

I。构造过程抽象(2)

- 表达式，值，**define**
- 过程的内部定义和块结构
- 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

表达式的一些情况

- 变量
 - 如果一个变量没定义，对它求值是错误，求值中断
 - 如果变量有定义，求值得到它当时的关联值
- 内部过程
 - 对内部过程名求值得到某种特殊信息。如（不同系统可能不同）
`> +` 得到 `#<procedure:+>`
- 组合过程
 - 对自己定义的过程名求值也得到特殊信息。如
`> (define (square x) (* x x))`
`> square` 得到 `#<procedure:square>`
- 特殊形式的名字不能求值
 - 例如，对 **define** 求值将出错

基本类型和表达式（自求值表达式）

- **Scheme** 有一组基本类型

- 数值类型

- 整数，如 **1**, **17**, **2035**

Scheme (Racket) 支持任意大的整数及其精确运算，如
(* 123456789 987654321 999999999999999999)

- 实数，如 **3.57**,
- 其他数值类型

- 数值类型的运算：

- 基本算术运算：**+** **-** ***** ...
- 比较和判断（得到逻辑值）：**>** **<** ... **zero?** **even?** ...
- 常用函数：**abs** **quotient** **remainder** **module** **sin** ...
- 随机数生成，如 **(random 10)**, **(random 1.0)**

基本类型和表达式

- 布尔类型，逻辑值 **#t** 和 **#f**，逻辑运算

逻辑真表达式：**#t**

逻辑假表达式：**#f**

- 字符类型

写起来比较麻烦，编写简单程序时使用较少

#\a **#\B** **#\{** **#\space** **#\newline**

- 字符类型的操作

- 字符比较：**char=?** **char>?** ...
- 其他操作：**char-upcase** ...
- 数值字符转换：**char-digit** **digit-char** ...

基本类型和表达式（自求值表达式）

■ 字符串类型

"Peking University", "Mathematics", "information science"

■ 字符串运算（操作）

常用操作: **string-length** **string-ref** (取串中字符)

字符串比较: **string=?** **string<?**

模式匹配: (**string-search-forward** *pattern string*)

其他: **string-append** (拼接) (**substring** *m n*)

■ 还有一些其他类型

包括下一章要讨论的组合数据类型

有关情况可以查阅语言手册和用户手册

特殊形式 **define** 的两种形式和意义

■ **define** 给名字（变量名/组合过程名）建立约束值

建立什么值的情况依赖于 **define** 表达式的形式

■ 给变量建立约束值

形式 (**define** *变量名 值表达式*)

例 (**define** *x* (+ 3 (* 5 7)))

意义 求出*值表达式*的值，给*变量*关联这个值

■ 定义过程

形式 (**define** (*过程名 形式参数...*) *过程体表达式*)

其中 *过程名*是一个名字，*形式参数*可以有 0 个或多个

*过程体表达式*可以是多个表达式，其中可以引用*形式参数*

意义 为*过程名*约束一个组合过程，其*形式参数*和*过程体*由这个 **define** 的成分给出，这里的*过程体表达式*不求值

define 定义过程的一些情况

- 定义简单过程

```
(define (f x) (+ (* (- x 1) x) (* (+ x 1) (+ x 2))))
```

使用 (f 5)

- 定义多个参数的过程

```
(define (g m n) (+ (* m m) (* n n)))
```

使用 (g 10 17)

- 定义无参过程

```
(define (h) (...))
```

使用 (h)

注意与 (define h ...) 和使用 h 之间的不同

- 形参的**作用域**是组合过程的**体**，在这个体里出现了形参名，就表明是引用相应的形参

将过程看作黑箱抽象

现在重新考察 sqrt 过程的定义，看能从中学到些什么

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                  x)))
```

```
(define (improve guess x) (average guess (/ x guess)))
```

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

```
(define (sqrt x) (sqrt-iter 1.0 x))
```

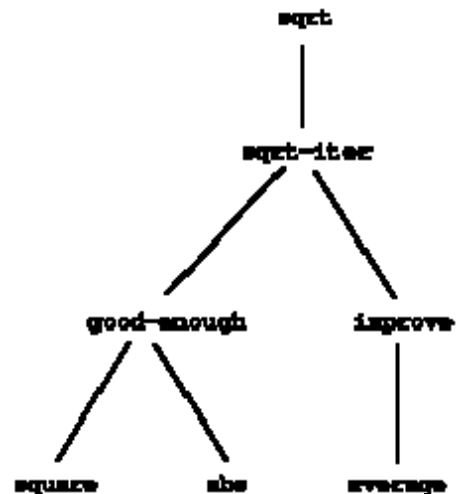
- 首先，sqrt-iter 是递归定义的，基于其自身定义

- 需要考虑这种“自循环定义”是否确实有意义
- 后面详细讨论

过程作为黑箱抽象

- **sqrt** 的实现由一些部分构成（图）
 - 每项工作用一个独立过程完成
 - 形成了原问题的一种分解
 - 分解的合理性？
 - 定义一个过程时可能用到其他过程
 - 使用的是那些过程的功能
 - 例如，需要平方，任何能计算平方的过程都可以用

使用形式应符合要求
 - 应该把被用的过程看作黑箱，只关注其功能，不关心其实现
- 减少写程序时的思维负担



过程作为黑箱抽象

- 例如，只考虑功能（做什么），下面两个 **square** 定义没差别：
(define (square x) (* x x))
(define (square x) (exp (double (log x))))；其中用到下面过程
(define (double x) (+ x x))
这种情况很好：抽象使程序中的部件具有可替代性
- 过程抽象的本质是一种功能分解：
 - 定义过程时，关注计算的**过程式描述**（怎样做），使用时只关注其**说明式描述**（功能，做什么）
 - 一个过程总（应该）隐藏一些实现细节，使用者不需要知道它如何工作。所用过程可以是其他人开发，或由程序库提供
 - 过程抽象是控制和分解程序复杂性的一种重要手段，也是记录和重用已有开发成果的单位

其他抽象机制也有类似作用，后面还会讨论

过程抽象：局部名字

- 过程隐藏的最简单细节是局部的名字。下面两个定义没区别：

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

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

- 过程定义里使用的形参只是占位符：
 - 从程序的语义看，具体采用的名字并不重要（对程序的可读性有意义），重要的是哪些地方用了同一个形参
 - 形参是过程体的约束变量（概念来自数理逻辑），作用域是整个过程体，对约束变量统一换名不改变过程意义。其他名字是自由的
- 过程 `good-enough?` 的定义

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

这里的 `x` 一定与 `square` 的 `x` 不同，否则执行时不可能有所需效果

过程抽象：局部名字

- 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）

过程的意义正确，有赖于两个约束变量（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）的意义合适

- 形参与自由变量重名将导致该变量被“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）的意义由运行环境确定，它们可能是
 - 内部过程或复合过程，计算中需要应用它（的意义）
 - 已经有约束值的外部变量，计算中需要用它的值
 - 否则，是无定义

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、局部变量名等。非局部定义的名字应该是全局定义的（变量、函数、类型等）。（不考虑宏定义）
 - 同样有局部名字遮蔽外围名字的问题
- C 语言里的名字还具有不同的地位和划分，不同类别是
 - 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
 - 其他为一般标识符，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
 - 尽可能少用全局变量
 - 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全局变量
 - 被一部分函数共享的外部变量，应该考虑能否定义为 **static**
- C++ 还提供了 **namespace** 特征，用于支持更细致的名字划分

过程抽象：内部定义和块结构

- `sqrt` 的相关定义，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（可能是其他地方定义的）

- 使用者实际上只关心 `sqrt`

让其他辅助过程也出现在全局环境里，只会干扰人的思维和工作
还污染了全局名字环境，例如，不能再定义同名过程了

过程抽象：内部定义和块结构

- 对一个结构局部的东西应该定义在这个结构内部

Scheme 支持过程内的局部定义，允许把过程定义放在过程里面

- 通过局部定义重新组织好的 `sqrt` 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))
```

三个内部过程定义

过程体表达式

- 在一个过程内部 `define` 的东西（过程或变量）只能在该过程内使用。
过程之外看不到这些名字。

内部定义和块结构

- 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，是组织程序的重要手段
- 基本想法：
 - 定义的局部化
 - 确定一种作用域单位（块，**block**）
 - 作用单位可以嵌套
 - 在局部定义的变量/过程只能在局部使用，外部看不到局部化使程序更清晰，减少全局名字，减少相互干扰
- 写大型程序时，特别需要控制名字的使用
 - 控制其作用范围（作用域）
 - 防止不同部分之间的名字冲突和误用
 - 信息的尽可能局部化是良好程序设计的重要特征

内部定义和块结构

- 采用局部定义，还可能简化过程定义
 - 由于局部过程在形参 **x** 的作用域里，可直接用 **x**（不必作为参数）
 - 修改后的定义：

```
(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
 - 后来的新语言为程序组织提供了许多专门机制（未必采用块结构）

C 程序结构

- C 程序的结构比较简单：
 - 不支持局部函数定义
 - 受限的程序组织方式（函数都在同一层）
- 程序设计语言发展中，这方面有两条路线：
 - 从 Fortran 到 C 以及 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
- 允许过程嵌套定义，组织程序的方式更丰富
 - 根据需要建立嵌套的过程结构，方便相关信息的局部化
 - C 语言不采用嵌套的过程结构，主要考虑是语言实现简单，目标程序的执行效率高
- 也可能以其他方式支持信息局部化
如建立数据抽象的结构，面向对象语言的类结构等

C 程序结构

- C 语言的组织机制较弱
 - 最高层机制是函数，没有函数之上的组织机制（平坦结构）
 - 后来的编程语言在这方面有很多发展
- C 语言里可以利用程序的物理结构（通过多个源文件）组织程序
static 函数和 **static** 全局变量实现信息局部化。多一层组织结构
- OO 语言的嵌套类也是为了帮助组织程序
有兴趣的同学可以分析一下它与函数嵌套的异同
- 例，为 **sqrt** 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) {...}
```

过程与其产生的计算

- 要真正理解程序设计，还需要理解程序的行为
 - 有许多不同方式可以完成同一件工作，如何选择？为什么？
 - 编程专家必须理解程序蕴涵的计算，理解一个过程（**procedure**）运行时产生什么样的计算进程（**process**）
- 一个过程（是一个文本描述）可以看作一种计算模式，它
 - 描述了一种特定计算的演化进程和方式
 - 对一组适当的参数，确定了一个具体计算（一个计算进程实例，表现为一系列具体的演化步骤）
 - 要完成一件工作，我们可能写出很多大不相同的过程
 - 完成同一工作的两个过程导致的计算进程可能大不相同
- 下面通过例子讨论几个简单过程产生的计算进程的“形状”，观察其中各种资源消耗的情况（主要是时间和空间）

得到的认识可供我们在编写其他程序时参考

线性递归和迭代

- 考虑做阶乘计算。一种看法（递归的观点）：
n 的阶乘就是 n 乘以 n - 1 的阶乘
$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1 = n \cdot [(n-1) \cdot \dots \cdot 2 \cdot 1] = n \cdot (n-1)!$$

- 过程定义：

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

- 用代换模型推导

由 (factorial 6) 得到的
计算进程见图

```
(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 开始顺序乘各个自然数，乘到 n 就得到结果

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

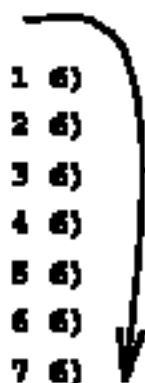
- 按这种观点写出程序：

```
(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)))
```

请重写为采用内部过程的定义

对应计算进程

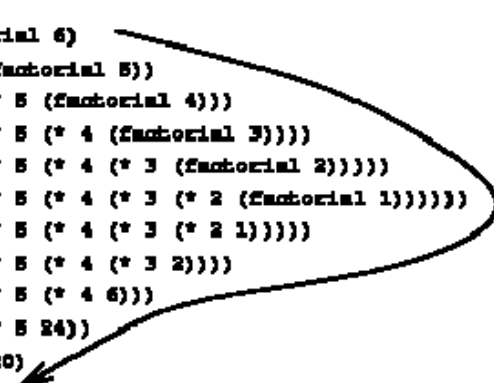
```
(factorial 6)  
(fact-iter 1 1 6)  
(fact-iter 1 2 6)  
(fact-iter 2 3 6)  
(fact-iter 6 4 6)  
(fact-iter 24 5 6)  
(fact-iter 120 6 6)  
(fact-iter 720 7 6)  
720
```



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
```

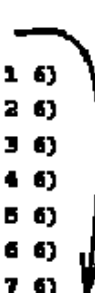


先展开后收缩：展开中积累一些计算，
收缩是完成这些计算

解释器要维护待执行计算的轨迹，轨迹
长度与后续计算的次数成正比

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
```



没有展开/收缩，直接计算

计算轨迹中的信息量为常量，
只要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

- 在迭代计算进程中，所需的所有信息都保存在几个变量里

- 可以在计算中任何一步中断和重启
- 只要有这组变量的当前值，就可以恢复并继续计算

- 例：

对调用 **(fact-iter 24 5 6)**

可知：还需计算 **24 乘 5** 后再乘 **6**

对调用 **(fact-iter 24 5 20)**

可知：还需继续乘 **5** 到 **20** 的整数，才能得到最后的结果

线性递归和迭代：分析

- 在线性递归中，相关变量的信息不足以反映计算进程的情况

- 解释器需要在内部保存一些“隐含”信息
- 这种信息的量随着计算进程的长度而线性增长

- 看阶乘的例子

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

- 假设当前调用是 **(factorial 5)**

无法知道外面还有多少遗留下来尚未进行的计算

不知是从哪里开始递归到求 **5** 的阶乘

如 **(factorial 6)** 和 **(factorial 20)** 执行中都会调用 **(factorial 5)**

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”
 - “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情况
 - “用递归方式定义过程”说的是程序写法，在定义过程的代码里出现了对这个过程本身的调用
- 常规语言里都提供专门循环结构（**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，如果其中对本过程的递归调用都是过程执行的最后一个表达式
 - 虽然是递归定义过程，计算所需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
- 有些常规语言也实现了尾递归优化，有兴趣可以试试

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

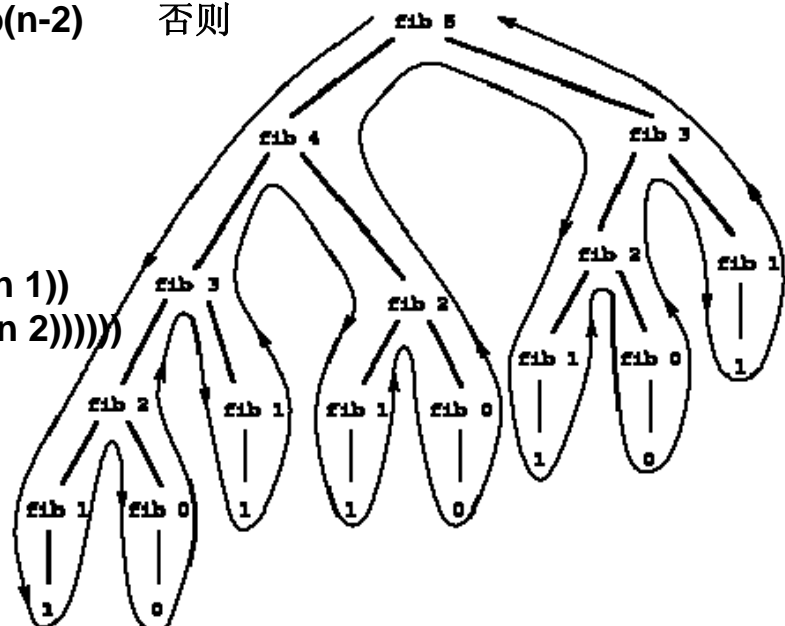
$\text{Fib}(n) = 0$ 若 $n = 0$
 $= 1$ 若 $n = 1$
 $= \text{Fib}(n-1) + \text{Fib}(n-2)$ 否则

- 相应的过程定义：

```
(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 **Fib(n)** 的增长与 **n** 成指数关系（练习1.13），因此 **fib(n)** 的计算量增长与 **n** 的增长成指数关系。这种情况很糟糕
- 考虑 **Fibonacci** 数的另一算法：取变量 **a** 和 **b**，分别初始化为 **Fib(0)** 和 **Fib(1)** 的值，而后反复地同时执行更新操作：

```
a ← a + b
b ← a
```

- 过程定义（形成的计算进程是线性迭代）：

```
(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))))
```

- 采用递归方式写的程序可能低效，为什么还值得关心？
1) 并不必然低效；2) 有些问题用递归描述特别自然，看一个例子

树形递归：换硬币的不同方式

- 人民币的硬币有1元，5角，1角，5分，2分和1分
问题：给了一定的人民币，问有多少种不同方式将它换成硬币？
这个问题用递归方式解决比较简单和自然
- 首先要分析问题，规划出一种对问题的递归观点（算法）
例如：确定一种硬币排列，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
- 递归的观点（递归的分析）
设法把解决原问题归结为在**一定条件下**解决一个/几个相对**更简单的同类问题**（或许还有另外的可以直接解决的问题）
例如：阶乘函数把求 **n** 的阶乘归结为求 **n - 1** 的阶乘
把求 **Fibonacci** 数 **F_n** 归结为求 **F_{n-1}** 和 **F_{n-2}**

换硬币的不同方式

- 这里把用 k 种硬币得到币值 a 归结为两个更简单的情况
 - 用 $k - 1$ 种硬币得到 a （减少一种硬币）
 - 用 k 种硬币得到较少的币值（前面说的 $a - b$ ，减少了币值）
- 递归算法还需把问题最终归结为一种/几种能直接得到结果的基本情况
 - 对阶乘，1 的阶乘是 1
 - 对 Fibonacci 数， F_0 和 F_1 可以直接得到
- 换硬币的几种基本情况：
 - $a = 0$ ，计 1 种方式
 - $a < 0$ ，计 0 种方式，因为不合法
 - 货币种类 $n = 0$ 但 a 不是 0，计 0 种方式，因为已无货币可用
- 综合这些考虑，不难写出一个递归定义的过程

换硬币的不同方式

- 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
```

```
                      (coin-value kinds-of-coins))
```

```
                  kinds-of-coins))))))
```

```
(define (coin-value 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)))
```

思考题，倒转各种硬币的排列顺序会怎么样（自己做试验）：

程序还正确吗？

效率会改变吗？

树形递归

- 实现递归计算过程的过程也有价值：
 - 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）
 - 代码通常更简单，很容易确认它们解决了原来的问题要做出与之对应的实现迭代过程的过程，可能困难得多
- 换零钱不同方式用递归描述很自然，它蕴涵着一个树形递归进程
能写出解决这个问题的迭代程序吗？请试试！
- 有时清晰简单的递归描述的计算代价很高，而对应的高效迭代过程可能很难写。人们也一直在研究：
 - 能否自动地从清晰易写的程序生成出高效的程序？
 - 如不能一般地解决这个问题，是否存在一些有价值的问题类，或一些特定的描述方式，对它们有解决办法？
 - 这个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行程序设计的研究

C 语言里的递归和迭代

- C 与其他常规语言一样通过几种循环语句描述线性迭代式的计算进程
- 常规语言中允许递归定义过程是从 **Algol 60** 开始的
 - 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
- C 语言和其他常规语言都支持递归
 - 一些实现不支持尾递归优化（试试你用的系统）
 - 对尾递归函数，这种运行系统还是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
 - 实现尾递归优化的系统，也可能在一些情况下不做优化

增长的阶

- 算法和数据结构课都讨论计算代价，其中的主要想法
 - 在某种抽象意义上考虑计算代价（增长的阶）
 - 考虑特定计算中各种资源的消耗如何随着问题规模的增长而增长
- 有关问题不再讨论
 - 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（不一定是上确界）
 - 总希望考虑尽可能紧的上界（更准确地反映算法的性质）
- 应该记得：
$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \dots < O(2^n) <$$

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：
$$b^0 = 1 \qquad b^n = b \cdot b^{(n-1)}$$
直接对应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))))
```

线性时间和常量空间（ $O(1)$ 空间）

实例：求幂

- 实际上，求 b^8 的 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

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

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

- 按这种想法定义的过程

```
(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 ? 作为谓词过程名的最后字符，是 Scheme 的编程习惯

本过程求幂所需乘法的次数是 $O(\log n)$ ，是重大改进

实例：素数检查

- 问题：判断整数 n 是否素数。下面讨论两种方法，前一个的复杂性是 $O(\sqrt{n})$ ，另一个是概率算法，复杂性是 $O(\log n)$

- 确定因子的直接方法是用顺序的整数去除。找最小因子的过程：

```
(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）

- 下面考虑的概率算法的基础是费马小定理：

若 n 是素数， a 是任小于 n 的正整数，则 a 的 n 次方与 a 模 n 同余
数 a 除以 n 的余数称为 a 取模 n 的余数，简称 a 取模 n

两个数模 n 同余：它们除以 n 的余数相同

n 不是素数时多数 $a < n$ 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 ，那么 n 不是素数
- 否则重复上述过程

- n 通过检查的次数越多，是素数的可能性就越大

但并不能保证是素数，因此，得到的结论是概率性的

实例：素数的费马检查

- 为实现这一算法，需要定义一个计算自然数的幂并取模的过程：

```
(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))))
```

- 过程 `expmod` 利用了一个数学关系

$(a * b) \bmod c = ((a \bmod c) * (b \bmod c)) \bmod c$

这样做，可以保证计算的中间结果不会变得太大

实例：素数的费马检查

- 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 (- n 1)) 得到区间 $[0, n-2]$ 中的随机数

- “判断”是否素数需要反复做费马检查。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

- 这里的真假只有概率的意义

概率算法

- 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，存在不是素数但却能通过检查的数

人们已提出了其他检查方法，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
 - 概率算法已经发展成了一个重要研究领域，有许多重要应用
 - 在实际中，很多时候也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，用语言里的迭代控制结构（循环结构）实现
- 请大家用 C 语言或其他自己熟悉的语言改写书上程序
- 实现素数判断的概率算法时，利用前面有关 **mod** 的等式可以避免计算中出现很大的中间结果
 - 对 **Scheme** 程序而言这是优化
 - 对常规语言（如 **C**）必须用 **mod**，否则会出现溢出

注意，如果取的模很大，乘法仍可能溢出

总结

- **Scheme** 的一些情况
 - 类型和操作
 - **define** 的两种情况
- 内部过程定义，方法和意义
- 计算进程的形状
 - 线性迭代
 - 线性递归
 - 树形递归
- 算法复杂性