编程基础 IV - 函数式编程范式

刘 钦

Outline

- 避免重复
- 函数式编程范式
- 编程的现实考量
- 证明程序的正确性
- 习题课

避免重复

知道程序员最喜欢的
2个操作是什么吗？

ctrl c
ctrl v

重复输出

ctrl c

ctrl c

ctrl c

ctrl c

代码重复

```
System.out.println(“ctrl c ”);
System.out.println(“ctrl c ”);
System.out.println(“ctrl c ”);
System.out.println(“ctrl c ”);
```

重复输出

ctrl c ctrl c ctrl c ctrl c

重复

```
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
System.out.println(“ctrl c”);
```

消除重复

- for(i=0;i<4;i++)
    - System.out.print(“ctrl c”);

重复代码

- for (int i = 0; i < 4; i++)
- {
-     sum1 += array1[i];
- }
- average1 = sum1/4;
-
- for (int i = 0; i < 4; i++)
- {
-     sum2 += array2[i];
- }
- average2 = sum2/4;

消除重复 — 抽象

```
• for (int i = 0; i < 4; i++)
• {
•   sum1 += array1[i];
• }
• average1 = sum1/4;
•
• for (int i = 0; i < 4; i++)
• {
•   sum2 += array2[i];
• }
• average2 = sum2/4;
```

```
int calcAverage (int* Array_of_4)
{
   int sum = 0;
   for (int i = 0; i < 4; i++)
   {
       sum += Array_of_4[i];
   }
   return sum/4;
}
```

```
int average1 = calcAverage(array1);
int average2 = calcAverage(array2);
```

重复代码 - if-for

```python
def logo_play():
    pic = [[' ' for i in range(10)] for j in range(10)]

    x_position = 0
    y_position = 0

    dictionary_diction = {'R': 1, 'L': -1, 'U': 1, 'D': -1}
    n = int(input())
    draw = '0'

    while n:
        n -= 1

        command = input().split()

        tep = int(command[1])
        direction = command[0]
        if len(command) == 3:
            draw = command[2]

        if direction == 'L':
            for place in range(x_position - tep, x_position):
                pic[y_position][place] = draw
            x_position -= tep
        elif direction == 'R':
            for place in range(x_position + 1, x_position + 1 + tep):
                pic[y_position][place] = draw
            x_position += tep
        elif direction == 'U':
            for place in range(y_position - tep, y_position):
                pic[place][x_position] = draw
            y_position -= tep
        elif direction == 'D':
            for place in range(y_position + 1, y_position + 1 + tep):
                pic[place][x_position] = draw
            y_position += tep

        if x_position<0 or x_position>9 or y_position<0 or y_position>9:
            print('Error!')
            return

    for i in range(10):
        for j in range(10):
            print(pic[i][j], end='')
        if i < 10:
            print()
```

消除重复 - for-if

```
def logo_play():
    logo_list = [[' ' for i in range(10)] for j in range(10)]
    x, y, sth = 0, 0, ' '  # 请问这样赋初始值好吗
    num = int(input())
    for i in range(num):
        order = input().split()
        if len(order) > 2:
            sth = order[2]
        for j in range(int(order[1])):
            if order[0] == 'R':
                x += 1
            elif order[0] == 'L':
                x -= 1
            elif order[0] == 'D':
                y += 1
            elif order[0] == 'U':
                y -= 1
            logo_list[y][x] = sth
            if x < 0 or x > 9 or y < 0 or y > 9:
                print('Error!')
                return
    for i in range(10):
        for j in range(10):
            print(logo_list[i][j], end='')
        print()
    return
```

消除重复 — 数据过滤

```
private Double getTotalSumExcludeCancelAmount(List amounts) {
    double totalToPay = 0.00;
    Iterator amountsIterator = amounts.iterator();
    while (amountsIterator.hasNext()) {
        Amount amount = (Amount) amountsIterator.next();
        if (!amount.getIsToCancel()) { // Additional condition comparing to the first method.
            if (!cancelstatuses.contains(amount.getStatus())) {
                totalToPay += amount.doubleValue();
            }
        }
    }
    return new Double(totalToPay);
}
```

```
private Double getTotalSum(List amounts) {
    double totalToPay = 0.00;
    Iterator amountsIterator = amounts.iterator();
    while (amountsIterator.hasNext()) {
        Amount amount = (Amount) amountsIterator.next();
        if (!cancelstatuses.contains(amount.getStatus())) {
            totalToPay += amount.doubleValue();
        }
    }
    return new Double(totalToPay);
}

private Double getTotalSumExcludeCancelAmount(List amounts) {
    //1. filter the list with “!amount.getIsToCancel()” condition
     List newamounts = filter(amounts);
    //2. get sum
    return getTotalSum(newamounts);
}
```

Outline

- 避免重复
- 函数式编程范式
    - 过程抽象（高阶函数）
    - 数据抽象（有序对、层次性数据、符号数据）
    - 信号流（枚举器+过滤器+映射+累积器）
    - Java 8 stream api
    - 函数式编程范式的特点
    - Hy
- 编程的现实考量
- 证明程序的正确性
- 习题课

过程抽象

高阶函数

- 也是一种抽象
- 过程作为参数

案例

- 考虑三个过程
  - 计算从a到b的各个整数之和
  - 计算给定范围内的整数的立方之和
  - 计算下列序列之和
    - 1/(1*3)+1/(5*7)+1/(9*11)+…

计算从a到b的各个整数之和

- (define (sum-integers a b)
-     (if (> a b)
-         0
-         (+ a (sum-integers(+ a 1) b))))

计算给定范围内的整数的立方之和

- (define (sum-cubes a b)
-     (if (> a b)
-         0
-         (+ (cube a) (sum-cubes(+ a 1) b))))

计算下列序列之和
1/(1*3)+1/(5*7)+1/(9*11)+…

```
• (define ( sum-pi a b)
•     (if (> a b)
•         0
•         (+ (/ 1.0 (* a (+ a 2))) (sum-pi( + a 4) b))))
```

模板

- (define (<name> a b)
- (if (> a b)
- 0
- (+ (<term> a) (<name>(<next> a) b))))

形参是函数

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

计算从a到b的各个整数之和 - 高阶函数版

- (define (identity x) x)
- (define (sum-integer a b)
-     (sum identity a inc b))

计算给定范围内的整数的立方之和 - 高阶函数版

- (define (inc n) (+ n 1))
- (define (sum-cubs a b)
-     (sum cube a inc b))

计算下列序列之和

1/(1*3)+1/(5*7)+1/(9*11)+… - 高阶函数版

- (define (sum-pi a b)
-     (define (pi-term x)
    - (/ 1.0 (* x (+ x 2))))
-     (define (pi-next x)
    - (+ x 4)
    - (sum pi-term a pi-next b))

利用lambda表达式构造过程

- 表达高阶函数
- (define (pi-sum a b)
    - (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
        - a
        - (lambda (x) (+ x 4))
        - b))

例子

- (define (f x y)
-   (define (f-helper a b)
-     (+ (* x (square a))
-        (* y b)
-        (* a b)))
-   (f-helper (+ 1 (* x y))
-             (- 1 y)))

利用lambda表达式构造过程

- 表达局部变量
- (define (f x y)
-   ((lambda (a b)
-      (+ (* x (square a))
-         (* y b)
-         (* a b)))
-    (+ 1 (* x y))
-    (- 1 y)))

let关键字

- (define (f x y)
- (let ((a (+ 1 (* x y)))
- (b (- 1 y)))
- (+ (* x (square a))
- (* y b)
- (* a b))))

数据抽象

有理数

- 有理数表示为两个整数的有序对

有理数

- (define (make-rat n d) (cons n d))
- (define (numer x) (car x)
- (define (denom x) (cdr x))

考虑有理数约化到最简形式

- (define (make-rat n d)
  - (let ((g (gcd n d)))
    - (cons (/ n g) (/ d g))))

链表

- 1-》 2 -》 3-》 4

链表

- (list 1 2 3 4)
- 等价于
- (cons 1
  - (cons 2
    - (cons 3
      - (cons 4 nil))))

list-ref

- 返回第n个项（从0开始计数）
  - n = 0 list-ref 返回表的car
  - 否则，返回表的cdr的第n-1个项

list-ref

- (define (list-ref items n)
  - (if (= n 0)
    - (car items)
    - (list-ref (cdr items) (- n 1))))

scale-list

- 将一个表里所有元素按给定的因子做一次缩放

scale-list

```
•  (define (scale-list items factor)
•    (if (null? items)
•        nil
•        (cons (* (car items) factor)
•              (scale-list (cdr items) factor))))
```

map

- 返回将这一过程应用于表中各个元素得到的结果形成的表

map

- `(define (map proc items)`
- `  (if (null? items)`
- `      nil`
- `      (cons (proc (car items))`
- `            (map proc (cdr items)))))`

map的应用

- (map abs (list -10 2.5 -11.6 17))
- (10 2.5 11.6 17)
- (map (lambda (x) (* x x))
-      (list 1 2 3 4))
- (1 4 9 16)

- Now we can give a new definition of scale-list in terms of map:

- (define (scale-list items factor)
-   (map (lambda (x) (* x factor))
-        items))

层次性结构

- ()
- () ()
- 1 () 3 ()
- 2 nil 4 nil

- （（1，2），（3，4））

层次性结构

- (cons (list 1 2) (list 3 4))
- (cons (cons 1(cons 2 nil))(cons 3 (cons 4 nil)))

count-leaves

- 数树的叶节点个数
    - 空表的叶节点个数为0；
    - 一个树叶的叶节点为1；
    - 否则为 car的叶节点个数+ cdr的叶节点个数。

count-leaves

- (define (count-leaves x)
  - (cond ((null ? x) 0)
    - ((not (pair ? x)) 1)
    - (else (+ (count-leaves (car x))
      - (count-leaves (cdr x))))))

scale-tree

- 数值树
- 返回一颗具有同样形状的树，树中的每个数值都乘以这个因子

scale-tree

- (define (scale-tree tree factor)
-     (cond ((null? tree) null)
-             ((not (pair? tree)) (* tree factor))
-             (else (cons (scale-tree (car tree) factor)
-                         (scale-tree (cdr tree) factor)))))

scale-tree lambda表达

```
• (define (map proc items)
•   (if (null? items)
•       nil
•       (cons (proc (car items))
•             (map proc (cdr items)))))
```

```
• (define (scale-tree tree factor)
•   (map (lambda (sub-tree)
•          (if (pair? sub-tree)
•              (scale-tree sub-tree
  factor)
•              (* sub-tree factor)))
•        tree))
```

求导

`(variable? e)` `is_variable(e)`
Is `e` a variable?
`(same-variable? v1 v2)` `is_same_variable(v1, v2)`
Are `v1` and `v2` the same variable?
`(sum? e)` `is_sum(e)`
Is `e` a sum?
`(addend e)` `addend(e)`
Addend of the sum `e`.
`(augend e)` `augend(e)`
Augend of the sum `e`.
`(make-sum a1 a2)` `make_sum(a1, a2)`
Construct the sum of `a1` and `a2`.
`(product? e)` `is_product(e)`
Is `e` a product?
`(multiplier e)` `multiplier(e)`
Multiplier of the product `e`.
`(multiplicand e)` `multiplicand(e)`
Multiplicand of the product `e`.
`(make-product m1 m2)` `make_product(m1, m2)`
Construct the product of `m1` and `m2`.

$$\begin{aligned}
\frac{dc}{dx} &= 0 \quad \text{for } c \text{ a constant or a variable different from } x \\
\frac{dx}{dx} &= 1 \\
\frac{d(u+v)}{dx} &= \frac{du}{dx} + \frac{dv}{dx} \\
\frac{d(uv)}{dx} &= u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)
\end{aligned}$$

求导

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
          (if (same-variable? exp var) 1 0))
        ((sum? exp)
          (make-sum (deriv (addend exp) var)
                    (deriv (augend exp) var)))
        ((product? exp)
          (make-sum
            (make-product (multiplier exp)
                          (deriv (multiplicand exp) var))
            (make-product (deriv (multiplier exp) var)
                          (multiplicand exp))))
        (else
          (error "unknown expression type -- DERIV" exp))))
```

求导

```scheme
(define (variable? x) (symbol? x))

(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))

(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))

(define (addend s) (cadr s))

(define (augend s) (caddr s))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))

(define (multiplier p) (cadr p))

(define (multiplicand p) (caddr p))
```

Make-sum

```
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
```

乗积Make-product?

Make-product

```
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

信号流

信号流

- 枚举器-》过滤器-》映射-》累积器

计算值为奇数的叶子的平方和

- 枚举器-》过滤器-》映射-》累积器
- tree leaves-》odd? -》square-》+，0

计算值为奇数的叶子的平方和

```
• (define (sum-odd-squares tree)
•   (cond ((null? tree) 0)
•         ((not (pair? tree))
•          (if (odd? tree) (square tree) 0))
•         (else (+ (sum-odd-squares (car tree))
•                  (sum-odd-squares (cdr tree))))))
```

所有偶数的fibnacci数列

- 枚举器-》映射-》过滤器-》累积器
- integer-》fib -》even?》cons,()

所有偶数的fibnacci数列

```
• (define (even-fibs n)
•  (define (next k)
•   (if (> k n)
•     nil
•     (let ((f (fib k)))
•      (if (even? f)
•        (cons f (next (+ k 1)))
•        (next (+ k 1))))))
•  (next 0))
```

案例

- 创建一个Messages Collection
- List<Message> messages = new ArrayList<>();
- messages.add(new Message("aglover", "foo", 56854));
- messages.add(new Message("aglover", "foo", 85));
- messages.add(new Message("aglover", "bar", 9999));
- messages.add(new Message("rsmith", "foo", 4564));

- 具体来讲，我希望找到Message当中所有延迟周期超过3000秒的条目并计算它们的总计延迟时长。

普通Java青年写法

```
• long totalWaitTime = 0;
• for (Message message : messages)
• {
•         if (message.delay > 3000)
•         {
•                 totalWaitTime += message.delay;
•         }
• }
```

文艺Java 8青年写法

- Java 8 Stream API
    - long totWaitTime = messages.stream().filter(m -> m.delay > 3000).mapToLong(m -> m.delay).sum();

函数式编程特点

函数式编程

- From wiki：
  - In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- 参考 WIKI 的解释，函数式编程（ Functional programming ）指的是一种编程范式（ Programming paradigm ），将计算机运算看作是数学中函数的计算，并且避免了状态以及变量的概念。
- 　　我们所了解到的面向对象编程、指令式编程以及所要讲的函数式编程均为不同的编程范式。编程范式的主要作用在于提供且同时决定了程序员对于程序执行的看法。例如，在面向对象编程中，程序员认为程序是一系列相互作用的对象，而在函数式编程中一个程序会被看作是一个无状态的函数计算的序列。

函数式编程范式

A. Functions As First-class Citizens

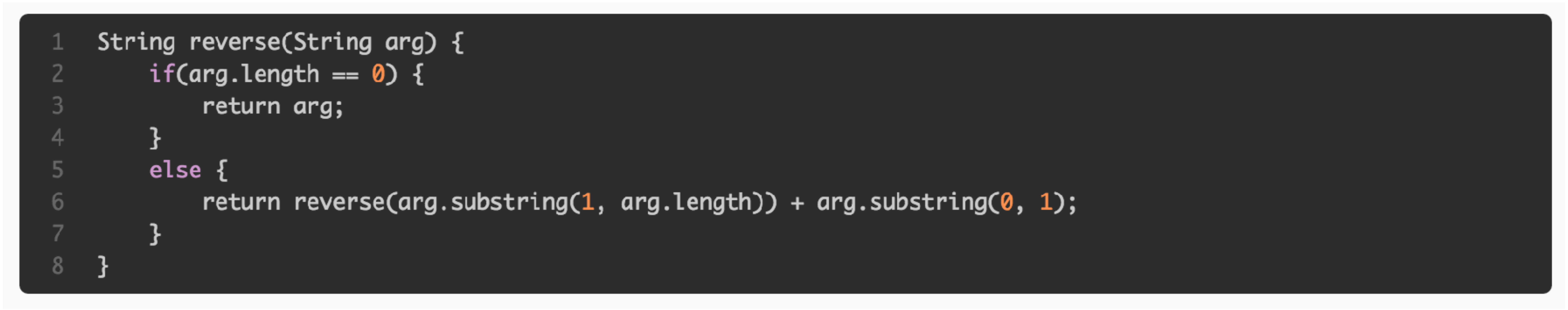
   A. 函数被当作头等公民，意味着函数可以作为别的函数的参数、函数的返回值，赋值给变量或存储在数据结构中，通常以高阶函数（ Higher Order Functions ）的形式存在。

B. No Side Effects

   A. 在计算机科学中，函数副作用（ Side Effect ）指当调用函数时，除了返回函数值之外，还对外部作用域产生附加的影响，例如修改全局变量（函数外的变量）或修改参数。而严格的函数式编程是要求函数必须没有副作用，这意味着影响函数返回值的唯一因素就是它的参数。

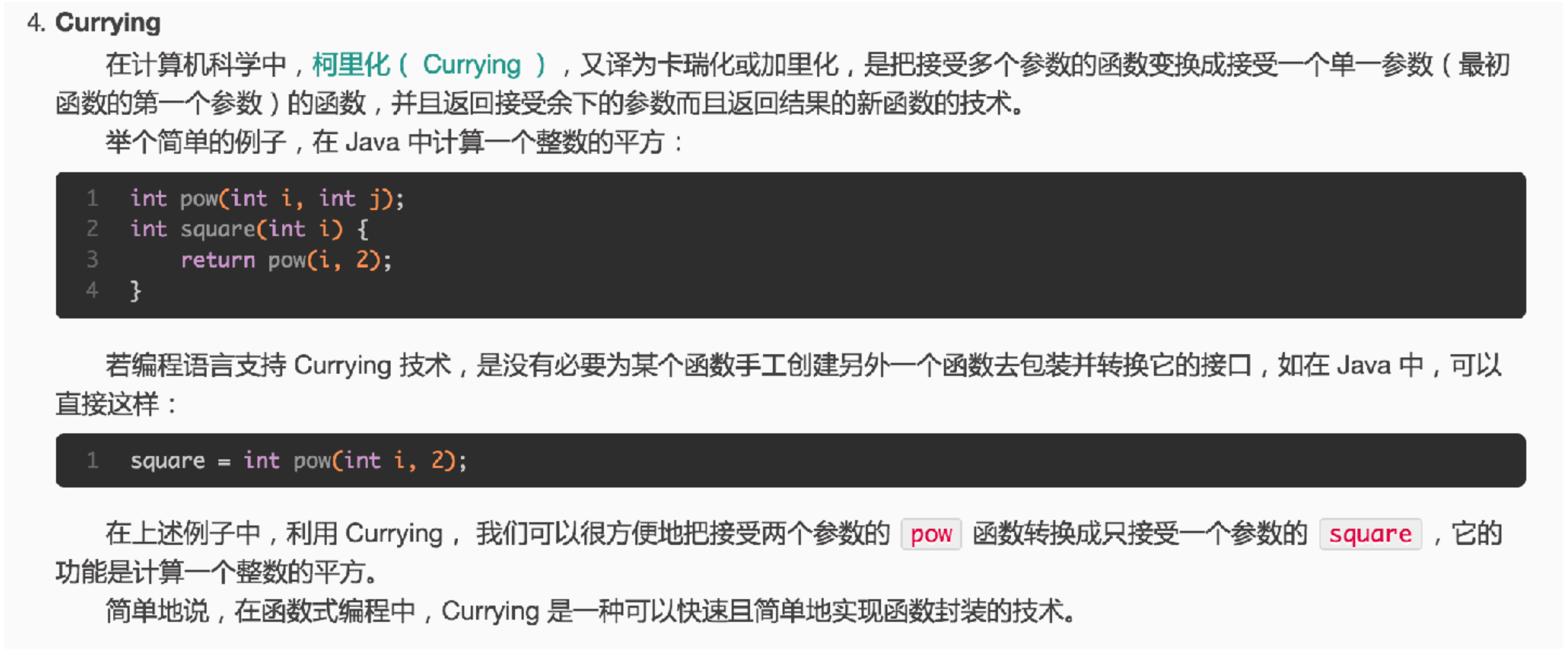
C. No Changing-state

   A. 在函数式编程中，函数就是基础元素，可以完成几乎所有的操作，哪怕是最简单的计算，也是用函数来完成的，而我们平时在其他类型中所理解的变量（ 可修改，往往用来保存状态 ）在函数式编程中，是不可修改的，这意味着状态（ State ）不能保存在变量中，而事实上函数式编程是使用函数参数来保存状态，最好的例子便是递归。



```
String reverse(String arg) {
    if(arg.length == 0) {
        return arg;
    }
    else {
        return reverse(arg.substring(1, arg.length)) + arg.substring(0, 1);
    }
}
```

翻转一个字符串



4. **Currying**

在计算机科学中，柯里化（ Currying ），又译为卡瑞化或加里化，是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

举个简单的例子，在 Java 中计算一个整数的平方：

```
int pow(int i, int j);
int square(int i) {
    return pow(i, 2);
}
```

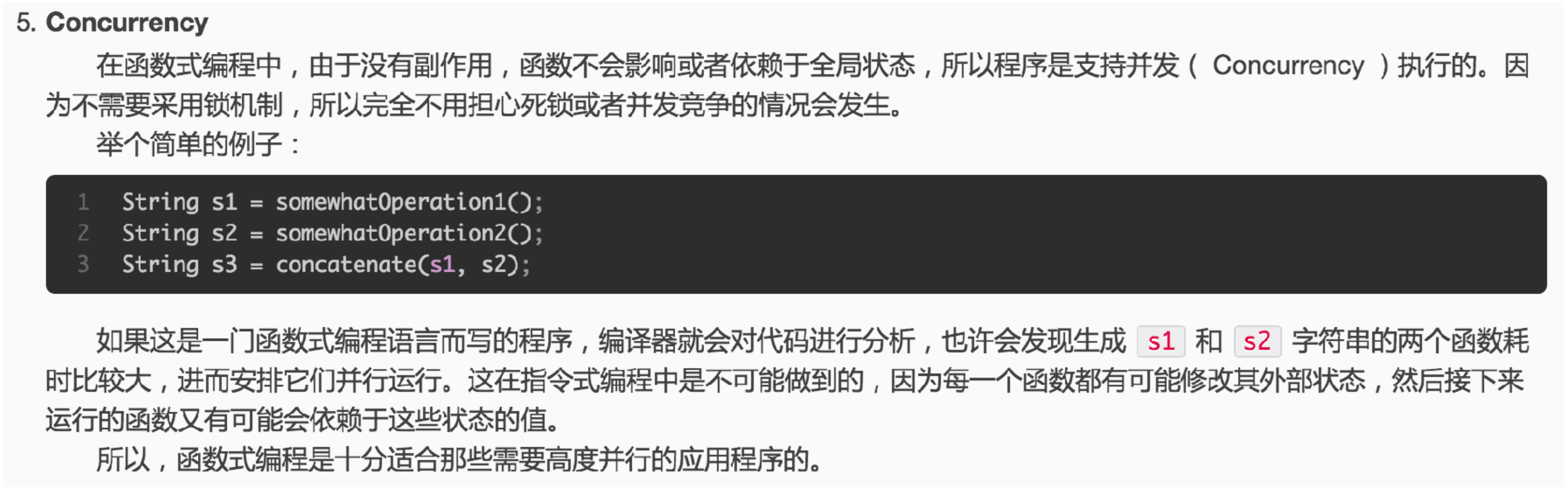
若编程语言支持 Currying 技术，是没有必要为某个函数手工创建另外一个函数去包装并转换它的接口，如在 Java 中，可以直接这样：

```
square = int pow(int i, 2);
```

在上述例子中，利用 Currying， 我们可以很方便地把接受两个参数的 `pow` 函数转换成只接受一个参数的 `square` ，它的功能是计算一个整数的平方。

简单地说，在函数式编程中，Currying 是一种可以快速且简单地实现函数封装的技术。

Currying



5. **Concurrency**

在函数式编程中，由于没有副作用，函数不会影响或者依赖于全局状态，所以程序是支持并发（ Concurrency ）执行的。因为不需要采用锁机制，所以完全不用担心死锁或者并发竞争的情况会发生。

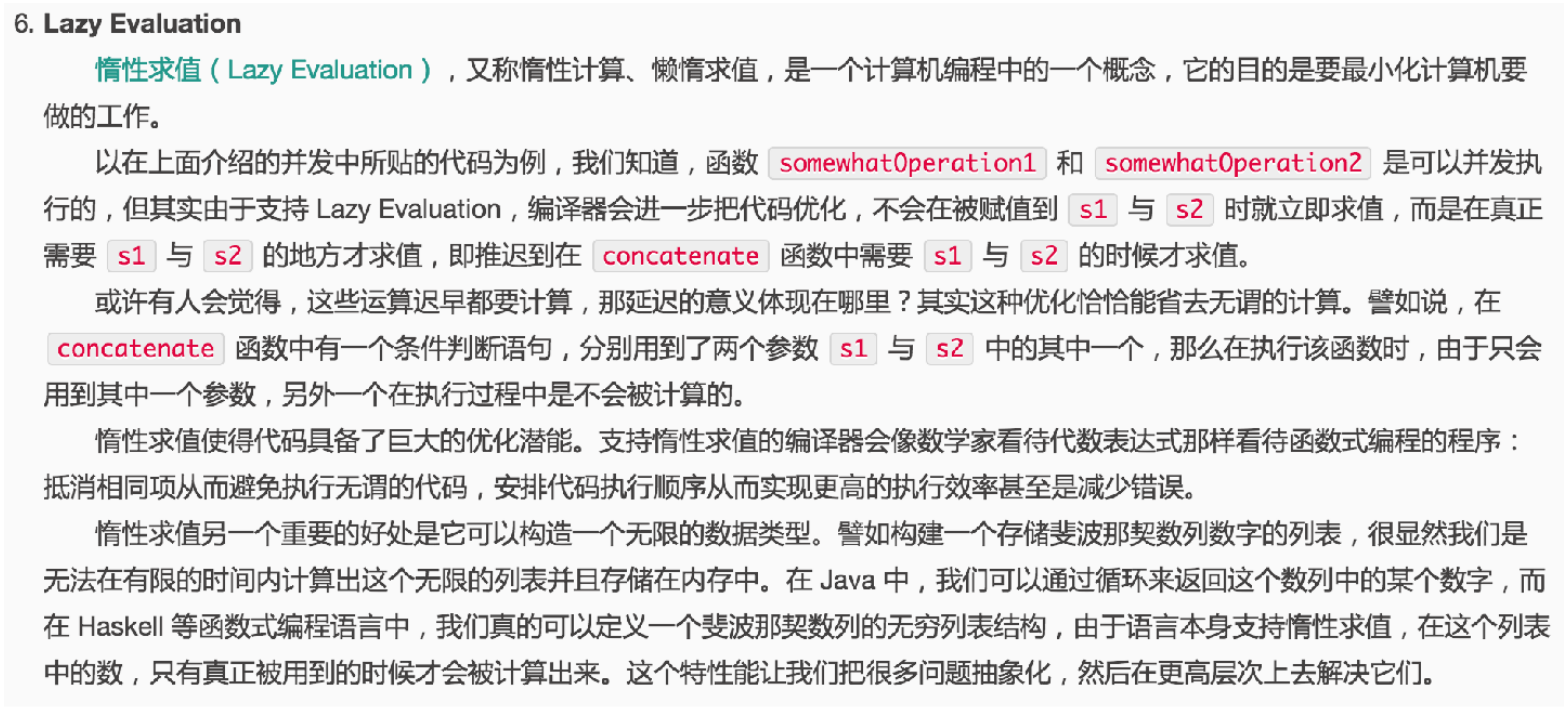
举个简单的例子：

```
String s1 = somewhatOperation1();
String s2 = somewhatOperation2();
String s3 = concatenate(s1, s2);
```

如果这是一门函数式编程语言而写的程序，编译器就会对代码进行分析，也许会发现生成 `s1` 和 `s2` 字符串的两个函数耗时比较大，进而安排它们并行运行。这在指令式编程中是不可能做到的，因为每一个函数都有可能修改其外部状态，然后接下来运行的函数又有可能会依赖于这些状态的值。

所以，函数式编程是十分适合那些需要高度并行的应用程序的。

Concurrency



6. **Lazy Evaluation**

惰性求值（Lazy Evaluation），又称惰性计算、懒惰求值，是一个计算机编程中的一个概念，它的目的是要最小化计算机要做的工作。

以在上面介绍的并发中所贴的代码为例，我们知道，函数 `somewhatOperation1` 和 `somewhatOperation2` 是可以并发执行的，但其实由于支持 Lazy Evaluation，编译器会进一步把代码优化，不会在被赋值到 `s1` 与 `s2` 时就立即求值，而是在真正需要 `s1` 与 `s2` 的地方才求值，即推迟到在 `concatenate` 函数中需要 `s1` 与 `s2` 的时候才求值。

或许有人会觉得，这些运算迟早都要计算，那延迟的意义体现在哪里？其实这种优化恰恰能省去无谓的计算。譬如说，在 `concatenate` 函数中有一个条件判断语句，分别用到了两个参数 `s1` 与 `s2` 中的其中一个，那么在执行该函数时，由于只会用到其中一个参数，另外一个在执行过程中是不会被计算的。

惰性求值使得代码具备了巨大的优化潜能。支持惰性求值的编译器会像数学家看待代数表达式那样看待函数式编程的程序：抵消相同项从而避免执行无谓的代码，安排代码执行顺序从而实现更高的执行效率甚至是减少错误。

惰性求值另一个重要的好处是它可以构造一个无限的数据类型。譬如构建一个存储斐波那契数列数字的列表，很显然我们是无法在有限的时间内计算出这个无限的列表并且存储在内存中。在 Java 中，我们可以通过循环来返回这个数列中的某个数字，而在 Haskell 等函数式编程语言中，我们真的可以定义一个斐波那契数列的无穷列表结构，由于语言本身支持惰性求值，在这个列表中的数，只有真正被用到的时候才会被计算出来。这个特性能让我们把很多问题抽象化，然后在更高层次上去解决它们。

Lazy Evaluation

Outline

- 避免重复
- 函数式编程范式
- 编译器的结构
- 编程的现实考量
- 证明程序的正确性
- 习题课

现实考量

现实考量

- 复杂性与规模
- 逻辑与物理
- 时间与空间

An intellectual challenge

- The art of programming is the art of organising complexity.
- -- Edsger W. Dijkstra

Size of problem

- “Difference in scale is one of the major source of our difficulties in programming”
- - E.W. Dijkstra

An “imperfect” Implementation

- real number a,b,c
    - a == b and b == c 能推倒出 a == c
- 计算机实现是否perfect?
    - 浮点数的精度问题

Proof vs Implementation

- 逻辑实现
- 物理实现

Dynamic vs Static

- Static
  - 代码
  - 存储
  - 数据结构
- Dynamic
  - 执行
  - 处理器
  - 算法

Storage Space vs Computation Speed

- Processor
  - active
  - computation speed
  - fast
- Store
  - passive
  - storage space
  - large

案例 1： 想要得到f(x)的值

- 方法一：算法驱动
    - 只在需要f(x)的值的时候，通过计算得到。
- 方法二：数据驱动
    - 先通过计算，保存所有x对应的f(X)值，需要计算的时候直接访问。

案例 2： PhoneKeyPad

```
• public class PhoneKeyPad{
•     /**
•      * input:"AbcDEGj"
•      * output:"2,2,2,3,3,4,5"
•      * @param s
•      * @return output
•      */
•     public static String convert(String s){
•             return "";
•     }
• }
```

方法一：　算法驱动

```java
public class PhoneKeyPad {
	/**
	 * input:"AbcDEGj"
	 * output:"2,2,2,3,3,4,5"
	 * @param s
	 * @return output
	 */
	public static String convert(String s){
		String lower = s.toLowerCase();
		char[] characters = lower.toCharArray();
		String result = "";
		int count = 0;
		for(int i = 0;i < characters.length;i++){
			if((characters[i] == 'a')||(characters[i] == 'b')||(characters[i] == 'c')){
				result = result + "2";
			} else {
			if((characters[i] == 'd')||(characters[i] == 'e')||(characters[i] == 'f')){
				result = result + "3";
			} else {
			if((characters[i] == 'g')||(characters[i] == 'h')||(characters[i] == 'i')){
				result = result + "4";
			} else {
			if((characters[i] == 'j')||(characters[i] == 'k')||(characters[i] == 'l')){
				result = result + "5";
			} else {
			if((characters[i] == 'm')||(characters[i] == 'n')||(characters[i] == 'o')){
				result = result + "6";
			} else {
			if((characters[i] == 'p')||(characters[i] == 'q')||(characters[i] == 'r')||(characters[i] == 's')){
				result = result + "7";
			} else {
			if((characters[i] == 't')||(characters[i] == 'u')||(characters[i] == 'v')){
				result = result + "8";
			} else {
			if((characters[i] == 'w')||(characters[i] == 'x')||(characters[i] == 'y')||(characters[i] == 'z')){
				result = result + "9";
			}}}}}}}}
			count++;
			if(count != s.length()){
				result = result + ",";
			}
		}
		return result;
	}
}
```

```
public class PhoneKeyPad {
	private static String[] keyPad =
	{ "", "", "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TUV", "WXYZ" };

	public static String convert(String s) {
		String resultString = "";
		s = s.toUpperCase();
		String sp = "";
		for (int i = 0; i < s.length(); i++) {
			char c = s.charAt(i);
			for (int j = 2; j < 10; j++) {

				if (keyPad[j].contains(c + "")) {
					resultString += sp + j;
				}
			}
			sp = ",";
		}

		return resultString;

	}
}
```

方法二： 数据驱动

案例 3： 消息处理

- 假设有一个程序，需要处理其他程序发送的消息，消息类型是字符串，每个消息都需要一个函数进行处理。

```
• void msg_proc(const char *msg_type, const char *msg_buf)
• {
•     if (0 == strcmp(msg_type, "inivite"))
•     {
•         inivite_fun(msg_buf);
•     }
•     else if (0 == strcmp(msg_type, "tring_100"))
•     {
•         tring_fun(msg_buf);
•     }
•     else if (0 == strcmp(msg_type, "ring_180"))
•     {
•         ring_180_fun(msg_buf);
•     }
•     else if (0 == strcmp(msg_type, "ring_181"))
•     {
•         ring_181_fun(msg_buf);
•     }
•     else if (0 == strcmp(msg_type, "ring_182"))
•     {
•         ring_182_fun(msg_buf);
•     }
•     else if (0 == strcmp(msg_type, "ring_183"))
•     {
•         ring_183_fun(msg_buf);
•     }
•     else if (0 == strcmp(msg_type, "ok_200"))
•     {
•         ok_200_fun(msg_buf);
•     }
```

方法一： 算法驱动

```
•     。。。。。。
•     else if (0 == strcmp(msg_type, "fail_486"))
•     {
•         fail_486_fun(msg_buf);
•     }
•     else
•     {
•         log("未识别的消息类型%s\n", msg_type);
•     }
• }
```

- 上面的消息类型取自sip协议（不完全相同，sip协议借鉴了http协议），消息类型可能还会增加。看着常常的流程可能有点累，检测一下中间某个消息有没有处理也比较费劲，而且，没增加一个消息，就要增加一个流程分支。

方法二： 数据驱动

- typedef void (*SIP_MSG_FUN)(const char *);

- typedef struct __msg_fun_st
- {
-     const char *msg_type;//消息类型
-     SIP_MSG_FUN fun_ptr;//函数指针
- }msg_fun_st;

- msg_fun_st msg_flow[] =
- {
-         {"inivite", inivite_fun},
-         {"tring_100", tring_fun},
-         {"ring_180", ring_180_fun},
-         {"ring_181", ring_181_fun},
-         {"ring_182", ring_182_fun},
-         {"ring_183", ring_183_fun},
-         {"ok_200", ok_200_fun},

-         。。。。。。
-         {"fail_486", fail_486_fun}
- };

- void msg_proc(const char *msg_type, const char *msg_buf)
- {
-     int type_num = sizeof(msg_flow) / sizeof(msg_fun_st);
-     int i = 0;

-     for (i = 0; i < type_num; i++)
-     {
-         if (0 == strcmp(msg_flow[i].msg_type, msg_type))
-         {
-             msg_flow[i].fun_ptr(msg_buf);
-             return ;
-         }
-     }
-     log("未识别的消息类型%s\n", msg_type);
- }

Outline

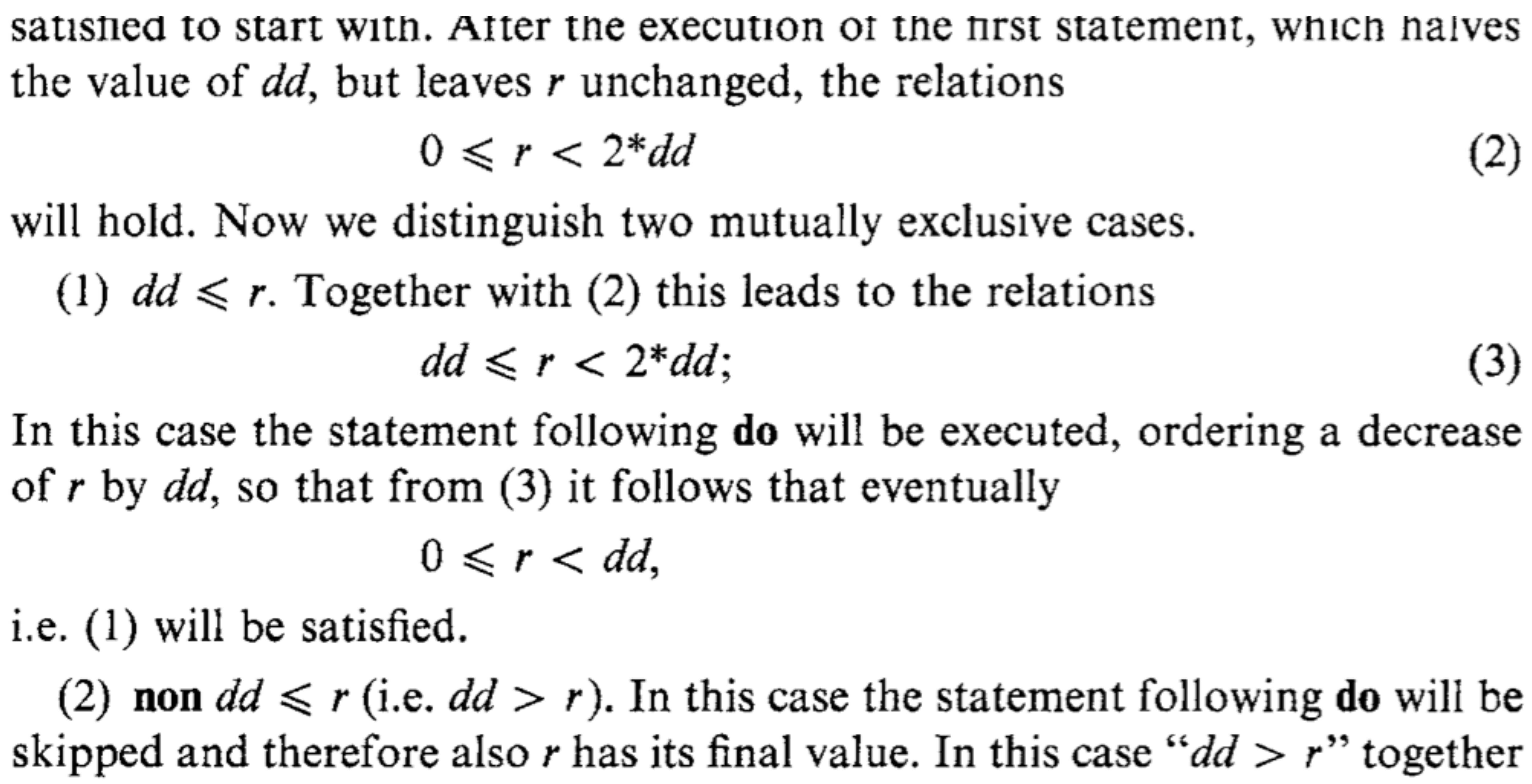
- 避免重复
- 函数式编程范式
- 编程的现实考量
- 证明程序的正确性
  - Dijkstra
  - Hoare
- 习题课

How to proof your program’s correctness

- Edsger W. Dijkstra
  - Enumeration
  - Mathematical induction
  - Abstraction
- C. A. R. HOARE
  - Axioms proof

How to proof — Edsger W. Dijkstra

- Two statements
    - int r = a; int dd =d;
    - while (dd<=r)
        - dd=2*dd;
    - while (dd!=d){
        - dd = dd/2;   //halve dd;
        - if（dd<=r）{ r= r-dd;} //reduce r modulo dd
    - }
- operating on the variable “r” and “dd” leaves the relations
    - 0 <= r < dd
        - which is satisfied to start with



satisfied to start with. After the execution of the first statement, which halves the value of *dd*, but leaves *r* unchanged, the relations

$$0 \leqslant r < 2*dd \qquad (2)$$

will hold. Now we distinguish two mutually exclusive cases.

(1) $dd \leqslant r$. Together with (2) this leads to the relations

$$dd \leqslant r < 2*dd; \qquad (3)$$

In this case the statement following **do** will be executed, ordering a decrease of *r* by *dd*, so that from (3) it follows that eventually

$$0 \leqslant r < dd,$$

i.e. (1) will be satisfied.

(2) **non** $dd \leqslant r$ (i.e. $dd > r$). In this case the statement following **do** will be skipped and therefore also *r* has its final value. In this case "$dd > r$" together

Enumeration

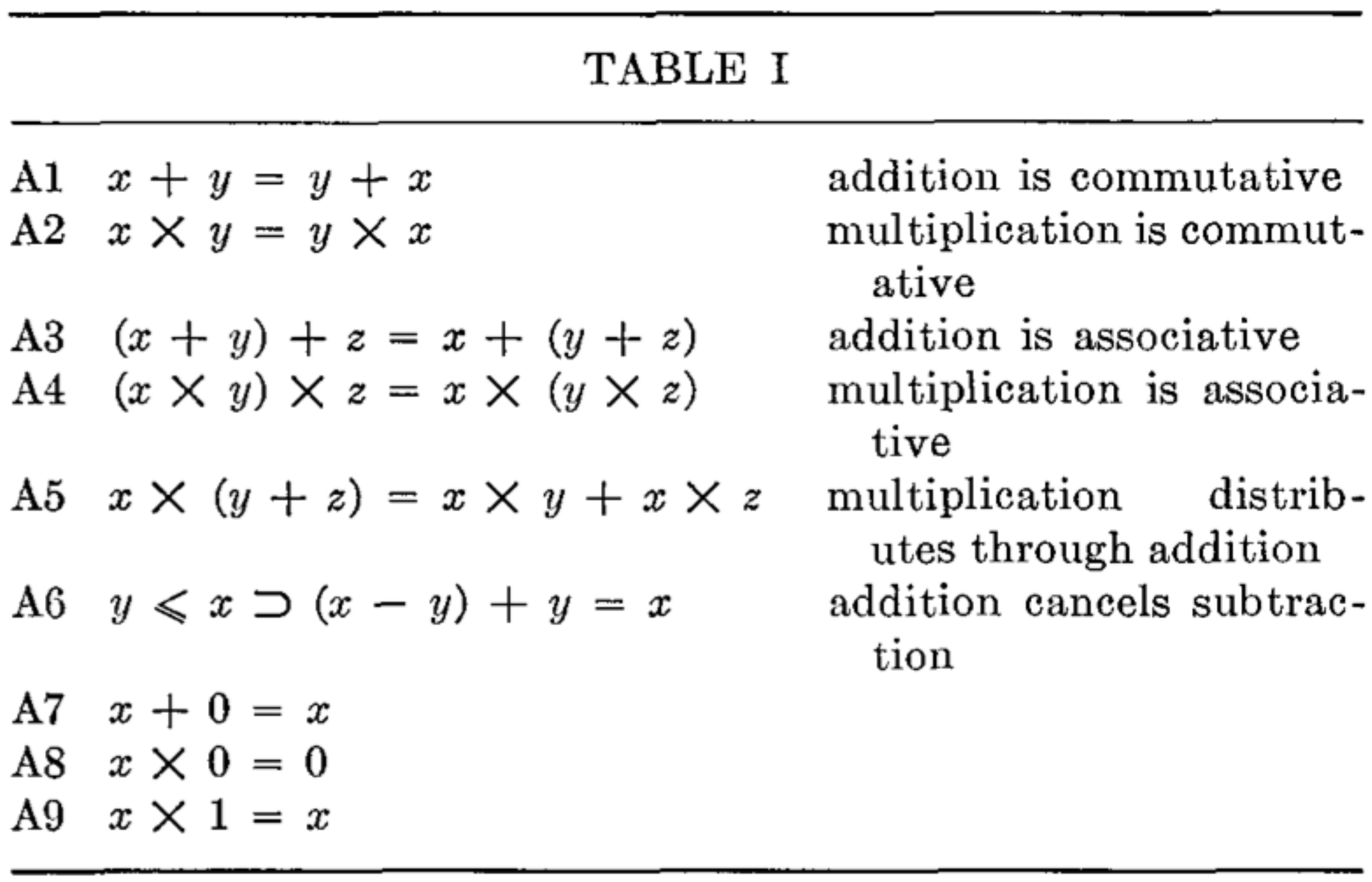
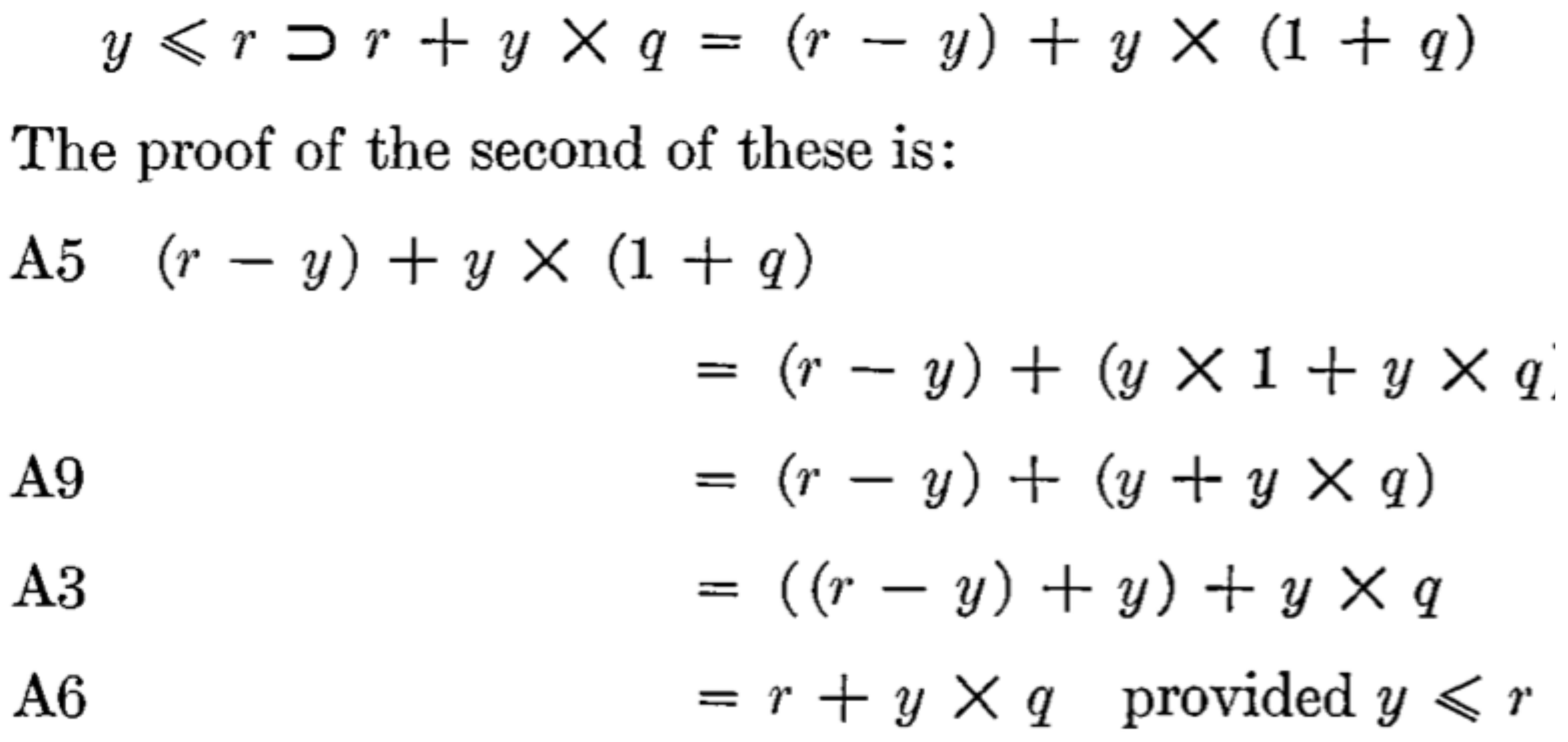


TABLE I

A1	$x + y = y + x$	addition is commutative
A2	$x \times y = y \times x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \times y) \times z = x \times (y \times z)$	multiplication is associative
A5	$x \times (y + z) = x \times y + x \times z$	multiplication distributes through addition
A6	$y \leqslant x \supset (x - y) + y = x$	addition cancels subtraction
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	



$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

How to proof — Hoare

D0 Axiom of Assignment

$\vdash P_0 \{x := f\}\, P$

where

x is a variable identifier;

f is an expression;

P_0 is obtained from P by substituting f for all occurrences of x.

D1 Rules of Consequence

If $\vdash P\{Q\}R$ and $\vdash R \supset S$ then $\vdash P\{Q\}S$

If $\vdash P\{Q\}R$ and $\vdash S \supset P$ then $\vdash S\{Q\}R$

D2 Rule of Composition

If $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1 ; Q_2)\}R$

D3 Rule of Iteration

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P$

TABLE III

Line number	Formal proof	Justification
1	$\textbf{true} \supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0\{r := x\}x = r + y \times 0$	D0
3	$x = r + y \times 0\ \{q := 0\}\ x = r + y \times q$	D0
4	$\textbf{true}\ \{r := x\}\ x = r + y \times 0$	D1 (1, 2)
5	$\textbf{true}\ \{r := x;\ \ q := 0\}\ x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leqslant r \supset x = (r-y) + y \times (1+q)$	Lemma 2
7	$x = (r-y) + y \times (1+q)\{r := r-y\}x = r + y \times (1+q)$	D0
8	$x = r + y \times (1+q)\{q := 1+q\}x = r + y \times q$	D0
9	$x = (r-y) + y \times (1+q)\{r := r-y;\ q := 1+q\}\ x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leqslant r\ \{r := r-y;\ q := 1+q\}\ x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q\ \{\textbf{while}\ y \leqslant r\ \textbf{do}\ (r := r-y;\ \ q := 1+q)\}\ \neg y \leqslant r \wedge x = r + y \times q$	D3 (10)
12	$\textbf{true}\ \{((r := x;\ \ q := 0);\ \ \textbf{while}\ y \leqslant r\ \textbf{do}\ (r := r-y;\ \ q := 1+q))\}\ \neg y \leqslant r \wedge x = r + y \times q$	D2 (5, 11)

Outline

- 避免重复
- 函数式编程范式
- 编程的现实考量
- 证明程序的正确性
- 习题课
    - 迭代和递归
    - Lambda验算
    - 过程建模和数据建模
    - 高阶函数
    - 函数式编程

递归和迭代

+ 加法

- 下面几个过程定义了一种加起来两个正整数的方法，他们都是基于过程inc（他将参数加1）和dec（他将参数减少1）

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
```

```
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

- 求值(+ 4 5)时的计算过程

Fibonacci数列

- Fib(n)
- = 0 n=0
-    1 n=1
-    Fib(n-1)+Fib(n-2) 否则

Fibonacci数列

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

最大公约数

- 如果r是a除以b的余数，那么a和b的公约数正好也是b和r的公约数

最大公约数

- (define (gcd a b)
  - (if (= b 0)
    - a
    - (gcd b (remainder a b))))

换零钱

- 将1美元换成半美元、四分之一美元、10美分、5美分、1美分总共有多少种换法?

换零钱

```
• (define (count-change amount)
•    (cc amount 5))

• (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) 5)
•          ((= kinds-of-coins 3) 10)
•          ((= kinds-of-coins 4) 25)
•          ((= kinds-of-coins 5) 50)))
```

Lamdar演算

Lambda> SUB FOUR TWO

- **SUCC** = $\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
- **PLUS** = $\lambda m.\lambda n.m$ **SUCC** n
- **PRED** = $\lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u)$
- **SUB** = $\lambda m.\lambda n.n$ **PRED** m

- Lambda> SUB FOUR TWO
- \f.\x.f (f x)

SUB FOUR TWO

- SUB FOUR TWO
- =\ m.\n. n PRED m FOUR TWO
- = TWO PRED FOUR
- = \f.\x. f(f(x)) PRED FOUR
- = PRED (PRED (FOUR))
- = TWO

IF (EQ ONE TWO) a b

- Lambda> TRUE = \x.\y.x
- Lambda> FALSE = \x.\y.y
- Lambda> AND = \p.\q.p q p
- Lambda> OR = \p.\q.p p q
- Lambda> NOT = \p.\a.\b.p b a
- Lambda> IF = \p.\a.\b.p a b
- Lambda> ISZERO = \n.n (\x.FALSE) TRUE
- Lambda> LEQ = \m.\n.ISZERO (SUB m n)
- Lambda> EQ = \m.\n. AND (LEQ m n) (LEQ n m)

- Lambda> IF (EQ ONE TWO) a b
- b

IF (EQ ONE TWO) a b

- EQ ONE TWO
- = \m.\n. AND (LEQ m n) (LEQ n m) ONE TWO
- = AND (LEQ ONE TWO) (LEQ TWO ONE)
- = AND (LEQ TWO ONE) (LEQ ONE TWO)
- = AND (\m.\n. ISZERO (SUB m n) TWO ONE) (LEQ ONE TWO)
- = AND (ISZERO ONE) (LEQ ONE TWO)
- = AND (\n.n (\x.F) T ONE) (LEQ ONE TWO)
- = AND (ONE (\x.F) T) (LEQ ONE TWO)
- = AND (\f.\x.f(x) (\x.F) T) (LEQ ONE TWO)
- = AND ((\x.F) T) (LEQ ONE TWO)
- = AND (F) (LEQ ONE TWO)
- = \p.\q. p q p F (LEQ ONE TWO)
- = F (LEQ ONE TWO) F
- =\x.\y.y (LEQ ONE TWO) F
- = F

LENGTH NIL

- Lambda> Y
- \g.(\x.g (x x)) \x.g (x x)
- $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$//Y的定义带入F
- Lambda> CONS = \x.\y.\f. f x y
- Lambda> CAR = \p.p TRUE
- Lambda> CDR = \p.p FALSE
- Lambda> NIL = \x. TRUE
- Lambda> NULL = \p.p (\x.\y.FALSE)
- Lambda> LENGTH = Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x))) ZERO

- Lambda> LENGTH NIL
- \f.\x.x

LENGTH

- //YF=F(YF)
- LENGTH NIL
- = Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x))) ZERO NIL
- = (\g.\c.\x. NULL x c (g (SUCC c) (CDR x))) (Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x)))) ZERO NIL
- = ZERO
- LENGTH (CONS a NIL)
- = Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x))) ZERO (CONS a NIL)
- = (\g.\c.\x. NULL x c (g (SUCC c) (CDR x))) (Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x)))) ZERO (CONS a NIL)
- = NULL (CONS a NIL) ZERO ((Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x)))) ONE NIL)
- = Y (\g.\c.\x. NULL x c (g (SUCC c) (CDR x)))) ONE NIL
- = ONE

过程建模

累 + * cons

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

- (accumulate + 0 (list 1 2 3 4 5))
- 15
- (accumulate * 1 (list 1 2 3 4 5))
- 120
- (accumulate cons nil (list 1 2 3 4 5))
- (1 2 3 4 5)

多项式求值

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

$$(\cdots(a_n x + a_{n-1})x + \cdots + a_1)x + a_0$$

- 想想 <??>里面填什么

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms) <??>)
              0
              coefficient-sequence))
```

多项式求值

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

$$(\cdots (a_n x + a_{n-1}) x + \cdots + a_1) x + a_0$$

```
(define (horner-eval x coefficient-sequence)
  (accumulate (lambda (this-coeff higher-terms)
                (+ this-coeff (* x higher-terms)))
              0
              coefficient-sequence))
```

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

- (assert-equal 10 (horner-eval 0 ‘(10)))= a0 =10
- (assert-equal 13 (horner-eval 10 '(3 1)))= a0+ 10*a1 =13
- (assert-equal 79 (horner-eval 2 '(1 3 0 5 0 1)))= a0+2*a1+2*2*2*a3+2*2*2*2*2*a5=1+2*3+8*5+32*1=79

accumulate-n

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs)) nil
     (cons (accumulate   op init (map car seqs))
        (accumulate-n op init (map cdr seqs)))))
```

- (assert-equal '(22 26 30) (accumulate-n + 0 '((1 2 3)
- (4 5 6)
- (7 8 9)
- (10 11 12))))
- =(cons (accumulate + 0 (1 4 7 10)) (accumulate-n + 0 ‘((2 3) (5 6) (8 9)(11 12)))

数据建模

矩阵

- ;;   +-   -+
- ;; | 1 2 3 4 |
- ;; | 4 5 6 6 |
- ;; | 6 7 8 9 |
- ;;   +-   -+
- ;;
- ;; is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))'.

Map

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
            (map proc (cdr items)))))
```

Map

- Scheme standardly provides a map procedure that is more general than the one described here. This more general map takes a procedure of n arguments, together with n lists, and applies the procedure to all the first elements of the lists, all the second elements of the lists, and so on, returning a list of the results. For example:

- (map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
- (741 852 963)

- (map (lambda (x y) (+ x (* 2 y)))
-     (list 1 2 3)
-     (list 4 5 6))
- (9 12 15)

dot-product

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

apply

- apply will take at least two arguments, the first of them being a procedure and the last a list. It will call the procedure with the following arguments, including those inside the list:
  - (apply + '(2 3 4))
  - ; => 9
  - This is the same as (+ 2 3 4)
  - (apply display '("Hello, world!"))
  - ; does not return a value, but prints "Hello, world!"

Function list

- There are two constructors for lists. The function list takes any number of arguments and returns a list with those arguments as its elements.

- > (list (+ 2 3) 'squash (= 2 2) (list 4 5) remainder 'zucchini)
- (5 SQUASH #T (4 5) #<PROCEDURE> ZUCCHINI)

- The other constructor, cons, is used when you already have a list and you want to add one new element. Cons takes two arguments, an element and a list (in that order), and returns a new list whose car is the first argument and whose cdr is the second.

- > (cons 'for '(no one))
- (FOR NO ONE)

- > (cons 'julia '())
- (JULIA)

矩阵运算

- `# (define (f . xs) ...)` which will allow you to call f with an arbitrary number of arguments (e.g. `(f 1 2 3 4 5)`) and `xs` will be a list containing those arguments.
- (define (identity-n . x) x)
- (define (matrix-*-vector m v)
-   (map (lambda (row) (dot-product v row)) m))
- (define (transpose mat)
-   (apply map (cons identity-n mat)))
- (define (transpose-slow mat)
-   (accumulate-n cons '() mat))
- (define (matrix-*-matrix m n)
-   (let ((cols (transpose n)))
-     (map (lambda (row) (matrix-*-vector cols row)) m)))

transpose mat

```
(define (transpose mat)
  (apply map (cons identity-n mat)))

//(apply map (cons list A)) is the same as (apply map list A)

    (apply map  list '((1 2 3)  (10 20 30)) )

= (apply map (cons list '((1 2 3)  (10 20 30))))

= (apply map (list list  '(1 2 3) '(10 20 30) ))

= ( map list  '(1 2 3) '(10 20 30)  )

= '((1 10) (2 20) (3 30))
```

结果

- (assert-equal 7 (dot-product '(1 2 3) '(-1 1 2)))
- ;; [2 -1 1]
- ;; [0 -2 1] * [1 2 3] = [3 -1 -3]
- ;; [1 -2 0]
- (define v '(1 2 3))
- (define m '((2 -1 1) (0 -2 1) (1 -2 0)))
- (assert-equal '(3 -1 -3) (matrix-*-vector m v))

- (assert-many (lambda (f)
-                 (assert-equal '((2 0 1) (-1 -2 -2) (1 1 0)) (f m)))
-               transpose-slow
-               transpose)

- (assert-equal '((19 22) (43 50)) (matrix-*-matrix '((1 2) (3 4))
-   '((5 6) (7 8))))

高阶函数

计算

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

- 还记得

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

计算

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

```
(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)
- .249999875000001

Scheme十诫

第一诫 递归

```scheme
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? (car lat) a)
                (member? a (cdr lat)))))))
```

第二诫

使用cons来构建列表

(rember a lat) 构建一个剔除了首个a的lat

```scheme
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) a) (cdr lat))
      (else (cons (car lat)
                  (rember a (cdr lat)))))))
```

第三诫

构建一个列表的时候，描述第一个典型的元素，之后 cons 该元素到一般性递归上

（first l）l中每个元素的第一个元素构成一个新的list

```scheme
(define firsts
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      (else (cons (car (car l))
                  (firsts (cdr l)))))))
```

第四诫

在递归时总是改变至少一个参数。该参数必须向着不断接近结束条件而改变。改变的参数必须在结束条件中得以测试：当使用 cdr 时，用 null? 测试是否结束。

(multisubst new old lat) 多重替换

```scheme
(define multisubst
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) old)
        (cons new
              (multisubst new old (cdr lat))))
      (else (cons (car lat)
                  (multisubst new old (cdr lat)))))))
```

第五诫

当用 plus 构建一个值时，总是使用 0 作为结束代码行的值，因为加上 0 不会改变加法的值。

当用 × 构建一个值时，总是使用 1 作为结束代码行的值，因为乘以 1 不会改变乘法的值。

当用 cons 构建一个值时，总是考虑把 () 作为结束代码行的值。

(tup+ tup1 tup2) list对应元素相加

```scheme
(define tup+
  (lambda (tup1 tup2)
    (cond
      ((and (null? tup1) (null? tup2)) (quote ()))
      (else (cons (plus (car tup1) (car tup2))
                  (tup+ (cdr tup1) (cdr tup2)))))))
```

(rember* a l)
移除列表 l 中的全部原子 a，以此类推。

```
(define rember*
  (lambda (a l)
    (cond
      ((null? l) (quote ()))
      ((atom?  (car l) )
        (cond
          ((eq? (car l) a) (rember* a (cdr l)))
          (else (cons (car l)
                      (rember* a (cdr l))))))
      (else (cons (rember* a (car l))
                  (rember* a (cdr l)))))))
```

第六诫

简化工作只在功能正确之后开展。

第七诫
对具有相同性质的 subparts 进行递归调用：

列表的子列表

算术表达式的子表达式

(value nexp)返回可计数算术表达式（中缀）的一般性值。

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      ((eq? (car (car nexp)) (quote +))
        (plus (value (car nexp))
              (value (car (cdr (cdr nexp))))))
      ((eq? (car (car nexp)) (quote ×))
        (× (value (car nexp))
           (value (car (cdr (cdr nexp))))))
      (else
        (↑ (value (car nexp))
           (value (car (cdr (cdr nexp))))))))))
```

第八诫
使用辅助函数来抽象表达方式

如果 0 表示为 ()，1 是 (())，2 是 (() ())，以此类推。

edd1加一

```
(define edd1
   (Lambda (n)
      (Cons (quote()) n)))
```

(a-pair? x)
定义：判断 S-表达式 x 是不是一个 pair
（仅有两个 S-表达式组成的列表）。

```scheme
(define a-pair
  (lambda (x)
    (cond
      ((atom? x) #f)
      ((null? x) #f)
      ((null? (cdr x)) #f)
      ((null? (cdr (cdr x))) #t)
      (else #f))))
```

第九诫
用函数来抽象通用模式

重写value

```
(define atom-to-function
  (lambda (x)
    (cond
      ((eq? x (quote +)) plus)
      ((eq? x (quote ×)) ×)
      (else ↑))))
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      (else
        ((atom-to-function (operator nexp))
          (value (1st-sub-exp nexp))
          (value (2nd-sub-exp nexp)))))))
```

第十诫
构建函数，一次搜集多个值

(evens-only* l)

定义：去除 l 中所有的奇数。

```
(define even?
  (lambda (n)
    (= (× (÷ n 2) 2) n)))

(define evens-only∗
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
        (cond
          ((even? (car l))
            (cons (car l)
                  (evens-only∗ (cdr l))))
          (else (evens-only∗ (cdr l)))))
      (else (cons (evens-only∗ (car l))
                  (evens-only∗ (cdr l)))))))
```

evens-only*&co 函数，其从一个列表中移除所有奇数项，以构建一个偶数项的嵌套列表，同时求出该列表所有偶数项的乘积和奇数项的和。

```
(define evens-only*&co
  (lambda (l col)
    (cond
      ((null? l) (col (quote ()) 1 0))
      ((atom? (car l))
        (cond
          ((even? (car l))
            (evens-only*&co (cdr l)
              (lambda (newl p s)
                (col (cons (car l) newl) (× (car l) p) s))))
          (else (evens-only*&co (cdr l)
            (lambda (newl p s)
              (col newl p (plus (car l) s)))))))
      (else (evens-only*&co (car l)
        (lambda (al ap as)
          (evens-only*&co (cdr l)
            (lambda (dl dp ds)
              (col (cons al dl)
                   (× ap dp)
                   (plus as ds)))))
)))))
```