为什么函数程序设计如此重要

John Hughes, Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN. rjmh@cs.chalmers.se

这篇论文最早写于1984年，之后在 Chalmers 大学流传了多年。稍微修订过一些的两个版本分别在1989和1990年发布（[2, Hug89]和[3, Hug90]）。下面这个版本以最早的 nroff 源文件为基础，遵循 LaTeX 的习惯做了些修改以便更接近可出版的效果，同时也包含一两处小错误修正。各位对于老旧的排版，以及没有使用 Haskell 来编写样例代码可以尽情的抱怨！

摘要

随着软件变得越来越复杂，使软件具有良好的结构也显得越来越重要。结构良好的软件易编写、易调试，并且提供了一组可以重用的模块，从而降低未来的编程费用。传统的语言在将问题如何模块化方面有概念上的局限性。函数程序设计语言越过了这些局限性。在这篇论文中，我们将提出函数程序设计的两个非常有助于模块化的特性：高阶函数和惰性计算。我们将以表和树、数值算法、alpha-beta启发算法（人工智能中用于博弈的算法）作为例子。因为模块化是程序设计成功的关键，函数程序设计语言对现实世界是至关重要的。

目录

1 简介

本论文试图向"现实世界"证明函数程序设计是至关重要的，并且帮助函数程序设计员充分探讨函数程序设计的优点。

函数程序设计意指每个程序均为一个函数。主程序本身也是一个函数，它将程序的输入作为函数的自变量，程序的输出作为函数的结果。一般地，主程序是由其他函数定义的，而这些函数也是由另外一些函数来定义的，直到最底层由程序设计语言提供的原始函数。所有这些函数都非常类似于数学上的函数，在本文中函数的定义将使用普通的等式。我们将沿用 Turner 设计的语言 Miranda(TM)[8, Tur85]的表示法，但是，即使没有函数程序语言背景的读者也是容易理解的（Mirande是 Research Software Ltd. 的商标）。

函数程序设计的特点和优点通常可以大致概括如下。函数程序不包含赋值语句，所以，一个变量一旦给定一个值，其值将永远不变。更一般地说，函数程序绝对没有副作用。一个函数的调用只计算其值，没有任何其他作用。这一特点可以排除程序错误的一个主要渊源，而且使得程序的执行顺序无关紧要-因为没有副作用可以改变一个表达式的值，表达式可以在任何时候被计算。因此，程序员不需要担忧控制流的编写。因为表达式可以在任何时候被计算，我们可以自由地将变量用其值代替，或者将值代换为变量，也就是说，程序是"引用透明的"（referential transparent）。这种自由度使得函数程序比传统的程序在数学上更容易处理。

但是，如果函数程序设计外行并不认为这些是优点，我们也不应该吃惊。这些优点讲了许多函数程序设计不是什么（没有赋值语句，没有副作用，没有控制流），但没有说它是什么。对于那些更喜欢物质利益的人们，这些"优点"不能非常令人信服。

函数程序设计宣称其具有极大的物质利益——函数程序设计员的生产效率比一般程序员高一个数量级，因为函数程序比普通程序短一个数量级。为什么是这样呢？唯一似乎可信的理由是普通程序的 90% 由赋值语句构成，而这些赋值语句在函数程序中可以全部省去。这显然是荒唐的——如果省略赋值语句能带来这么大的利益，FORTRAN程序员不会编写了20年的程序。通过省略某些特征而使得一个语言表达力更强，无论这些特征如何糟，这在逻辑上是不可能的。

即使一个函数程序设计员也会对这些所谓的优点不满，因为这些对于挖掘函数程序语言的能力毫无帮助。我们不可能编写一个特别缺乏赋值语句，或者特别引用透明的程序。如果没有程序质量的尺度，我们也便没有奋斗的目标。

显然，这样刻画函数程序设计是不恰当的。我们必须既能解释函数程序设计的表达力，又能给函数程序设计员清楚地指出应该追求的目标。

2 与结构程序设计的对比

对比函数程序设计与结构程序设计是很有帮助的。在过去，结构程序设计的特征与优点大致概括如下：结构化程序不包含goto语句。结构程序中的程序块没

有多入口和多出口。结构程序在数学上比非结构程序更容易处理。结构程序设计的这些优点非常类似于我们前面讨论的函数程序的优点。这些优点基本上是否定语句，已经陷入了有关"基本goto"的无意义讨论之中。

作为事后诸葛亮，显然，结构程序设计的这些性质并没有点到问题的核心。结构程序与非结构程序最重要的区别在于结构程序是由模块化的方式设计的。模块化设计带来了生产率的极大提高。首先，小模块容易快速编写。其次，通用模块可以被重用，因此加快了随后的开发过程。第三，模块可以被单独测试，减少了调试的时间。

Goto语句的不存在及其效应与模块化没有多少关系，它仅对"小规模程序设计"有帮助，而模块化有助于"大规模程序设计"。因此，尽管程序员需要做更多的工作，程序员仍然可以享受使用 FORTRAN 或者汇编语言进行结构化程序设计带来的好处。

现在人们普遍认为模块化设计是程序设计成功的关键，程序设计语言如 Modula-II[9, Wir82]，Ada[5, oD80] 和标准 ML 都包含了特别用于改善模块化设计的特征。然而，有一点经常被忽视了。当我们编写一个模块化程序以解决一个问题时，通常将一个问题分解为一些子问题，然后解决这些子问题，并将这些解结合起来。分解问题的方法直接依赖于将这些解粘合在一起的方法。因此，欲在概念上增强模块化问题的能力，我们必须在程序设计语言中提供新的粘合方法。复杂的作用域规则和分别编译只对书写细节有帮助，而并没有提供分解问题的新概念工具。

让我们返回函数程序设计。我们将在下面论证函数程序设计提供了两个新的、非常重要的粘合方法。我们将给出许多可以用新方法模块化的，而且大大简化的程序例子。这是函数程序设计的能力的关键——它能极大地改进模块化。这也是函数程序设计员应该争取的目标——设计简短和更通用的模块，然后用我们即将描述的新方法将其粘合在一起。

3　将函数粘合在一起

第一种粘合方法可以将简单的函数粘合在一起形成更复杂的函数。我们将用一个简单的表处理问题——把表中的元素加在一起--来解释之。我们定义表如下:

```
listof X :: = nil | cons X (listof X)
```

其意义为: 一个 X（X 是任意的）上的表或者是 nil，即一个不含任何元素的表；或者是 X 的一个元素及 X 上的一个表的 cons。一个 cons 表示其第一个元素为 X，其第二个及其后的元素为 X 上的表的元素。X 在这里表示任意类型，例如，如果 X 是"整数"，那么，以上的定义表示一个整数表或者是空，或者是一个整数和另一个整数表的 cons。我们将按照惯例，将一个表的元素列在一对方括弧内表示之，而不显式地使用 nil 和 cons。这只是为了表示的方便。例如:

```
[]      ⇒ nil
[1]     ⇒ cons 1 nil
[1,2,3] ⇒ cons 1 (cons 2 (cons 3 nil))
```

我们可以用一个递归函数 sum 将一个表的元素加在一起。sum 需要定义在两种参数上：一个空表(nil)，以及一个 cons。因为空表的和为 0，我们定义：

```
sum nil = 0
```

因为一个cons的和是其第一个元素与另一个表之和相加。故有：

```
sum (cons num list) = num + sum list
```

检查以上定义，我们发现在计算和时只有带筐的部分是特定的：

```
          +---+
sum nil = | 0 |
          +---+
                            +---+
sum (cons num list) = num   | + | sum list
                            +---+
```

这表示 sum 的计算可以通过把一个一般递归模式与带框部分粘合起来模块化。这种递归模式通常称为归约（reduce），因此 sum 可以表示为：

```
sum = reduce add 0
```

为方便起见，我们给 reduce 传递了一个二元函数 add 而非一个运算符号。add 的定义如下：

```
add x y = x + y
```

函数 reduce 的定义可以将 sum 参数化，由下列方程式导出：

```
(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

这里我们使用了括号 (reduece f x) 以表示其代表 sum。通常括号可以省略，所以 ((reduce f x) l) 可写作 (reduce f x l)。像 reduce 这样的三元函数只作用于两个参数时可视为一元函数。一般地，一个n元函数作用于 m(<n) 个参数可视为 n−m 元函数。今后我们将沿用此习惯。

将 sum 如此模块化后，我们可以重用一些函数。其中最有趣的部分是 reduce，可用于编写求表中元素积的函数：

```
product = reduce multiply 1
```

还可用于测试一个布尔表中是否存在元素为 true:

```
anytrue = redeuce or false
```

或者一个布尔表的所有元素全为 true:

```
alltrue = redeuce and true
```

一种理解 (reduce f a) 的方法是将表中的所有 cons 用 f 代替，所有的 nil 用 a 代替。以 [1,2,3] 为例，因为它表示:

```
cons 1 (cons 2 (cons 3 nil))
```

所以 (reduece add 0) 将上表转换为:

```
add 1 (add 2 (add 3 0)) = 6
```

同样，(reduce multiply 1) 将上表转换为:

```
multiply 1 (multiply 2 (multiply 3 1)) = 6
```

现在我们可以明显地看出 (reduce cons nil) 恰恰复制一个表。因为两个表的连接可以通过将第一个表的元素 cons 到第二个表上获得，所以:

```
append a b = reduce cons b a
```

例如,

```
append [1,2] [3,4] = redeuce cons [3,4] [1,2]
                   = (reduece cons [3,4]) (cons 1 (cons 2 nil))
                   = cons 1 (cons 2 [3,4])
(□cons □□cons□[3□4] □□nil)
                   = [1,2,3,4]
```

将表中所有元素加倍的函数可以定义为:

```
doubleall = reduce doubleandcons nil
```

其中 doubleandcons num list = cons (2*num) list

函数 doubleandcons 可以进一步模块化，第一步为:

```
doubleandcons = fandcons double
fandcons f el list = cons (f el) list
```

其中 double n = 2*n

然后,

```
fandcons f = cons . f
```

这里"."（标准函数合成运算符）定义为:

```
(f . g) h = f (g h)
```

将fandcons作用于一些参数，我们可以看出fandcons的定义是正确的:

```
fandcons f el = (cons . f) el
              = cons (f el)
```

所以 fandcons f el list = cons (f el) list

最后的结果是:

```
doubleall = reduce (cons . double) nil
```

进一步模块化， 我们有:

```
doubleall = map double
map f = reduce (cons . f) nil
```

其中 map 将函数 f 应用于表的所有元素。map是又一个常用函数。

我们还可以把距阵的所有元素相加，其中矩阵表示为一个由表构成的表。其定义为:

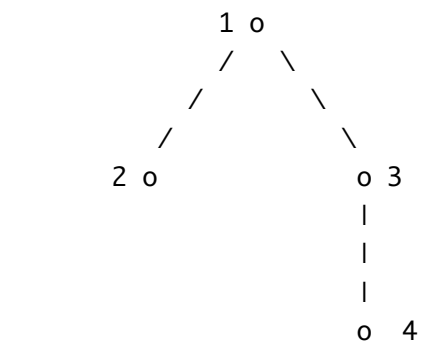
```
summatrix = sum . map sum
```

在这里，map sum 计算每一行的和， 而最左边的 sum 将这些和相加，从而得到整个距阵之和。

这些例子应该足以使读者相信，模块化带来的益处是极大的。通过把一个简单函数 sum 模块化为一个"高阶函数"和一些简单参数的组合，我们得到一个可以用之轻易地编写许多表上函数的构件（reduce）。我们不应该停留在表上的函数。作为另一个例子，考虑如下定义的数据类型有序标号树:

```
treeof X ::= node X (listof (treeof X))
```

此定义说，X 上的树是一个标有 X 上元素的节点和一列子树，每个子树都是 X 上的树。例如，下图所示的树:



可以表示为:

```
node 1
   (cons (node 2 nil)
      (cons (node 3
            (cons (node 4 nil) nil))
         nil))
```

我们将直接定义一个类似于 reduce 的函数 redtree，而非由一个例子抽象出来。回想 reduce 带有两个参数，一个用以代替 cons，一个用以代替 nil。因为树是由 node，cons 和 nil 构成的，所以 redtree 必须带三个参数，以代替以上三个部分。因为树和表具有不同的类型，我们需要在树和表上各定义一个函数。为此，我们定义:

```
redtree f g a (node label subtrees) =
      f label (redtree' f g a subtrees)
redtree' f g a (cons subtree rest) =
      g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a nil = a
```

通过粘合 redtree 和其他函数，我们可以定义许多有趣的函数。例如，对于一个标签为数的树，所有标签之和可由下列函数求得:

```
sumtree = redtree add add 0
```

以先前的树为例，应用 sumtree 得到:

```
add 1
   (add (add 2 0)
      (add (add 3
            (add (add 4 0) 0))
         0))
= 10
```

一棵树上所有标签构成的表可由下式求得:

```
labels = redtree cons append nil
```

将其应用于先前的树得到如下的表:

```
cons 1
   (append (cons 2 nil)
      (append (cons 3
            (append (cons 4 nil) nil))
            nil))
= [1,2,3,4]
```

最后，我们可以定义一个类似于 map 的函数，它将一个函数作用于树上的所有标签:

```
maptree f = redtree (node . f) cons nil
```

所有这些结果的取得是因为那些在常规语言中不可拆分的函数在函数程序语言中可以表示为一些部分的组合: 一个通用的高阶函数和一些特殊函数。这些高阶函数一旦定义好，许多运算的定义变得非常容易。对于每一个数据类型，应该定义相应的高阶函数处理之。这样不仅使得数据结构的处理变得容易，而且也使得数据类型表示的细节知识局部化。与常规语言的最好类比是语言的可扩充性——函数程序设计语言好像在任何需要的时候都可以扩充以新的控制结构。

4 将程序粘合在一起

函数程序语言提供的另一种粘合方式可以将多个程序粘合在一起。记住，一个完全函数程序是一个由它的输入到它的输出的函数。如果f和g是这样的程序，那么（g . f）也是一个程序，当它作用于其输入时，其计算方式如下:

```
g (f input)
```

程序 f 计算其输出，此输出又成为程序 g 的输入。通常可以将 f 的输出存储在一个临时文件中来实现这个计算过程。但问题是临时文件可能占用太多的内存，用这种方式粘合程序是不可行的。函数语言为此提供了一种解决方法。程序 f 和 g 严格地同步执行。只有 g 试图读输入时 f 才开始运行，并且 f 只运行到其输出足以满足g的输入要求。然后 f 暂停运行，直至 g 试图读取下一个输入。如果 g 没有读取 f 的所有输入而停止运行，那么f便退出运行，这是一个意外的收获。f 甚至可以是一个非终止的程序，产生无穷的输出，因为 f 的运行将随着 g 的运行终止而被迫结束。这便使得运行终止条件与循环体分离开来，这是一个非常有力的模块化工具。

由于这种计算方法使 f 的运行达到最小，故称之为"惰性计算"。惰性计算使得将一个程序模块化为一个产生大量可能结果的生成器，以及一个从中选择适当答案的选择器成为可能。虽然其他系统也允许程序以这种方式一起运行，但是，只有函数语言对每个函数调用一致地使用惰性计算，允许程序的每一部分都可以用这种方式模块化。惰性计算也许是函数程序员最有力的模块化工具。

4.1 Newton-Raphson平方根

我们将编写几个数值算法来显示惰性计算的威力。首先考虑 Newton-Raphson 求平方根算法。这个求一个数 N 的平方根的算法是：由一个初始近似值 a0 开始，应用下面的公式计算平方根更精确的近似值:

```
a(n+1) = (a(n) + N/a(n)) / 2
```

如果近似值收敛于某个极限 a, 那么

```
a = (a + N/a) / 2
```

那么

```
2a = a + N/a
a = N/a
a * a = N
a = squareroot(N)
```

事实上，近似值很快收敛于一个极限。平方根程序允许一个误差 eps, 并当两个连续近似值之差小于 eps 时停止运行。

此算法通常是如下编写的（FORTRAN）:

```
C  N IS CALLED ZN HERE SO THAT IT HAS THE RIGHT TYPE
X = A0
Y = A0 + 2.*EPS
C  THE VALUE OF Y DOES NOT MATTER SO LONG AS ABS(X-Y).GT.EPS
```

```
100  IF (ABS(X-Y).LE.EPS) GOTO 200
     Y = X
     X = (X + ZN/X) / 2
     GOTO 100
200  CONTINUE
     C  THE SQUARE ROOT OF ZN IS NOW IN X
```

在常规语言中这个程序是不可分解的。我们将使用惰性计算把这个程序用更模块化的方式表示出来，然后展示这些函数的其他用途。

因为 Newton-Raphson 算法计算一个近似值序列，很自然地在程序中可以用一个近似值列显式地表示之。每个近似值均可用下面的函数由前一个近似值导出:

```
next N x = (x + N/x) / 2
```

所以，(next N)是将一个近似值映射到下一个近似值的函数。称此函数为f, 近似值序列变为:

```
[a0, f a0, f(f a0), f(f(f a0)), …]
```

我们可以定义一个函数计算此序列:

```
repeat f a = cons a (repeat f (f a))
```

所以，近似值序列可由下式计算得:

```
repeat (next N) a0
```

repeat 是有"无穷"输出的函数例子，但是这并不要紧，近似值的计算不会超出程序所要求的范围。无穷只是潜在的: 如果需要，任意个数的近似值都可被计算出来，repeat 本身并未设置任何限制。

求平方根的剩余工作由函数 within 完成，给定一个误差和一列近似值，within 在此列中寻找两个连续近似值，其差不超过给定的误差。 其定义为:

```
within eps (cons a (cons b rest)) =
   = b,  if abs(a-b) <= eps
   = within eps (cons b rest), otherwise
```

将这几部分合在一起:

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

我们可以把求平方根的组成部分以不同的方式组合。一种可能的修改是要求相邻近似值之比趋向于 1，而非其差趋向于 0。这种度量对于非常小的数（此时相邻的近似值之差可能小至计算难以开始）和非常大的数（此时舍入误差可能大于给定的误差）更合适。我们只需定义一个替换within的函数:

```
relative eps (cons a (cons b rest)) =
   = b, if abs(a-b) <= eps*abs b
   = relative eps (cons b rest), otherwise
```

现在我们可以定义一个新的求平方根函数:

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

在此我们不须重新编写产生近似值的那部分函数。

4.2 数值微分

我们在前面求平方根的例子中重用了近似值序列。当然，我们也可以将 within 和 relative 重用于任何输出近似值序列的算法。下面我们将其运用于一个数值微分算法中。

一个函数在某一点的微分是函数在此点的图像的斜率。这个斜率可以用此点和附近另外一点间的直线的斜率来估计。我们假设，当这两点非常接近时，函数在这两点间的图像近似于直线。为此，有下面的定义:

```
easydiff f x h = (f(x+h)-f x) / h
```

为了得到精确的近似值，h 的值必须很小。不幸的是，当 h 很小时，f(x+h) 与 f(x) 很接近，其差的舍入误差可能干扰结果。那么如何选择 h 的值呢？一种解决办法是由一个适当的h值开始，在计算近似值序列时，使h的值越来越小。这样的序列应该收敛于导数值，但是，由于舍入的误差，最终将变得不精确。如果用 (within) 来选择足够精确的第一近似值，那么舍入误差影响结果的风险可以大大减小。我们需要一个函数计算下列序列:

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

其中 h 的初始值是 h0，其后的值是前一个值的一半。应用这个函数，任一点的导数均可由下式计算:

```
within eps (differentiate h0 f x)
```

此解并不令人非常满意，因为近似值序列收敛速度很慢。使用一点数学知识会很有帮助。序列的元素可表示为：正确答案 + 与h有关的误差项。

并且在理论上可以证明，误差项约正比于 h 的幂，因此随着 h 变小，误差项会变得越来越小。假设正确答案是 A, 误差项是 B*h**n。因为计算每个近似值时 h 之值被使用两次，同时也在下一个近似值中出现两次，所以任意两个连续近似值可表示如下：

```
a(i) = A + B*(2**n)*(h**n)
a(i+1) = A + B*(h**n)
```

消去误差项后得到

```
      a(i+1)*(2**n) - a(i)
A = ------------------------
           2**n -1
```

当然，因为误差项只是大致可表示为h的幂，所以这个结果也是近似的，但是， 它是一个好得多的近似值。通过下面的函数， 这个改进可应用于任何一对连续的近似值：

```
elimerror n (cons a (cons b rest)) =
   cons ((b*(2**n)-a)/(2**n-1))(elimerror n (cons b rest))
```

消去误差项后得到的近似值序列其收敛速度要快得多。

在使用 elimerror 之前， 我们还有一个问题需要解决——确定 n 的值。

n 值的预测一般是困难的，但它的度量是容易的。不难证明，下列函数能正确地估算 n 的值，其证明略去。

```
Order (cons a (cons b (cons c rest)))
   = round(log2((a-c)/(b-c) -1))
```

其中，round x = x四舍五入到最近的整数，log2 x = 以2为底x的对数。

一个改进近似值序列的通用函数可定义如下：

```
improve s = elimerror (order s) s
```

应用函数improve可以更有效地计算一个函数的导数如下：

```
within eps (improve (differentiate h0 f x))
```

函数 improve 只适用于由参数 h 计算的序列，其中 h 在每次逼近时均被 2 除。但是，improve 作用于这种性质的序列的结果仍然具有这种性质！这意味着一个逼近序列可以被改进多次。每次的改进都消去一个不同的误差项，由此得到的序列收敛得越来越快。因此，使用下式可以很有效地计算导数：

```
within eps (improve (improve (improve (differentiate h0 f x))))
```

用数值分析的术语来讲，这很可能叫四阶方法，并能很快地计算出精确结果。我们甚至可以定义:

```
super s = map second (repeat improve s)
second (cons a (cons b rest)) = b
```

其中 repeat improve 用于得到一个不断改进的逼近序列，然后再由每个改进的序列中的第二逼近值构造一个新的序列（结果表明，第二个逼近是最好的选择，因为它更精确而且不需要额外的计算）。此算法确实很复杂——在计算逼近的过程中，它使用越来越好的数值方法。用如下的程序确实可以很有效地计算导数:

```
within eps (super (differentiate h0 f x))
```

举这个例子也许是大材小用，但是这里要表达的思想是: 即使像 super 这样复杂的算法也可以利用惰性计算很容易地模块化。

4.3 数值积分

本节讨论的最后一个例子是数值积分。

问题的陈述很简单: 给定一个一元实函数 f 和两个端点 a 和 b，估算在曲线 f 下介于 a 和 b 之间的面积。估算此面积的最简单方法是假设 f 近似于直线，在这种情况下此面积为:

```
easyintegrate f a b = (f a + f b)*(b-a)/2
```

遗憾的是这种估计可能是很不精确的，除非 a 和 b 非常接近。一个更好的估算方法是将 a 和 b 之间的区间一分为二，计算每部分的面积并将结果相加。我们可以定义一个不断逼近积分的序列: 使用上面的公式计算第一个近似值，然后将每个二分之一区间上的更精确近似值相加以计算序列的其他近似值。这个序列由以下函数计算:

```
integrate f a b = cons (easyintergrate f a b)
                       (map addpair (zip (integrate f a mid)
                                         (integrate f mid b)))
```

其中 mid = (a+b)/2

zip 是另一个标准的表处理函数。它作用于两个表，返回一个二元组表，每个二元组由两个表的相应元素组成。因此，第一个二元组由第一个表的第一个元素和第二个表的第一个元素组成，等等。zip 可如下定义:

```
zip (cons a s) (cons b t) = cons (pair a b)(zip s t)
```

在求积分过程中，zip 计算对应于两个子区间的积分的二元组序列，map addpair 将二元组的元素相加得到一个原区间近似值的序列。

实际上，这个积分过程效率很低，因为它不停地重复计算 f 的值。如程序所示，easyintegrate 计算 f 在 a 和 b 的值，integrate 的递归调用又重复计算这些值。同样，(f mid) 也在每次递归调用中被重复计算。为此，下述过程更可取，它永远不会重复计算 f 的值:

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb = cons ((fa +fb)*(b-a)/2)
                        (map addpair (zip (integ f a m fa fm)
                           (integ f m b fm fb)))
```

其中

```
m = (a+b)/2
fm = f m
```

integrate 计算一个逐步逼近积分的无穷序列，正如前一节微分的逼近一样。因此，我们可以用以下的程序计算任意精度的积分:

```
within eps (integrate f a b)
relative eps (integrate f a b)
```

这个积分算法有同前一小节中微分算法一样的缺点——其收敛速度非常慢。同上所述，收敛速度可以改进。计算第一个逼近（由 easyintergrate）仅使用两个点，间距为 b-a 。第二个逼近的计算也使用中点，所以两个邻点的间距仅为 (b-a)/2。第三个逼近值的计算在每个二分之一区间使用同一方法，所以相邻点的间距仅为 (b-a)/4。显然，在一个逼近值与下一个逼近值间邻点间的间距缩减了一半。用 h 记此间距，我们可以用前一节定义的 ‘‘improve’’ 函数 改进此逼近序列。所以，我们可以编写快速收敛于积分的逼近序列，例如:

```
super (integrate sin 0 4)
improve (integrate f 0 1)
```

其中 `f x = 1/(1+x*x)`

注: 后一个序列是计算 pi/4 的 8 阶方法。第二个近似值可精确到小数点五位，尽管只需计算5次 f 的值。

在这一节中，我们列举了几个使用函数程序设计的数值算法，应用了惰性计算将计算的各部分粘合在一起。为此，我们能够用新的方式模块化这些算法，分解成如 within，relative 和 improve 这样的通用函数。使用不同的方式结合这些函数，我们很容易地编写了一些很好的数值算法。

5 一个人工智能的例子

我们提出函数程序语言具有很强的表现力，其主要原因是这种语言提供了两种新的粘合方法：高阶函数和惰性计算。在本节我们将借用人工智能中一个较大的例子，应用这两种粘合使得编程很容易地实现。

我们选择的例子是 alpha-beta 启发算法，用于估算博弈者所处局势的优劣。算法检查博弈会如何发展，并且避开那些不利的步骤。

假设博弈由类型为 ``position'' 的对象表示。这个类型将随着不同的博弈类型而变化，我们对此不做任何假设。对于一个博弈局势，我们必须知道下一步可以如何走。假定下面的函数把一个博弈局势映射到博弈下一步可以到达的所有局势构成之列。例如，连三子博弈 tic-tac-toe 的 moves 函数：

```
       | |          X| |     |X|     | |
      -+-+-         -+-+-   -+-+-   -+-+-
moves | |     =   [ | | ,   | | ,   |X| ]
      -+-+-         -+-+-   -+-+-   -+-+-
       | |          | |     | |     | |

       | |          O| |     |O|
      -+-+-         -+-+-   -+-+-
moves |X|     =   [ |X| ,   |X| ]
      -+-+-         -+-+-   -+-+-
       | |          | |     | |
```

我们假定由当前的局势可以确定下一步轮哪一位选手着子。在连三子博弈中，我们可以通过计算每一种棋子的个数来决定。对于像象棋这样的博弈，类型 ``position'' 必须显式地包含这个信息。

给定函数 ``moves''，第一步是构造一个博弈树。树的节点用局势标记，并使得每一个节点的所有孩子的标记是从父节点一步可以到达的局势。也就是说，如果一个节点的标记为 p，那么它的孩子要使用 (moves p) 标记。一个博弈树可以是无穷的，比如博弈可以永远继续下去，没有任何一方取胜。博弈树就如我们在第二节讨论的树—— 每个节点有一个标记（代表局势）和一列子节点。因此，我们可以用相同的数据类型来表示它们。

一个博弈树是通过重复地应用 moves 构造出来的。从根开始，利用 moves 生成其子树根的标记。然后利用 moves 生成子树的子树，以此类推。这种递归模式可以用一个高阶函数表示：

```
reptree f a = node a (map (reptree f) (f a))
```

利用这个函数，我们可以定义另一个高阶函数，它从一个特定的局势开始构造一个博弈树：

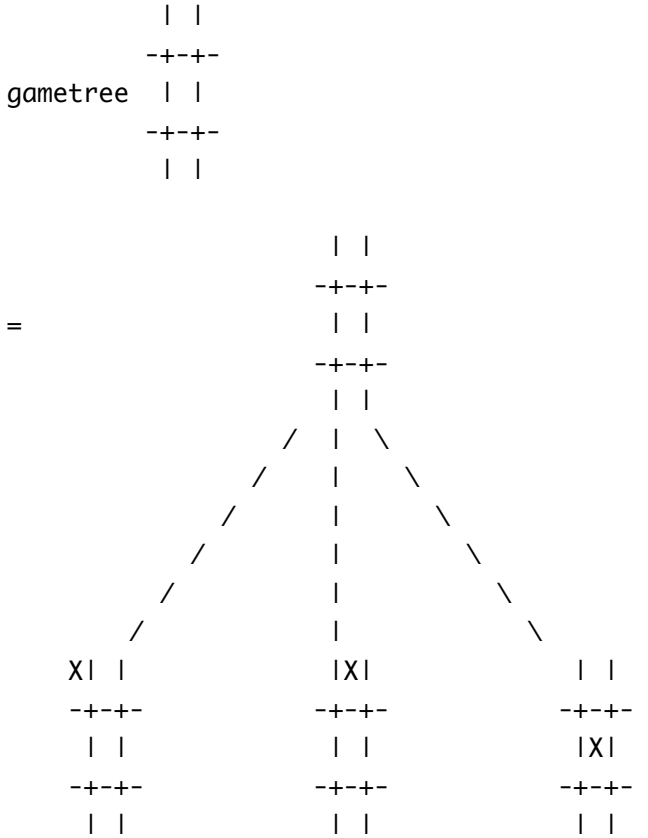
```
gametree p = reptree moves p
```

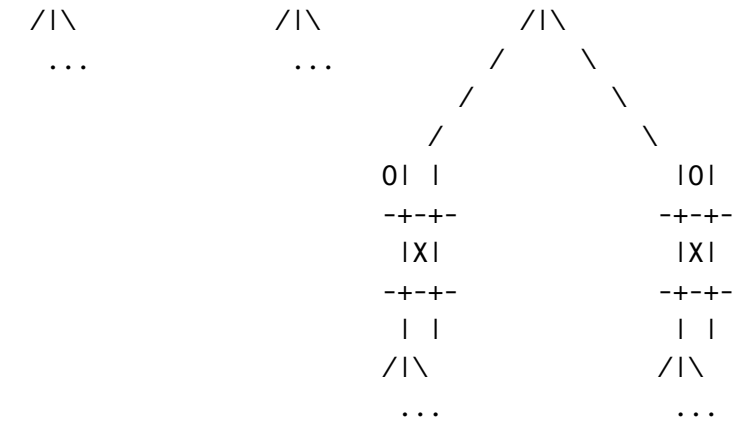
例如下图，这里使用的高阶函数 (reptree) 类似于前一节构造无穷序列的函数 repeat。

我们关注的 alpha-beta 算法由给定的结点向前观测，预测博弈的发展是否有利。为此，算法必须能够在不向前观察时对一个结点做出大概的估计。这种"静态估算"必须限于向前观测有限步，并可用于及早指导算法。静态估算的结果是计算机一方对一结点的前景的度量（假设计算机的对手是人）。此度量值越大，此局势对计算机越有利；此度量值越小，此局势对计算机越不利。最简单的一种函数是对于计算机赢的局势返回1，输的局势返回-1，其他局势返回0。事实上，静态函数度量使一个局势"看好"各种的因素，例如，象棋中棋子的因素和中心的控制。假设我们有这样的一个函数:

```
static: position -> number
```

因为博弈树是一个 (treeof position)， 它可以被函数 (maptree static) 转换为一个 (treeof number)。 函数 (maptree static) 静态地计算树上的所有局势（这里的树可以是无穷的），函数 maptree 在第二节有定义。





给定一个静态计算树，树中各局势的真正值（true value）是什么呢？特别地，根的局势应该赋什么样的值呢？静态值是不合适的，因为它只是大概的估计。一个节点之值应该由其子节点之值来决定。我们假定每个选手都选择最理想的步骤。因为一个节点的值越大，表明此节点表示的局势对计算机越理想，所以，当计算机在此结点着子时，它将在其子节点中选择具有最大值的子节点。类似地，对手也将选择走到其具有最小值的子节点。假定计算机和其对手轮流着子，那么当计算机走子时一个节点的值由函数 maximise 计算，否则由 minimise 计算:

```
maximize (node n sub) = max (map minimize sub)
minimize (node n sub) = min (map maximize sub)
```

其中，max 和 min 是返回数列中最大值和最小值的函数。以上定义是不完全的，因为它们相互递归调用但无初始定义。我们必须定义无后继的节点之值，并将其定义为静态值。因此，静态值用于当一个选手已经取胜或者不能继续走子时的节点。函数 maximise 和 minimise 的完整定义如下:

```
maximize (node n nil) = n
maximize (node n sub) = max (map minimize sub)
minimize (node n nil) = n
minimise (node n sub) = min (map maximise sub)
```

我们几乎可以定义计算一点之真值的函数:

```
evaluate = maximize . maptree static . gametree
```

这个定义有两个问题: 首先，它不适用于无穷树。maximise 使用递归直至找到树叶，如果没有树叶，它便不能返回任何值。第二个问题是，即使是有穷博弈树，此树也可能会很大。计算整个博弈树是不现实的，搜索应该限定在未来的有限几步内。我们可以将树剪接到一定的深度:

```
prune 0 (node a x) = node a nil
prune n (node a x) = node a (map (prune (n-1) x)
```

(prune n) 将一棵树中与根节点距离大于 n 的所有节点裁剪掉。如果博弈树被裁剪，maximise 将在深度为 n 的节点上被迫使用静态值，而不会进一步递归。因此，evaluate 可定义为:

```
evaluate = maximize . maptree static . prune 5 .gametree
```

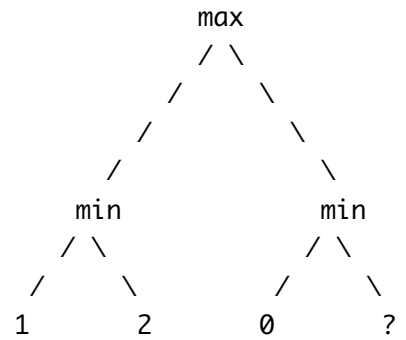
这里我们选择了向前观察5步。

在以上的讨论中我们已经使用了高阶函数和惰性计算。高阶函数 reptree 和 maptree 使我们易于构造和处理博弈树。更重要的是，惰性计算允许我们将 evaluate 如此模块化。因为博弈树是潜在无穷的，没有惰性计算，此程序将永远不会终止运行。如果不使用 prune 5 . gametree，我们将不得不将这两个函数合成一个仅构造博弈树前5层的函数。更糟的是，即使前5层也已经大至内存不可能容纳。我们在程序中使用了函数:

```
maptree static . prune 5 . gametree
```

仅构造博弈树中 maximise 所需要的部分，并且一旦 maximise 完成对此部分的计算，此部分博弈树便可被丢弃（由无用单元回收器 (Garbage collector) 收集），整个博弈树永远不会驻留在内存。在任何时候，只有博弈树的一部分存储在内存。因此，此惰性程序是有效的。因为这种有效性依赖于 maximise（合成链中的最后一个函数）和 gametree（第一个函数）之间的交互作用，如果没有惰性计算，它只能通过将链中所有的函数合成为一个函数来完成。这样一来，模块化大大降低，然而，这便是通常的解决办法。我们可以通过修改每一部分来改进算法 evaluate, 这要相对容易得多。而一般程序员必须将整个程序作为一个整体来修改，其难度要大得多。

到目前为止，我们仅描述了简单极大化。alpah-beta 算法的核心思想是我们通常无需观察整个树便可计算极大值和极小值。考虑下面的树:



奇怪的是，为了计算此树，我们并不需要计算问号的值。左边的极小值是 1，而右边的极小值显然是一个小于或等于 0 的值。因此，这两个极小值的极大值必然是 1。这个发现可以推广到 maximise 和 minimise。

首先将 maximise 分解成 max 作用于一列数:

```
maximise = max . maximise'
```

注: minimise 的分解是类似的。因为 minimise 和 maximise 是完全对称的，我们将讨论 maximise，minimise 可以类似地处理。

这样分解 maximise 后，maximise 可以利用 maximise′（而非 minimise）找出 minimise 是在哪些数中求得最小值。因此，maximise 可以丢弃某些无关的数。有了惰性计算，如果 maximise 无需观察所有的数列，那么某些数列便无需计算，因此极大地节省了计算的时间。

由 maximise 的定义很容易将 max ‘‘分离出来’’:

```
maximise' (node n nil) = cons n nil
maximise' (node n l)   = map minimise l
                       = map (min . minimise') l
                       = map min (map minimise' l)
                       = mapmin (map minimise' l)
where mapmin =  map min
```

因为 minimise′ 返回一个数列（其极小值是 minimise 的结果），所以 (map minimise′ l) 返回一个数列的数列。maximise′ 应该返回各个数列极小值所构成的数列。然而，只有其中的最大值是我们关心的。我们将定义一个新的 mapmin，它将忽略其极小值无关紧要的数列。

```
mapmin (cons nums rest)
   =  cons (min nums) (omit (min nums) rest)
```

函数omit接受一个潜在的极大值——目前的极大值——并且忽略所有小于这个极大值的极小值。

```
omit pot nil =  nil
omit pot (cons nums rest)
   = omit pot rest,                              if minleq nums pot
   = cons (min nums) (omit (min nums) rest), otherwise
```

minleq 接受一个数列和一个潜在极大值，如果数列的极小值小于或等于此潜在极大值，则返回真。要完成这个任务，minleq无需扫描整个数列！如果数列中有任何元素小于或等于潜在极大值，那么数列的极小值也必然小于或等于潜在极大值。为此，minleq 无需扫描数列的剩余部分，其定义如下:

```
minleq nil pot = false
minleq (cons num rest) pot = true,            if  num <= pot
                           = minleq rest pot, otherwise
```

完成 maximise 和 minimise′ 的定义后，很容易编写一个新的计算程序:

```
evaluate = max . maximise' . maptree static . prune 8 . gametree
```

利用惰性计算，maximise′ 只需扫描更小的一部分树，这使得整个程序的运行更有效，正如 prune 只须检查无穷树的一部分使得程序可以终止运行。尽管 maximise′ 的优化很简单，它已经对计算程序的速度产生了非常明显的效果，并且使得计算程序 evaluate 能够预测得更多。

我们可以对计算程序进行其他的优化。例如，alpha-beta 算法在棋手选择最好步骤时达到最优。为此，我们可以将每个节点的子树排序：对于计算机走棋的节点，其子节点的值应该从大到小排列，而对于对手走棋的节点，其子节点的值从小到大排列。下列的函数可以完成这样的任务:

```
highfirst (node n sub) =  node n  (sort higher (map lowfirst sub))
lowfirst (node n sub) =  node n (sort (not.higher) (map highfirst sub))
higher (node n1 sub1) (node n2 sub2) =  n1 > n2
```

其中sort是个一般的排序程序。此时，计算程序变为:

```
evaluate = max . maximise' . highfirst . maptree static . prune 8 . gametree
```

读者可能觉得，为了限制搜索的深度，计算机与其对手只需考虑最好的三步即可。要做这样的修改，只需将 highfirst 用 (taketree 3 . highfirst) 代替即可，其中:

```
taketree n =  redtree (nodett n) cons nil
    nodett n label sub = node label (take n sub)
```

taketree 将一棵树中的所有结点替换为至多有 n 个子节点的节点，其中使用了返回一个表中的前 n 个元素（当列中元素个数小于 n 时，返回的元素个数也小于 n）的函数 (take n)。

另一个改进是细化 pruning。前述程序向前观测一个固定的深度，即使这些棋局是动态的。比如，程序会在女王受到威胁的棋局前停止搜索。通常我们定义一些‘‘动态’’棋局，并且不允许在这些棋局前停止搜索。假设函数 ‘‘dynamic’’ 可以识别这些棋局，我们只需增加一个方程以细化裁剪:

```
prune 0 (node pos sub) = node pos (map (prune 0) sub), if dynamic pos
```

在以上的模块化程序中做这样的更改是很容易的。如上所述，因为程序完全依赖于函数链中最后一个函数 maximise 和第一个函数 gametree 之间的交互作用的效率，如果没有惰性计算，此程序只能写成一个整体的函数。 这样的程序既难于编写，难于修改，也难于理解。

6 结论

我们在本文提出模块化是程序设计的成功的关键。一个能提高生产率的程序设计语言必须能很好地支持模块化程序设计。但是，仅有新的作用域规则和分别编译是不够的——模块化不仅仅意味着模块。将一个问题分解成更小问题的能力直接依赖于我们将这些解粘合在一起的能力。一个语言若要支持模块化程序设计，必须提供新的粘合方法。函数程序提供了两个这样的方法——高阶函数和惰性计算。

使用这些方法我们可以用令人耳目一新的方法将程序模块化。我们已经列举了许多例子。更通用的小模块可以被更广泛地重用，因此简化了程序设计。这便解释了为什么函数程序比常规程序更简短，更容易编写。同时也为函数程序员提供了一个目标。如果程序的某个部分变得混乱或者复杂，程序员应该设法将其模块化、通用化。高阶函数和惰性计算便是合适的工具。

当然，并不是我们第一个指出高阶函数和惰性计算的表达力和简洁性。例如，Turner 指出如何将两者应用于一个化学结构的生成程序中[7, Tur81]。Alelson 和 Sussman 强调指出流（惰性列表）是组织程序的有力工具[1, AS86]。Henderson 使用流组织函数操作系统[?, P.H82]。本文的主要贡献在于断言函数程序语言的表达能力的关键是它具有更好的模块化。

本文也涉及有关惰性计算的争议。有些人认为函数程序语言应该是惰性的，而有些人则认为不应该。也有些人则取折衷的态度，认为函数程序语言只提供惰性列表，并且用特别的语法来构造之（如在 SCHEME[1, AS86] 中）。本文提供了进一步的证据：惰性计算对于函数语言是非常重要的，不应该成为二等公民。它或许是函数程序员最有力的工具。我们不应该失去这样的关键工具。

7 鸣谢

本文很大程度上得益于作者与牛津程序设计研究组 Phil Wadler 和 Richard Bird 的讨论。哥德堡 Chalmers 大学的 Magnus Bondesson 指出较早的草稿中一个数值算法的严重错误，也因此使得其他算法的编写得以顺利进行。本文是在 UK Science and Engineering Researvh Council 的资助下完成的。

参考资料

[1] H. Abelson and G.J. Sussman. Structure and Interpretation of Computer Programs, MIT Press, Boston, 1986

[2] J. Hughes. Why Functional Programming Matters, Computer Journal, 32(2), 1989

[3] J. Hughes, Why Functional Programming Matters, In D. Turner, editor, Research Topics in Functional Programming, Addison Wesley, 1990

[4] R. Milner and M. Tofte and R. Harper, The Definition of Standard ML, MIT Press, 1990

[5] United States Department of Defense, The Programming Language Ada Reference Manual, Springer-Verlag, 1980

[6] P. Henderson, Purely Functional Operating Systems, 1982

[7] D. A. Turner, The Semantic Elegance of Applicative Languages, Proceedings 1981 Conference on Functional Languages and Computer Architecture, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981

[8] D. A. Turner, Miranda: A non-strict language with polymorphic types, Proceedings 1985 Conference on Fucntional Programming Languages and Computer Architecture, pp.1-16, Nancy, France, 1985

[9] N. Wirth, Programming in Modula-II, Springer-Verlag, 1982