

I。构造过程抽象(2)

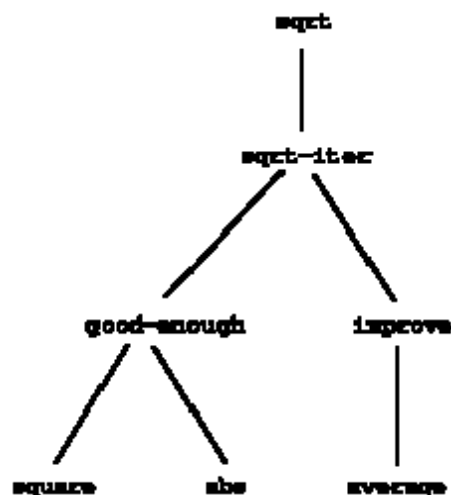
要点:

- 过程内部定义和块结构
- 分析过程（静态，描述）产生的计算进程（动态，行为）
- 计算进程的类型
 - 线性递归
 - 线性迭代
 - 树形递归
- 计算的代价
- 高阶过程：以过程为参数和/或返回值

过程作为黑箱抽象

重新考察 **sqrt** 过程的定义，看看能学到什么

- 首先，它是递归定义的，是基于自身定义的
需要考虑这种“自循环定义”是否真有意义，后面将详细讨论
- **sqrt** 分成一些部分实现
 - 每项工作用一个独立过程完成
 - 构成原问题的一种分解
- 分解合理与否的问题值得考虑
- 定义新过程时，用到的过程应看作黑箱，只关注其功能，不关心其实现
 - 例如，需要用求平方时，任何能计算平方的过程都可以用
 - 只要求它们的使用形式相同



过程作为黑箱抽象

- 例如，只考虑功能（做什么），下面两个定义没有差别：

(define (square x) (* x x))

(define (square x) (exp (double (log x)))) ; 用的过程定义如下

(define (double x) (+ x x))

这是一件好事，程序里的部件应该有可替代性

- 过程抽象的本质：

- 定义过程时，关注所需计算的**过程式描述**细节（怎样做），使用时只关注其**说明式描述**（做什么）
- 一个过程总（应该）隐藏起一些实现细节，使用者不需要知道如何写就可以用。所用过程可能是其他人写的，或是库提供的
- 过程抽象是控制和分解程序复杂性的重要手段，也是记录和重用已有开发成果的单位

其他抽象机制也都有类似作用

过程抽象：局部名字

- 过程中隐藏的最简单细节是局部名。下面两个定义没区别：

(define (square x) (* x x))

(define (square y) (* y y))

- 过程体里的形参：

- 具体名字不重要，重要的是哪些地方用了同一个形参
- 是过程体的约束变量（概念来自数理逻辑），作用域是过程体，**约束变量统一换名不改变结构的意义**。其他名字是自由的

- 看过程 **good-enough?** 的定义：

(define (good-enough? guess x)
 (< (abs (- (square guess) x)) 0.001))

这里的 **x** 必然与 **square** 里的 **x** 不同

否则程序执行时不可能得到所需的效果

- 在 `good-enough?` 的定义里：

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

`guess` 和 `x` 是约束变量，`<`、`-`、`abs` 和 `square` 是自由（变量）

这个过程的意义正确，依赖于两个约束变量（形参）与四个自由变量的名字不同，四个自由变量（在环境里关联）的意义正确

- 形参与所需自由变量重名导致该变量被“捕获”（原定义被屏蔽）：

```
(define (good-enough? guess abs)
  (< (abs (- (square guess) abs)) 0.001))
```

- 自由变量（名字）的意义由运行时的环境确定，它可以是
 - 内部过程或复合过程，需要应用它
 - 有约束值的变量，计算中需要它的值

C 语言程序中名字的意义

- C 函数里的名字可能是
 - 局部参数名或局部定义的变量名等。未在局部定义的名字应该是全局定义的（变量、函数、类型等）。这里不讨论宏定义
 - 同样有局部名字遮蔽外围名字的问题
- C 语言里的名字有不同的地位和划分，除关键字外的类别有
 - 每个函数里的标号名
 - `struct/union/enum` 标记名各为一类
 - 每个 `struct` 或 `union` 下的成员名各为一类
 - 一般标识符，包括变量名、函数名、`typedef` 名字、枚举名
- C 程序的名字解析是编译器的工作
 - C 中的名字（标识符）是静态的概念，运行时没有名字问题
 - Scheme 的变量名在运行中始终存在，以支持程序的动态行为

C 程序里的变量定义

- 现在考虑 C 程序里的变量定义
 - 为什么把一些变量定义为外部的全局变量？
 - 为什么把一些变量定义为局部变量？
- 例如：需要定义一个 1000000 个元素的 **double** 数组
 - 定义在 **main** 里面和外面有什么不同？
- C 变量定义的几个原则
 - 尽可能减少全局变量
 - 变量定义尽可能靠近使用的位置
 - 大型、唯一、公用的变量应该定义为外部全局的
 - 被部分函数共享的外部变量，应考虑能否定义为 **static**

过程抽象：内部定义和块结构

- **sqrt** 的相关定义包括几个过程：

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
(define (improve guess x)
  (average guess (/ x guess)))
```

其中 **abs** 和 **average** 是通用的，可能在其他地方定义。
- 注意：使用者只关心 **sqrt**。其他辅助过程出现在全局环境，只会干扰人的思维和工作（例如，不能再定义同名过程）
- 写大型程序时需要控制名字的使用，控制其作用范围（作用域）

过程抽象：内部定义和块结构

- 信息的尽可能局部化是良好程序设计的重要特征
- 局部的东西应定义在内部。**Scheme** 支持过程内的局部定义，允许把过程定义放在过程里面。按这种考虑组织好的程序：

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

- 这种嵌套定义形式称为块结构（**block structure**），由早期的重要语言 **ALGOL 60** 引进，是一些语言里组织程序的重要手段

过程抽象：内部定义和块结构

- 函数定义局部化使程序更清晰，减少了非必要的名字污染，还可能简化过程定义：由于局部过程定义在形参（**x**）的作用域里，因此可以直接使用（不必再作为参数传递）
- 按这种观点修改后的 **sqrt** 定义：

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

- 块结构对控制程序的复杂性很有价值。后来的各种新语言都为程序组织提供了一些专门机制（未必是块结构）

C 程序结构

- C 程序的结构比较简单：
 - 不支持局部函数定义（基于其他考虑）
 - 这种规定限制了 C 语言的程序组织方式
- 程序设计语言发展中对此有两条路线：
 - 从 Fortran 到 C 以及后来的 C++，Java 等，都不允许过程嵌套
 - 从 Algol 60 到 Pascal，Modula，Ada 等，都允许过程嵌套
- 允许过程的嵌套定义，益处是有更多的方式组织程序
 - 根据需要建立嵌套的过程结构
 - 容易做到相关信息的局部化
- C 语言没有采纳嵌套的程序结构，主要考虑：
 - 实现简单，目标程序的执行效率高
 - 可能以其他方式得到信息局部化（数据抽象，面向对象技术）

C 程序结构

- C 语言的组织机制比较弱
 - 语言中最高层次的机制就是函数，没有函数之上的组织机制，又不允许函数嵌套。函数都处于一个平坦的层次
 - 以后的编程语言在这方面有很多进步
- 在 C 语言里还可以利用程序的物理结构
 - 通过 static 函数和 static 全局变量，可实现一定的信息局部化
- 对 sqrt 实例建立一个独立文件，内容是

```
static double sqrt_iter (double guess, double x){...}
static double improve (double guess, double x){...}
static int good_enough (double guess, double x){...}
static double average (double x, double y){...}
double sqrt (double x) {...}
```
- 可实现多一层信息组织，但不支持多层嵌套的作用域。OO 的类和嵌套类也可用于帮助组织（请分析它与多重函数嵌套的异同）

过程与其产生的计算

- 要真正理解程序设计，只学会使用语言功能把程序写出来还不够
 - 完成同一工作有多种不同方式，应如何选择？为什么？
 - 要成为程序设计专家，必须能理解所写程序蕴涵的计算，理解一个过程（**procedure**）产生什么样的计算进程（**process**）
- 一个过程（描述）可看作一个计算的模式
 - 描述一个计算的演化进程，说明其演化方式
 - 对一组适当的参数，确定了一个具体的计算进程（一个计算实例，是一系列具体的演化步骤）
 - 完成同一件工作，完全可能写出多个大不相同的过程
 - 完成同一工作的两个不同过程导致的计算进程也可能大不相同
- 下面通过例子讨论一些简单过程产生的计算进程的“形状”，观察其中各种资源的消耗情况（主要是时间和空间）
 - 从这里得到的认识可供写其他程序时参考

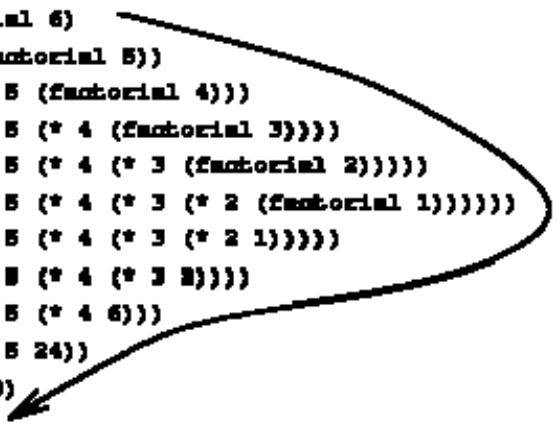
线性递归和迭代

- 考虑阶乘计算。一种看法（递归的观点）：
$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1 = n \cdot [(n-1) \cdot \dots \cdot 2 \cdot 1] = n \cdot (n-1)!$$
- 相应的过程定义：

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

- 采用代换模型推导
由 (factorial 6) 得到

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```



线性递归和迭代

- 另一观点： $n!$ 是从 1 开始逐个乘各自然数，乘到 n 就得到了阶乘值

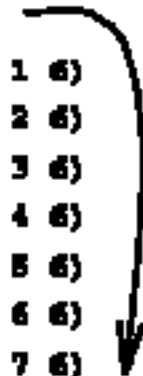
```
product ← counter · product  
counter ← counter + 1
```

- 按这种观点写出程序：

```
(define (factorial n)  
  (fact-iter 1 1 n))  
  
(define (fact-iter product  
                  counter max-count)  
  (if (> counter max-count)  
      product  
      (fact-iter (* counter product)  
                  (+ counter 1)  
                  max-count)))
```

对应计算进程

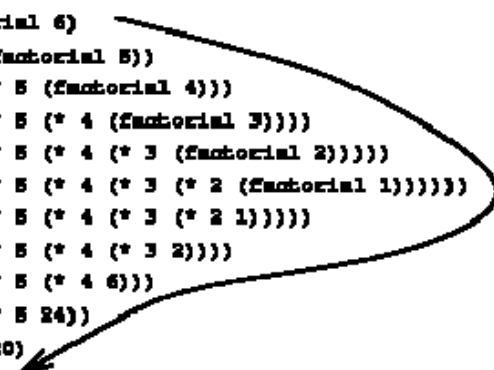
```
(factorial 6)  
(fact-iter 1 1 6)  
(fact-iter 1 2 6)  
(fact-iter 2 3 6)  
(fact-iter 6 4 6)  
(fact-iter 24 5 6)  
(fact-iter 120 6 6)  
(fact-iter 720 7 6)  
720
```



线性递归和迭代

- 对比两个计算进程：

```
(factorial 6)  
(* 6 (factorial 5))  
(* 6 (* 5 (factorial 4)))  
(* 6 (* 5 (* 4 (factorial 3))))  
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))  
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))  
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))  
(* 6 (* 5 (* 4 (* 3 2))))  
(* 6 (* 5 (* 4 6)))  
(* 6 (* 5 24))  
(* 6 120)  
720
```

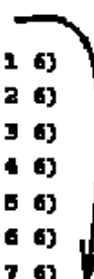


先展开后收缩：展开过程中积累一系列计算，收缩就是完成这些计算

解释器需要维护待执行计算的轨迹，轨迹长度大致等于后续计算的次数

积累长度为线性的，计算序列的长度也为线性，称为线性递归进程

```
(factorial 6)  
(fact-iter 1 1 6)  
(fact-iter 1 2 6)  
(fact-iter 2 3 6)  
(fact-iter 6 4 6)  
(fact-iter 24 5 6)  
(fact-iter 120 6 6)  
(fact-iter 720 7 6)  
720
```



无展开/收缩，直接进行计算

计算轨迹的信息量为常量，只需维护几个变量的当前值

计算序列的长度为线性的

具有这种性态的计算进程称为线性迭代进程

线性递归和迭代：分析

- 迭代进程中，计算的所有信息都在几个变量里
 - 可以在计算中的任何一步中断和重启计算
 - 只要有这组变量的当前值，就可以恢复并继续计算
- 在线性递归进程中，相关变量里的信息不足以反映计算进程的情况
 - 解释器需要保存一些“隐含”信息（在系统内部）
 - 这种信息的量将随着计算进程的长度线性增长
- 看阶乘的例子

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

只根据当前调用 (**factorial 5**)，无法得知外面有多少遗留下未进行的计算（不知道是从哪里开始递归到求 **5** 的阶乘）

线性递归和迭代：分析

- 注意区分“递归计算进程”和“用递归方式定义的过程”
 - 用递归方式定义过程说的是程序的写法，定义一个过程的代码里调用了这个过程本身
 - 递归计算进程，说的是计算中的情况和执行行为，反映计算中需要维持的信息的情况
- 常规语言用专门循环结构（**for**, **while**等）描述迭代计算
 - **Scheme** 采用尾递归技术，可以用递归方式描述迭代计算
- 尾递归形式和尾递归优化
 - 一个递归定义的过程称为是尾递归的，如果其中的对本过程的递归调用都是执行中的最后一个表达式
 - 虽然是递归定义的过程，使用的存储却不随递归深度增加。尾递归技术，就是重复使用原过程在执行栈里的存储，不另行分配
- 常规语言都没有实现尾递归优化，有兴趣可以自己考虑可能吗/为什么

树形递归

- 另一常见计算模式是树形递归，典型例子是Fibonacci数的计算

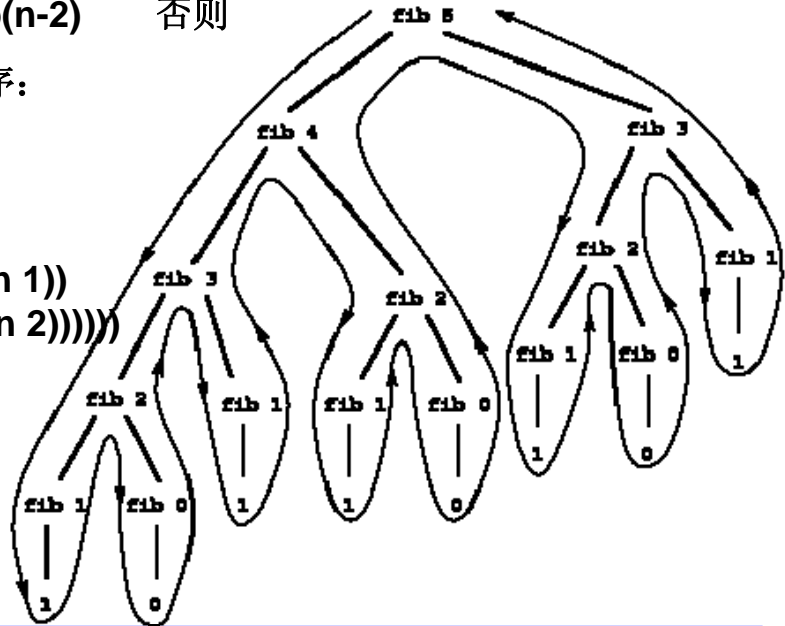
$$\begin{aligned}\text{Fib}(n) &= 0 && \text{若 } n = 0 \\ &= 1 && \text{若 } n = 1 \\ &= \text{Fib}(n-1) + \text{Fib}(n-2) && \text{否则}\end{aligned}$$

- 根据定义直接写出的程序:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

- (fib 5) 产生的计算

进程图示:



树形递归

- 已知Fibonacci数 $\text{Fib}(n)$ 的增长与 n 成指数关系（练习1.13），可知 $\text{fib}(n)$ 的计算量增长与 n 的增长成指数关系。很糟糕
- 考虑Fibonacci数的另一算法：取变量 a 和 b ，将它们初始化为 $\text{Fib}(0)$ 和 $\text{Fib}(1)$ 的值，而后反复同时执行更新操作：

$$\begin{aligned}a &\leftarrow a + b \\ b &\leftarrow a\end{aligned}$$

- 写出过程定义:

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

这一函数形成的计算进程是线性迭代

树形递归：换零钱的不同方式

- 有些问题很容易用递归描述，但不易写出线性迭代程序
- 一个问题：人民币硬币有1元，5角，1角，5分，2分和1分。给了一定的人民币，问有多少种不同方式将它换成硬币？
- 用递归过程描述，先要有对问题的一种递归式的看法。
- 一种看法：确定一种硬币排列，总币值 a 换为硬币的不同方式等于：
 - 将 a 换为不用第一种硬币的方式，加上
 - 用一个第一种硬币（设币值为 b ）后将 $a-b$ 换成各种硬币的方式
- 几种基础情况：
 - $a = 0$ ，计 1 种方式
 - $a < 0$ ，计 0 种方式，因为不合法
 - 货币种类 $n = 0$ ，计 0 种方式，因为已无货币可用
- 很容易把这一套考虑翻译为递归定义的过程

树形递归：换零钱的不同方式

- 过程定义（只计算不同换法的数目，不考虑换的方式）

```
(define (count-change amount) (cc amount 6))
```



```
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount (- kinds-of-coins 1))
                  (cc (- amount
                        (first-denomination kinds-of-coins))
                      kinds-of-coins))))))
```



```
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 2)
        ((= kinds-of-coins 3) 5)
        ((= kinds-of-coins 4) 10)
        ((= kinds-of-coins 5) 50)
        ((= kinds-of-coins 6) 100)))
```

请思考，翻转硬币的排列顺序会怎么样？

程序还正确吗？

效率会改变吗？

- 允许写实现递归过程的过程，确实有价值：
 - 是某些问题的自然表示，如一些复杂数据结构操作（如树遍历）
 - 编写更简单，容易确认它与原问题的关系
 - 做出对应复杂递归进程的迭代过程的过程，常需要付出很多智力
- 换零钱不同方式，用递归过程描述很自然，它蕴涵一个树形递归进程
 - 写出解决这个问题的迭代不太容易，大家自己做一做
- 递归描述常常比较清晰简单，但却可能是实现了一种代价很高的计算。而高效的迭代过程可能很难写。人们一直在研究：
 - 能不能自动地从清晰易写的程序生成出高效的程序？
 - 如果不能一般性地解决这个问题，是否存在一些有价值的问题类，或一些特定的描述方式，对它们有解决的办法？
- 这一问题在计算机科学技术中处处可见，永远值得研究。例如，今天蓬勃发展的有关并程序序设计的研究

C 语言里的递归和迭代

- **C** 和其他常规语言一样，通过一套迭代语句（循环语句）支持描述线性迭代式的计算
- 常规语言里允许以递归方式定义程序始于 **Algol 60**
 - 后来的高级语言都允许递归定义的程序
 - **Fortran** 从 **Fortran 90** 开始也支持递归方式的程序
 - 支持递归的语言实现必须采用运行栈技术，在运行栈上为过程调用的局部信息和辅助信息分配空间，带来不小开销
 - **RISC** 计算机的一个重要设计目标就是提高运行栈的实现效率
- 虽然 **C** 语言（和其他常规语言）都支持递归
 - 但都不支持尾递归优化
 - 即使写尾递归形式的程序，语言的运行系统仍会为每次递归调用分配新空间，程序空间开销与运行中的递归深度成线性关系

增长的阶

- 算法和数据结构课程讨论了计算代价的问题，其中最主要想法
 - 在抽象意义上考虑计算的代价（增长的阶）
 - 考虑计算中的各种资源消耗如何随着问题规模的增长而增长
- 这方面问题不再讨论
 - 书中用 $\Theta(f(n))$ 表示增长的阶是 $f(n)$
 - 我们下面用 $O(f(n))$ 表示上界（不要求精确下界）
 - 总希望考虑尽可能紧的上界（更好反映算法的性质）
- 应记得：
$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \dots < O(2^n) <$$

常量	对数	线性		平方	立方	指数
----	----	----	--	----	----	----
- 下面看两个例子

实例：求幂

- 求 b^n 最直接方式是利用下面递归定义：
$$b^n = b \cdot b^{(n-1)} \quad b^0 = 1$$
- 直接写出的程序需要线性时间和线性空间（线性递归计算）：

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```
- 不难改为实现线性迭代的过程（仿照前面阶乘程序）

```
(define (expt b n) (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                  (- counter 1)
                  (* b product))))
```

需要线性时间和常量空间（ $O(1)$ 空间）

实例：求幂

- 用反复乘的方式，求 b^8 要 7 次乘法，实际上可以只做 3 次

$$b^2 = b \cdot b$$

$$b^4 = b^2 \cdot b^2$$

$$b^8 = b^4 \cdot b^4$$

- 对一般整数 n ，有

$$n \text{ 为偶数时} \quad b^n = (b^{(n/2)})^2$$

$$n \text{ 为奇数时} \quad b^n = b \cdot b^{(n-1)} \quad \text{请注意，} n-1 \text{ 是偶数}$$

- 根据这一认识写的过程

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

```
(define (even? n) (= (remainder n 2) 0))
```

用 ? 作为谓词函数名最后字符是 Scheme 的编程习惯

这一过程求幂所需乘法次数是 $O(\log n)$ ，是重大改进

实例：素数检测

- 判断整数 n 是否素数。下面给出两种方法，一个复杂性是 $O(\sqrt{n})$ ，另一个概率算法的复杂性是 $O(\log n)$

- 找因子的直接方法是用顺序的整数去除。下面过程找出最小因子：

```
(define (smallest-divisor n)
  (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (divides? a b)
  (= (remainder b a) 0))
```

- 素数就是“大于 2 的最小因子就是其本身”的整数：

```
(define (prime? n)
  (= n (smallest-divisor n)))
```

n 非素数时一定有不大于其平方根的因子。需检查 $O(\sqrt{n})$ 个整数

实例：素数的费马检查

- 概率算法的基础是费马小定理：若 n 是素数， a 是任一小于 n 的正整数，那么 a 的 n 次方与 a 模 n 同余。（两个数模 n 同余：它们除以 n 余数相同。数 a 除以 n 的余数称为 a 取模 n 的余数，简称 a 取模 n ）
- n 非素数时多数 $a < n$ 都不满足上述关系。这样就得到一个“算法”
 - 随机取一个 $a < n$ ，求 a^n 取模 n 的余数
 - 如果结果不是 a 则 n 不是素数；否则重复这一过程
- n 通过检查的次数越多，是素数的可能性就越大
- 实现这一算法，需要一个计算自然数的幂取模的过程：

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                     m))))
```

实例：素数的费马检查

- 过程 **expmod** 利用了一个数学关系，保证计算的中间结果不太大
$$(a * b) \bmod c = ((a \bmod c) * (b \bmod c)) \bmod c$$
- 执行费马检查需要随机选取 1 到 $n-1$ 之间的数，过程：

```
(define (fermat-test n)
  (define (try-it a) (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
```

random 取得随机数，(**random** (- n 1)) 得到 0 到 $n-2$ 间的随机数

- “判断”是否为素数需要反复做费马检查。可以把次数作为参数：

```
(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

在被检查的数通过了 **times** 次检查后返回真，否则返回假

- 上述算法只有概率意义上的正确性：

随着检查次数增加，通过检查的数是素数的概率越来越大

- 一点说明：费马小定理只说明素数能通过费马检查，并没说通过检查的都是素数。确实存在不是素数的数能通过费马检查
- 人们已找到了别的检查方法，能保证通过检查的都是素数
- 这一算法的结论只在概率上有意义
 - 结果只有概率意义的算法称为概率算法，概率算法已发展成了一个重要研究领域，有许多重要应用
 - 实际中，很多时候只需要有概率性的保证

C 语言里的过程和计算

- **C 语言里用“函数”实现过程**
 - 线性递归和树形递归用递归的方式描述
 - 线性迭代计算，需要用语言里的迭代控制结构（循环结构）实现
 - 请自己用 **C** 语言或其他熟悉的语言改写书上的程序，设法弄清所用实现能不能自动完成尾递归优化
 - 在实现素数判断的概率算法时，有一个问题值得注意
 - 前面说利用 **mod** 的性质，可避免计算中出现很大的中间结果
 - 对 **Scheme** 程序而言，这是一种优化。因为 **Scheme** 和许多函数式语言（和各种数学软件）都支持任意范围的整数，支持任意大的整数计算（受限于硬件内存或具体系统的实现）
 - 对常规语言（如 **C**），计算中不断取 **mod** 是必须的。因为 **C** 的整数的表示范围优先，非此很容易出现溢出
- 还需注意，要取的模很大时，乘法仍可能溢出

高阶过程

- 过程是抽象，一个过程描述对数据的某种复合操作。如求立方过程：

`(define (cube x) (* x x x))`

- 完全可以不写过程，总直接用系统操作写组合式：

```
(* 3 3 3)
(* x x x)
(* y y y)
```

- 这就是只在系统操作的层面上工作，不能提高描述层次
虽然能计算立方，但程序里没有立方概念
为公共计算模式命名就是建立概念。过程抽象可以起这种作用
- 只能以数作为参数也限制了建立抽象的能力
有些计算模式可以适用于多种不同的过程
为这类计算模式建立抽象，就需要以过程作为参数或返回值

高阶过程

- 以过程作为参数或返回值的，操作过程的过程称为[高阶过程](#)
 - 下面讨论如何用高阶过程作为抽象的工具
 - 帮助理解高阶过程可以如何提高语言的表达能力
- 常规语言里也有一定的定义高阶过程的能力
 - 但相关功能通常很有限，限制了相应的发挥
 - **Java**、**C#** 等引进 **lambda** 表达式，就是为了在这方面有所前进
- 下面的讨论就是要帮我们看清高阶过程抽象的价值
 - 有助于理解为什么 **Java** 等语言要不遗余力地引进相关功能
 - 帮助了解 **lambda** 表达式的构造和使用

以过程作为参数

- 考虑下面几个过程：

<pre>(define (sum-integers a b) (if (> a b) 0 (+ a (sum-integers (+ a 1) b))))</pre>	$a + \dots + b$
<pre>(define (sum-cubes a b) (if (> a b) 0 (+ (cube a) (sum-cubes (+ a 1) b))))</pre>	$a^3 + \dots + b^3$
<pre>(define (pi-sum a b) (if (> a b) 0 (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))</pre>	$\frac{1}{1 \cdot 3} + \frac{1}{5 \cdot 7} + \frac{1}{9 \cdot 11} + \dots$

虽然各过程的细节不同，但它们都是从参数 **a** 到参数 **b**，按一定步长，对依赖于参数 **a** 的一些项求和

以过程作为参数

- 这几个过程的公共模式是：

```
(define (<pname> a b)
  (if (> a b)
      0
      (+ (<term> a)
        (<pname> (<next> a) b))))
```

许多过程有一个公共模式，说明这里存在一个有用的抽象。如果所用语言足够强大，就可以利用和实现这种抽象

- **Scheme** 允许将过程作为参数，下面的过程实现上述抽象

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
        (sum term (next a) next b))))
```

其中的 **term** 和 **next** 是计算一个项和下一个 **a** 值的过程

以过程作为参数

- 有了 **sum**，前面函数都能按统一方式定义（提供适当的 **term/next**）

```
(define (inc n) (+ n 1))  
(define (sum-cubes a b) (sum cube a inc b))  
  
(define (identity x) x)  
(define (sum-integers a b) (sum identity a inc b))  
  
(define (pi-sum a b)  
  (define (pi-term x) (/ 1.0 (* x (+ x 2))))  
  (define (pi-next x) (+ x 4))  
  (sum pi-term a pi-next b))
```

- 使用的例子：

```
(sum-cubes 1 10)  
3025  
  
(* 8 (pi-sum 1 1000))  
3.139592655589783
```

收敛非常慢，到 **pi/8**

- **sum** 实现的是线性递归计算进程
- 练习1.30 要求写完成同样功能的高阶过程，实现线性迭代

以过程作为参数：数值积分

- 一个抽象真的有用，就表现在它可用于形式化其他的概念。如 **sum** 可用于实现数值积分，公式是

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

- 其中 **dx** 是很小的步长值。实现：

```
(define (integral f a b dx)  
  (define (add-dx x) (+ x dx))  
  (* (sum f (+ a (/ dx 2.0)) add-dx b)  
     dx))
```

```
(integral cube 0 1 0.01)  
.24998750000000042
```

```
(integral cube 0 1 0.001)  
.2499998750000001
```

x³ 在 **[0, 1]** 积分的精确值是 **1/4**