第六章　*Maple* 程序设计

前面，我们使用的是 Maple 的交互式命令环境. 所谓交互式命令环境，就是一次输入一条或几条命令，然后按回车，这些命令就被执行了，执行的结果显示在同一个可执行块中. 对于大多数用户来说，利用交互式命令环境解决问题已经足够了，但如果要解决一系列同一类型的问题或者希望利用 Maple 编写需要的解决特定问题的函数和程序，以期更加充分地利用 Maple 的强大功能，提高大规模问题的计算效率，掌握一定的程序设计是必要的. 幸运的是，Maple 自身提供了一套编程工具，即 Maple 语言. Maple 语言实际上是由 Maple 各种命令以及一些简单的过程控制语句组成的.

1 编程基础

1.1 算子

所谓算子，是从一个抽象空间到另一个抽象空间的函数. 在数学上算子的含义通常是函数到函数的映射. 在 **Maple** 中，算子常用“箭头”记号定义(也称箭头操作符)：

```
> f:=x->a*x*exp(x);
```

$$f := x \to a\, x\, \mathrm{e}^{x}$$

```
> g:=(x,y)->a*x*y*exp(x^2+y^2);
```

$$g := (x, y) \to a\, x\, y\, \mathrm{e}^{(x^2+y^2)}$$

另外，函数 **unapply** 也可以从表达式建立算子：

```
> unapply(x^2+1,x);
```

$$x \to x^2 + 1$$

```
> unapply(x^2+y^2,x,y);
```

$$(x, y) \to x^2 + y^2$$

当我们依次把算子 f 作用到参数 0, a, x^2+a 时即可得平常意义上的函数值：

```
> f:=t->t*sin(t);
```

$$f := t \to t\sin(t)$$

```
> f(0);
```

$$0$$

```
> f(a);
```

$$a\sin(a)$$

```
> f(x^2+a);
```

$$(x^2+a)\sin(x^2+a)$$

上述结果是函数作用的例子. 而最后一个结果$(x^2+a)\sin(x^2+a)$实际上是算子 f 与算子 g:=t->t^2+a 复合后再作用到参数 x 的结果.

从数学上讲, 作用与复合是不同的，它们产生的结果是有区别的, 但在使用它们时, 两者还是有些重叠的. 在 Maple 中, 可以依赖于语法把它们区分开:

(1) 当复合两个算子时, 结果仍是算子, 两个算子的定义域必须是相容的；

(2) 当把一个算子作用于一个参数(参数必须在算子的定义域中)时, 结果是一个表达式;

(3) 在 Maple 中，函数作用的语法是使用括号( )，如函数 f 作用到参数 u 写作 f(u). 而复合算子的符号是@，多重复合时使用符号@@.

通过进一步的例子可以清楚区分作用与复合的功能: f 和 g 复合的结果是算子 $f\circ g := t\mapsto f(g(t))$，而把这个算子作用到参数 x 得到表达式 $f(g(x))$. 例如, $f = t\mapsto \sin(t+\varphi), g = u\mapsto \exp(u)$, 则 $f\circ g := z\mapsto \sin(\exp(z)+\varphi)$ 是一个算子, 而 $f(g(x)) = \sin(\exp(x)+\varphi)$ 是一个表达式, 因为 x 是一个实数. 试比较下述两例：

```
> D(g@f);
```

$$(\mathrm{D}(g))@f\ \mathrm{D}(f)$$

```
> D(g*h);
```

$$\mathrm{D}(g)\,h + g\,\mathrm{D}(h)$$

另外一个应引起注意的问题是算子(函数)与表达式的异同，在第一章 2.2.2 中曾探讨过函数与表达式的区别，这里再通过几个例子说明其中的微妙差异：

```
> f1:=x^2+1;
> f2:=y^2+1;
```

$$f1 := x^2+1$$

$$f2 := y^2 + 1$$

```
> f3:=f1+f2;
```

$$f3 := x^2 + 2 + y^2$$

再看下面的例子：

```
> g1:=x->x^2+1;
> g2:=y->y^2+1;
```

$$g1 := x \to x^2 + 1$$

$$g2 := y \to y^2 + 1$$

```
> g3:=g1+g2;
```

$$g3 := g1 + g2$$

与前面例子不同的是，两个算子(函数) **g1**, **g2** 相加的结果依然是函数名 **g3**，出现这个问题的主要原因是 g1 与 g2 分别为 x，y 的函数，Maple 认为它们的定义域不相容. 要得到与前例的结果，只需稍作改动：

```
> g3:=g1(x)+g2(y);
```

$$g3 := x^2 + 2 + y^2$$

下面的例子想说明生成 Maple 函数的两种方式“箭头操作符”及“unapply”之间微妙的差异：

```
> x:='x': a:=1: b:=2: c:=3:
> a*x^2+b*x+c;
```

$$x^2 + 2\,x + 3$$

```
> f:=unapply(a*x^2+b*x+c,x);
```

$$f := x \to x^2 + 2\,x + 3$$

```
> g:=x->a*x^2+b*x+c;
```

$$g := x \to a\,x^2 + b\,x + c$$

由此可见，f 中的 a,b,c 已经作了代换，而 g 中则显含 a,b,c。再看下面实验：

```
> f(x); g(x);
```

$$x^2 + 2\,x + 3$$

$$x^2 + 2\,x + 3$$

f 与 g 两者相同，再对其微分：

```
> D(f); D(g);
```

$$x \to 2x + 2$$

$$x \to 2ax + b$$

再改变常数 c 的值，观察 f 与 g 的变化：

```
> c := 15;
```

$$c := 15$$

```
> f(x); g(x);
```

$$x^2 + 2x + 3$$

$$x^2 + 2x + 15$$

由此可见，在利用 Maple 进行函数研究时，对同一问题应该用不同方法加以校验，而这一切的支撑是数学基础！

1.2 编程初体验

利用算子可以生成最简单的函数—单个语句的函数，但严格意义上讲它并非程序设计，它所生成的数据对象是子程序. 所谓子程序，简单地说，就是一组预先编好的函数命令，我们由下面的简单程序来看看 **Maple** 程序的结构：

```
> plus:=proc(x,y)
    x+y;
  end;
```

这个程序只有 2 个参数，在程序内部它的名称是 *x*, y，这是 Maple 最简单的程序结构，仅仅在 **proc( )**和 **end** 中间加上在计算中需要的一条或者多条命令即可，**Maple** 会把最后一个语句的结果作为整个子程序的返回结果，这一点需要引起注意. 再看下例：

```
> P:=proc(x,y)
     x-y;
     x*y;
     x+y;
 end:
> P(3,4);
```

$$7$$

显然，尽管程序 **P** 有三条计算命令，但返回的只是最后一个语句 **x+y;**的结果. 要想输出所有的计算结果，需要在程序中增加 **print** 语句:

```
> P:=proc(x,y)
```

```
   print(x-y);
   print(x*y);
   print(x+y);
 end:
> P(3,4);
```

$$-1$$

$$12$$

$$7$$

再看下面几个例子：

```
> for i from 2 to 6 do
    expand((x+y)^i );
  od;
```

$$x^2 + 2\,x\,y + y^2$$

$$x^3 + 3\,x^2\,y + 3\,x\,y^2 + y^3$$

$$x^4 + 4\,x^3\,y + 6\,x^2\,y^2 + 4\,x\,y^3 + y^4$$

$$x^5 + 5\,x^4\,y + 10\,x^3\,y^2 + 10\,x^2\,y^3 + 5\,x\,y^4 + y^5$$

$$x^6 + 6\,x^5\,y + 15\,x^4\,y^2 + 20\,x^3\,y^3 + 15\,x^2\,y^4 + 6\,x\,y^5 + y^6$$

```
> F:=proc(n::integer)
  if n mod 12=0 then true
  else false
  fi
  end:
> F(123^123), F(1234567890^9);
```

$$\mathit{false}, \mathit{true}$$

从上面几个简单的例子可以看出 **Maple** 子程序主要包含以下一些内容：

(i) 把定义的子程序赋值给程序名 **procname**，以后就可以用子程序名 **procname** 来调用程序；

(ii) 子程序一律以 **proc(**)开头，括号里是程序的输入参数，如果括号中什么都没有，表示这个子程序没有任何输入参数；

(iii) 子程序中的每一个语句都用分号(或冒号)分开(这一点不是主要的，程序设计时，

在可能的时候—过程当中的最后一个语句、for-循环、if 语句中的最后一个语句省略终结标点也是允许的，这并不是为了懒惰，而是因为在终结语句后面插入一个语句产生的影响要比仅仅执行一个新语句产生的影响大)；

(iv) 在定义完子程序之后，Maple 会显示它对该子程序的解释 (除非在 end 后用冒号结束)，它的解释和你的定义是等价的，但形式上不一定完全相同；

(v) Maple 会自动地把除了参数以外的变量都作为局部变量(local variable)，这就是说，它们仅仅在这个子程序的定义中有效，和子程序以外的任何同名变量无关.

在定义了一个子程序以后，执行它的方法和执行任何 Maple 系统子程序一样—程序名再加上一对圆括号( )，括号中包含要调用的参数，如果子程序没有参数，括号也是不能省略的.

除了上面给出的程序设计方法外，在 Maple 中还可以直接由“->” (箭头)生成程序，如下例：

```
> f:=x->if x>0 then x else -x fi;
```

$f := \mathbf{proc}\ (x)\ \mathbf{option}\ \ operator, arrow;\ \mathbf{if}\ 0 < x\ \mathbf{then}\ x\ \mathbf{else}\ -x\ \mathbf{end\ if}\ \ \mathbf{end\ proc}$

```
> f(-5),f(5);
```

5, 5

甚至于程序名也可以省略，这种情况通常会在使用函数 map 时遇到：

```
> map(x->if x>0 then x else -x fi,[-4,-3,-2,0,1]);
```

[4, 3, 2, 0, 1]

如果需要察看一个已经定义好的子程序的过程，用 **eval** 命令，查看 Maple 中源程序(如 factor 函数)使用下述组合命令：

interface(verboseproc=2);

print(factor);

再看一个更为有用的简单程序：代数方程的参数解。该程序在代数方程 f(x,y)=0 求解中使用了一个巧妙的代换 y=tx 得到了方程的参数解，它的主要用途是用来画图、求积分、求微分和求级数. 程序如下：

```
> parsolve:=proc(f,xy::{list(name),set(name)},t::name)
  local p,x,y;
  x:=xy[1];
  y:=xy[2];
  p:={solve(subs(y=t*x,f),x)}minus{0};
  map((xi,u,xx,yy)->{xx=xi,yy=u*xi},p,t,x,y)
end:
```

调用该程序可以方便求解：

```
> parsolve(u^2+v^2=a^2,[u,v],t);
```

$$\left\{\left\{u=-\frac{a}{\sqrt{1+t^2}},v=-\frac{t\,a}{\sqrt{1+t^2}}\right\},\left\{u=\frac{a}{\sqrt{1+t^2}},v=\frac{t\,a}{\sqrt{1+t^2}}\right\}\right\}$$

```
> f:=randpoly([x,y],degree=3,sparse);
```

$$f := -53\,x + 85\,x\,y + 49\,y^2 + 78\,x^3 + 17\,x\,y^2 + 72\,y^3$$

```
> parsolve(f,[x,y],t);
```

$$\left\{\left\{y=\frac{1}{2}\frac{t\,(-49\,t^2-85\,t+\sqrt{2401\,t^4+23594\,t^3+10829\,t^2+16536}\,)}{72\,t^3+78+17\,t^2},\right.\right.$$
$$\left.x=\frac{1}{2}\frac{-49\,t^2-85\,t+\sqrt{2401\,t^4+23594\,t^3+10829\,t^2+16536}}{72\,t^3+78+17\,t^2}\right\},\left\{\right.$$
$$y=\frac{1}{2}\frac{t\,(-49\,t^2-85\,t-\sqrt{2401\,t^4+23594\,t^3+10829\,t^2+16536}\,)}{72\,t^3+78+17\,t^2},$$
$$\left.\left.x=\frac{1}{2}\frac{-49\,t^2-85\,t-\sqrt{2401\,t^4+23594\,t^3+10829\,t^2+16536}}{72\,t^3+78+17\,t^2}\right\}\right\}$$

1.3 局部变量和全局变量

Maple 中的全局变量，是指那些在交互式命令环境中定义和使用的变量，前面所使用的变量几乎都属于全局变量. 而在编写子程序时，需要定义一些只在子程序内部使用的变量，称其为局部变量. 当 **Maple** 执行子程序时，所有和局部变量同名的全局变量都保持不变，而不管在子程序中给局部变量赋予了何值. 如果要把局部变量定义为全局变量，需要用关键词 **global** 在程序最开始加以声明，而局部变量则用 **local** 声明，虽然这是不必要的，但在程序设计时，声明变量是有一定好处的.

下面通过实例演示局部变量与全局变量的不同. 为了更清楚地观察子程序对全局变量的影响，在子程序外先设定一个变量 a 的值：

```
> a:=1;
```

$$a := 1$$

```
> f:=proc( )
   local a;
   a:=12345678/4321;
   evalf(a/2);
end;
> f();
```

$$1428.567230$$

```
> a;
```

$$1$$

```
>g:=proc( )
   global a;
   a:=12345678/4321;
   evalf(a/2);
end;

>g();
```

$$1428.567230$$

```
>a;
```

$$\frac{12345678}{4321}$$

显然，在前一个程序中，由于在子程序外已经赋值给 a，a 是全局变量，它的值不受子程序中同名局部变量的影响；而在后一个子程序中，由于重新把 a 定义为全局变量，所以子程序外的 a 随着子程序中的 a 值的变化而变化.

子程序中的输入参数，它既不是全局的，也不是局部的. 在子程序内部，它是形式参数，也就是说，它的具体取值尚未被确定，它在程序调用时会被替换成真正的参数值. 而在子程序外部，它们仅仅表示子程序接受的参数的多少，而对于具体的参数值没有关系.

1.4 变量 nargs, args 与 procname

在所有程序中都有三个有用的变量：**nargs**, **args** 与 **procname**. 前两个给出关于调用参量的信息: **nargs** 变量是调用的实际参量的个数，**args** 变量是包含参量的表达式序列，**args** 的子序列通过范围或数字的参量选取. 例如，第 i 个参量被调用的格式为: **args[i]**. **nargs, args** 变量通常在含有可选择参量的程序中使用. 下面看一个例子：

```
>p:=proc( )
local i;
RETURN(nargs,[seq(i^3,i=args)])
end proc:
>p(1,2,3,4,5,6,7,8,9,10);
```

$$10, [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]$$

该程序利用 **Maple** 函数 **RETURN** 返回了输入参量的个数以及参量序列的立方列表，**RETURN** 函数使用时必须在其后加圆括号，即使无结果返回时也得如此。下面是一个关于求任意序列最大值的复杂程序：

```
>maximum:=proc( )
  local r, i;
```

```
  r:=args[1];
  for i from 2 to nargs do
    if args[i]>r then r:=args[i]
    end if
  end do;
     r;
end proc:
>maximum(12^4,5^7,69^3,10^4+4.9*1000);
```

328509

如果变量 **procname** (程序被调用的名字)存在的话，它可以用来直接访问该程序，通常用 **procname(args)** 完成调用:

```
>f:=proc(a)
if a>0 then RETURN(a^(1/2))
else RETURN('procname(args)')
end if
end proc:
>f(100-3^6),f(10^2+90-9*4);
```

$$f(-629),\sqrt{154}$$

2 基本程序结构

所有的高级程序设计语言都具有程序结构，因为为了完成一项复杂的任务，仅仅按照语句顺序依次执行是远远不够的，更需要程序在特定的地方能够跳转、分叉、循环，……，与此同时，程序结构就产生了.

2.1 for 循环

在程序设计中，常常需要把相同或者类似的语句连续执行多次，此时，通过 **for** 循环结构可以更便捷地编写程序. 试看下面几个例子：

① 求 1 至 5 自然数的和：

```
>total:=0:
  for i from 1 to 5 do
    total:=total+i:
  od;
```

$$total := 1$$
$$total := 3$$
$$total := 6$$
$$total := 10$$

$$total := 15$$

② 列示 2，4，6，8，10 及其平方数、立方数：

```
>for i from 2 to 10 by 2 do
'i'=i, 'i^2'=2^i, 'i^3'=i^3;
od;
```

$$i = 2, i^2 = 4, i^3 = 8$$

$$i = 4, i^2 = 16, i^3 = 64$$

$$i = 6, i^2 = 64, i^3 = 216$$

$$i = 8, i^2 = 256, i^3 = 512$$

$$i = 10, i^2 = 1024, i^3 = 1000$$

③ 列示第 100 到第 108 个素数(第 1 个素数是 2)：

```
>for j from 100 to 108 do
  prime[j]=ithprime(j);
 od;
```

$$prime_{100} = 541$$
$$prime_{101} = 547$$
$$prime_{102} = 557$$
$$prime_{103} = 563$$
$$prime_{104} = 569$$
$$prime_{105} = 571$$
$$prime_{106} = 577$$
$$prime_{107} = 587$$
$$prime_{108} = 593$$

④ 列示 5 到 10 的阶乘及其因数分解：

```
>for n in seq(i!,i=5..10) do #离散循环
  n=ifactor(n);
  od;
```

$$120 = (2)^3 (3) (5)$$

$$720 = (2)^4 (3)^2 (5)$$

$$5040 = (2)^4\ (3)^2\ (5)\ (7)$$

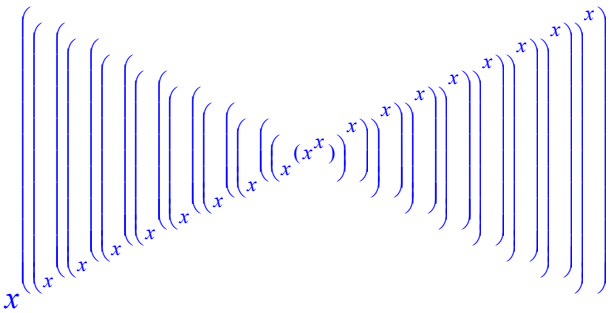
$$40320 = (2)^7\ (3)^2\ (5)\ (7)$$

$$362880 = (2)^7\ (3)^4\ (5)\ (7)$$

$$3628800 = (2)^8\ (3)^4\ (5)^2\ (7)$$

⑤ 一个复杂的幂指函数生成：

```
>f:=proc(x,n)
 local i, t;
 t:=1;
 for i from 1 to n do
     t:=x^(t^x);
 od;
 t;
 end:
> f(x,10);
```



通过上述例子可以看出，Maple 的 **for** 循环结构很接近于英语语法. 在上面例子中, *i* 称为循环变量, 用于循环计数, **do** 后面的部分, 称为循环体, 就是需要反复执行的语句, 可以是一条语句, 也可以是多条语句. 在循环体中, 也可以引用循环变量. 循环体的结束标志, 也就是整个 for 循环结构的结束标志, 是字母“od”. for 循环结构的语法可总结为：

for 循环变量 from 初始值 to 终止值 (by 步长) do

循环体

od

在 for 循环结构中，初始值和终了值必须是数值型的, 而且必须是实数, 如果初始值

是 1，可以省略不写，而步长用 by 引出.

对于上面程序中的重复循环似乎可有可无，但实际科学计算中，重复次数往往成千上万，或者重复的次数无法确定，循环就必不可少了．下面再看一个实用程序—从 1 到 n 的自然数的求和程序：

```
> SUM:=proc(n)
    local i, total;
    total:=0;
    for i from 1 to n do
        total:=total+i;
    od;
end;
> SUM(1000);
```

$$500500$$

该程序仅仅具有练习例举的意义，因为在 Maple 中像这样一个问题只需要一个简单语句就可以完成：

```
> Sum(n,n=1..1000)=sum(n,n=1..1000);
```

$$\sum_{n=1}^{1000} n = 500500$$

更一般地，对于确定的求和可用 **add** 命令(该命令无惰性函数形式)：

```
> add(n,n=1..1000);
```

$$500500$$

再看下面一个动画制作例子(图略去)：

```
> for i from -20 to 30 do
    p||i:=plots[implicitplot](x^3+y^3-5*x*y=1-i/8,x=-3..3,y=-3..3,numpoints=800,
          tickmarks=[2,2] )
  od:
> plots[display]( p||(-20..30), insequence=true );
```

2.2 分支结构(条件语句)

所谓分支结构，是指程序在执行时，依据不同的条件，分别执行两个或多个不同的程序块，所以常常称为条件语句. if 条件结构的语法为：

if 逻辑表达式 1(条件 1) **then** 程序块 1

elif 逻辑表达式 2(条件 2) **then** 程序块 2

else 程序块 3

fi

下面是一个判断两个数中较大者的例子：

```
>bigger:= proc(a,b)
    if a>=b then a
      else b
    fi
  end;
```

再如下例：

```
>ABS:=proc(x)
    if x<0 then -x
        else x;
    fi;
end;
```

显然，这个绝对值函数的简单程序，对于任意实数范围内的数值型变量计算都是没有问题的，但对于非数值型的参数则无能为力．为此，我们需要在程序中添加一条判断参数是否为数值型变量的语句：

```
>ABS:=proc(x)
    if type(x,numeric) then
        if x<0 then -x else x;
        fi;
      else
        'ABS'(x);
    fi;
end:
```

这里，我们用到一个 **if** 语句的嵌套，也就是一个 **if** 语句处于另一个 **if** 语句当中．这样的结构可以用来判断复杂的条件．我们还可以用多重嵌套的条件结构来构造更为复杂的函数，例如下面的分段函数：

```
> HAT:=proc(x)
    if type(x,numeric) then
        if x<=0 then 0;
        else
            if x<=1 then x;
                else 0;
            fi;
```

```
        fi;
    else
        'HAT'(x);
    fi;
end;
```

尽管在上面的程序中我们用了不同的缩进来表示不同的层次关系，这段程序还是很难懂. 对于这种多分支结构，可以用另一种方式书写，只需在第二层的 if 语句中使用 **elif**，这样就可以把多个分支形式上写在同一个层次中，以便于阅读.

```
> HAT:=proc(x)
    if type(x,numeric) then
        if x<=0 then 0;
        elif x<=1 then x;
        else 0;
        fi;
     else
        'HAT'(x);
    fi;
end;
```

和许多高级语言一样，这种多重分支结构理论上可以多到任意多重.

再来看看前面的绝对值函数的程序，似乎是完美的，但是，对于乘积的绝对值则没有办法，下面用 map 命令来修正，注意其中的 type(…, `*`)是用来判断表达式是否为乘积，进而对其进行化简.

```
> ABS:=proc(x)
    if type(x,numeric) then
        if x<0 then -x else x fi;
    elif type(x,`*`) then map(ABS,x);
    else
        'ABS'(x);
    fi;
end:
> ABS(-1/2*b);
```

$$\frac{1}{2}\,\mathrm{ABS}(b)$$

分段函数是一类重要的函数，条件语句在定义分段函数时具有明显的优越性，下面学习几个实例：

(1) $f(x)=\begin{cases} x^2+1 & x<0 \\ \sin(\pi x) & 0\le x \end{cases}$

```
> f:=proc( x )
  if x<0 then x^2+1 else sin(Pi*x) fi;
```

```
end;
```

(2) $g(x)=\begin{cases} x^2+x & x\le 0 \\ \sin(x) & x<3\pi \\ x^2-6\pi x+9\pi^2-x+3\pi & 3\pi\le x \end{cases}$

```
>g:=proc(x)
   if x<=0 then
     x^2+x
   else
     if x<3*Pi then
       sin(x)
     else
       x^2-6*x*Pi+9*Pi^2-x+3*Pi
     fi
   fi
end;
```

2.3 while 循环

for 循环在那些已知循环次数，或者循环次数可以用简单表达式计算的情况下比较适用. 但有时循环次数并不能简单地给出，我们要通过计算，判断一个条件来决定是否继续循环，这时，可以使用 while 循环. while 循环标准结构为：

while 条件表达式 do

循环体

od

Maple 首先判断条件是否成立，如果成立，就一遍遍地执行循环体，直到条件不成立为止.

下面看一个简单的程序，是用辗转相除法计算两个自然数的最大公约数(**Euclidean** 算法).

```
> GCD:=proc(a::posint, b::posint)
    local p,q,r;
    p:=max(a,b);
    q:=min(a,b);
    r:=irem(p,q);
    while r<>0 do
      p:=q;
      q:=r;
```

```
        r:=irem(p,q);
    od;
    q;
end:
> GCD(123456789,987654321);
```

9

在上面程序中的参数 a、b 后面的双冒号“::”指定了输入的参数类型. 若类型不匹配时输出错误信息. 再看下面一个扩展 **Euclidean** 算法的例子：

```
> mygcdex:= proc(a::nonnegint,b::nonnegint,x::name,y::name)
  local a1,a2,a3,x1,x2,x3,y1,y2,y3,q;
  a1:=a;  a2:=b;
  x1:=1;  y1:=0;
  x2:=0;  y2:=1;
  while (a2<>0) do
    a3:= a1 mod a2;
    q:= floor(a1/a2);
    x3:=x1-q*x2;
    y3:=y1-q*y2;
    a1:=a2;  a2:= a3;
    x1:=x2;  x2:= x3;
    y1:=y2;  y2:= y3;
  od;
  x:=x1;  y:=y1;
  RETURN(a1)
 end:
> mygcdex(2^10,6^50,'x','y');
```

1024

2.4 递归子程序

正如在一个子程序中我们可以调用其他的子程序一样(比如系统内部函数, 或者已经定义好的子程序), 一个子程序也可以在它的内部调用它自己, 这样的子程序我们称为递归子程序. 在数学中, 用递归方式定义的例子很多, 如 Fibonacci 数列：

$$f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2} (n \geq 2)$$

下面我们在 Maple 中利用递归子程序求 Fibonacci 数列的第 n 项：

```
> Fibonacci:=proc(n::nonnegint)::list
    if n<=1 then n;
```

```
      else
      Fibonacci(n-1)+Fibonacci(n-2);
    fi;
end:
> seq(Fibonacci(i),i=0..19);
 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181
> Fibonacci(20);
                                6765
```

但是，应用上述程序计算 Fibonacci(2000)时则不能进行，原因是要计算 F2000，就先得计算 F1999、F1998、……，无限制地递归会耗尽系统的堆栈空间而导致错误.

由数据结构有关理论知道，递归程序理论上都可以用循环实现，比如我们重写上面的程序如下：

```
> Fibonacci:=proc(n::nonnegint)
    local temp,fnew,fold,i;
    if n<=1 then n;
    else
       fold:=0;
       fnew:=1;
       for i from 2 to n do
           temp:=fnew+fold;
           fold:=fnew;
           fnew:=temp;
       od;
    fi;
end:
> Fibonacci(2000):

> time(Fibonacci(2000));
                                .019
```

利用循环，程序不像前面那么易懂了，但同时带来的好处也是不可忽视的，循环结构不仅不会受到堆栈的限制，而且计算速度得到了很大的提高.

我们知道，Maple 子程序默认情况下把最后一条语句的结果作为子程序的结果返回. 我们可以用 RETURN 命令来显式地返回结果，比如前面的递归结果可等价地写成：

```
> Fibonacci:=proc(n::nonnegint)
    option remember;
```

```
    if n<=1 then RETURN(n);
    fi;
    Fibonacci(n-1)+Fibonacci(n-2);
end:
```

程序进入 n<=1 的分支中，执行到 RETURN 命令就跳出程序，而不会接着执行后面的语句了．另外，使用 RETURN 命令在一定程度上可以增加程序的可读性.

在第二章中曾提到 Maple 的自动求导功能，下面通过实例说明。第一个例子是关于分段函数求导问题：

```
> F:=x->if x>0 then sin(x) else arctan(x) fi;
```

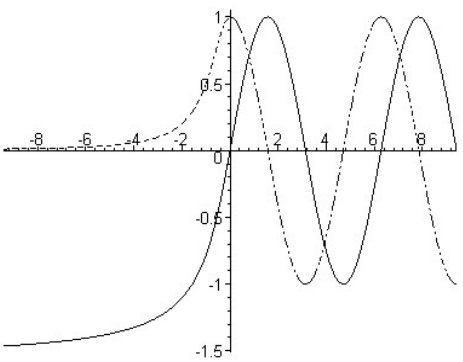
$$F := \mathbf{proc}(x)$$
$$\mathbf{option}\ \ operator, arrow;$$
$$\quad \mathbf{if}\ 0 < x\ \mathbf{then}\ \sin(x)\ \mathbf{else}\ \arctan(x)\ \mathbf{end\ if}$$
$$\mathbf{end\ proc}$$

```
> Fp:=D(F);
```

$$Fp := \mathbf{proc}(x)$$
$$\mathbf{option}\ \ operator, arrow;$$
$$\quad \mathbf{if}\ 0 < x\ \mathbf{then}\ \cos(x)\ \mathbf{else}\ 1/(1 + x\text{^}2)\ \mathbf{end\ if}$$
$$\mathbf{end\ proc}$$

将分段函数及其导数的曲线绘制在同一个坐标系中，实线为 F，虚线为 Fp：

```
> plot({F,Fp},-3*Pi..3*Pi,linestyle=[1,4],color=black);
```



第二个例子试图说明 Maple 能够完成对多个函数的自动求导：

```
> f:=proc(x)
  local s,t;
  s:=ln(x);
  t:=x^2;
  s*t+3*t;
```

```
end;
```

$$f := \mathbf{proc}\ (x)\ \mathbf{local}\ \ s, t;\ s := \ln(x);\ t := x\text{^}2;\ s \times t + 3 \times t\ \mathbf{end\ proc}$$

```
> fp:=D(f);
```

$$fp := \mathbf{proc}\ (x)\ \mathbf{local}\ \ tx, s, t, sx;\ sx := 1/x;\ s := \ln(x);\ tx := 2 \times x;\ t := x\text{^}2;\ sx \times t + tx \times s + 3 \times tx\ \mathbf{end\ proc}$$

程序 fp 是怎样构造的呢？对比 f 和 fp 的源代码可以帮助我们了解 Maple 自动求导机理：

在 fp 的源代码中，每一条赋值语句 V:=f(v1,v2, … vn)之前是 Vx:= fp(v1,v2, … vn)，此处 vi 是局部变量或形式参数，fp(v1,v2, … vn)是对 f(v1,v2, … vn)在形式上的微分，用该赋值语句的的导数取代最后一条语句或返回值。这种方法被称为向前自动微分法。在 Maple 内部对这种方法有一些限制：不能有递归程序，不能有记忆选项，不能对全局变量赋值，只能有常数的循环变量，等。

下面一个例子说明自动求导在计算时间和存储空间上的优越性，这是一个利用递归定义的 $f_1 = x, f_n = x^{f_{n-1}}\ (n > 1)$ 的幂函数的自动求导问题：

```
> f:=proc(x,n)
   local i,t;
   t:=1;
   for i to n do
       t:=x^t
   od;
   t
end;
```

$$f := \mathbf{proc}\ (x, n)\ \mathbf{local}\ \ i, t;\ t := 1;\ \mathbf{for}\ i\ \mathbf{to}\ n\ \mathbf{do}\ t := x\text{^}t\ \mathbf{end\ do}\ ;\ t\ \mathbf{end\ proc}$$

```
> f1_3:=D[1$3](f):        # 求 f 对 x 的三阶导数
> setbytes:=kernelopts(bytesused):     # 使用寄存器内存量
> settime:=time():                     # 开始计时
> f1_3(1.1,22);
```

$$18.23670379$$

```
> cpu_time:=(time()-settime)*seconds;     # 计时时间
```

$$cpu\_time := .230\ seconds$$

```
> memory_used:=evalf((kernelopts(bytesused)-setbytes)/1024*kbytes,5);#存储空间大小
```

$$memory\_used := 671.86\ kbytes$$

再利用符号微分重复上述步骤：

```
> f22:=unapply(f(x,22),x):
> setbytes:=kernelopts(bytesused):
> settime():
> (D@@3)(f22)(1.1);
```

$$18.23670379$$

```
> cpu_time:=(time()-settime)*seconds;
```

$$cpu\_time := 119.160\ seconds$$

```
> memory_used:=evalf((kernelopts(bytesused)-setbytes)/1024*kbytes,5);
```

$$memory\_used := 63423.\ kbytes$$

显然Maple自动求导在计算时间和存储空间上具有明显的优越性。巧妙利用自动微分解决较复杂的函数求导问题有十分现实的意义。

3 子程序求值

在 Maple 子程序中的语句，求值的方式和交互环境中的求值有所不同，这在某种程度上是为了提高子程序的执行效率而考虑的. 在交互式环境中，Maple 对于一般的变量和表达式都进行完全求值(除了数组、映射表等数据结构外). 比如先将 b 赋给 a，再将 c 赋给 b，则最终 a 的值也指向 c.

```
> a:=b:
  b:=c:
  a+1;
```

$$c + 1$$

但在子程序内部情况就不一样了，试看下述实验：

```
> f:=proc()
   local a,b;
   a:=b;
   b:=c;
   a+1;
end:
```

结果居然是：

```
> f();
```

$$b+1$$

这是因为 a 和 b 都是局部变量，Maple 对于子程序中的局部变量只进行一层的求值. 下面，我们针对子程序中的不同的变量，系统介绍其求值机制.

3.1 参数

子程序中的参数，就是那些在 proc( )的括号中的变量. 参数是一类特殊的变量，在调用子程序时，会把它们替换成实际的参数.

```
> sqrt1:=proc(x::anything,y::name)
      y:=x^2;
end:
> sqrt1(d,ans);
```

$$d^2$$

```
> ans;
```

$$d^2$$

我们再来试试别的参数，比如前面赋值过的 a 作为第一个参数，第二个参数仍用 ans，只不过加了单引号：

```
> sqrt1(a,'ans');
```

$$c^2$$

```
> ans;
```

$$c^2$$

事实上，Maple 在进入子程序以前就已经对参数进行了求值，因为是在子程序外进行求值，所以求值规则服从调用时所在的环境. 上面是在交互式环境下调用 sqrt1 得到的结果，作为对照看看在子程序内部调用上面的 sqrt1 子程序会有什么不同.

```
> g:=proc()
   local a,b,ans;
   a:=b;
   b:=c;
   sqrt1(a,ans);
end:
> g();
```

$$b^2$$

因为这次调用是在程序内部，所以只进行第一层求值，进入 sqrt1 的参数值为 b.

Maple 对于子程序的参数，都只在调用之前进行一次求值，而在子程序内部出现的地方都用这次求值的结果替换，而不再重新进行求值. 如前所述之 sqrt1：

```
> sqrt1:=proc(x::anything,y::name)
      y:=x^2;
      y+1;
end:
> sqrt1(d,'ans');
```

$$ans + 1$$

可见，对参数赋值的作用只有一个—返回一定的信息. 因为在子程序内部，永远不会对参数求值. 我们可以认为，参数是一个 0 层求值的变量.

3.2 局部变量和全局变量

Maple 对于局部变量只进行一层求值，也就相当于用 eval(a, 1)得到的结果. 这种求值机制不仅可以提高效率，而且还有着更重要的作用. 比如在程序中，我们往往需要把两个变量的值交换，一般地，我们使用一个中间变量来完成这样的交换. 但如果求值机制和交互式环境中一样，将达不到我们的目的. 试看，在交互式环境下，我们企图用这样的方法交换两个未被赋值的变量会有什么后果：

```
> temp:=p:
  p:=q:
  q:=temp;
```

$$q := q$$

不管在交互式环境下还是在程序中，Maple 对于每一个全局变量都进行完全求值，除非它是数组、映射表或者子程序. 对于数组、映射表和子程序, Maple 采用赋值链中的上一名称来求值.

除了上面这些特殊数据对象及下面一个特例外，总结起来，Maple 对于子程序的参数进行 0 层求值，对于局部变量进行 1 层求值，而对于全局变量，则进行完全求值.

对于上面说的求值规则，在 Maple 中还有个特例，就是“同上”操作符“%”需要引起注意. 就其作用来说，它应该属于局部变量. 在进入一个子程序时，Maple 会自动地把该子程序中的%设置成 NULL(空). 但是，对于这个“局部变量”，Maple 不遵循上面的规则，无论在那儿都会将其完全求值. 我们下面通过一个例子说明它在子程序中的求值机制：

```
> f:=proc()
```

```
   local a,b,c,d;
   print("At the beginning, [%] is", %);
   a:=b;
   b:=c;
   c:=d;
   a+1;
   print("Now [%] is",%);
end:
> f();
```

"At the beginning, [%] is"

"Now [%] is" , $d + 1$

可以看出, 尽管在子程序内部, 局部变量a仅进行一层求值, 但指代a+1的同上操作符却进行了完全求值, 得到 d+1 的结果.

3.3 环境变量

所谓环境变量就是从一个程序退出时系统自动设置的全局变量。Maple 中有几个内建的环境变量，如 Digits，Normalizer，Testzero, mod, printlevel 等. 从作用域来看, 它们应该算作是全局变量. 在求值上, 它们也像其他全局变量一样, 始终都是完全求值. 但是如果在子程序中间对环境变量进行了设置, 那么, 在退出该子程序时, 系统会自动地将其恢复成进入程序时的状态, 以保证当前行时的环境. 正因如此, 我们称其为环境变量. 试看下面程序:

```
> f:=proc()
   print("Entering f. Digits is", Digits);
   Digits:=20;
   print("Now Digits has become", Digits);
end:
> g:=proc()
   print("Entering g. Digits is", Digits);
   Digits:=100;
   print("Now Digits is", Digits);
   f();
   print("Back in g from f, Digits is", Digits);
end:
> f();
```

"Entering f. Digits is" , 10

```
                    "Now Digits has become" , 20
> g();
                    "Entering g. Digits is" , 10
                    "Now Digits is", 100
                    "Entering f. Digits is" , 100
                    "Now Digits has become" , 20
                    "Back in g from f, Digits is" , 100
> Digits;
                                10
```

可以看出，从子程序 g 中返回时，Digits 又被自动地设成了原来的值(10). 如果你需要自己定义一些具有这样的特性的环境变量，也十分简单—Maple 会把一切以_Env 开始的变量认作是环境变量.

4 嵌套子程序和记忆表

4.1 嵌套子程序

很多情况下，可以在一个子程序的内部定义另一个子程序. 实际上，我们在写这样的程序时常常没有意识到它是嵌套子程序. 用于对每一个元素操作的 map 命令在嵌套程序设计中很有用. 比如，编写一个子程序，它返回有序表中的每一个元素都将被第一个元素除的结果.

```
> nest:=proc(x::list)
      local v;
      v:=x[1];
      map(y->y/v,x);
  end:
> lst:=[2,4,8,16,32]:
> nest(lst);
                          [1, 2, 4, 8, 16]
```

在上面的程序中，我们在子程序中定义了另一个子程序: y->y/v，这个子程序中有一个变量 v, Maple 根据有效域的范围，认为它就是外面的子程序 nest 中的同名变量 v. 那么这是一个全局变量还是一个局部变量？显然两者都不是，因为把上面的例子中的内部子程序的 v 声明成 local 或者 global，都无法达到我们的目的.

那么, Maple 究竟是怎么样来判断一个变量的作用域呢？首先, 它自里向外, 一层一层地寻找显式地用 local、global 声明的同名变量, 或者是子程序的参数. 如果找到了, 它就将其绑定在外层的同名变量之上, 实际上就是将两个变量视为同一, 就像上面例子中的情况. 如果没有找到, 就遵循下面的原则：如果变量位于赋值运算符“:=”的左边, 就视其为局部变量, 否则均认为它是全局变量. 再看下面一个实例：

```
> uniform:= proc(r::constant..constant)
    proc()
    local intrange, f;
    intrange:=map(x->round(x*10^Digits),evalf(r));
    f:= rand(intrange);
    evalf(f()/10^Digits);
  end:
end:
> U:=uniform(cos(2)..sin(1)): Digits:=4:
> seq(U(),i=1..10);
```

.4210, .7449, -.06440, .7386, .7994, -.2166, -.1174, .02740, .2570, .3538

4.2 记忆表

记忆表的目的是为了提高计算效率, 它把每一个计算过的结果存储在一个映射表中, 所以在下一次用相同的参数直接调用以避免重复计算.

(1) remember 选项

在程序中可以加入 remember 选项来建立该子程序的记忆表. remember 选项可以把对时间和空间的指数增长的需求约简到线性增长。下面再建立一个 Fibonacci 子程序：

```
> Fibonacci:=proc(n::nonnegint)
    option remember;
    if n<=1 then RETURN(n) fi;
    Fibonacci(n-1)+Fibonacci(n-2);
end:
```

在我们计算了 F3 之后, 它的记忆表中就有了 4 项(记忆表中的子程序的第 4 个元素—用 op (4, …)来获得), 可以用 op 命令来检查一个子程序的记忆表：

```
> Fibonacci(3);
```

2

```
> op(4,eval(Fibonacci));
```

table([0 = 0, 1 = 1, 2 = 1, 3 = 2])

但是，过分地使用 remember 选项也并非全是好事，一方面，使用 remember 选项的

程序对内存的需求比正确写出的不使用 remember 选项的程序要高，另一方面，remember 选项不重视全局变量和环境变量的值(除了在 evalf 中对 Digits 特别重视外)，从而可能导致错误。还有一点，remember 选项与表、阵列、向量不能混用，因为 remember 无法注意到表中元素的变化。

(2) 在记忆表中加入项

通常调用具有记忆表的子程序并且得到计算结果，Maple 会自动地把结果加到记忆表中去，但也可以手动在记忆表中添加内容. 试看下面的程序：

```
> F:=proc(n::nonnegint)
    option remember;
    F(n-1)+F(n-2);
 end:
```

现在对记忆表赋值，注意这样赋值不会使程序被调用：

```
> F(0):=0;F(1):=1;
```

$$F(0) := 0$$

$$F(1) := 1$$

试调用程序得到想要的结果：

```
> F(100);
```

$$354224848179261915075$$

由于程序 F 的代码很短，很容易忘记对初始条件赋初值，这样会导致永不终止的递归，从而产生死循环. 因此，程序 **Fibonacci** 较 F 要好一些.

(3) 在记忆表中删除项

记忆表是映射表的一种，可以方便地在其中添加或者删除特定的项，和一般变量一样，删除一个表项只需要将其名称用 evaln 赋给它本身就行了. 假设，因为输入错误，我们在前面一个程序中加入了一个错误的项：

```
> Fibonacci(2):=2;
```

$$Fibonacci(2) := 2$$

查看记忆表：

```
> T:=op(4,eval(Fibonacci));
```

$$T := table([2 = 2])$$

再将相应的项删除掉，则又回复原来的记忆表：

```
> T[2]:=evaln(T[2]);
```

$$T_2 := T_2$$

```
> op(4,eval(Fibonacci));
```

$$\text{table}([0 = 0, 1 = 1, 2 = 1, 3 = 2])$$

Maple 也可以自动删除子程序的记忆表，如果我们子程序定义时声明 system 选项，系统将会在回收无用内存时将记忆表删除，就如同大多数系统内部子程序一样.

对于记忆表的使用范围，一般地，它只适用于对于确定的参数具有确定的结果的程序，而对于那些结果不确定的程序，比如用到环境变量、全局变量，或者时间函数等的程序，就不能使用，否则将会导致错误的结果.

5 返回子程序的子程序

在所有的子程序中，编写返回值为一个子程序的程序所遇到的困难也许是最多的了. 编写这样的程序，需要对 Maple 的各种变量的求值规则、有效语句有透彻的理解. Maple 一些内部子程序就有这样的机制，比如随机函数 rand，返回的就是一个子程序，它会在给定的范围内随机地取值. 下面以牛顿迭代法和函数的平移为例说明这种程序设计技巧。

5.1 牛顿迭代法

牛顿迭代法是用来求解非线性方程数值解的常用方法之一. 首先，选择一个接近于精确解的初值点，接着，在曲线上该初值点处作切线，求出切线与 x 轴的交点. 对于大部分函数，这个交点比初值点更要接近精确解. 依次，重复迭代，就可以得到更精确的点. 牛顿迭代法的数学过程为：

对于方程 $f(x)=0$，选定初值 x_0，然后利用递推公式 $x_{k+1}=x_k-\dfrac{f(x_k)}{f'(x_k)}$ 来逐步得到精确解. 试看下面求解方程 $x-2\sqrt{x}=0$：

```
> MakeInteration:=proc(expr::algebraic,x::name)
    local interation;
    interation:=x-expr/diff(expr,x);
    unapply(interation,x);
  end:
> expr:=x-2*sqrt(x);
```

$$expr := x - 2\sqrt{x}$$

```
> Newton:=MakeInteration(expr,x);
```

$$Newton := x \to x - \frac{x - 2\sqrt{x}}{1 - \frac{1}{\sqrt{x}}}$$

然后我们利用已经得到的迭代递推函数 Newton，可以用来求解这个超越方程：

```
> x0:=3.0;
```

$$x0 := 3.0$$

```
> to 4 do x0:=Newton(x0);od;
```

$$x0 := 4.098076213$$
$$x0 := 4.000579795$$
$$x0 := 4.000000019$$
$$x0 := 4.000000001$$

可以看到，经过 4 次迭代结果已经相当满意了.

在上面的子程序 MakeInteration 的输入参数类型是代数表达式，我们可以编制用函数作为参数的子程序. 由于函数的求值具有特殊性，在程序中需要用 eval 对其进行完全求值.

```
> MakeInteration:=proc(f::procedure)
    (x->x)-eval(f)/D(eval(f));
end:
```

由于输入参数是一个程序，就不需要再显式地指出自变量了，我们用 D 运算符求它的导函数，并且作运算. 这里，**(x->x)**表示恒等函数 $f(x) = x$. 从这里，我们也可以知道，在 Maple 中，不仅代数表达式可以进行运算，具有相同自变量的函数也可以相互进行运算. 利用上面的程序求解超越方程 $x^2 - \cos x = 0$：

```
> g:=x->x^2-cos(x);
```

$$g := x \to x^2 - \cos(x)$$

```
> Newton:=MakeInteration(g);
```

$$Newton := (x \to x) - \frac{x \to x^2 - \cos(x)}{x \to 2\,x + \sin(x)}$$

```
> x0:=1.0;
```

$$x0 := 1.0$$

```
> to 4 do x0:=Newton(x0);od;
```

$$x0 := .8382184099$$

$$x0 := .8242418682$$

$$x0 := .8241323190$$

$$x0 := .8241323123$$

5.2 函数的平移

考虑这样一个简单的问题，已知一个函数 f，我们需要得到另一个函数 g，满足 $g(x) = f(x+1)$. 在数学中，这样的操作称为平移. 在 Maple 中，可以用简单的程序实现：

```
> shift:=(f::procedure)->(x->f(x+1));
```

$$shift := f::procedure \to x \to f(x+1)$$

```
> shift(sin);
```

$$x \to \sin(x+1)$$

```
> shift((x)->x);
```

$$x \to (x \to x)(x+1)$$

```
> %(1);
```

$$2$$

```
> shift:=proc(f::procedure)
    local x;
    unapply(f(x+1),x);
  end;
> shift(sin);
```

$$x \to \sin(x+1)$$

```
> h:=(x,y)->x*y;
```

$$h := (x, y) \to x\,y$$

```
> shift(h);
Error, (in h) h uses a 2nd argument, y, which is missing
> shift:=(f::procedure)->(x->f(x+1,args[2..-1]));
```

$$shift := f::procedure \to x \to f(x+1, \mathrm{args}_{2\,..\,-1})$$

```
> shift(sin);
```

$$x \to \sin(x+1, \mathrm{args}_{2\,..\,-1})$$

```
> hh:=shift(h);
```

$$hh := x \to h(x+1, \mathrm{args}_{2\,..\,-1})$$

```
> hh(x,y);
```

$$(x+1)\,y$$

```
> h:=(x,y,z)-> y*z^2/x;
```

$$h := (x, y, z) \to \frac{y\,z^2}{x}$$

```
> hh(x,y,z);
```

$$\frac{y\,z^2}{x+1}$$

```
> shift := proc(f::procedure)
    loca l F;
    subs('F'=eval(f),x->F(x+1,args[2..-1]));
  end;
> H:=shift(h);
```

$$H := x \to \left((x, y, z) \to \frac{y\,z^2}{x} \right)(x+1, \mathrm{args}_{2\,..\,-1})$$

```
> h:=45;
```

$$h := 45$$

```
> H(x,y,z);
```

$$\frac{y\,z^2}{x+1}$$

```
> f:=proc(x) x^2 end;
```

$$f := \mathbf{proc}\ (x)\ x\text{^}2\ \mathbf{end\ proc}$$

```
> op(f);
```

$$\mathbf{proc}\ (x)\ x\text{^}2\ \mathbf{end\ proc}$$

6 局部变量的进一步探讨

局部变量的作用域是当前的子程序，而且，对于该子程序的每一次运行，局部变量都是不同的变量. 简单来说，每一次调用子程序，都产生一些新的变量；如果你两次调用

相同的子程序，那么，第二次所用的局部变量和第一次的局部变量是不同的.

有时候，在退出一个子程序时，局部变量并不消失. 比如在子程序中将局部变量作为结果返回了，那么在子程序结束后，这些局部变量仍然存在. 对于这些变量，常常难以捉摸，因为它们可以和全局变量同名，但却都是不同的变量—它们在系统内存中占有不同的位置，修改其中一个变量的值不会影响到其他值.

为了对这种情况有一定的了解，我们首先定义一个返回局部变量的子程序：

```
> make_a:=proc()
    local a;
    a;
  end;
```

检查所产生的变量是否相同，可以根据中间元素的唯一性(自动删除相同的元素).

```
> test:={a,a,a};
```

$$test := \{a\}$$

```
> test:=test union {make_a()};
```

$$test := \{a, a\}$$

每次调用程序 make_a 生成的同名变量，是互不相同的，而且，它们和全局变量 a 也不相同. 注意：Maple 并不是只依据变量名来区别变量的. 在交互式环境中键入变量 a 时，Maple 认为它是全局变量. 所以，可以用上面的同名变量集合 test 中很容易地找到全局变量.

```
> member(a,test,num);
```

$$true$$

```
> num;
```

$$1$$

同名变量的最大用途在于可以用来写出一串通常用 Maple 很难得到的表达式. 举例来说，Maple 会自动地把表达式 a+a 代简成 2a，要得到表达式 a+a=2a 是一件不容易的事. 现在，我们可以利用上面的子程序 make_a 来轻松地写出这样的式子.

```
> a+make_a()=2*a;
```

$$a + a = 2\,a$$

对于 Maple 来说，虽然它们具有相同的变量名，但等号左边的两个 a 是不同的，所以，它没有自动地把它化简成 2a. 对于全局变量，可以用变量名直接引用，于是对于另一个用 make_a 得到的变量，它的引用就需要费一番周折. 我们可以通过 remove 命令去掉等式左边的全局变量 a 来得到另一个 a.

```
> eqn:=%;
```

$$eqn := a + a = 2\,a$$

```
> another_a:=remove(x->eval(x=a),lhs(eqn));
```

$$another\_a := a$$

现在全局变量 another_a 指向了那个难以捉摸的 a，我们要做的是把那个 a 赋成全局变量 a，用赋值语句显然是不行的，我们用 assign 命令将其赋值. assign 命令和赋值语句的不同在于被赋值的变量是作为参数传给 assign 的，所以 Maple 会自动地将它求值为它所指的 a，这样，就可以为 a 赋值了.

```
> assign(another_a);
> eqn;
```

$$a + (\ ) = 2\,a$$

这时，等式两边都是作为全局变量的 a 了. 我们用 evalb 可以检验等式的正确性(虽然这是显然的，但对于复杂的等式，这却是有必要的).

```
> evalb(%);
```

$$false$$

在这一小节中，我们引入了令人费解的“越界”的局部变量. 而实际上，可能在以前的学习中就已经碰到过这样的问题. 已经学习过的 assume 命令，它将一个变量赋予另一个有确定范围的变量—新的变量只是在原来的变量后面加上一个“~”. 这个具有波浪线的变量就是一个“越界”的局部变量—如果在交互式环境中输入它(注意：需要用一对反向撇号“`  `”括起来), Maple 将不能识别，因为编译程序认为输入的是全局变量.

```
> assume(b>0);
> x:=b+1;
```

$$x := b\sim + 1$$

```
> subs('b'=c,x);
```

$$b\sim + 1$$

Maple所做的是把局部变量b~赋给全局变量b. 如果我们将b另外赋值，b~仍然存在，则由它所生成的表达式也仍然存在.

```
> b:='b';
```

$$b := b$$

```
> x;
```

$$b\sim + 1$$

7 扩展 Maple 命令

虽然编写程序可以满足各种不同的需要，但是有时候，根据需要扩展 Maple 原有命令可以事半功倍，事实上，Maple 更大的特点提供了一种“平台”.

在大多数现代编程语言中，都支持自定义的数据结构和数据类型，Maple 也具有这

样的功能，只需将一个结构类型赋值给`type/TypeName`， TypeName 就可以作为一个类型使用了. 这样，对于结构本身只需要写一次，减少了出错的可能性，也减少了工作量.

```
> `type/Variables`:={name,list(name),set(name)};
```

$$type/Variables := \{name, \mathrm{list}(name), \mathrm{set}(name)\}$$

```
> type(x,Variables);
```

$$true$$

```
> type({x[1],x[2]},Variables);
```

$$true$$

在这个例子中，我们把几个类型的集合赋给了`type/Variables`，也就是说我们定义的数据类型 Variables 是这几个类型的集合. 所以，不管单个变量还是变量的集合，都是 Variables 类型的.

另外，还可以将一个子程序赋给`type/TypeName`. 在测试一个数据对象是否具有 TypeName 类型时，Maple 会自动调用该子程序. 但是子程序必须返回布尔值(true 或者 false).

作为例子，下面定义一个全排列(permutation)的数据类型，也就是检测一个有序表是否含有从 1 到 n 的所有自然数(且每一个只出现一次). 提取有序表中所有元素的集合，再与 1 到 n 的自然数集比较，就可以实现这一要求.

```
> `type/permutation`:=proc(p)
    local i;
    type(p,list) and {op(p)}={seq(i,i=1..nops(p))};
end;
> type([2,3,1,4],permutation);
```

$$true$$

```
> type([1,2,3,1],permutation);
```

$$false$$

自定义的类型检测函数可以具有多于一个的参数. 比如，要检测一个表达式 expr 是否具有类型 TypeName(parameters)，Maple 就会用如下形式调用检测函数 TypeName(expr,parameters). 例如，定义一元线性表达式类型 LINEAR，它是一个未知变量的一次多项式.

```
> `type/LINEAR`:=proc(f,v::name)
    type(f,polynom(anything,v)) and degree(f,v) =1;
end;
```

$$type/LINEAR := \mathbf{proc}(f, v::name)\ \mathrm{type}(f, \mathrm{polynom}(anything, v))\ \mathbf{and}\ \mathrm{degree}(f, v) = 1\ \mathbf{end\ proc}$$

```
> type (x^2,LINEAR(x));
```

$$false$$

```
> type (a*x+b,LINEAR(x));
```

true

8 程序调试

任何语言编写的程序都不能保证没有错误，Maple 也一样。有时候错误非常隐蔽，仅仅通过检查源程序或者数值试验可能无法排队程序中的错误。Maple 中提供了一些实用的调试工具。

Maple 的主要调试工具之一是 **printlevel**, 该工具对程序的执行过程产生详细的计算，包括程序调用的参量、子程序计算步骤和结果. 可以合理地设置 **printlevel** 来控制调试的深度(其默认值为1), 然后直接调用所需调试的程序即可看到程序的执行过程. 此时, 将打印出程序内的代码. 当 **printlevel** 的值增加, 附加的信息将会被打印出来, 这些信息当然对程序调试是重要的.

另一个可选择的调试工具是使用函数 **trace**, 该函数以被跟踪的函数作为参量. 除了指定程序的参量和结果, 跟踪过程会提供每一语句的执行结果. 除此之外在 trace 的输出上没有别的控制. 函数 untrace 被用作去掉 trace 的影响.

命令 **showstat** 可以显示源程序、语句对应的编号、断点的设置、以及当前的中断位置。它的调用格式为 **showstat**(*procdure,number*)，其中 procedure 是需要显示的子程序名称。对于无条件断点，showstat 将在其语句编号后显示星号“*”；对于条件断点，则显示问号“？”。可选参数 number，指定语句号或者语句号的范围。这时，将只显示所指定的语句，其余的语句都用“…”表示。

而函数 **stopat** 设置断点，它的调用格式为：**stopat**(*procedure, number ,condition*)，其中 procedure 是子程序名称；number 是需要设置中断的语句编号，它可以省略，默认情况下将在该程序的第一条语句之前设置断点；condition 是在该断点处中断的条件，它是一个布尔表达式，可以包含任意全局变量，该子程序中的局部变量和参数，在省略这一参数时，stopat 将设置无条件断点。无条件断点的语句号后面将用星号标记，而条件断点则吉林省记以问号。

命令 **unstopat** 用来清除程序中的断点，它的调用格式为：unstopat(procedure , number)，其中 number 是需要删除的断点的语句编号，也是可选参数，如果省略，将删除该程序中所有的断点。

在用户自己编写的源程序中，也可以用调用函数 DEBUG 的方法加入一个显式的断点：**DEBUG**(*condition*)，其中 *condition* 是一个布尔表达式，表示中断的条件，在它的值为 true 时引起中断，如果为 false 或者 FAIL 时将忽略这一中断语句；它是可先参数，默认情况下交导致无条件中断。DUBEG 的参数也可以是非布尔类型的表达式，这使用程序将无条件中断，并且显示这个表达式的值。

显式断点的调用结果和调试断点的结果相同，都是使程序中断，并显示上一执行语句的结果以及下一条将要执行的语句。

监视断点可以对变量进行监视，用命令 stopwhen 加入，它有两种调用格式：

stopwhen(*globalVariableName*)或 **stopwhen**([*procedure , VariableName*]). 第一种调用设置了对全局变量 *globalVariableName* 的监视，还可以用它来设置对环境变量的监视；第二种调用则在程序 procedure 内部监视变量 *VariableName*（可以为全局变量或者局部变量）。

在 Maple 对所监视的变量进行赋值以后，监视断点就中断程序的运行，并显示对应的赋值语句（对于所赋的值已进行化简），而不是显示结果。然后，依照惯例显示下一条语句。需要注意的是，监视断点在赋值之后引起中断，而不是赋值以前。

要清除监视断点，可以调用命令 unstopwhen，它的参数和 stopwhen 完全一样，也可以在调试环境中调用。如果不提供任何参数，unstopwhen 将清除所有的监视断点。

出错断点可以截获 Maple 的错误住处并引起程序中断，进入调试状态。命令 stoperror 可以设置出错中断，它的调用格式为 **stoperror** (“*error message* “)，其中 *errormessage* 指定了需要截获的 Maple 出错信息，也可以使用 all 作为参数，以截获所有的错误。stoperror 的返回值是反有设置出错断点的错误有序表.

Maple 中还有一个实用工具 **mint**，它可以检查 Maple 句法错误, 而且比运行调试工具速度要快得多. 另外，该函数还可以检查程序中是否有全局变量以及所有的局部变量是否都使用了.

虽然 Maple 提供了功能强大的程序调试工具，利用它可以监视程序的内部执行过程，可以更快地找到程序的错误并加以修改. 但是，由于程序设计本身的复杂性，因此，在编写程序时应严格按照相关算法设计程序，万勿把全部希望寄托在程序的调试上.