i]->term);
                    Pop_OPTR(&OPTR,&x);
                    break;
                case '>':  //如果栈顶元素优先于 c
                    Pop_OPTR(&OPTR,&theta);  //用 theta 带回运算符栈顶元素
                    Pop_OPND(&OPND,&b);      //用 a,b 带回运算数栈栈顶的两个元素
                    Pop_OPND(&OPND,&a);
                    result=Operate(a,theta,b); //进行运算
                    Push_OPND(&OPND,result);   //将运算结果压入运算数栈
                    i--; //回溯一个项
                    break;
                default:
                    break;
            }//switch
        }//else
    }//for loop
    GetTop_OPND(OPND,&result);    //用 result 带回运算数栈栈顶元素，即运算结果
    return result;
}

int TermType(char * Term)
{   //1-运算数，2-未知数，3-运算符，4-函数体
    c=Term;len=strlen(Term);
    if(In(*c,delimiter)&&strlen(c)==1)
        {type=3;}
    else
    {
```

```
        for(i=0;i<len;++i)
        {
            if(In(*c,digit))
                {IsDigit+=1;++c;}
            else if(In(*c,letter))
            {
                if(*c=='e'||*c=='E') {IsE+=1;IsLetter+=1;}
                else IsLetter+=1;
                ++c;
            }
            else if(*c=='-')
                {IsMinus+=1;++c;}
            else if(*c=='(')
                {LeftBracket+=1;++c;}
            else if(*c==')')
                {RightBracket+=1;++c;}
            else ++c;
        }
        c=Term;  //指针回溯
        if(IsDigit==strlen(c))   //纯数字
            type=1;
        else if((IsMinus==1)&&(IsDigit==strlen(c)-1)) //负数
            type=1;
        else if(IsLetter==1&&IsE==1&&IsMinus<=2)  //科学计数法
            type=1;
        else if(LeftBracket>=1&&LeftBracket==RightBracket)
            type=4;     //函数体
        else type=2;    //未知变量
    }
    return type;
} //TermType
```

2. 函数、函数库的数据类型及结构如下：

```
typedef struct FuncUnit {
```

```
    char      funcname[VARNAMESIZE];  //用于存放函数名
    char            func[FUNCSIZE];  //用于存放函数体
    char          var[VARNAMESIZE];  //存放变量名
    int                   FuncType;  //函数类型：1-简单函数，2-复合函数
    char        compo[VARNAMESIZE];  //若为复合函数，则用来存放被调用的函数名
    struct FuncUnit      *next;  //指针域
} FuncUnit;    //一个函数
typedef struct {
    int             length;   //储藏的函数个数
    FuncUnit        * head;   //头指针
} FuncWH;     //函数库
```

插入、查询、删除函数等操作与单链表中插入、查询、删除等操作一致。主要区别在于创建函数，具体伪码算法如下：

```
FuncUnit * CreateFuncUnit(char * FuncName, char * Func, char * VarName)
{   //创建新的函数
    pNew=(FuncUnit *)malloc(sizeof(FuncUnit));   //为新结点开辟空间
    pFuncName=FuncName;pfuncname=pNew->funcname;
    while((*pfuncname++=*pFuncName++)); //复制函数名
    pFunc=Func;pfunc=pNew->func;
    while((*pfunc++=*pFunc++));  //将输入的函数复制到函数库的单元格中
    pVar=VarName;pvar=pNew->var;
    while((*pvar++=*pVar++));   //复制变量名
    pNew->next=NULL;
    return pNew;
} //CreateFuncUnit
```

另外，由于复合函数中包含被调用函数，需要对被调用函数进行特殊处理，具体的伪码算法如下：

```
Status GetSubFunc(char * Func, char * SubFuncName, char * SubFunc)
{   //获取被调用函数信息
    spl[]="()";
    result=strtok(Func, spl);
    while(result!=NULL)
        {strcpy(dst[n++], result);result = strtok(NULL, spl);}
```

```
    strcpy(SubFuncName, dst[0]);
    strcpy(SubFunc, dst[1]);
    return OK;
} //GetSubFunc
```

函数及函数库的基本操作设置如下：

```
FuncUnit * DefSimple(void);
    //定义简单函数
float RunSimple(FuncUnit * Func);
    //运行简单函数
FuncUnit * DefCompo(FuncWH FuncWH);
    //定义复合函数
float RunCompo(FuncUnit * Func, FuncWH FuncWH);
    //运行复合函数
```

具体的伪码算法如下：

```
FuncUnit * DefSimple()
{
    scanf("%s %s",FuncName,VarName);
    scanf("%s",Func);
    SimpleFunc=CreateFuncUnit(FuncName, Func, VarName);
    SimpleFunc->FuncType=1;  //简单函数
    return SimpleFunc;
} //DefSimple

float RunSimple(FuncUnit * Func)
{   //运行简单函数
    ConvertStandard(Func->func);
    scanf("%s",input);
    VarVal=ConvertNum(input);
    result=EvaluateExpression(input,Func->func,1,VarVal);
    return result;
} //RunSimple

FuncUnit * DefCompo(FuncWH FuncWH)
```

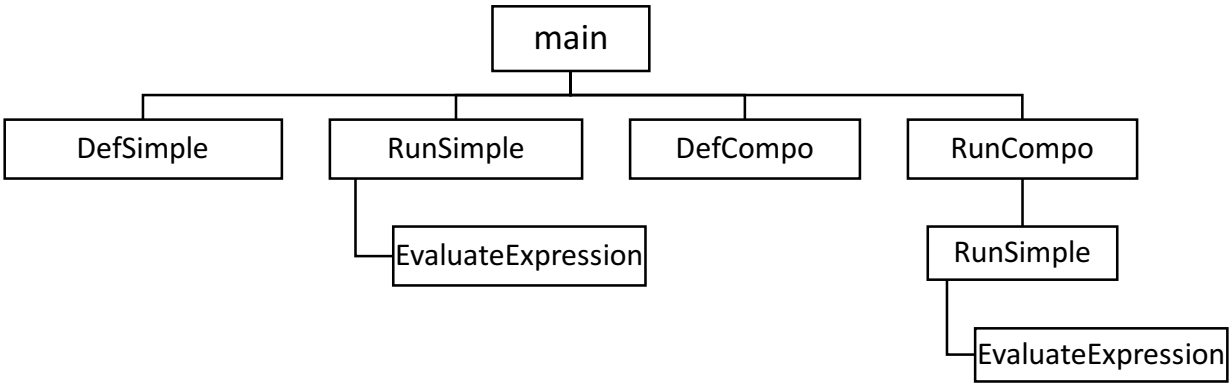
```
{   //
    PrintFuncWH(&FuncWH);     //打印出历史存储函数
    scanf("%d",&choice);      //选择调用函数
    ChosenFunc=GetFuncUnit(&FuncWH, choice);
    scanf("%s %s",FuncName,VarName); //复合函数名、变量名
    scanf("%s",Func); //复合函数体
    CompoFunc=CreateFuncUnit(FuncName, Func, VarName);
    CompoFunc->FuncType=2;   //复合函数
    strcpy(CompoFunc->compo, ChosenFunc->funcname);  //调用的函数名
    return CompoFunc;
} //DefCompo

float RunCompo(FuncUnit * Func, FuncWH FuncWH)
{   //运行复合函数，获得三个参数用于计算
    strcpy(MainFunc, Func->func);
    for(i=0;i<FuncWH.length;++i)
    {
        pTemp=GetFuncUnit(&FuncWH, i+1);
        if(strcmp(Func->compo,pTemp->funcname)==0)
            strcpy(SubFunc, pTemp->func);
    }
    scanf("%s",input);           //输入自变量的值
    VarVal=ConvertNum(input);    //转换成可运算的浮点数
    ConvertStandard(MainFunc);
    result=EvaluateExpression(SubFunc, MainFunc, 2, VarVal);
    return result;
} //RunCompo
```

3. 主函数的伪码算法：

```
void main()
{  //主函数
    f=DefSimple();
    RunSimple(f);
    Save(f);
```

```
    g=DefCompo(f);
    RunCompo(g);
} //main
```

4. 函数的调用关系图反映了演示程序的层次结构：



模块 VI 矩阵运算。实现矩阵的加、减、乘、转置、行列式求值运算。

1. 三元组的数据类型及结构如下：

```
typedef int ElemType;
typedef struct {
    int       i,j;
    ElemType    e;
} Triple;       //三元组稀疏矩阵
```

2. 稀疏矩阵的数据类型及结构如下：

```
typedef struct {
    Triple     *data[MAXSIZE+1];
    int                  taken;  //已插入的三元组个数
    int           rpos[MAXRC+1];  //每行首个非零元在改行的相对位置
    int            num[MAXRC+1];  //每行非零元个数
    int              mu,nu,tu;
} TSMatrix;
```

稀疏矩阵的基本操作设置如下：

```
Status InitSMatrix(TSMatrix *M);
    //初始化稀疏矩阵
```

```
Triple * CreateTriple(int i, int j, ElemType e);
    //创建三元组
Status InsertOrderSMatrix(TSMatrix *M, Triple * triple);
    //按顺序向稀疏矩阵插入三元组
Status CreateSMatrix(TSMatrix *M);
    //创建稀疏矩阵
Status DestroySMatrix(TSMatrix *M);
    //销毁稀疏矩阵
Status ResetSMatrix(TSMatrix *M);
    //将稀疏矩阵置空
int GetMatrixElem(TSMatrix M, int i, int j);
    //获取稀疏矩阵第 i 行第 j 列的元素
Status PrintSMatrix(TSMatrix M);
    //打印稀疏矩阵
Status CopySMatrix(TSMatrix M, TSMatrix *T);
    //复制稀疏矩阵
Status AddSMatrix(TSMatrix M, TSMatrix N, TSMatrix *Q);
    //将稀疏矩阵 M 和 N 相加，并将结果存入矩阵 Q 中
Status SubtSMatrix(TSMatrix M, TSMatrix N, TSMatrix *Q);
    //将稀疏矩阵 M 和 N 相减，并将结果存入矩阵 Q 中
Status MultSMatrix(TSMatrix M, TSMatrix N, TSMatrix *Q);
    //将稀疏矩阵 M 和 N 相乘，并将结果存入矩阵 Q 中
Status TransposeSMatrix(TSMatrix M, TSMatrix *T);
    //将稀疏矩阵 M 进行转置，并将结果存入矩阵 T 中
float DetVal(TSMatrix M);
    //求解 n 阶矩阵 M 对应的行列式
```

一些伪码算法如下：

```
Triple * CreateTriple(int i, int j, ElemType e)
{   //创建三元组
    pTriple=(Triple *)malloc(sizeof(Triple));
    if(!pTriple) exit(ERROR);
    pTriple->i=i;pTriple->j=j;pTriple->e=e;
    return pTriple;
```

```
} //CreateTriple

Status InitSMatrix(TSMatrix *M)
{   //对稀疏矩阵M进行初始化操作
    M->mu=0;M->nu=0;M->tu=0;M->taken=0;
    memset(M->data, 0, MAXSIZE+1);
    memset(M->rpos, 0, MAXRC+1);
    memset(M->num, 0, MAXRC+1);
    return OK;
} //InitSMatrix

Status InsertOrderSMatrix(TSMatrix *M, Triple * triple)
{   //按行序将三元组插入到稀疏矩阵M中
    if(triple->i>M->mu||triple->j>M->nu)
        return ERROR;
    //如果还未插入任何元素，则插在第一个位置上
    if(M->taken==0)
    {M->data[1]=triple;return OK;}
    for(pos=1;pos<M->taken+1;++pos)
    {
        pre=pos-1;post=pos+1;
        if(triple->i<M->data[pos]->i)        //在no之前插入
        {
            for(int p=M->taken;p>=pre;--p)
                M->data[p+1]=M->data[p];
            M->data[pre+1]=triple;
            return OK;
        }
        else if(triple->i==M->data[pos]->i)
        {
            if(triple->j<M->data[pos]->j)   //在no之前插入
            {
                for(int p=M->taken;p>pre;--p)
```

```
                M->data[p+1]=M->data[p];
            M->data[pre+1]=triple;
            return OK;
        }
        else if(triple->j==M->data[pos]->j)
        {
            return ERROR;
        }
        else if(triple->j>M->data[pos]->j)
            continue;
    }
    else if(triple->i>M->data[pos]->i)
        continue;
}
if(pos==M->taken+1)
{M->data[M->taken+1]=triple;return OK;}
return OK;
} //InsertOrderSMatrix

Status CreateSMatrix(TSMatrix *M)
{   //根据用户输入创建矩阵
    scanf("%d %d %d",&M->mu,&M->nu,&M->tu);
    for(no=1;no<M->tu+1;++no)
    {
        scanf("%d %d %d",&i,&j,&e);
        pData=CreateTriple(i, j, e);
        success=InsertOrderSMatrix(M, pData);
        if(success==1)
        {
            M->taken+=1;
            if(pData->j<M->rpos[pData->i])      //更新改行首个非零元元素的位置
                M->rpos[pData->i]=pData->j;
            M->num[pData->i]+=1;                //改行非零元个数加1
```

```
        }
        else if(success==-2)        //不成功，则回溯继续输入
            --no;
        else if(success==-1)        //不成功，则回溯继续输入
            --no;
    }
    return OK;
} //CreateSMatrix

Status CopySMatrix(TSMatrix M, TSMatrix *T)
{
    T->mu=M.mu;T->nu=M.nu;T->tu=M.tu;
    for(index=1;index<MAXRC+1;++index)
    {
        T->num[index]=M.num[index];
        T->rpos[index]=M.rpos[index];
    }
    for(pos=1;pos<M.taken+1;++pos)
        T->data[pos]=M.data[pos];
    T->taken=M.taken;T->tu=M.tu;
    return OK;
} //CopySMatrix

Status AddSMatrix(TSMatrix M, TSMatrix N, TSMatrix *Q)
{
    if(M.mu!=N.mu||M.nu!=N.nu)
        return ERROR;
    else
    {
        Q->mu=M.mu;Q->nu=M.nu;Q->taken=0;
        row=M.mu; col=M.nu;
        for(i=1;i<row+1;++i)
            for(j=1;j<col+1;++j)
```

```
            {
                elemM=GetMatrixElem(M, i, j);
                elemN=GetMatrixElem(N, i, j);
                sum=elemM+elemN;
                if(sum!=0)
                {
                    SumNum+=1;
                    Triple * pData=CreateTriple(i, j, sum);
                    Q->data[SumNum]=pData;
                    Q->taken+=1;Q->tu+=1;
                    if(pData->j<Q->rpos[pData->i])
                    //更新改行首个非零元元素的位置
                        Q->rpos[pData->i]=pData->j;
                    Q->num[pData->i]+=1;
                }
            }
        return OK;
    }
    return OK;
} //AddSMatrix

Status MultSMatrix(TSMatrix M, TSMatrix N, TSMatrix *Q)
{   //改进方法
    if(M.nu!=N.mu)
        return ERROR;
    else if(M.nu*N.mu==0) return ERROR;
    else
    {
        Q->mu=M.mu;Q->nu=N.nu;Q->tu=Q->taken=0; //Q 初始化
        row=Q->mu;col=Q->nu;
        for(i=1;i<row+1;++i)
            for(j=1;j<col+1;++j)
            {
```

```
                product=0;
                for(n=1;n<M.nu+1;++n)
                    product+=GetMatrixElem(M, i, n)*GetMatrixElem(N, n, j);
                if(product!=0)
                {
                    ProductNum+=1;
                    Triple * pData=CreateTriple(i, j, product);
                    Q->data[ProductNum]=pData;
                    Q->taken+=1;Q->tu+=1;
                    if(pData->j<Q->rpos[pData->i])
                    //更新改行首个非零元元素的位置
                        Q->rpos[pData->i]=pData->j;
                    Q->num[pData->i]+=1;
                }
            }
        return OK;
    }
    return OK;
} //MultSMatrix

Status TransposeSMatrix(TSMatrix M, TSMatrix *T)
{
    T->mu=M.nu;T->nu=M.mu;T->tu=T->taken=0;
    for(pos=1;pos<M.tu+1;++pos)
    {
        i=M.data[pos]->j;j=M.data[pos]->i;
        elem=M.data[pos]->e;
        pData=CreateTriple(i, j, elem);
        if(InsertOrderSMatrix(T, pData)==1)
        {
            T->taken+=1;T->tu+=1;
            if(pData->i<T->rpos[pData->j])      //更新改行首个非零元元素的位置
                T->rpos[pData->j]=pData->i;
```

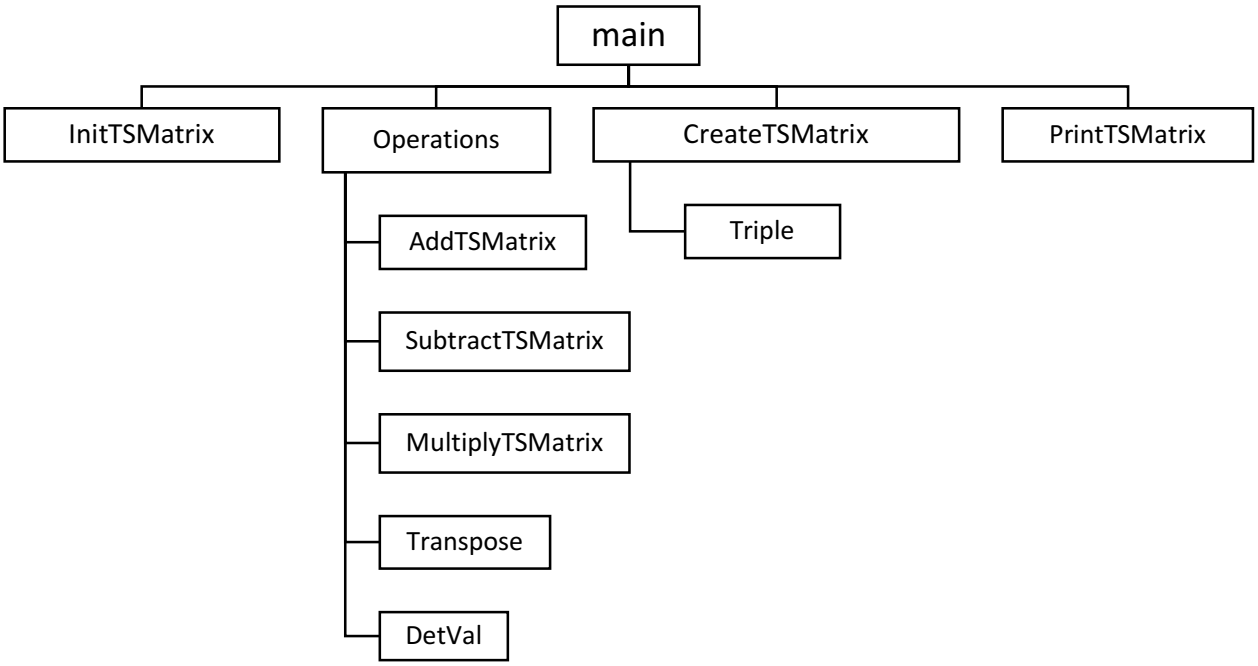
```
            T->num[pData->j]+=1;
        }
    }
    return OK;
} //TransposeSMatrix

float DetVal(TSMatrix M)
{   //对稀疏矩阵M对应的行列式进行求解
    if(M.mu!=M.nu)
        exit(ERROR);
    }
    int n=M.mu;   //行列式的阶数
    int z;
    float a[n][n],result=1.0,temp;
    memset(a, 0, sizeof(a));
    int i,j,e;      //记录行列坐标及对应的数值
    for(int no=0;no<M.tu;no++)    //还原矩阵
    {
        i=M.data[no+1]->i-1;
        j=M.data[no+1]->j-1;
        e=M.data[no+1]->e;
        a[i][j]=e;
    }
    i=0;j=0;

    for(z=0;z<n-1;z++)
        for(i=z;i<n-1;i++)
        {
            if(a[z][z]==0)
                for(i=z;a[z][z]==0;i++)
                {
                    {
                        for(j=0;j<n;j++)
```

```
                        a[z][j]=a[z][j]+a[i+1][j];
                    }
                    if(a[z][z]!=0)  break;
                }
            {
                temp=-a[i+1][z]/a[z][z];
                for(j=z;j<n;j++)
                    a[i+1][j]=temp*(a[z][j])+a[i+1][j];
            }
        }
    for(z=0;z<n;z++)
        result=result*(a[z][z]);

    return result;
} //DetVal
```

3. 主函数的伪码算法：

```
void main()
{  //主函数
    InitTSMatrix(&M);InitTSMatrix(&N);InitTSMatrix(&P);InitTSMatrix(&Q);
    CreateTSMatrix(&M);CreateTSMatrix(&N);CreateTSMatrix(&P);
    AddTSMatrix(M,N,&Q);PrintTSMatrix(Q);ResetTSMatrix(&Q);
    SubtractTSMatrix(M,N,&Q);PrintTSMatrix(Q);ResetTSMatrix(&Q);
    MultiplyTSMatrix(M,P,&Q);PrintTSMatrix(Q);ResetTSMatrix(&Q);
    InitTSMatrix(&S);CreateTSMatrix(&S);
    Transpose(&S);PrintTSMatrix(S);
    DetVal(P);
} //main
```

4. 函数的调用关系图反映了演示程序的层次结构：


main
InitTSMatrix
Operations
CreateTSMatrix
PrintTSMatrix
Triple
AddTSMatrix
SubtractTSMatrix
MultiplyTSMatrix
Transpose
DetVal

四、 调试分析

模块 I 用顺序表来完成任意同维度向量的计算，包括加法、减法、夹角余弦值

1. 虽然在逻辑上划分出了顺序表 SqList 和向量 Vector 两种抽象数据类型，但在实际编程过程中，将二者合为一体，直接以顺序表为原型构建向量并进行操作，这样避免重复。
2. 算法的时空分析

1）由于向量结构体中附带了 length 和 listsize 的信息，很多操作较为方便。InitVector, ExtendVector, EmptyVector, DimensionVector, GetElemVector 的时间复杂度均为 O(1)。

2) InsertVector, DeleteVector, PrintVector, DestoryVector, ClearVector 与线性表中插入、删除、打印、销毁、清除的时间复杂度一致，均为 O(n)。

3) AddVectors 和 SubtractVectors 需要先读取两个向量对应位置的值，进行加法/减法运算，并将和/差依次插入到另一个向量中，时间复杂度为 O(n)。

4) MultiplyVectors 需要读取两个向量对应位置的值，进行乘法运算，并把所有乘积相加，时间复杂度为 O(n)。

5) CosineVectors 需要先调用 MultiplyVectors 求得向量积(O(n))，再分别求得两个向量的模长(O(n))，再将向量积除以模长积，时间复杂度为 O(n)。

6) 所有操作都无需利用辅助空间，空间复杂度均为 O(1)。

模块 II 用顺序表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 虽然在逻辑上划分出了顺序表 SqList 和一元多项式 Polynomial 两种抽象数据类型，但在实际编程过程中，将二者合为一体，直接以顺序表为原型构建一元多项式并进行操作，这样避免重复。
2. 算法的时空分析

1）由于一元多项式结构体中附带了 length 和 listsize 的信息，很多操作较为方便。InitPolynomial, ExtendPolynomial, EmptyPolynomial, LengthPolynomial, GetElemPolynomial 的时间复杂度均为 O(1)。

2) InsertPolynomial, DeletePolynomial, PrintPolynomial, DestoryPolynomial, ClearPolynomial 与顺序表中插入、删除、打印、销毁、清除的时间复杂度一致，均为 O(n)。

3) InsertPolynomialOrder 需要先将待插入的元素与表中已有的元素进行比较大小，然后再挪动元素完成插入，比较的时间复杂度为 O(n)，挪动元素的时间复杂度为 O(n)，总的时间复杂度为 O(n)。

4) AddPolynomials 和 SubtractPolynomials 需要先后读取两个一元多项式的值，若待插入项的指数与表中已有的某项的指数相同，则合并同类项，否则按照升序进行插入，时间复杂度为 O((n+m)^2)。

5) MultiplyPolynomials 需要读取两个一元多项式对应位置的值，求得笛卡尔积，分别进行升序插入，时间复杂度为 O((n*m)^2)。

6) DerivativePolynomial 需要逐一读取一元多项式的项，分别求导，将结果依次插入新的顺序表中，时间复杂度为 O(n)。

7) 所有操作都无需利用辅助空间，空间复杂度均为 O(1)。

模块 III 用单链表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

3. 虽然在逻辑上划分出了单链表 LkList 和一元多项式 Polynomial 两种抽象数据类型，但在实际编程过程中，将二者合为一体，直接以单链表为原型构建一元多项式并进行操作，这样避免重复。
4. 算法的时空分析

1) 由于一元多项式结构体中附带了 length 和 listsize 的信息，很多操作较为方便。InitPolynomial, EmptyPolynomial, LengthPolynomial 的时间复杂度均为 O(1)；GetElemPolynomial 的时间复杂度为 O(n)。

2) InsertPolynomial, DeletePolynomial, PrintPolynomial, DestoryPolynomial, ClearPolynomial 与单链表中插入、删除、打印、销毁、清除的时间复杂度一致，均为 O(1)。

3) InsertPolynomialOrder 需要先将待插入的元素与表中已有的元素进行比较大小，然后再完成插入，比较的时间复杂度为 O(n)，插入的时间复杂度为 O(1)，总的时间复杂度为 O(n)。

4) AddPolynomials 和 SubtractPolynomials 需要先后读取两个一元多项式的值，若待插入项的指数与表中已有的某项的指数相同，则合并同类项，否则按照升序进行插入，时间复杂度为 O((n+m)^2)。

5) MultiplyPolynomials 需要读取两个一元多项式对应位置的值，求得笛卡尔积，分别进行升序插入，时间复杂度为 O((n*m)^2)。

6) DerivativePolynomial 需要逐一读取一元多项式的项，分别求导，将结果依次插入新的顺序表中，时间复杂度为 O(n)。

7) 所有操作都无需利用辅助空间，空间复杂度均为 O(1)。

模块 IV (1)四则运算表达式求值。操作符包括加(‘+’)、减(‘-’)、乘(‘*’)、除(‘/’)、幂(‘^’)、左括号(‘(’)、右括号(‘)’)，而操作数则包括整数、浮点数等不同类型的数

值。比如“30+4*2.5”，得到 40 或 40.0 等形式的结果。⑵含单变量的表达式求值。变量可以是 C/C++的标识符。比如“3+4*X2”，需要输入变量 X2 的值，然后计算结果。数字形式包括用科学计数法表示的，如 1.14e2，以及负数等。

1) 将表达式各项压栈、弹栈的操作与栈的基本操作一致，时间复杂度为 O(1)，空间复杂度为 O(1)。
2) 解析表达式需要逐字读取，并分割成项存入项库，时间复杂度和空间复杂度均为 O(n)。
3) 运算的时间复杂度为 O(1)。总体来说，进行表达式运算需要调用栈结构、项结构和未知变量结构，空间复杂度为 O(n)。

模块 V ⑴定义并运行简单函数。比如定义：f(X2)=3+4*X2，然后执行 f(5)，则得到结果 23。⑵保留函数定义历史，并可以运行历史函数。⑶函数的调用：比如已经定义了函数 f(X)，新定义函数 g(X)中调用了 f。

1) 基本操作与模块 IV 一致，但在处理复合函数的时候需要用到递归调用。这样处理保证了代码的简洁性和健壮性，若多层调用函数，可按照类似的思路进行扩展。
2) 总体来说，进行函数定义和运算需要调用栈结构、单链表结构、项结构和未知变量结构，空间复杂度为 O(n)。

模块 VI 矩阵运算。实现矩阵的加、减、乘、转置、行列式求值运算。

1) 在创建矩阵时，需要按行序升序插入三元组，待插入三元组需要与稀疏矩阵中现有的三元组进行逐一比较，时间复杂度为 O(n)。
2) 矩阵的加、减运算需要对三元组进行遍历，最坏的时间复杂度为 O(m*n)，最好的情况为 O(1)。
3) 矩阵的乘法运算利用稀疏矩阵中存储的 rpos 数组和 num 数组，时间复杂度为 $O(tu_1*tu_2)$。
4) 行列式求值需要将三元组矩阵还原成数组，空间复杂度为 O(m*n)。

五、 用户手册

模块 I 用顺序表来完成任意同维度向量的计算，包括加法、减法、夹角余弦值

1. 运行程序，对话框显示“创建向量：请输入向量维数”。
2. 输入一个正整数，敲击回车键完成输入。
3. 依次输入数字，构建向量；每次输入一个数字，敲击回车键完成当次输入。
4. 向量输入完毕构建成功以后，对话框将显示向量。
5. 依次运行向量加法、减法、乘法、求夹角余弦值，对话框将显示相应的结果。

模块 II 用顺序表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 运行程序，对话框显示“创建一元多项式：请输入项数”。
2. 输入一个正整数，敲击回车键完成输入。
3. 依次输入数字（整数、小数均可），构建一元多项式；每次输入一组数字（分别为系数和指数），敲击回车键完成当次输入。
4. 一元多项式输入完毕构建成功以后，对话框将显示一元多项式。
5. 依次运行一元多项式加法、减法、乘法、求导，对话框将显示相应的结果。

模块 III 用单链表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 运行程序，对话框显示“创建一元多项式：请输入项数”。
2. 输入一个正整数，敲击回车键完成输入。
3. 依次输入数字（整数、小数均可），构建一元多项式；每次输入一组数字（分别为系数和指数），敲击回车键完成当次输入。
4. 一元多项式输入完毕构建成功以后，对话框将显示一元多项式。
5. 依次运行一元多项式加法、减法、乘法、求导，对话框将显示相应的结果。

模块 IV ⑴四则运算表达式求值。操作符包括加(‘+’)、减(‘-’)、乘(‘*’)、除(‘/’)、幂(‘^’)、左括号(‘(’)、右括号(‘)’)，而操作数则包括整数、浮点数等不同类型的数值。比如“30+4*2.5”，得到 40 或 40.0 等形式的结果。⑵含单变量的表达式求值。变量可以是 C/C++的标识符。比如“3+4*X2”，需要输入变量 X2 的值，然后计算结果。数字形式包括用科学计数法表示的，如 1.14e2，以及负数等。

1. 运行程序，对话框显示“请输入表达式”。
2. 用户可以键入任意数值表达式，也可以键入含未知变量的表达式。注意负数要用括号括起。未知变量的命名规则与 C 语言变量命名规则一致。
3. 对话框将显示用户输入的表达式。

4. 程序将对表达式进行解析，若遇未知变量，则要求用户键入数字。
5. 完成表达式运算，输出结果。
6. 程序结束。

模块 V ⑴定义并运行简单函数。比如定义：f(X2)=3+4*X2，然后执行 f(5)，则得到结果 23。⑵保留函数定义历史，并可以运行历史函数。⑶函数的调用：比如已经定义了函数 f(X)，新定义函数 g(X)中调用了 f。

1. 运行程序，对话框显示“请定义简单函数”。
2. 用户先定义简单函数名和变量名，再输入函数解析式。注意负数要用括号括起。未知变量的命名规则与 C 语言变量命名规则一致。
3. 对话框将显示用户输入的简单函数式。
4. 对话框显示“请为未知变量赋值”，用户键入数值。
5. 完成运算，输出结果。
6. 存储用户定义的简单函数。
7. 对话框显示“请定义复合函数”，并打印出历史存储函数。
8. 用户选择调用的函数名，并定义复合函数名和变量名，再输入函数解析式。注意负数要用括号括起。未知变量的命名规则与 C 语言变量命名规则一致。
9. 对话框将显示用户输入的复合函数式。
10. 对话框显示“请为未知变量赋值”，用户键入数值。
11. 完成运算，输出结果。
12. 程序结束。

模块 VI 矩阵运算。实现矩阵的加、减、乘、转置、行列式求值运算。

1. 运行程序，对话框显示“创建矩阵：请输入行数和列数”。
2. 用户键入一对正整数以定义矩阵的行数和列数。
3. 对话框显示“请输入元素的行、列位置与数值”。
4. 用户依次输入三元数组以向稀疏矩阵中插入元素。
5. 完成矩阵加、减法要求两个矩阵的行列维数完全一致，否则报错。
6. 完成矩阵乘法要求左矩阵的列数等于右矩阵的行数，否则报错。
7. 完成行列式求值要求矩阵为方阵，否则报错。

六、 测试结果

模块 I 用顺序表来完成任意同维度向量的计算，包括加法、减法、夹角余弦值

1. 构建向量 Vec1: 键入数字“11”，确定向量维数为 11。
2. 依次键入数字 11、2、-45、32、-245、0、3442、5、0、25、56、-23 构建向量 Vec1=(2.0, -45.0, 32.0, -245.0, 0.0, 3442.0, 5.0, 0.0, 25.0, 56.0, -23.0)。
3. 构建向量 Vec2: 键入数字“11”，确定向量维数为 11。
4. 依次键入数字 8、90、55、-3、0、67、790、234、804、0、-687 构建向量 Vec2=(8.0, 90.0, 55.0, -3.0, 0.0, 67.0, 790.0, 234.0, 804.0, 0.0, -687.0)。
5. 执行向量加法，得结果为 Vec=(10.0, 45.0, 87.0, -248.0, 0.0, 3509.0, 795.0, 234.0, 829.0, 56.0, -710.0)。
6. 执行向量减法，得结果为 Vec=(-6.0, -135.0, -23.0, -242.0, 0.0, 3375.0, -785.0, -234.0, -779.0, 56.0, 664.0)。
7. 执行向量乘法，得结果为 268926.0
8. 执行向量夹角余弦值算法，得结果为 0.0579。

模块 II 用顺序表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

1. 构建一元多项式 Poly1: 键入数字“8”，确定项数为 8。
2. 依次键入数组(5,1000), (-1,828), (7,7), (-89,5),(-23,3),(90,2),(-235,1),(754,0)构建一元多项式$\mathrm{Poly1} = 5X^{1000} - 1X^{828} + 7X^{7} - 89X^{5} - 23X^{3} + 90X^{2} - 235X^{1} + 754X^{0}$。
3. 构建一元多项式 Poly1: 键入数字“7”，确定项数为 7。
4. 依次键入数组(-34,643), (554,103), (-7,7), (-89,5),(243,4),(-222,2),(1,0)构建一元多项式$\mathrm{Poly2} = -34X^{643} + 554X^{103} - 7X^{7} - 89X^{5} + 243X^{4} - 222X^{2} + 1X^{0}$。
5. 执行一元多项式加法，得结果为$\mathrm{Poly} = 755X^{0} - 235X^{1} - 132X^{2} - 23X^{3} + 243X^{4} - 178X^{5} + 0X^{7} + 554X^{103} - 34X^{643} - 1X^{828} + 5X^{1000}$。
6. 执行一元多项式减法，得结果为$\mathrm{Poly} = 753X^{0} - 235X^{1} + 312X^{2} - 23X^{3} - 243X^{4} + 0X^{5} + 14X^{7} - 554X^{103} + 34X^{643} - 1X^{828} + 5X^{1000}$。
7. 执行一元多项式乘法，得结果为$\mathrm{Poly} = 754X^{0} - 235X^{1} - 167298X^{2} + 52147X^{3} + 163242X^{4} - 119194X^{5} + 42785X^{6} + 888X^{7} + 3692X^{8} - 23811X^{9} + 8082X^{10} + 1701X^{11} + 0X^{12} - 49X^{14} + 417716X^{103} - 130190X^{104} + 49860X^{105} - 12742X^{106} - 49306X^{108} + 3878X^{110} - 25636X^{643} + 7990X^{644} - 3060X^{645} + 782X^{646} + 3026X^{648} - 238X^{650} - 1X^{828} + 222X^{830} - 243X^{832} + 89X^{833} + 7X^{835} - 554X^{931} + 5X^{1000} -$

$1110X^{1002} + 1215X^{1004} - 445X^{1005} - 35X^{1007} + 2770X^{1103} + 34X^{1471} - 170X^{1643}$。

8. 对一元多项式 Poly1 执行求导，得结果为$\text{DER Poly1} = 180X^{1} - 69X^{2} - 445X^{4} + 49X^{6} - 828X^{827} + 5000X^{999}$。
9. 对一元多项式 Poly2 执行求导，得结果为$\text{DER Poly2} = -444X^{1} + 972X^{3} - 445X^{4} - 49X^{6} + 57062X^{102} - 21862X^{642}$。

模块 III 用单链表来完成任意一元多项式的计算，包括加法、减法、乘法、导数（包括任意阶）等

与模块 II 的测试情况基本一致

模块 IV ⑴四则运算表达式求值。操作符包括加(‘+’)、减(‘-’)、乘(‘*’)、除(‘/’)、幂(‘^’)、左括号(‘(’)、右括号(‘)’)，而操作数则包括整数、浮点数等不同类型的数值。比如“30+4*2.5”，得到 40 或 40.0 等形式的结果。⑵含单变量的表达式求值。变量可以是 C/C++的标识符。比如“3+4*X2”，需要输入变量 X2 的值，然后计算结果。数字形式包括用科学计数法表示的，如 1.14e2，以及负数等。

1. 输入不含未知变量的表达式 1“(-1.8e2)*((-5.2)+(2*3-1))^3+8/2.3”。
2. 显示表达式为(-1.8e2)*((-5.2)+(2*3-1))^3+8/2.3。
3. 标准化以后得(-1.8e2)*((-5.2)+(2*3-1))^3+8/2.3#。
4. 运算求值，得结果为 4.918257。
5. 输入含未知变量的表达式 2“(-1.8e2)*(X1+(2*3-1))^3+8/X2”。
6. 显示表达式为(-1.8e2)*(X1+(2*3-1))^3+8/X2。
7. 标准化以后得(-1.8e2)*(X1+(2*3-1))^3+8/X2#。
8. 解析表达式，遇到未知变量 X1、X2，分别键入-5.2、2.3 进行赋值。
9. 运算求值，得结果为 4.918257。

模块 V ⑴定义并运行简单函数。比如定义：f(X2)=3+4*X2，然后执行 f(5)，则得到结果 23。⑵保留函数定义历史，并可以运行历史函数。⑶函数的调用：比如已经定义了函数 f(X)，新定义函数 g(X)中调用了 f。

1. 定义简单函数为 f(X)=6-X*2+1/X。
2. 运行 f(2)，得结果为 2.50。
3. 存储 f(X)。
4. 定义复合函数为 g(X)=2*f(X^2)-1/f(1/X)。
5. 运行 g(2)，得结果为-9.88。

模块 VI 矩阵运算。实现矩阵的加、减、乘、转置、行列式求值运算。

1. 创建矩阵 M、N、P、Q 如下：

M_{4*5}={(2,4,7), (1,5,2), (4,5,4)}

N_{4*5}={(1,1,1), (2,2,2), (3,3,3), (4,4,4), (4,5,5)}

P_{5*2}={(1,2,3), (3,1,5), (4,2,6)}

Q_{3*3}={(1,1,1), (2,2,2), (3,3,3)}

M_{4*5}+ N_{4*5}=

1	0	0	0	2
0	2	0	7	0
0	0	3	0	0
0	0	0	4	9

M_{4*5}- N_{4*5}=

-1	0	0	0	2
0	-2	0	7	0
0	0	-3	0	0
0	0	0	-4	-1

Q_{4*2}=M_{4*5}* P_{5*2}=

0	0
0	42
0	0
0	0

Transpose(Q_{4*2})=

0	0	0	0
0	42	0	0

DetVal(Q_{3*3})=6.00

七、 附录

源程序文件名清单：

main.c //主程序

Vectors.h //向量实现单元

PolyNSq.h //顺序表一元多项式实现单元

PolyNLk.h //单链表一元多项式实现单元

Expressions.h //表达式实现单元

Variables.h //表达式中的变量处理单元

Operations.h //表达式中的运算栈实现单元

Functions.h //函数实现单元

Matrix.h //三元组稀疏矩阵实现单元