今天的内容

- List的介绍（继续）
- For包含
- 可变状态对象

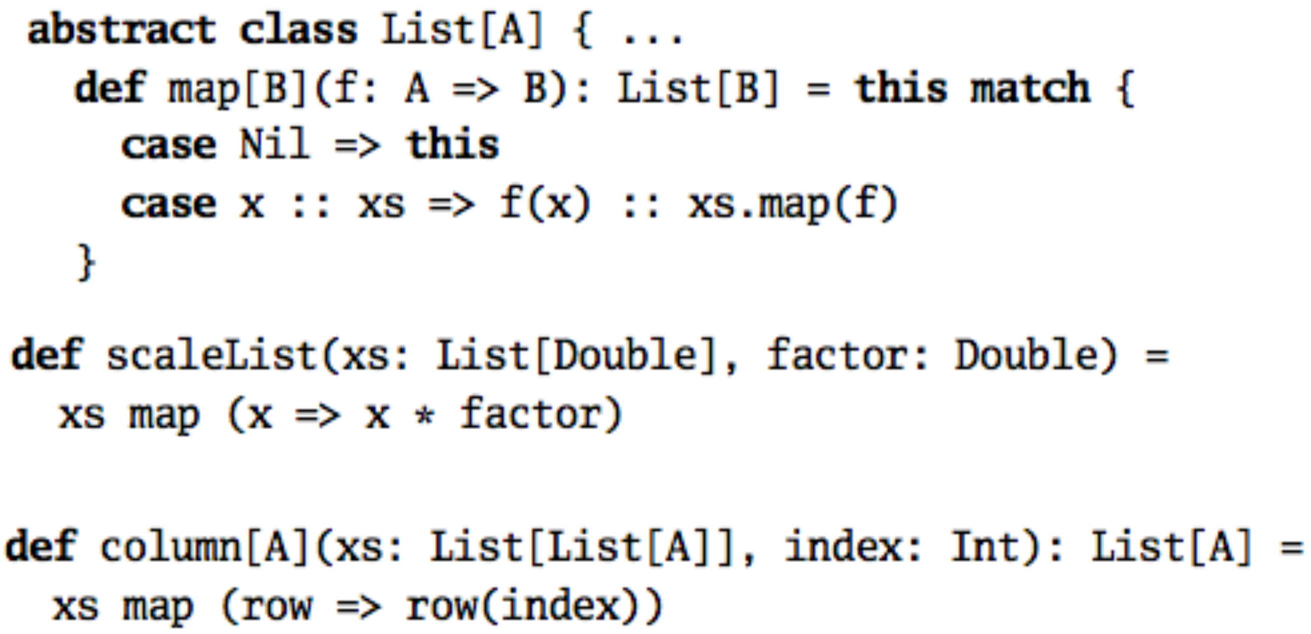
List 介绍继续

List 的高阶方法

List上的计算模式总结

- 计算模式
  - 对每个元素进行转换
  - 选出满足某个条件的所有元素
  - 对元素进行某种方式上的组合
- 通过高阶函数来实现以上模式
- List 的方法

Mapping (映射)



```
abstract class List[A] { ...
  def map[B](f: A => B): List[B] = this match {
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)
  }
```

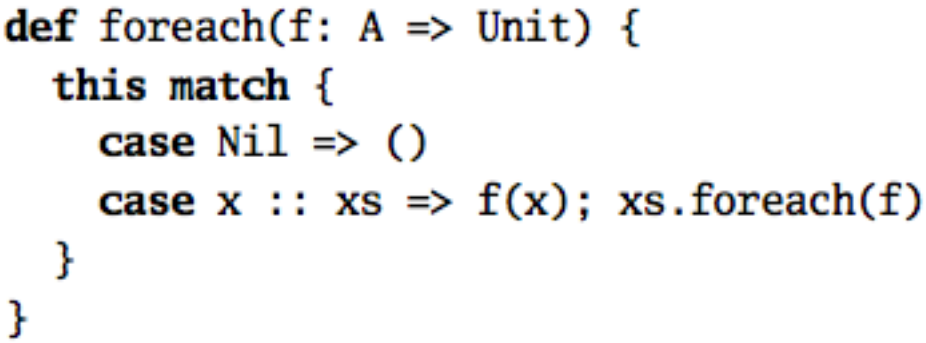
```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)
```

```
def column[A](xs: List[List[A]], index: Int): List[A] =
  xs map (row => row(index))
```

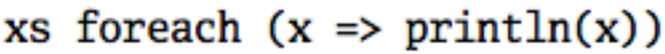
- 变换每个元素

For each 方法

- 对每个元素应用一个函数，但不返回一个列表结果
- 为了副作用(side effect)而设
  - In computer science, a function or expression is said to have a **side effect** if, in addition to returning a value, it also modifies some state or has an *observable* interaction with calling functions or the outside world. For example, a function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other side-effecting functions.



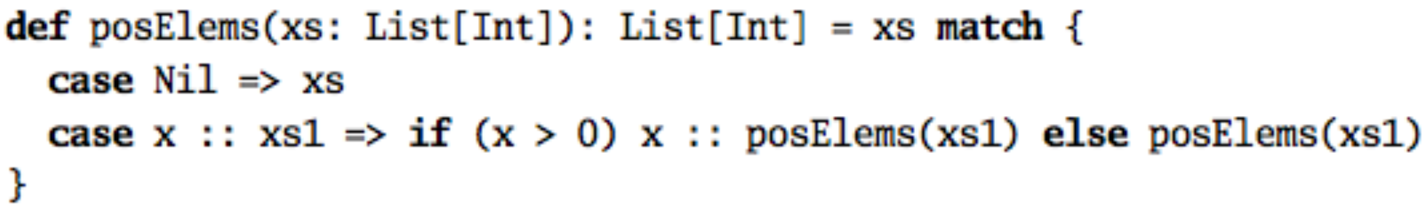
```
def foreach(f: A => Unit) {
  this match {
    case Nil => ()
    case x :: xs => f(x); xs.foreach(f)
  }
}
```



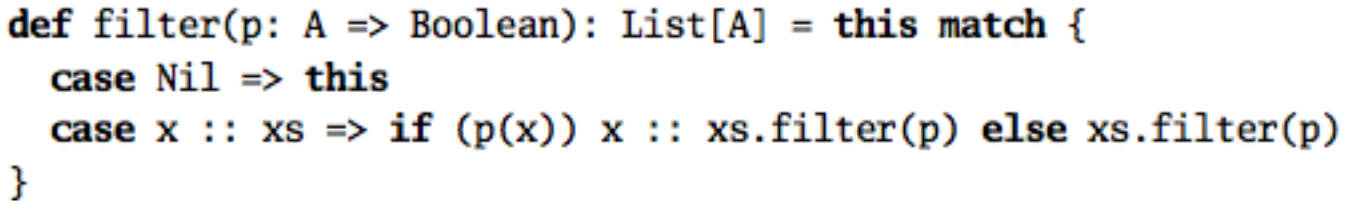
```
xs foreach (x => println(x))
```

Filtering（过滤列表）

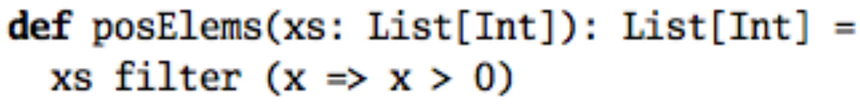
- 根据一个原则来选择元素



```
def posElems(xs: List[Int]): List[Int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```



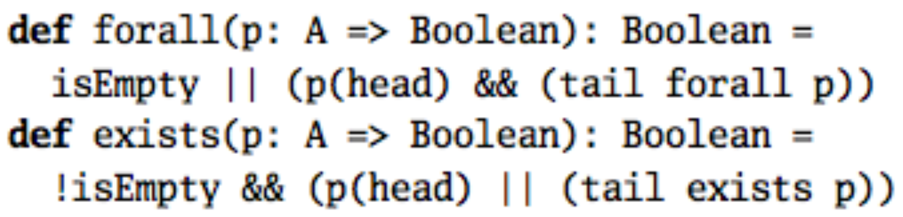
```
def filter(p: A => Boolean): List[A] = this match {
  case Nil => this
  case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
}
```



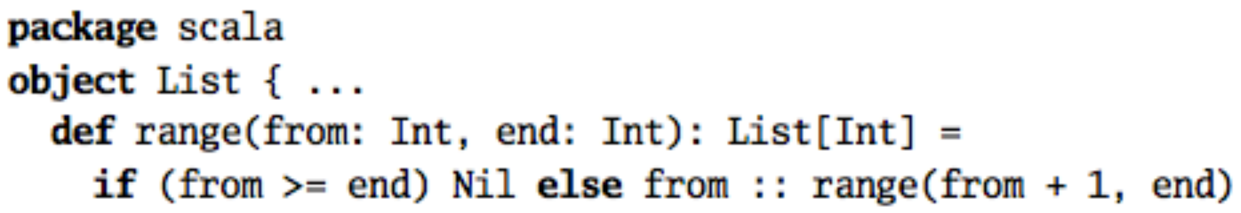
```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

Forall , exists

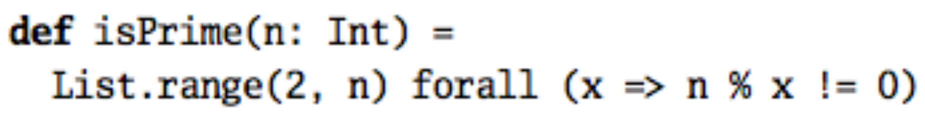
- Forall : all elements satisfy a condition
- Exists: exists an element that satisfies a condition



```
def forall(p: A => Boolean): Boolean =
  isEmpty || (p(head) && (tail forall p))
def exists(p: A => Boolean): Boolean =
  !isEmpty && (p(head) || (tail exists p))
```



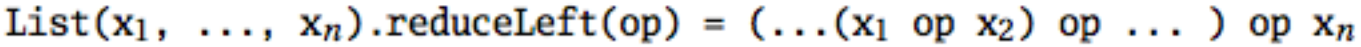
```
package scala
object List { ...
  def range(from: Int, end: Int): List[Int] =
    if (from >= end) Nil else from :: range(from + 1, end)
```

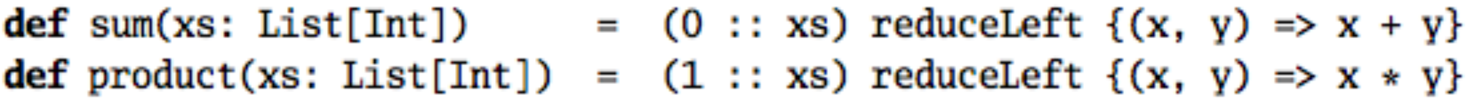


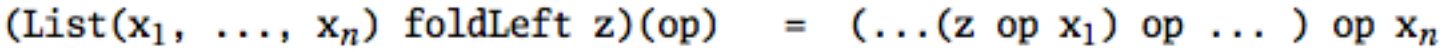
```
def isPrime(n: Int) =
  List.range(2, n) forall (x => n % x != 0)
```

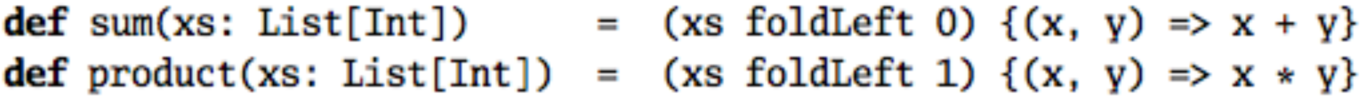
质数定义

折叠和减少列表(folding and reducing)


List(x_1, ..., x_n).reduceLeft(op) = (...(x_1 op x_2) op ...) op x_n

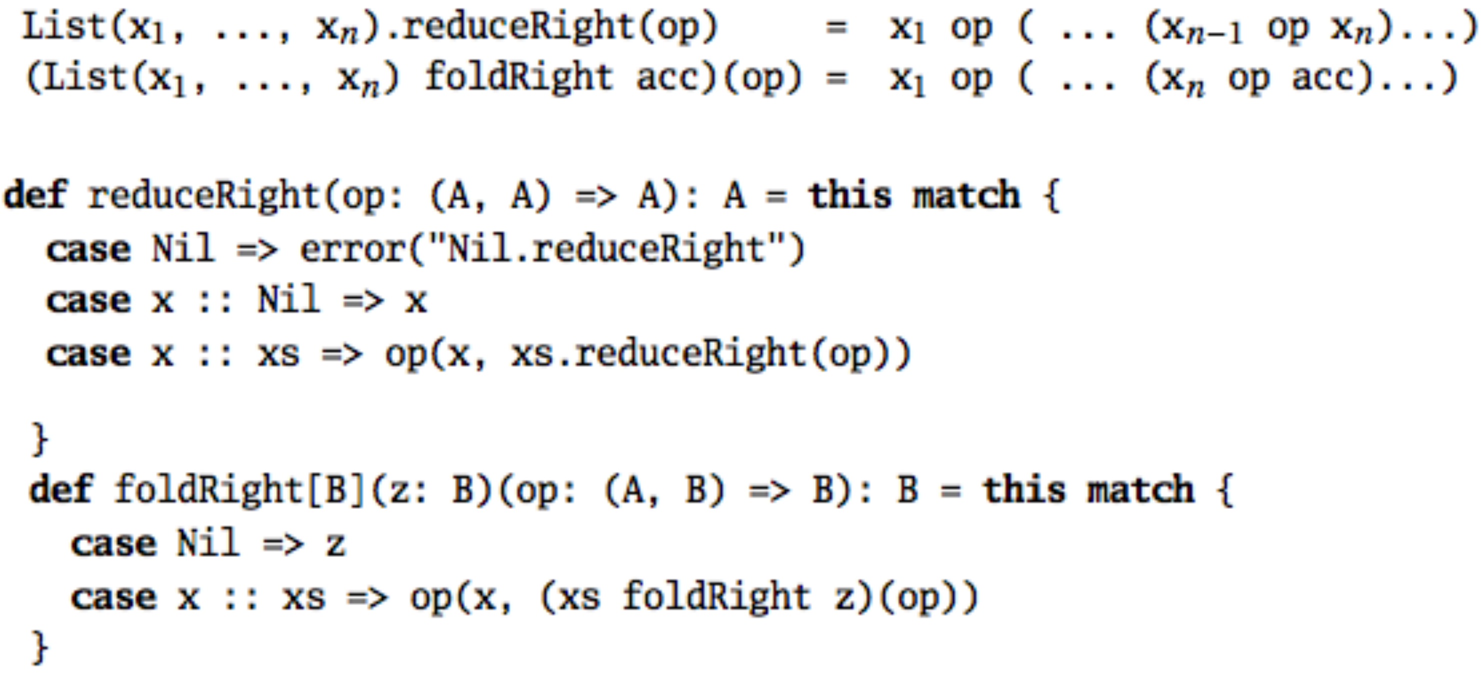

```
def sum(xs: List[Int])      =  (0 :: xs) reduceLeft {(x, y) => x + y}
def product(xs: List[Int])  =  (1 :: xs) reduceLeft {(x, y) => x * y}
```


(List(x_1, ..., x_n) foldLeft z)(op) = (...(z op x_1) op ...) op x_n


```
def sum(xs: List[Int])      =  (xs foldLeft 0) {(x, y) => x + y}
def product(xs: List[Int])  =  (xs foldLeft 1) {(x, y) => x * y}
```

- Combine elements of a list with some operator.

FoldRight , ReduceRight



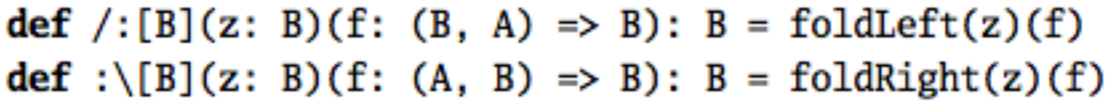
List(x_1, ..., x_n).reduceRight(op) = x_1 op (... (x_{n-1} op x_n)...)

(List(x_1, ..., x_n) foldRight acc)(op) = x_1 op (... (x_n op acc)...)

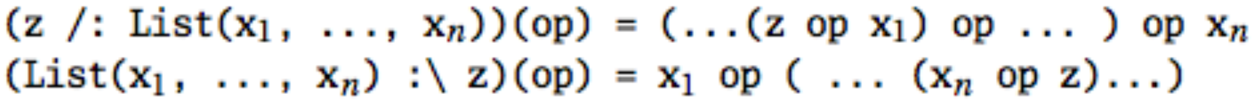
```
def reduceRight(op: (A, A) => A): A = this match {
  case Nil => error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))

 }
 def foldRight[B](z: B)(op: (A, B) => B): B = this match {
   case Nil => z
   case x :: xs => op(x, (xs foldRight z)(op))
 }
```

- Produce right-leaning trees.

Abbreviations for foldLeft and foldRight



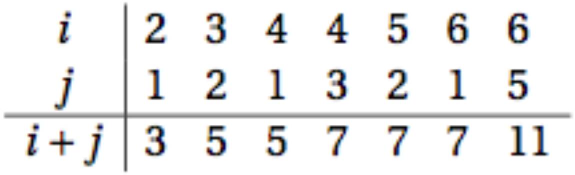
```
def /:[B](z: B)(f: (B, A) => B): B = foldLeft(z)(f)
def :\[B](z: B)(f: (A, B) => B): B = foldRight(z)(f)
```



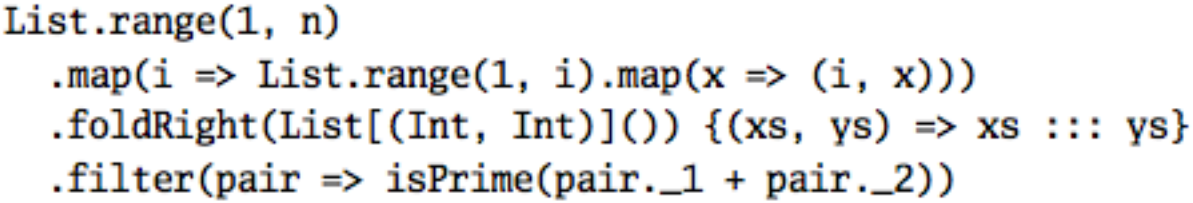
(z /: List(x_1, ..., x_n))(op) = (...(z op x_1) op ...) op x_n
(List(x_1, ..., x_n) :\ z)(op) = x_1 op (... (x_n op z)...)

Nested Mappings

- 高阶函数可以替代嵌套循环
- Find all pairs of positive integers I and j, where 1<=j<i<n such that i+j is prime.



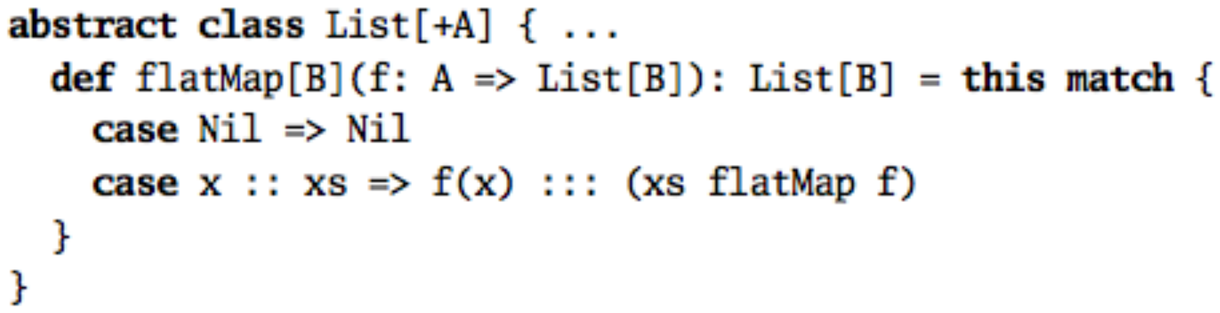
i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11



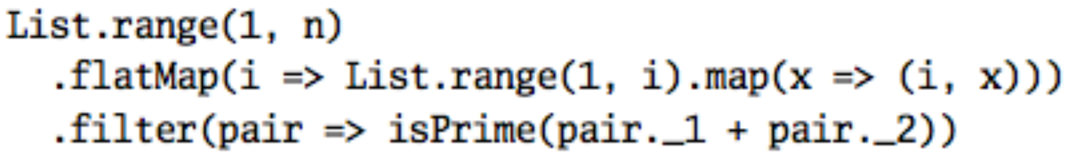
```
List.range(1, n)
  .map(i => List.range(1, i).map(x => (i, x)))
  .foldRight(List[(Int, Int)]()) {(xs, ys) => xs ::: ys}
  .filter(pair => isPrime(pair._1 + pair._2))
```

Flattening Maps

- flatMap
  - Combination of mapping and then concatenating sublists



```
abstract class List[+A] { ...
  def flatMap[B](f: A => List[B]): List[B] = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: (xs flatMap f)
  }
}
```



```
List.range(1, n)
  .flatMap(i => List.range(1, i).map(x => (i, x)))
  .filter(pair => isPrime(pair._1 + pair._2))
```

Pairs whose sum is prime

List 总结

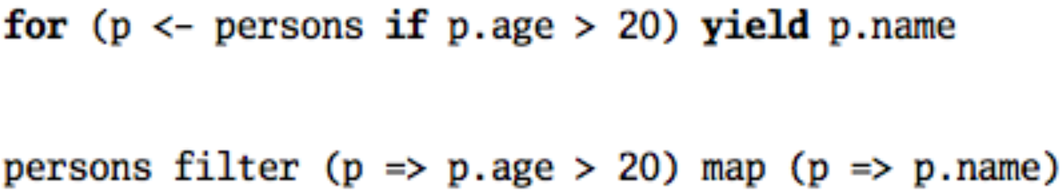
- 基本数据结构
- Immutable, common data type in functional programming
- 相当于array in imperative languages
- 访问模式不同，递归方式访问（借助模式匹配）
- 高阶函数抽象常用的计算模式

For-comprehensions(for 语句包含)

为什么用for comprehension

- Map, flatMap, Filter 抽象性会使得程序难以理解
- 增强可读性
- Build a bridge between set comprehensions in mathematics and for-loops in imperative language
- Resembles query notation of relational databases

表现行式

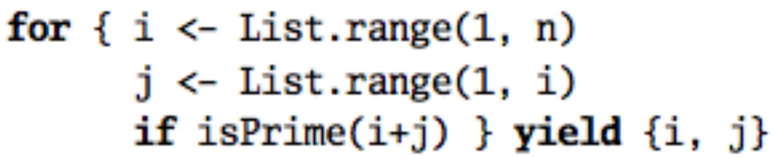


```
for (p <- persons if p.age > 20) yield p.name

persons filter (p => p.age > 20) map (p => p.name)
```

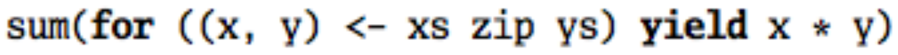
- For (s) yield e
  - S 是一系列generators, definitions, filters
  - Generator: val x <- e, e is a list-valued expression; 值绑定
  - Definition: val x = e; 引入一个别名
  - Filter：Boolean-typed expression; 过滤值

举例

- 找到所有质数整数对（i，j），1<=j<i<n， such that i+j is prime



```
for { i <- List.range(1, n)
      j <- List.range(1, i)
      if isPrime(i+j) } yield {i, j}
```

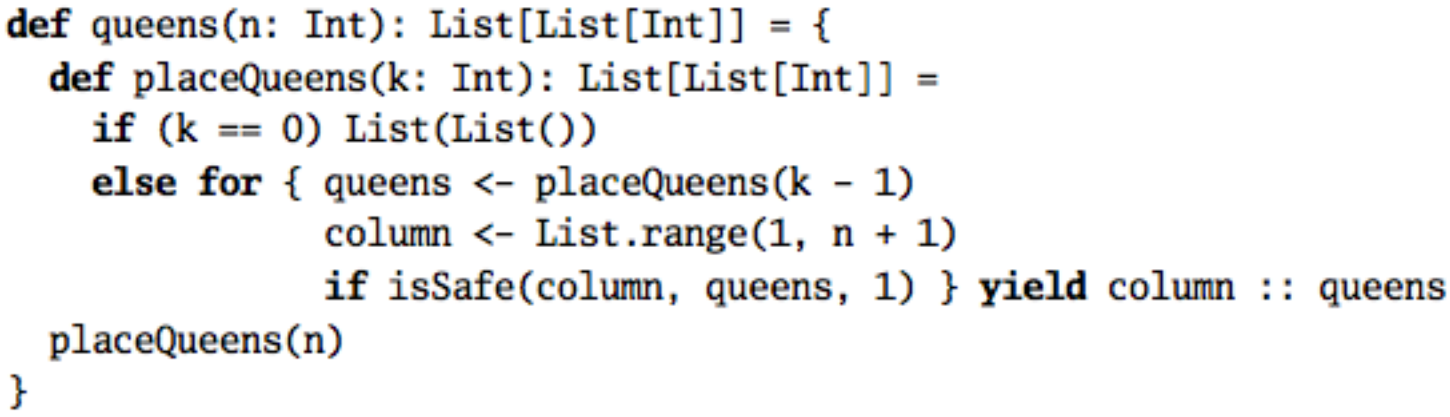


```
sum(for ((x, y) <- xs zip ys) yield x * y)
```

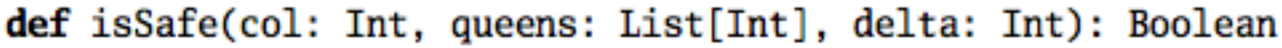
- Compute scalar product of two vectors xs and ys

求解组合问题：N-皇后问题

- Place a queen in each row without attacking other queens
- Assume already generated all solutions of placing k-1 queens



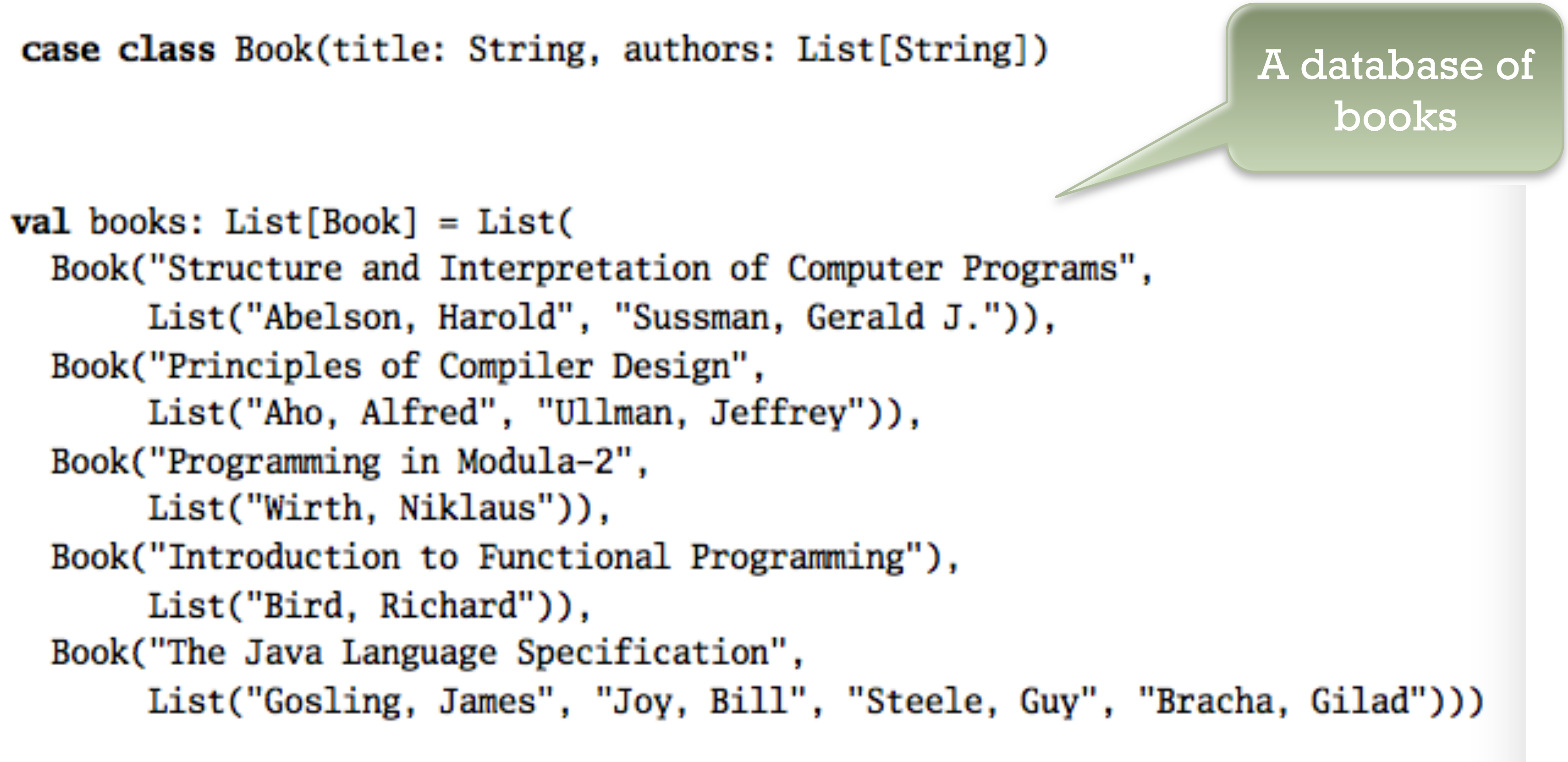
```
def queens(n: Int): List[List[Int]] = {
  def placeQueens(k: Int): List[List[Int]] =
    if (k == 0) List(List())
    else for { queens <- placeQueens(k - 1)
               column <- List.range(1, n + 1)
               if isSafe(column, queens, 1) } yield column :: queens
  placeQueens(n)
}
```



```
def isSafe(col: Int, queens: List[Int], delta: Int): Boolean
```

查询搜索

- Equivalent to common database query languages

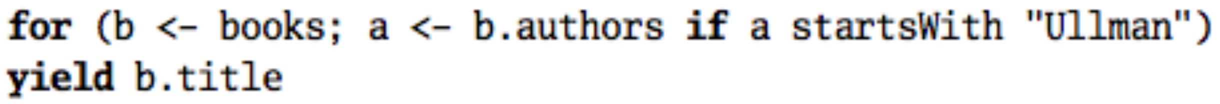


```
case class Book(title: String, authors: List[String])


val books: List[Book] = List(
  Book("Structure and Interpretation of Computer Programs",
       List("Abelson, Harold", "Sussman, Gerald J.")),
  Book("Principles of Compiler Design",
       List("Aho, Alfred", "Ullman, Jeffrey")),
  Book("Programming in Modula-2",
       List("Wirth, Niklaus")),
  Book("Introduction to Functional Programming"),
       List("Bird, Richard")),
  Book("The Java Language Specification",
       List("Gosling, James", "Joy, Bill", "Steele, Guy", "Bracha, Gilad")))
```

A database of books

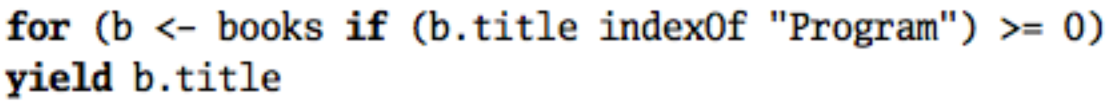
查询搜索

- To find titles of all books whose author’s last name is “Ullman”

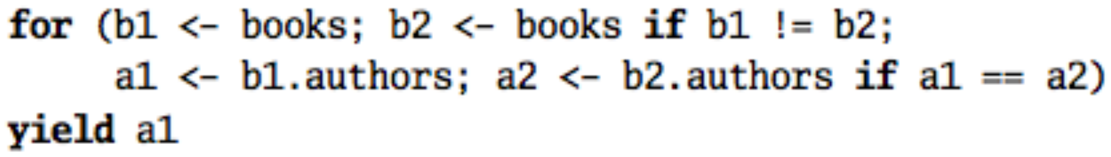


```
for (b <- books; a <- b.authors if a startsWith "Ullman")
yield b.title
```

Titles have string “Program”



```
for (b <- books if (b.title indexOf "Program") >= 0)
yield b.title
```

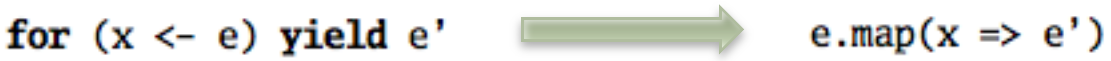


```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

Authors who have written at least two books in the database.

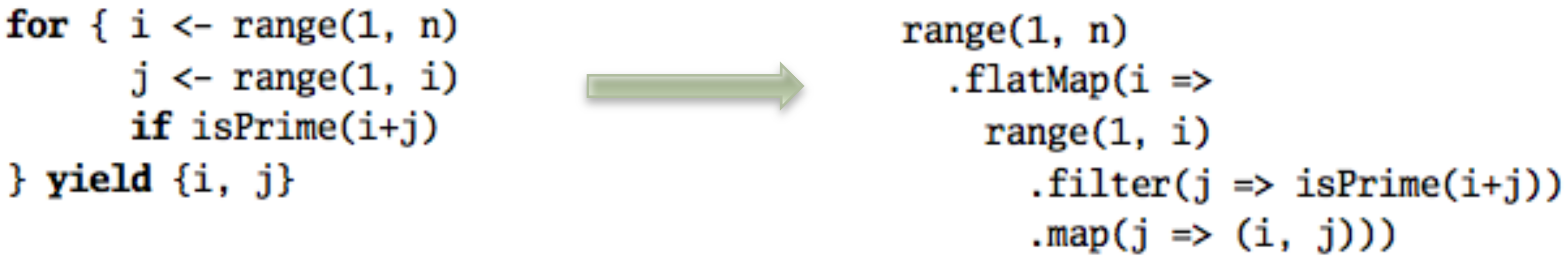
转换翻译

- 可以用高阶函数map, flatMap和filter来实现



```
for (x <- e) yield e'
```

```
e.map(x => e')
```

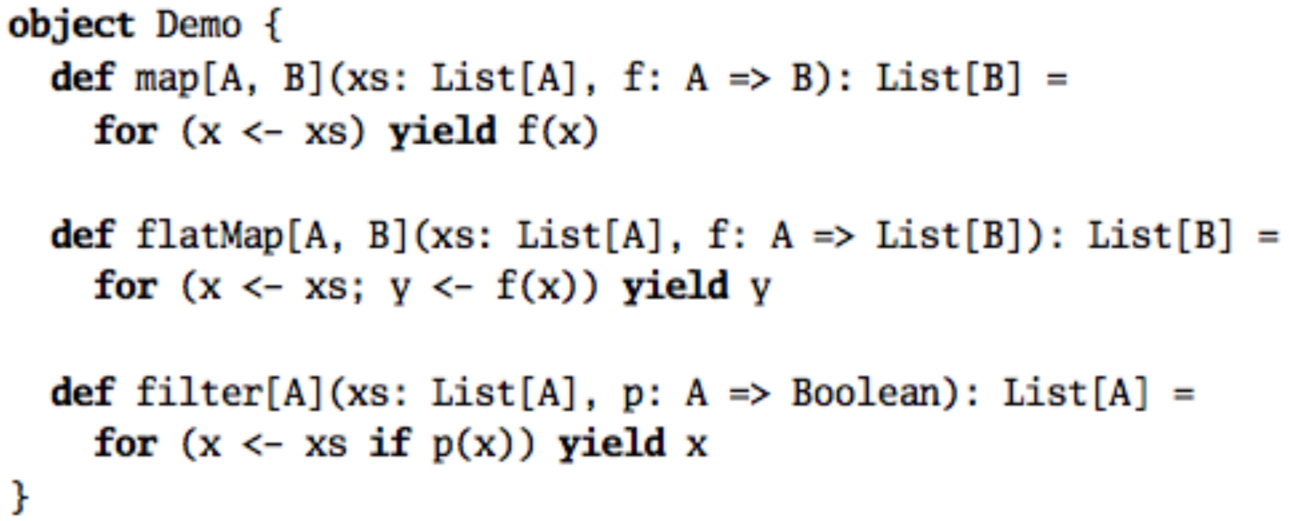


```
for { i <- range(1, n)
      j <- range(1, i)
      if isPrime(i+j)
} yield {i, j}
```

```
range(1, n)
  .flatMap(i =>
    range(1, i)
      .filter(j => isPrime(i+j))
      .map(j => (i, j)))
```

转换翻译

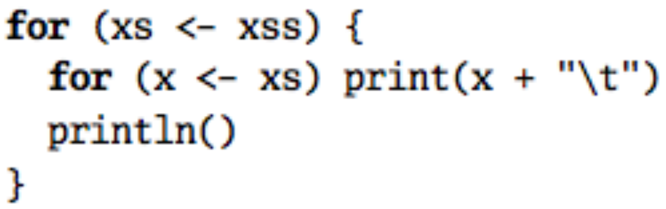
Map,flatmap, filter也可用for-comprehension来实现



```
object Demo {
  def map[A, B](xs: List[A], f: A => B): List[B] =
    for (x <- xs) yield f(x)

  def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
    for (x <- xs; y <- f(x)) yield y

  def filter[A](xs: List[A], p: A => Boolean): List[A] =
    for (x <- xs if p(x)) yield x
}
```

For 循环

- A variant of the for-comprehension syntax
- For (s) e ; key yield is missing



```
for (xs <- xss) {
  for (x <- xs) print(x + "\t")
  println()
}
```

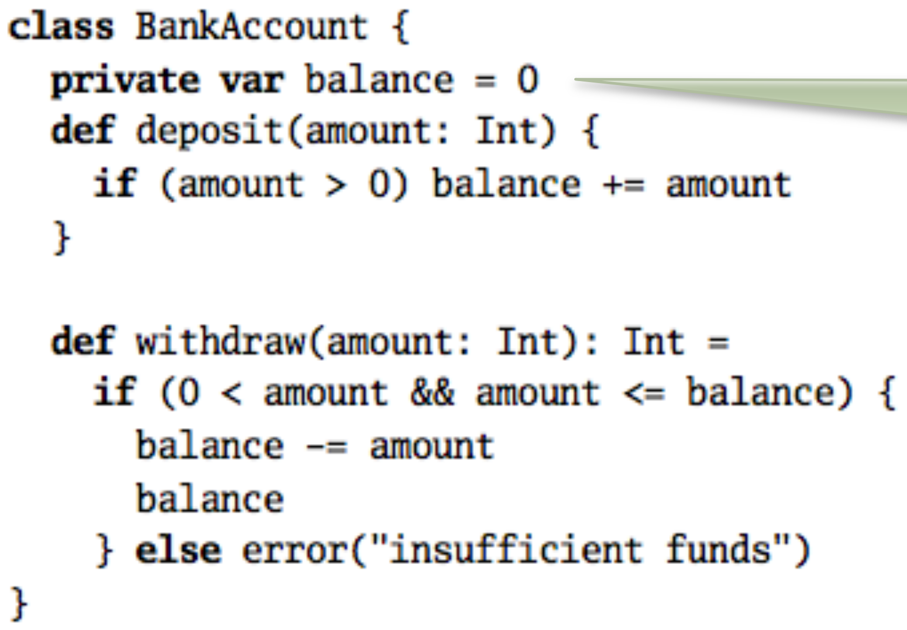
打印显示一个矩阵（列表的列表）的元素

可改变的状态(Mutable State)

Stateful Objects(有状态的对象)

- View the world as a set of objects, some have state that changes over time.
- A bank account object has state, “can I withdraw ￥100?” depends on different time
- 变量定义用 var
  - var count = 600
- Real-world objects with states represented by objects that have variables as members

Bank account example

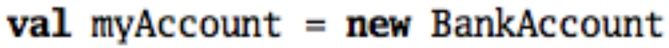


```
class BankAccount {
  private var balance = 0
  def deposit(amount: Int) {
    if (amount > 0) balance += amount
  }

  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance -= amount
      balance
    } else error("insufficient funds")
}
```

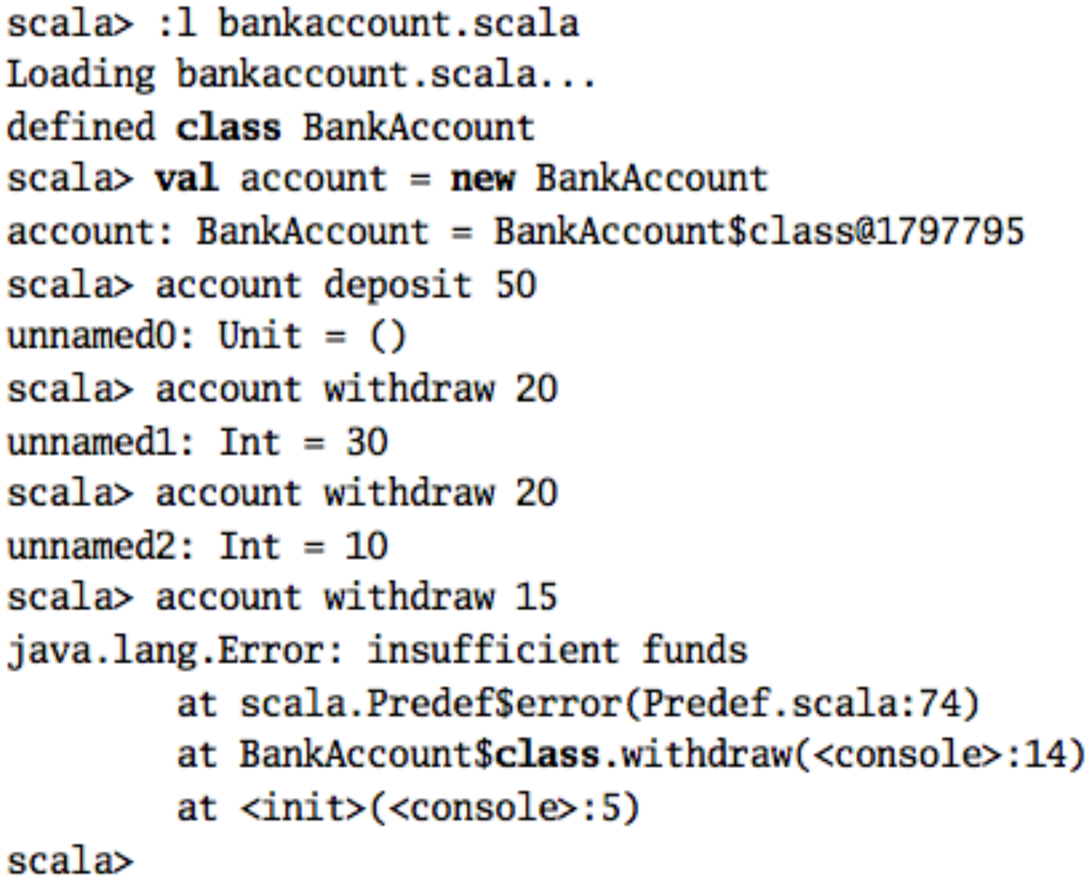
变量代表可能会改变的状态

- 私有变量

Bank account example



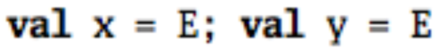
```
val myAccount = new BankAccount
```



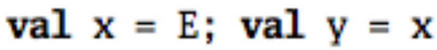
```
scala> :l bankaccount.scala
Loading bankaccount.scala...
defined class BankAccount
scala> val account = new BankAccount
account: BankAccount = BankAccount$class@1797795
scala> account deposit 50
unnamed0: Unit = ()
scala> account withdraw 20
unnamed1: Int = 30
scala> account withdraw 20
unnamed2: Int = 10
scala> account withdraw 15
java.lang.Error: insufficient funds
        at scala.Predef$error(Predef.scala:74)
        at BankAccount$class.withdraw(<console>:14)
        at <init>(<console>:5)
scala>
```

- Bank accounts are stateful objects

有状态对象的相同（sameness）比较

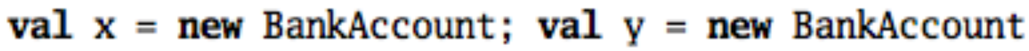


```
val x = E; val y = E
```



```
val x = E; val y = x
```

X， y 相同

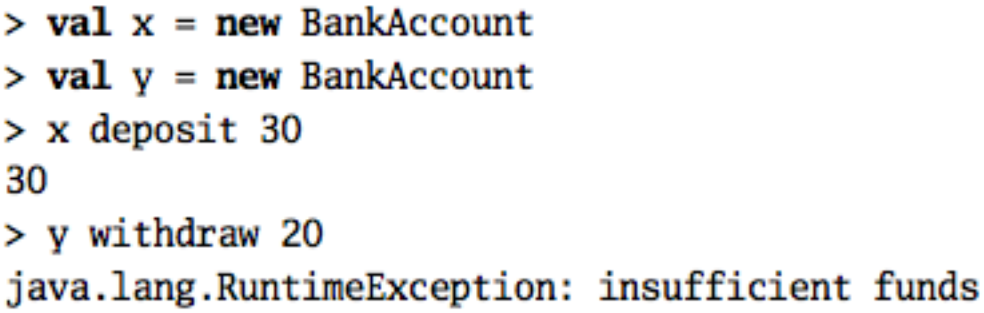


```
val x = new BankAccount; val y = new BankAccount
```

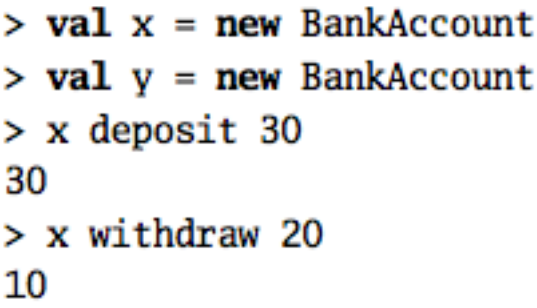
这里的x和y相同吗？

- E: arbitrary expression
- 操作结果比较法（operational equivalence）

有状态对象的相同（sameness）比较

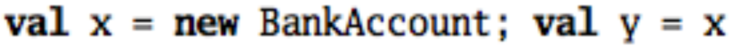


```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> y withdraw 20
java.lang.RuntimeException: insufficient funds
```



```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> x withdraw 20
10
```

- 操作的结果不同，说明**x**和**y**不相同
- 之前的替代计算模型在这里不能被使用

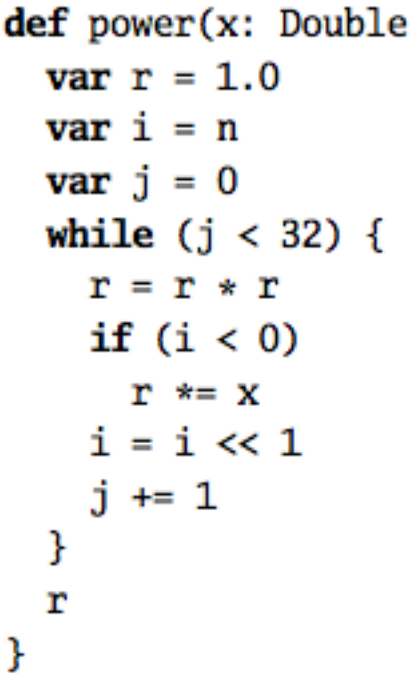


```
val x = new BankAccount; val y = x
```

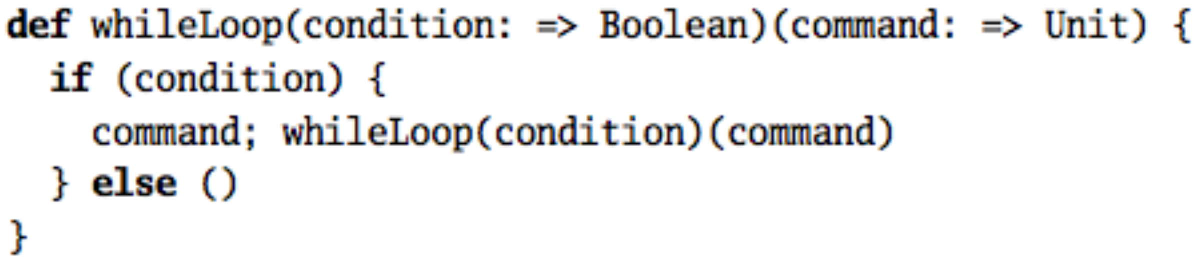
这样定义则相同。

Imperative control structures

- While, do-while, if (单个), return
- 可用函数来替代



```
def power(x: Double, n: Int): Double = {
  var r = 1.0
  var i = n
  var j = 0
  while (j < 32) {
    r = r * r
    if (i < 0)
      r *= x
    i = i << 1
    j += 1
  }
  r
}
```



```
def whileLoop(condition: => Boolean)(command: => Unit) {
  if (condition) {
    command; whileLoop(condition)(command)
  } else ()
}
```

- Passed by-name, evaluated repeatedly for each loop iteration;
- Tail recursive