第五章 函数式程序设计语言

- 过程式程序设计语言由于数据的名值分离，变量的时空特性导致程序难于查错、难于修改
- 命令式语言天生带来的三个问题只解决了一半
- 滥用goto已经完全解决
- 悬挂指针没有完全解决
- 函数副作用不可能消除

00解决了上述问题吗？

- 问题是程序状态的易变性(Mutability)和顺序性(Sequencing)
- Backus在图灵奖的一篇演说《程序设计能从冯·诺依曼风格下解放出来吗?》中极力宣扬发展与数学联系更密切的函数式程序设计语言

5.1 过程式语言存在的问题

（1）易变性难于数学模型

- 代数中的变量是未知的确定值，而程序设计语言的变量是对存储的抽象
- 根本解决：能不能不要程序意义的“变量”只保留数学意义的“变量”？
- 能不能消除函数的副作用?

例：有副作用的函数

```
int sf_fun(int x)
    static int z = 0;      //第一次装入赋初值
    return x + (z++);
sf_fun(3) = {3 |4 | 5 | 6 | 7 ...}
//随调用次数而异，不是数学意义的确定函数。
```

数学函数的特征：

- 映射表达式的求值顺序由递归表达式和条件表达式控制。
- 不依赖于任何外部值（无副作用），所以给定相同的自变量集合，总是映射值域集合中的同一个元素。

（2）顺序性更于难数学模型

- 顺序性影响计算结果，例如，急求值、正规求值、懒求值同一表达式就会有不同的结果。有副作用更甚，因而难于为程序建立统一的符号数学理论。
  - 应寻求与求值顺序无关的表达方式

变元求值策略/求值顺序的影响

ML：fun sqr(n : int) = n*n

若p=2，q=5有调用

sqr (p + q) = sqr (2 + 5) = 7 * 7 = 49

急求值：表达式先求值再入体。

正规求值：

(p+q) * (p+q) = (2+5)*(2+5) = 7 * 7 = 49

按λ演算，先置换原表达式，体中代入值计算

懒求值：

(p+q) * (p+q) = (2+5) * (2+5) = (2+5) * 7 = 49

只在界面置换原表达式，何时用该值何时计算，相同的只算一次

（2）顺序性更于难数学模型

- 理想的改变途径
  - 没有变量，就没有破坏性赋值，也不会有引起副作用的全局量和局部量之分。通过引用调用就没有意义。
  - 循环也没有意义，因为只有每次执行循环改变了控制变量的值，循环才能得到不同的结果。
  - 那么程序结构只剩下表达式、条件表达式、递归表达式。

- Church-Rosser性质：

表达式的完全求值，仅当它前后一致地按正规顺序求值，几种求值方式得的结果应一致。若一表达式能以几种不同的求值次序求值（包括混合使用几种求值方案），则所有这些求值次序得到的结果值应该是一样的。

急求值

- **急求值是严格求值（Strict），完全求值**
- **急求值对应值调用，最安全**

```
fun  cand (b1: bool, b2:bool)=
   if  b1  then  b2
       else  false
```

有函数调用cand（n>0，t/n>0.5）**若**n=0，t=0.8

若急求值，第二子表达式未结合即失败

满足Church-Rosser性质：
　表达式的完全求值，仅当它前后一致地按正规顺序求值，几种求值方式得的结果应一致。

正规求值

- **正规求值，对应名调用，支持递归**

```
if n>0  then  t/n >0.5  else  false
```

上述调用等效代入置换后再求值cand=false

若函数定义为F(P)， 函数应用为F(Exp)，则在函数(体)中的每次P出现均用Exp置换。这相当于前述名调用。显而易见，正规求值要作多次Exp计算。如果**Exp有副作用**，多次计算的Exp值不会是同一个值， 整个函数求值就难以预料了。

懒求值

- **懒求值可实现短路求值，也支持递归**
  - 懒(lazy)求值是正规求值的特例，它只到用到被结合的形参变量的值时才对实参变元表达式求值(第一次出现)，求值后将结果值束定于形参变量，以后形参变量再次出现就不用求值了。这是懒的第一个意思
  - 对于复杂的表达式如果子表达式求值对整个表达式求值没有影响就不再求它。这是懒的第二个意思。

懒求值

可以部分求值：

E = Exp1 and Exp2 and ...Expn

若Exp1求值为false，则E值已定为false。再如：

E = Exp1 or Exp2 or ...Expn

若Exp1为true，则E值为true不论其它表达式取何值。

这也叫短路(short circuit)求值，一般用作条件表达式，也叫短路条件。不仅效率高，而且能避免不必要的出错。

C、Ada，及近代函数式语言均采用懒求值

懒求值的讨论

- 核心：在函数调用中，同一个实际参数出现多次，该实参也只求值一次。
- 可以定义无限的数据结构。
  - Haskell：
    - Evens = [2, 4..]
    - Squares = [n * n | n<-[0..]]
- f(g(x))，假设g生成大量数据，每次处理少量，然后f必须处理这些数据，每次处理少量，f和g可隐式地同步执行
- 代价：语义复杂

5.2 可计算函数和 Λ演算

- 自然数的函数 F: N → N 是可计算函数，
  - 当且仅当存在着一个λ表达式 f，
  - 使得对于 N 中的每对 x, y 都有 F(x) = y
  - 当且仅当 f x == y，
  - 这里的 x 和 y 分别是对应于 x 和 y 的邱奇数。
- 这是定义可计算性的多种方式之一。

自然数集N是指满足以下条件的集合：
①N中有一个元素，记作1。
②N中每一个元素都能在 N 中找到一个元素作为它的后继者。
③1是0的后继者。
④0不是任何元素的后继者。
⑤不同元素有不同的后继者。
⑥归纳公理：N的任一子集M，如果1∈M，并且只要x在M中就能推出x的后继者也在M中，那么M=N。

λ演算是什么

- λ演算是符号的逻辑演算系统，它正好只有这三种机制，它就成为函数式程序设计语言的模型
- λ演算是一个符号、逻辑系统，其公式就是符号串并按逻辑规则操纵
- Church的理论证明，λ演算是个完备的系统， 可以表示任何计算函数，所以任何可用λ演算仿真实现的语言也是完备的。等价于图灵机。
- λ演算可以被称为最小的**通用**程序设计语言。它包括一条**变换规则**（变量替换）和一条函数定义方式。

- λ演算使函数概念形式化，是涉及变量、函数、函数组合规则的演算。
- λ演算基于最简单的定义函数的思想： 一为函数抽象λx.E，一为函数应用(λx.E)(a)。
- 一切变量、标识符、表达式都是函数或(复合)高阶函数。如λx.C(C为常量)是常函数。

5.2.1 术语和表示法

(1) λ 演算有两类符号：

- 小写单字符用以命名参数，也叫变量。
- →归约，＝ 等价
- 外加四个符号：（　　）　　.　　λ
- 大写单字符、特殊字符（如+、−、*、/）、大小写组成的标识符代替一个 λ 表达式。

(2) λ表达式/λ项

- 形如λy.F为λ函数表达式，以关键字λ开始，变量y为参数。
- 形如(FG)为λ应用表达式
  - xy也是一个合法的λ。项
- 为了清晰，λ表达式可以任加成对括号。
  - 左结合，f x y＝(f x) y

Λ演算公式举例

(3) 公式

- 若F，G是λ.项，则F→G，F=G是公式。

x	变量、公式、表达式。
(λx.((y)x))	函数， 体内嵌入应用。
(λz.(y(λz.x)))	函数， 体内嵌入应用，再次嵌入函数。
(λz.(z y))x)	应用表达式。
λx. λy. λz.(x λx.(u v)w)	复杂表达式

(4) 简略表示

- 缩写与变形表达　下例各表达均等效:

λa. λb. λc. λz.E = λabcz.E
= λ(abcz).E
= λ(a, b, c, z).E
=λa.(λb.(λc.(λz.E)))

(λa. E) (λb.E)(λc.E) (λz.E)

- 命名:以大写单字符或标识符命名其λ表达式

G = (λx.(y(yx)))
((λx.(y(yx)))(λx.(y(yx)))) = (G G) = H

由于λ演算中一切语义概念均用λ表达式表达。为了清晰采用命名替换使之更易读。

T = λx. λy. x　　//逻辑真值

F = λx. λy. y　　//逻辑假值

1 = λx. λy. x y　　//数1

2 = λx. λy. x(x y)　　//数2

　　//数0?

zerop = λn.n(λx.F)T　　//判零函数

注：zerop中的F、T可以用λ表达式展开

形式语法

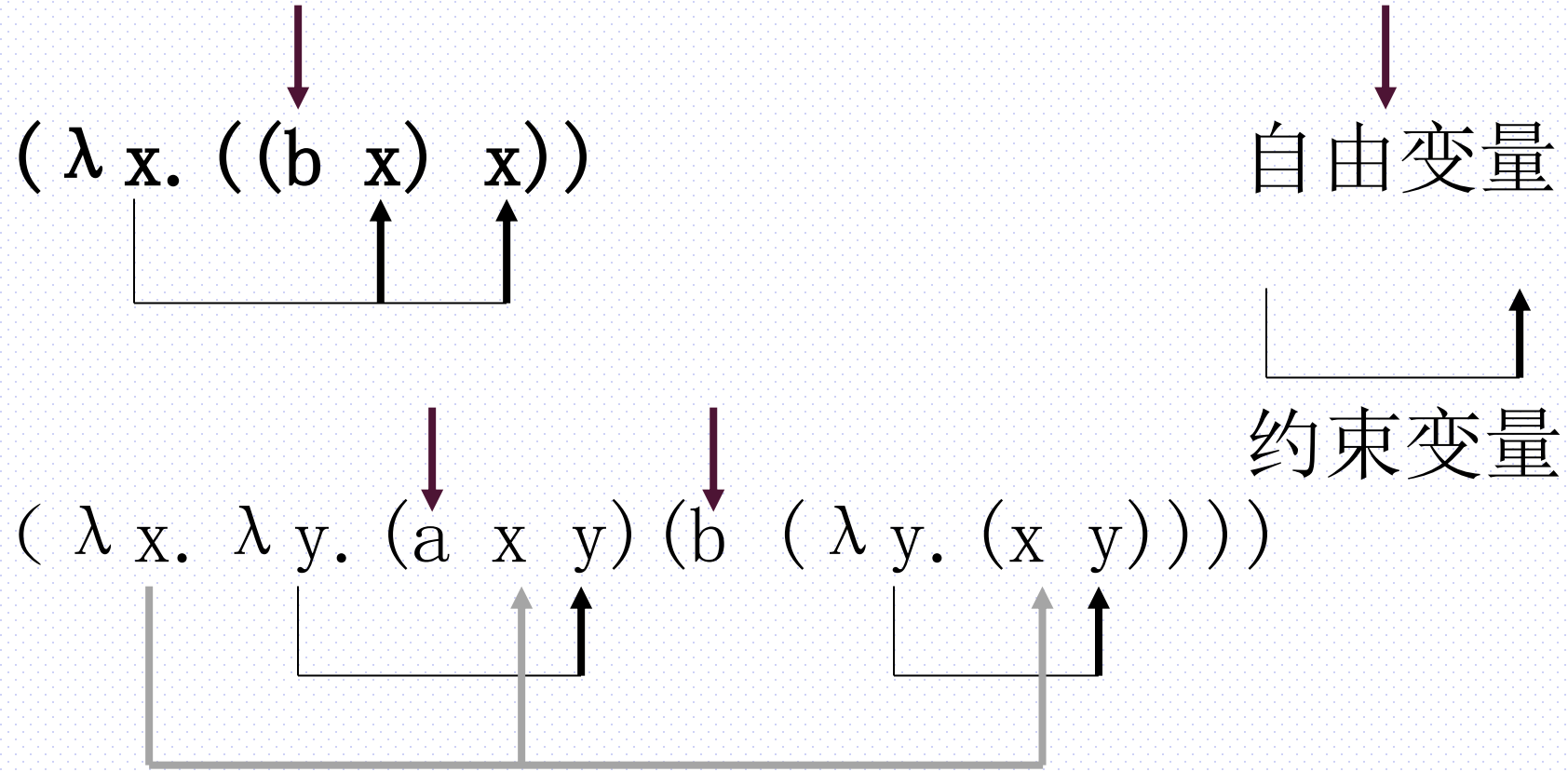
核心的λ演算没有类型，没有顺序控制等概念，程序和数据没有区分。语法极简单：

〈λ-表达式〉 :: = 〈变量〉
| λ〈变量〉.〈λ-表达式〉
| (〈λ-表达式〉〈λ-表达式〉)
| (〈λ-表达式〉)

〈变量〉 :: =〈字母〉

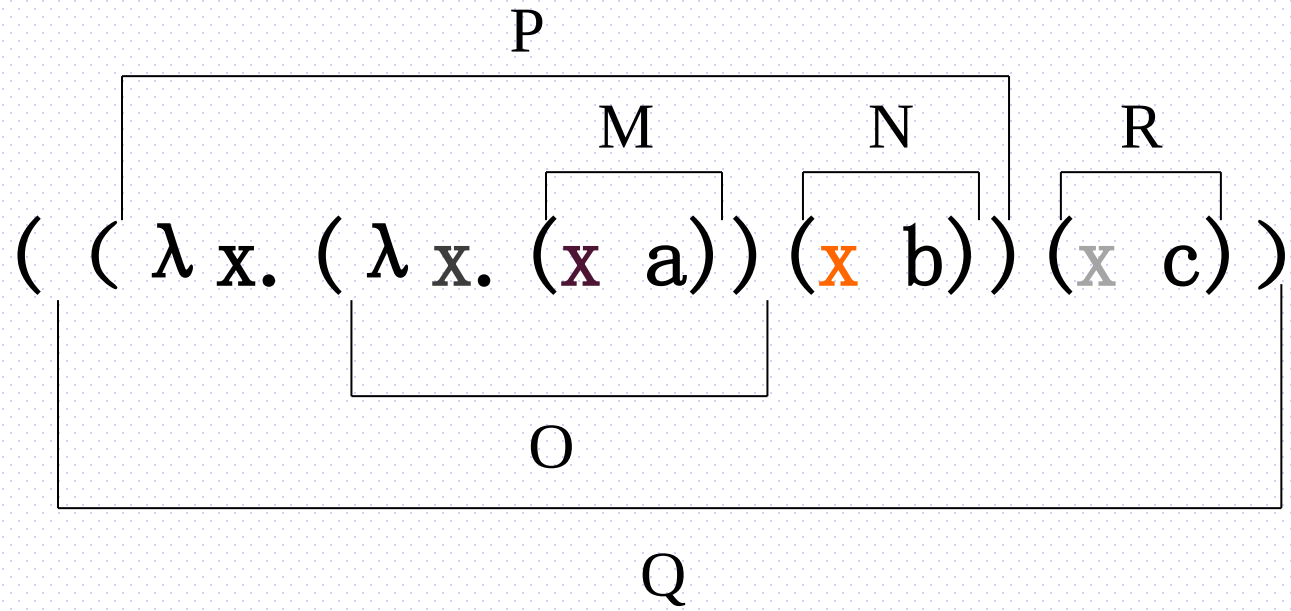
为什么没有数字？
字母不够怎么办？

5.2.2 约束变量和自由变量



λ*x*.(*x y*) 这样的 lambda 表达式并未定义一个函数，因为变量 *y* 的出现是**自由**的，即它并没有被**绑定**到表达式中的任何一个 λ 上。

Λ表达式中的作用域覆盖



第1个x约束于P
第2个x约束于O
第3个x在M中自由，约束于O，P
第4个x在N中自由，约束于P
第5个x在R，Q中自由，在Q中前4个x均约束出现，a，b，c自由出现。

LAMBDA变量绑定规则

1. 在表达式x中，如果x本身就是一个变量，那么x就是一个单独的自由出现。
2. 在表达式 λx. F 中，自由出现就是F中所有的除了x的自由出现。这种情况下在F中所有x的出现都称为被表达式中x前面的那个λ所绑定。
3. 在表达式(FG)中，变量的自由出现就是F和G中所有变量的自由出现。

a) $free(x) = x$

b) $free(F\ G) = free(F) \cup free(G)$

c) $free(\lambda x.\ F) = free(F) - \{x\}$

基本函数

- TRUE 和FALSE的λ表达式

  T = λx. λy. x

  F = λx. λy. y

- 整数的λ表达式:

  0= λx. λy. y

  1= λx. λy. x y

  2= λx. λy. x(x y)

  n= λx. λy. x(x(...(x y))

  n个

- 基本操作函数

not=λz.((zF)T)=λz.((zλx.λy.y)(λx.λy.x))

and=λa.λb.((ab)F) = λa.λb.((ab)λx.λy.y))

or = λa.λb.((aT)b) = λa.λb.((a λx.λy.x)b)

以下是算术操作函数举例:

+ = add = λx.λy.λa.λb.((xa)(ya)b)

* = multiply = λx.λy.λa.((x(ya)))

** = sqr = λx.λy.(yx)

identity = λx.x　　//同一函数

succ = λn.(λx.λy.nx(x y))　　//后继函数

zerop = λn.n(λx.F)T

=λn.n(λz.λx.λy.y)(λx.λy.x)　//判零函数

例：3+4。写全了是：

$(\lambda x.\ \lambda y.\ \lambda a.\ \lambda b.((x\ a)(y\ a)\ b\ ))$

$(\lambda p.\ \lambda q.(p(p(p\ q)))(\lambda s.\ \lambda t.(s(s(s(s\ t)))$

$\equiv \lambda a.\ \lambda b.(a(a(a(a(a(a(a\ b))))))))$

$+ = add = \lambda x.\ \lambda y.\ \lambda a.\ \lambda b.((xa)(ya)b)$

$3= \lambda x.\ \lambda y.x(x(xy))$

归约与范式

归约将复杂的表达式化成简单形式，　即按一定的规则对符号表达式进行置换。

例：归约数1的后继

(succ 1) => (λn.(λx. λy.n x(x y)) 1)

=> (λx. λy.1 x(x y))

=> (λx. λy.(λp. λq. p q) x(x y))

=> (λx. λy.(λq. x q)(x y))

=> (λx. λy.x(x y))=2

注：succ和1都是函数(1是常函数)，第一步是λn束定的n被1置换。展开后，x置换p，(xy)置换q，最后一行不能再置换了，它就是范式，语义为2。

(1) B归约:

- 归约的表达式是一个λ应用表达式(λx.F G)， 其左边子表达式是λ函数表达式，右边是任意λ表达式。
- 归约以右边的λ表达式置换函数体F中λ指明的那个形参变量。形式地，我们用[N/x,F]表示对（λx.F G）的置换。
- 关键的问题是注意函数体中要置换的变量是否自由出现，如:

((λx.x(λx.(xy)))(zz))

=>(zz)(λx.((zz)y)) //错误，第二x个非自由出现。

=>(zz)(λx.(xy)) //正确

B 归约的内涵

- Beta-归约规则表达的是函数作用的概念。
- 它陈述了若所有的 E' 的自由出现在 E [V/E'] 中仍然是自由的情况下，有 ((λ V. E) E') == E [V/E'] 成立。
- == 关系被定义为满足上述两条规则的最小等价关系 (即在这个等价关系中减去任何一个映射，它将不再是一个等价关系)。
- 对上述等价关系的一个更具操作性的定义可以这样获得：只允许从左至右来应用规则。

有以下规则：

1. $(\lambda x.M)N \rightarrow M[x/N]$ 如果N中所有变量的自由出现都在M[x/N]中保持自由出现
2. $M \rightarrow M$
3. $M \rightarrow N, N \rightarrow L \Rightarrow M \rightarrow L$
4. $M \rightarrow M' \Rightarrow ZM \rightarrow ZM'$
5. $M \rightarrow M' \Rightarrow MZ \rightarrow M'Z$
6. $M \rightarrow M' \Rightarrow \lambda x.M \rightarrow \lambda x.M'$
7. $M = M' \Rightarrow M=M'$
8. $M=M' \Rightarrow M'=M$
9. $M=N, N=L \Rightarrow M=L$
10. $M=M' \Rightarrow ZM = ZM'$
11. $M=M' \Rightarrow MZ = M'Z$
12. $M=M' \Rightarrow \lambda x.M = \lambda x.M'$

例:高层表示的 B 归约

- (λn.add n n) 3
  => add 3 3　　//3置换n后取消λn
  => 6
- (λf. λx.f(f x)) succ 7
  => λx.succ (succ x) 7
  => succ (succ 7)
  => succ (8) = 9

- **范式**
  - 如果一个λ-项M中不含有任何形为((λx.N1)N2)的子项，则称M是一个范式，简记为n.f.。如果一个λ.项M通过有穷步β-归约后，得到一个范式，则称M有n.f.，没有n.f.的λ-项称为n.n.f.。

Church-Rosser定理

但β归约有时并不能简化.

如：(λx.xx)(λx.xx)，归约后仍是原公式，　这种λ表达式称为不可归约的。

对应为程序设计语言中的无限递归。

(2) η 归约是消除一层约束的归约：λ x.fx => f， x不是f中的自由出现

(3) α 换名:归约中如发生改变束定性质，则允许换名(λ 后跟的变量名)，以保证原有束定关系。例如：

(λ x. (λ y.x)) (z y) //(zy)中y是自由变量

=> λ y.(zy) //此时(zy)中y被束定了，错误!

=> (λ x.(λ w.x))(zy) //因(λ y.x)中函数体无y， 可换名

=> λ w.(zy) // 正确!

(4)归约约定

- 顺序:每次归约只要找到可归约的子公式即可归约，λ演算没有规定顺序。
- 范式:符号归约当施行（除α规则外）所有变换规则后没有新形式出现，则这种λ表达式叫范式。
- 解释:范式即λ演算的语义解释，

  形如 x x，(y (λx.z))就只能解释为数据了。

上述基本函数均为范式，在它的上面取上有意义的名字可以构成上一层的函数，

如：pred =λn.(subtract n 1)

（5） 综合规约例题：以λ演算规约3**2

3**2=**(3)(2)

** = sqr = λx. λy.(yx)

2= λx. λy.x(x y)

$=\underline{\lambda x.}\ \lambda y.(y\ \underline{x})\ \underline{(3)}\ (2)$

$>_\beta (\underline{\lambda y.}\ (\underline{y}\ 3))\ \underline{(2)}$

$>_\beta (\underline{(2)}\ 3)$

$= (\underline{\lambda f.}\ \lambda c.\underline{f}\ (\ \underline{f}\ c\ )\ )\ \underline{(3)}$

$>_\beta \underline{\lambda c.}\ (\underline{(}\ \underline{3}\ \ (\ 3\ c\ )\ )$

$= \lambda c.(\underline{\lambda f.\ \lambda c.(f(f(f(f(c)))))}(3\ c))$

// 有c不能置换c

$>_\alpha \lambda c.(\underline{\lambda f.}\ \lambda z.\underline{(f}\ \underline{(f}\ \underline{(f}\ (z)))))(3\ c)$

$>_\beta \lambda c.(\underline{\lambda z.}\ (\underline{(3}\ c)(\underline{(3}\ c)(\underline{(3}\ c)(z)))))$

// 再展3

$= \lambda c.\ \lambda z.\ (((\ \lambda f.\ \lambda c.\ (f(f(f(c)))c)((3c)((3c)(z))))$

$>_{\alpha}\ \lambda c.\ \lambda z.\ (((\lambda f.\ \lambda w.\ (f(\ f(\ f(w)))c)$
$((3c)((3c)(z))))$

$3 = \lambda x.\ \lambda y.\ x(x(xy))$

$>_{\beta}\ \lambda c.\ \lambda z.\ (((\lambda w.\ (c(c(c(w))))((3c)((3c)(z))))$

//同理展开第二个c,第三个c

$= \lambda c.\ \lambda z.\ (((\lambda w.\ (c(c(c(w))))((\lambda p.\ (c(c(c(p)))))((\ \lambda q.\ (c(c(c(q)))))(z))))$

$>_{\beta}\ \lambda c.\ \lambda z.\ (((\lambda w.\ (c(c(c(w))))((\ \lambda p.\ (c(c(c(p)))))$
$(((c(c(c(z)))))))$

$>_{\beta}\ \lambda c.\ \lambda z.\ (((\ \lambda w.\ (c(c(c(w))))$
$(((c(c(c(c(c(c(z)))))))))))$

$>_{\beta}\ \lambda c.\ \lambda z.\ (c(c(c(c(c(c(c(c(c(z)))))))))) = 9$

增强Λ演算

只用最底层λ演算是极其复杂的。用高层命名函数，语义清晰。不仅如此，保留一些常见关键字，语义更清晰。

例如，我们可以定义一个if_then_else为名的函数：if_then_else = λp.λm.λn.p m n，当p为'真'时，执行m否则为n。我们先验证其真伪。

例：当条件表达式为真时if_then_else函数的归约

(if_then_else) T M N
=> (λp. λm. λn. p m n) T M N
=> (λm.λn. (T m n))M N
=> (λm. λn. (λx.λy.x) m n) M N
=> (λm. λn. (λy.m) n) M N
=> (λm. λn. m) M N
=> (λn. M) N
=>M

T = λx.λy.x

F = λx.λy.y

- if表达式

  可保留显式if-then-else形式:

  (if_then_else) E1 E2 E3= if E1 then E2
  else E3

  其中E1， E2， E3为λ表达式。

- Let/where表达式

如果有高阶函数：

(λn. multiply n (succ n)) (add i 2)

=> multiply (add i 2) (succ (add i 2))

//n 和 add i 2置换变元得

=> multiply n (succ n)

// let n = add i 2 in multiply n (succ n)

let a = b in E ≡ (λa. E) b ≡ E where a=b

(λf. E2) (λx.E1) = let f = λx.E1 in E2

= let f x = E1 in E2

其中形如f=λx.E1的λx. 可移向左边为f x = E1 。 如:

sqr = λn. multiply n n //整个是λ函数表达式

sqr n = multiply n n //两应用表达式也相等

let表达式在ML. LISP中直接采用，Miranda用where关键字使程序更好读， let直到E完结构成一个程序块。Miranda只不过把where块放在E之后。

- 元组表达式

一般情况下n元组是p=($x_1, x_2, \ldots x_n$)，建立在p上函数有：

let f($x_1, x_2, \ldots x_n$)=E1 in E2

≡ let fp=E1 in

let x_1=first p in

let x_2=second p in

.

.

.

let x_n=n_th p in E2

函数式描述方法

关于函数式描述方法

- 函数式语言的特点
  - 引用透明性；高阶性；模式匹配；并行性；
- 函数式语言的组成部分
  - 程序结构
  - 类型及其操作
  - 表达式
- 用函数式语言来描述算法（解释器）
  - 函数空间
  - 函数定义（方程）

关于函数式描述方法

- 函数式语言的组成部分
  - 程序结构
    - 函数定义
    - 目标表达式
  - 类型及其操作
    - 标准类型
    - 幂集类型
    - 联合类型
    - 函数类型
    - 抽象类型

- 集合类型
- 元组类型
- 序列类型
- 递归类型

关于函数式描述方法

- 函数式语言的组成部分
  - 表达式
    - 非let表达式（常量，变量，λ表达式，条件表达式，以及各种操作）
    - let表达式

      let x = E' *in* E
    - letrec表达式
      - letrec是一种用于递归形式的let语句；

      letrec x = E_1 *in* E
    - 在表达式中增加类型说明

```
let z=0 in
let y=add z 1 in
let x=add y 1 in mul (add x y) z
```

等价

```
letrec x=add y 1
and y=add z 1
and z=0 in add (add x y) z
```

关于函数式描述方法

- 用函数式语言来描述算法
  - 函数空间：$INT^* \times INT \rightarrow BOOL$
  - 函数定义（方程）

    lookup L a =

    (null L→FALSE，a=hd L→TRUE，lookup (tl L) a)

函数式语言提供很少的原始函数，可以构造复杂函数，再提供函数应用的操作，以及某种表示数据的结构，使用这些结构来表示参数以及函数计算的值。

5.3 函数式语言怎样克服命令式语言的缺点

- 5.3.1 消除赋值/变量

  赋值语句在过程语言中起什么作用。在函数式语言中取消会有什么问题:

  [1] 存储计算子表达式的中间结果。

  [2] 条件语句的重要组成。

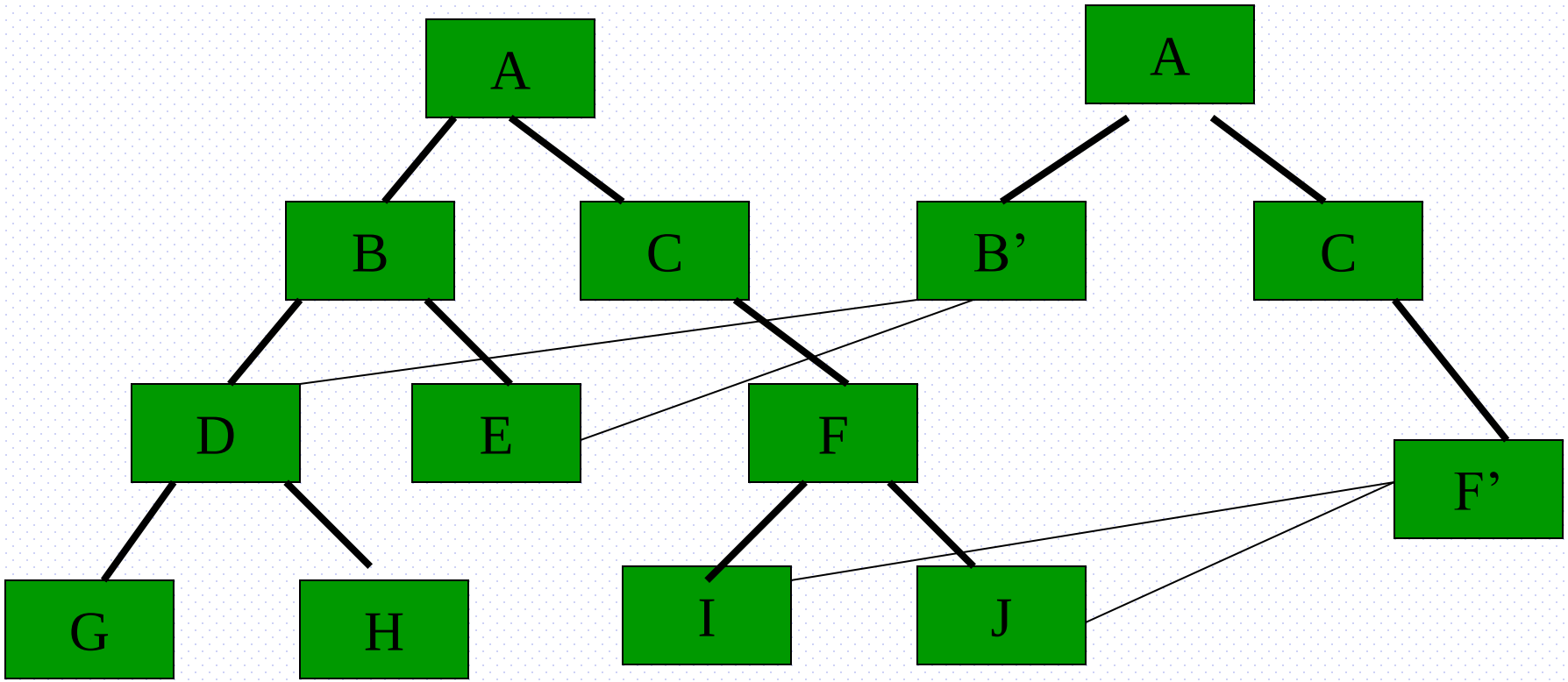
  [3] 用于循环控制变量。

  [4] 处理复杂数据结构(增删改某个成分)。

解决办法

1. 保留全局量、局部量(符号名)以及参数名。
2. 用条件表达式代替条件语句，其返回值通过参数束定或where子句束定于名字
3. 函数式语言都要定义表数据结构，因为归约和递归计算在表上很方便。对整个表操作实则是隐式迭代,不用循环控制变量。
   - 对于顺序值也都用表写个映射函数即可隐式迭代。其它显式循环要用递归。

[4]:禁止赋值意味着数据结构一旦创建不得修改,故采用如下函数式语言结构数据修改方式



5.3.2 以递归代替WHILE_DO

- 要点是把递归函数体写入条件表达式。
- 一般采用尾递归。
- 用递归仿真循环的终止条件是条件测试部分，函数如有返回值放在then部分，递归函数体放在else部分，如果不需返回值则取消一部分(else)。

5.3.3用嵌套代替语义相关的顺序

- 语句控制方案：顺序和嵌套。函数式语言要求函数间逻辑无关。
- 一旦语义相关无法传递数据，非得写成嵌套函数不可(返回值自动束定到外套函数的变元上)

pascal实现：

```
c：=a+b;
s：=sin(c * c);    。
write(a，b，  c，  s);  //上面计算不进行是无法打印的，逻辑上要有顺序。
```

LISP 实现：

```
(print (let ((c(+ a b)))
  let ((s (sin (* c c))))
     list a b c s ))))  //仍有顺序但在一个表达式内。自左至右处理即隐式顺序。
```

Miranda实现：

```
Answere a b = (a b c s )
              where
              s= sin (c* c)
              c= a+b          //全然没有顺序，逻辑清晰
```

- 用懒求值代替顺序
- 利用卫式进一步消除顺序性

Miranda的卫式表达式

```
gcd a b= gcd (a-b) b,   a>b
       = gcd a (b-a),   a<b
       = a,             a=b
```

Scala的条件选择

```
def gcd(a: Int,b: Int):Int = {
        if(a>b) gcd(a-b,b)
        else if (a<b) gcd(a,b-a) else a
}
```

LISP的条件选择

```
(define GCD (a b)
   (cond (( greaterp a b) (GCD ((diference a b ) b)))
         ((Lessp a b) (GCD (a (diference b a ))))
         ( T a )))
```

5.3.4 输出问题

- 输出利用了副作用。
- 懒求值、正规求值方案选择和头等程序对象之间的矛盾。
- 利用数据对象内部原有的顺序。语言支持的任何形式(交互、非交互)的输出都可以用在表和元组上。构造输入输出流形成值的无限表。
- 无限表尾不表示任何值，它是函数对象，每当调用到它时，它按规定计算表头值，并构造一新的函数对象放在表尾，以便再展其它项，它就是新的无限表尾的头，这个过程一直延续到需要的表长已达到。

5.3.5 表闭包

表闭包是一个任意复杂结构的(无限)表。其语法是：

〈ZF表达式〉::=‘[’〈体〉‘|’〈限定符表〉‘]’

〈限定符表〉::=〈产生器〉{‘;’〈产生器〉|〈过滤器〉}

〈产生器〉::=〈变量〉←〈表_表达式〉

〈过滤器〉::=〈布尔表达式〉

〈体〉::= 〈表_表达式〉

表闭包示例：

ZF 表达式	解释	
[n*2 | n←[2，4，6，8，10]]	[4，8，12，16，20]	[1]
[n*n | n ←[1，2，...]]	[1，4，9，...]	[2]
[x+y | x← [1..3]； y←[3，7]]	[4，5，6，8，9，10]	[3]
[x*y | x ←[1..3]； y←[1..3]; x>=y]	[1，2，4，3，6，9]	[4]

[1]行只有一个生成器， 表值2，4，6，8，10束定于n得出2倍表。
[2]行是无限表也只有一产生器，[3]行[4]行有两个产生器， [4]行还有一过滤器， 否则要对称出9个数。

5.3.6 高阶函数

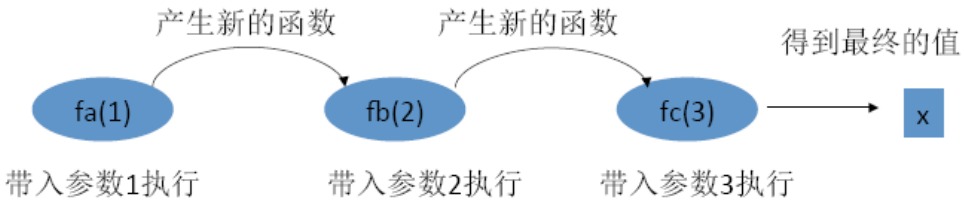
- 高阶函数主要有两种：
  - 一种是将一个函数当做另外一个函数的参数（即函数参数）；
  - 另外一种是返回值是函数的函数。
- （1）函数参数，即传入另一个函数的参数是函数
  - def valueAtOneQuarter(f: (Double) => Double) = f(0.25)
  - valueAtOneQuarter(ceil)
  - valueAtOneQuarter(sqrt)　//res0: 0.5，一个非负实数的平方根函数
- （2）返回值是函数的函数，高阶函数可以产生新的函数，即函数返回值是一个函数，也就是闭包。
  - 闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量；
  - def multiplyBy(factor:Double)=(x:Double)=>factor*x
  - val x=multiplyBy(10)
  - x(50)　//res0: Double = 500.0

5.3.7 柯里化(Currying)

- **柯里化(Currying)指的是将原来接受两个参数的函数变成新的接受一个参数的函数的过程。新的函数返回一个以原有第二个参数为参数的函数。**

f(1)(2)(3) → ((f(1))(2))(3)

柯里化过程示意图



柯里化实例：

原始多参数函数：
def sum(x:Int, y:Int)=x+y

转换成接收一个参数：
def first(x:Int)=(y:Int)=>x+y

调用方式一：
val second=first(1)
second(2)
调用方式二：first(1)(2)

柯里化：
def sum(x:Int)(y:Int) =x+y

调用方式：
sum(1)(2) //res0: Int = 3

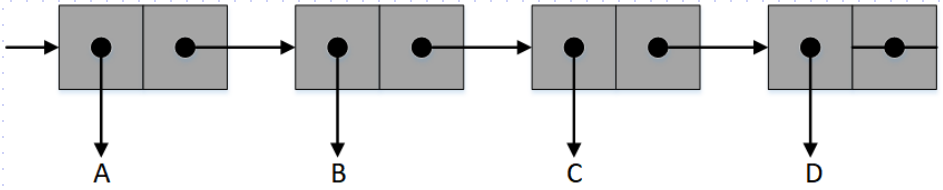
5.4 函数式语言

- 函数是头等程序对象
  - 引用透明性
  - 高阶函数
  - 组合与柯里化
  - 表闭包
- Lambda函数
  - 匿名函数
  - 闭包

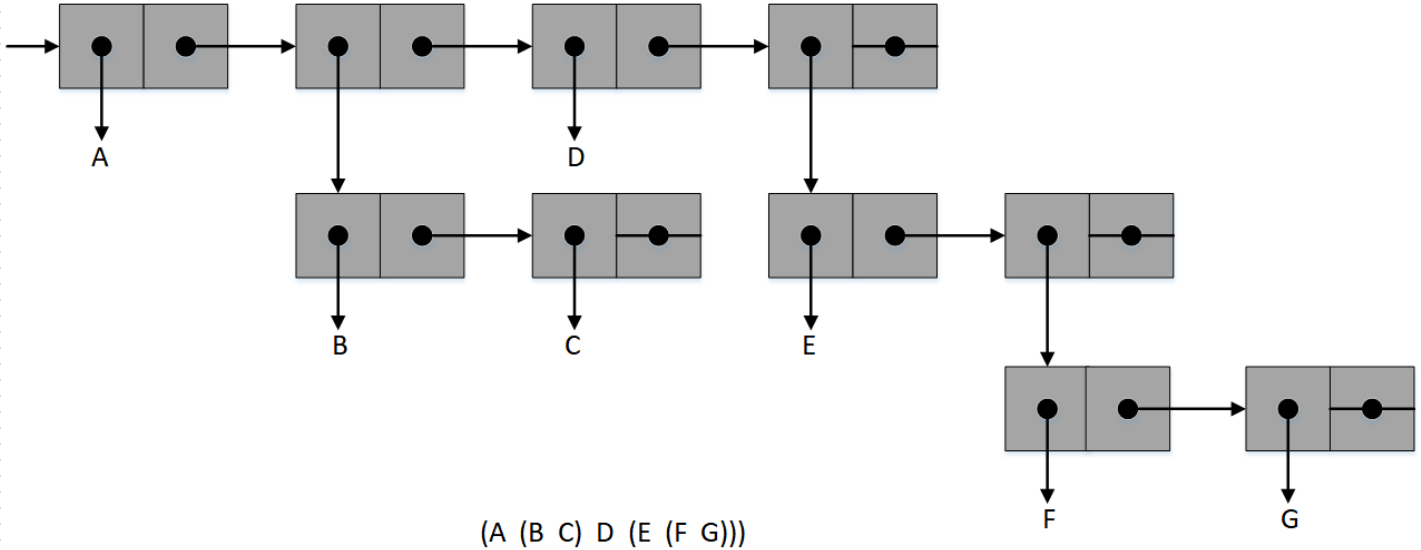
函数式语言分类

1. 函数式语言(Functional language)

- 纯函数式语言，如Lisp、Scheme、Haskell等；
- 混合型的函数式语言(支持一些非函数式编程的特性)
  - 如Scala、clojure(基于Lisp的方言)等语言；
- 严格的说不算是函数式编程语言，但具备部分函数式语言的特性
  - 如：Python、Ruby等；



(A B C D)



(A (B C) D (E (F G)))

- 纯函数语言中的数据对象
  - LISP：*List Processor*
  - 原子：符号或数学常量
  - 链表：
    - 简单链表：（A B C D），元素为原子
    - 嵌套链表：(A(B C)D(E(F G)))，(A ((B B))(((C C C))))

- 纯函数语言中的表达式与运算
  - S-表达式：原子或链表
    - 如果表达式e得出值v，我们说e返回v。
    - 如果一个表达式是一个表，那么我们把表中的第一个元素叫做操作符，其余的元素叫做自变量。
    - 数据即程序、程序即数据
    - **通用函数——EVAL表达式与基本操作符**
      - **quote、atom、eq、car、cdr、cons、cond**
    - 宏（Macro）:自定义一个语法结构

```
(define    macro-name
          (syntax-rules   ()
           ( (template)  operation)
            ……) )
```

```
( define-syntax my-when
    ( syntax-rules ()
      [ ( pred body ... )
        ( if pred ( begin body ...
) ( void ) ) ] ) )

( my-when    ( = 2 1 )
  ( display 1 )
  ( display 2 ) )
```

表(LIST)

- Miranda表的表示法

- [ ] //空表
- [1..n] //1到n, 域表示
- odd_number = [1, 3, ..100] //1到100内奇数表，头两项及最后项必写
- eleven_up = [11...] //10以上， 无限表表示
- evens = [10, 12...100] //10以上偶数表至100，头两项及最后项必写
- evens_up = [12, 14...] //10以上偶数无限表，
- week _days = [“Mon”, “Tue”, “Wed”, “Thur”, “Fri”] //五个串值的表

- Scheme
  - Scheme程序是一个函数定义集合
    - 原始函数：只处理原子
    - 定义函数：lambda表达式，如LAMBDA(x) (* x x)
      - 用LET函数简化（创建局部静态作用域）
    - 输出函数
    - 数值谓词函数：#T|#F（是否为空表）
    - 控制流：顺序，COND，递归
    - 链表函数

```
fun sq(x) = x*x

(define pi 3.14)
(define (sq x)(* x x)
(define sq (lambda (x) (* x x)))
(lambda (x) (* x x)
```

匿名函数值

支持递归定义

```
(* E1 E2)
(E1 E2 E3)
(if P E1 E2)
(cond (P1 P2) (P2 E2)
(else E3))
```

- Scheme

EVAL(E1), EVAL(E2), EVAL(E3)

```
(let ((x1 E1) (x2 E2)) E3)
(let* ((x1 E1) (x2 E2)) E3)
```

let val x1=E1
val x2=E2
in E3
end

```
(define x 0)
  x
(let ((x 2) (y x)) y)     ;bind y before redefining x
 0
(let* ((x 2) (y x)) y)   ; bind y after redefining x
 2
```

```
(quote sun)
(quote (mon tue wed))
(list E1 E2 E3)
```

- Scheme
  - 表的运算：car、cdr与cons
    - caar、cadr、cddr、cdar
    - cons(a x)，可以用(a.x)代替
  - 表的操作：length、rev、append、map、remove-if、reduce

```
(length '()) = 0
(length (cons a y)) = (+1 (length y))
```

```
(length x) = (+1 (length (cdr x)))
```

```
(define (length x)
    cond (null? x) 0
              (else (+1 (length (cdr x)))  ))
```

- Scheme
  - 表的操作：map、reduce，关联表
  - map把对元素的函数扩展到一个表
  - map的语义

(map f ‘( )) = ‘( )
(map f (cons a y) = (cons (f a) (map f y))

例子

```
define (square n) (* n n)
(map square ‘(1 2 3 4 5))
                    (1 4 9 16 25)
```

- reduce（fold）函数的功能

(reduce f ‘( ) v) = v

(reduce f (cons a y) v = f(a, reduce f y v)

接收一个列表、一个初始值以及一个函数，将该函数作为特定的组合方式，将其递归地应用于列表的所有成员，并返回最终结果。

例：fold + ‘0 ‘(1,2,3)，其具体的操作流程是(((0+1)+2)+3)，并最终返回其总和。

- Scheme
  - 例子：链表原子元素求和

```
(DEFINE (adder lis)
    (COND
        ((NULL? lis) 0)
        (ELSE (+ (CAR lis) (adder (CDR  lis))))
))
```

```
(DEFINE (adder lis)
    (COND
        ((NULL? lis) 0)
        (ELSE (EVAL (CONS  '+ lis)))
))
```

```
(adder  '(3 4 6))
->  (+ 3 4 6)
```

练习题（10分钟）

- 编写一个Scheme函数guess，返回一个链表，这个链表中的元素是所给两个链表（lis1，lis2）的交集。
  - quote(')、atom、eq、car、cdr、cons、cond

```
Member函数：原子是否在表中
( DEFINE ( Member  atm  lis )
     ( COND
          (( NULL?  lis)  #F )
          (( EQ?  atm ( CAR lis )) #T)
          (ELSE ( member atm (CDR lis)))
))
```

编写一个Scheme函数（tlrm），将给定的链表（lis）中最后一个元素移除。

member函数：原子是否在表中

```
( DEFINE ( member  atm  lis )
     ( COND
          (( NULL?  lis)  #F )
          (( EQ?  atm ( CAR lis )) #T)
          (ELSE ( member atm (CDR lis)))
))
```

```
(DEFINE (guess lis1 lis2)
  (COND
       ((NULL? lis1)  '())
       ((member (CAR lis1) lis2)
            (CONS (CAR lis1)  (guess (CDR lis1) (lis2)))
       ( ELSE (guess (CDR lis1) lis2)
```

- ML
  - ML是有类型的，Scheme没有
    - ML中的是同类型的，或空表，[ ]
    - hd——car，tl——cdr，a::y
    - 表上的运算：
      - fun f(x) = if 表x为空 then ……
        else 做某种牵扯到hd(x), tl(x)和f的事情
      - fun append(x, z)
      - fun reverse(x, z) = if null(x) then z else reverse(tl(z), hd(x)::z)
      - reverse([1, 2, 3], [ ]) = [3, 2, 1]
      - reverse(a::y, z) = reverse(y, a::z)
    - 函数的分情况说明：任何时候都要考虑空表
    - 模式：由变量和值构造符组成。fun first(x, y) = x;
    - 匿名函数：fn <formal-parameter> => <body>
      - fn x => x*2
    - 选择性复制（remove_if)

- ML的类型
  - ML有类型，编译时检查，但是无需类型声明
    - 类型推理，或者< expression > : < type >
      - fun succ n = n+1;　//ok
      - fun add(x+y) = x+y;　//error
    - 参数化多态性：
      - 表的操作
    - 值构造符：datatype
      - 递归申明
      - datatype bitree = leaf | nonleaf of bitree * bitree　//构造符有一个参数bitree * bitree

5.5 函数式语言SCALA

- Scala 是一个多范式（multi-paradigm）的编程语言，设计初衷是要集成OOP和函数式编程（FP）的各种特性,平滑地集成了面向对象和函数式语言的特性。Scala只有很少几个特性是原创的：
  - 面向对象特性：Scala是一种纯面向对象的语言，每个值都是对象；
  - 函数式编程：即函数也能当成值来使用。可以定义匿名函数、高阶函数、多层嵌套函数，并支持柯里化；
  - 静态类型：即Scala具有的类型系统，编译时先检查类型，保证了代码的安全性、一致性；
  - 扩展性
  - 并发性：使用Actor作为并发模型，但它不同于Thread，它相当于Thread的实体。在Scala中，多个Actor可以分别复用/使用同一个线程，因此可以在程序中可以使用数百万个Actor,而线程只能创建数千个。

Scala简单的例子

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, world!")
    }
}
```

Scala的基本数据类型与Java中的基本数据类型是一一对应的，不同的是Scala的基本数据类型头字母必须大写，比如Int、Long、String、Char、Double、Float等，数据结构有列表、元组、集、映射等。

数据结构

- 声明值和变量

val:常量；　　例：val answer = 8 * 5 + 2

var:值可变的变量；　例：var counter = 0

counter = 1 // OK,可以改变一个var

- 元组(tuple)

元组是不同类型的值的聚集。元组的值是通过将单个的值包含在圆括号中构成的。例如：

(1, 3.14, "Fred")

- 列表(list)

类似于数组，它们所有元素的类型都相同，列表是不可变的，值一旦被定义了就不能改变，其次列表 具有递归的结构。所有的列表可以使用两种基本的构建模块来定义，一个无尾**Nil**和 **::**。**Nil**代表了空列表。

```
val nums: List[Int] = List(1, 2, 3, 4)
// 整型列表
val nums = 1 :: (2 :: (3 :: (4 :: Nil)))

// 空列表
val empty = Nil
```

- 集(set)　**Set是没有重复的对象集合，所有的元素都是唯一的，集不保留元素插入的顺序。**

  val set = Set(1,2,3)

  val nums: Set[Int] = Set()

- 映射(map)　**Map(映射)是一种可迭代的键值对（key/value）结构。**

  任何值都可以根据键来进行检索。键在映射中是唯一的，但值不一定是唯一的。定义Map时，需要为键值对定义类型。

  var A:Map[Char,Int] = Map()

  val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")

  如果需要添加 key-value 对，可以使用 + 号

  A + = ('J' -> 5)

内定义操作

在Scala中任何方法都可以是操作符

Scala定义了常规的算术运算符(+、-、*、/、%)并按中缀表示使用
a+b 其实是如下方法调用的简写
a.+(b)

Scala内定义了一些有用的表操作：

```
L1 ::: L2          //   表L2并接到表L1的末尾
L1 ++: L2          //   表L1并接到表L2的头部
A +: List         // 在列表List的头部添加一个元素A
List :+ A         //在列表List的尾部添加一个元素A
(A /:List)(_+_) //对List中所有的元素进行相同的操作+A
```

定义函数

- Scala源文件中可以定义两类函数：
  - 类方法：由类实例进行调用；
  - 局部函数：在函数内部定义，作用域仅限于定义它的函数内部；

- Scala函数声明具有以下形式：

```
def functionName ([list of parameters]) : [return type]
```

- Scala函数定义具有以下形式：

```
def functionName ([list of parameters]) : [return type] = {
    function body
    return [expr]
}
```

1. 基本特性：

Scala是面向对象的

鉴于一切值都是对象，可以说Scala是一门纯面向对象的语言。对象的类型和行为是由类和特质来描述的。类可以由子类化和一种灵活的、基于mixin的组合机制（它可作为多重继承的简单替代方案）来扩展。

Scala是函数式的

鉴于一切函数都是值，又可以说Scala是一门函数式语言。

Scala为定义匿名函数提供了轻量级的语法，支持高阶函数，允许函数嵌套及柯里化。

Scala的样例类和内置支持的模式匹配代数模型，在许多函数式编程语言中都被使用。

对于那些并非类的成员函数，单例对象提供了便捷的方式去组织它们。

object，class，trait

- class：该关键字定义一个类，与java基本一致；
- object：
  - Scala 比 java 更面向对象的一点，是 Scala 的类不允许有静态（static）成员。对此类使用场景，Scala提供了单例对象（singleton object）。
  - 单例对象是一种特殊的类，有且只有一个实例。和惰性变量一样，单例对象是延迟创建的，当它第一次被使用时创建。
  - 是头等程序对象。单例对象的定义看上去和类定义很像，只不过 class 关键字被换成了 object 关键字。它和类的区别是，object对象不能带参数。

当一个单例对象和某个类共享一个名称时，这个单例对象称为 伴生对象。同理，这个类被称为是这个单例对象的伴生类。类和它的伴生对象可以互相访问其私有成员。使用伴生对象来定义那些在伴生类中不依赖于实例化对象而存在的成员变量或者方法。

```
1  import scala.math._
2
3  class Circle(radius: Double) {
4    import Circle._
5    def area: Double = calculateArea(radius)
6  }
7
8  object Circle {
9    private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
10 }
11
12 val circle1 = Circle(5.0)
13
14 circle1.area
```

object，class，trait

- trait：特质 (Traits) 用于在类 (Class)之间共享程序接口 (Interface)和字段 (Fields)。 它们类似于Java的接口。 将瘦接口扩充为富接口。
- 类和对象 (Objects)可以扩展特质，但是特质不能被实例化，因此特质没有参数。

特征作为泛型类型和抽象方法非常有用。

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}
```

使用 extends 关键字来扩展特征。然后使用override 关键字来实现trait里面的任何抽象成员：

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}

class IntIterator(to: Int) extends Iterator[Int] {
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Int =  {
    if (hasNext) {
      val t = current
      current += 1
      t
    } else 0
  }
}


val iterator = new IntIterator(10)
iterator.next()  // returns 0
iterator.next()  // returns 1
```

当某个特质被用于组合类时，被称为混入。

类D有一个父类B和一个混入C。一个类只能有一个父类但是可以有多个混入（分别使用关键字extend和with）。混入和某个父类可能有相同的父类。

```
abstract class A {
  val message: String
}
class B extends A {
  val message = "I'm an instance of class B"
}
trait C extends A {
  def loudMessage = message.toUpperCase()
}
class D extends B with C

val d = new D
println(d.message)  // I'm an instance of class B
println(d.loudMessage)  // I'M AN INSTANCE OF CLASS B
```

柯里化(Currying)

Currying也是一个Scala的语言特性:

```
def add(x:Int): (Int)=>Int = {
    def square(t:Int): Int = {
        t*t
    }
    x + square(_)
 } // add(1)(2) is 5
// 等价于下面的实现方式
def add(x:Int, y:Int): Int = {
    x + y*y
}
```

Currying通过把函数作为函数值，所以支持类似f(x1)(x2)(x3)这种调用方式。这样做能够简化函数的定义方式。

Scala面向对象特征

一切值都是对象，从这一点来说，是一门纯面向对象的语言，对象的类型和行为由类和特质描述。这方面类似java。

（1）封装性

Scala 访问修饰符基本和Java的一样，分别有：private，protected，public。

- 用 private 关键字修饰，带有此标记的成员仅在包含了成员定义的类或对象内部可见，同样的规则还适用内部类。Scala 中的 private 限定符，比 Java 更严格，在嵌套类情况下，外层类不能访问被嵌套类的私有成员；

（1）封装性

- 在 scala 中，保护（Protected）只允许在定义了该成员的类的子类中被访问。而在java中，用protected关键字修饰的成员，除了定义了该成员的类的子类可以访问，同一个包里的其他类也可以进行访问；
- 如果没有指定访问修饰符，默认情况下，Scala 对象的访问级别都是public， public成员在任何地方都可以被访问。

（1）封装性

作用域保护：在Scala中，访问修饰符可以通过使用限定词强调。格式为：

private[x] 或 protected[x]

这里的x指代某个所属的包、类或单例对象。如果写成private[x],读作"这个成员除了对[…]中的类或[…]中的包中的类及它们的伴生对象可见外，对其它所有类都是private。

这种技巧在横跨了若干包的大型项目中非常有用，它允许你定义一些在你项目的若干子包中可见但对于项目外部的客户却始终不可见的东西。

（2）继承

Scala继承体系结构设计非常巧妙，它没有特殊地对待「基本数据类型」，将万物视为对象。此外，Scala在顶层引入Any，它是所有类的父类；而在底层引入了Nothing，它是所有类的子类，整个系统的设计保持一致和完整。

Scala继承机制跟Java很相似，使用extends关键字，只允许继承一个父类。但重写一个非抽象方法或子类中的属性val要覆盖父类中的属性，必须使用override修饰符。

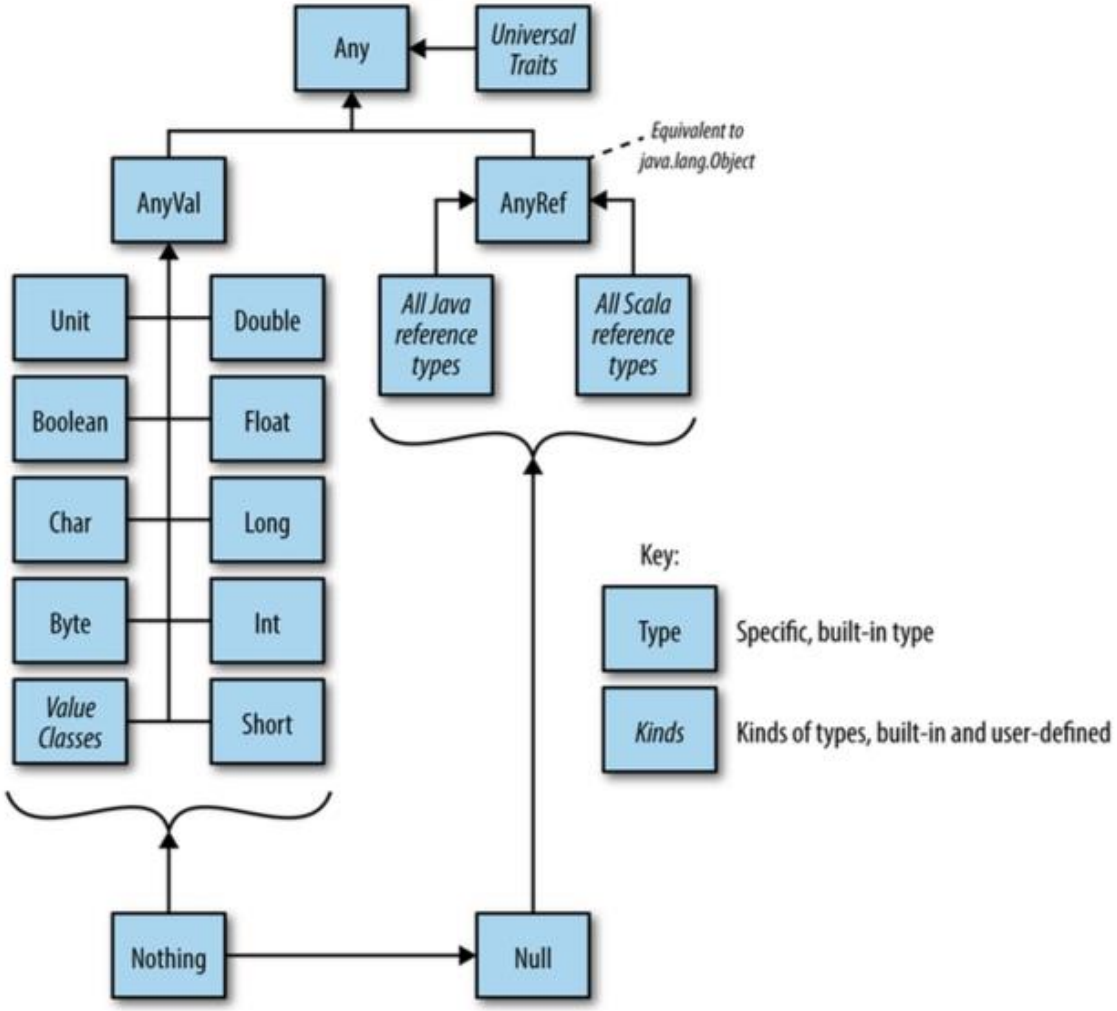
Scala类结构

（2）继承

Scala的对象可分为两个类型：

- 引用类型(Reference Types)：继承自AnyRef
- 值类型(Value Types)：继承自AnyVal

而**Any则是AnyVal**,AnyRef的父类。也就是说，Any是所有Scala类型的父类，它内置于Scala内部，由编译器实现。

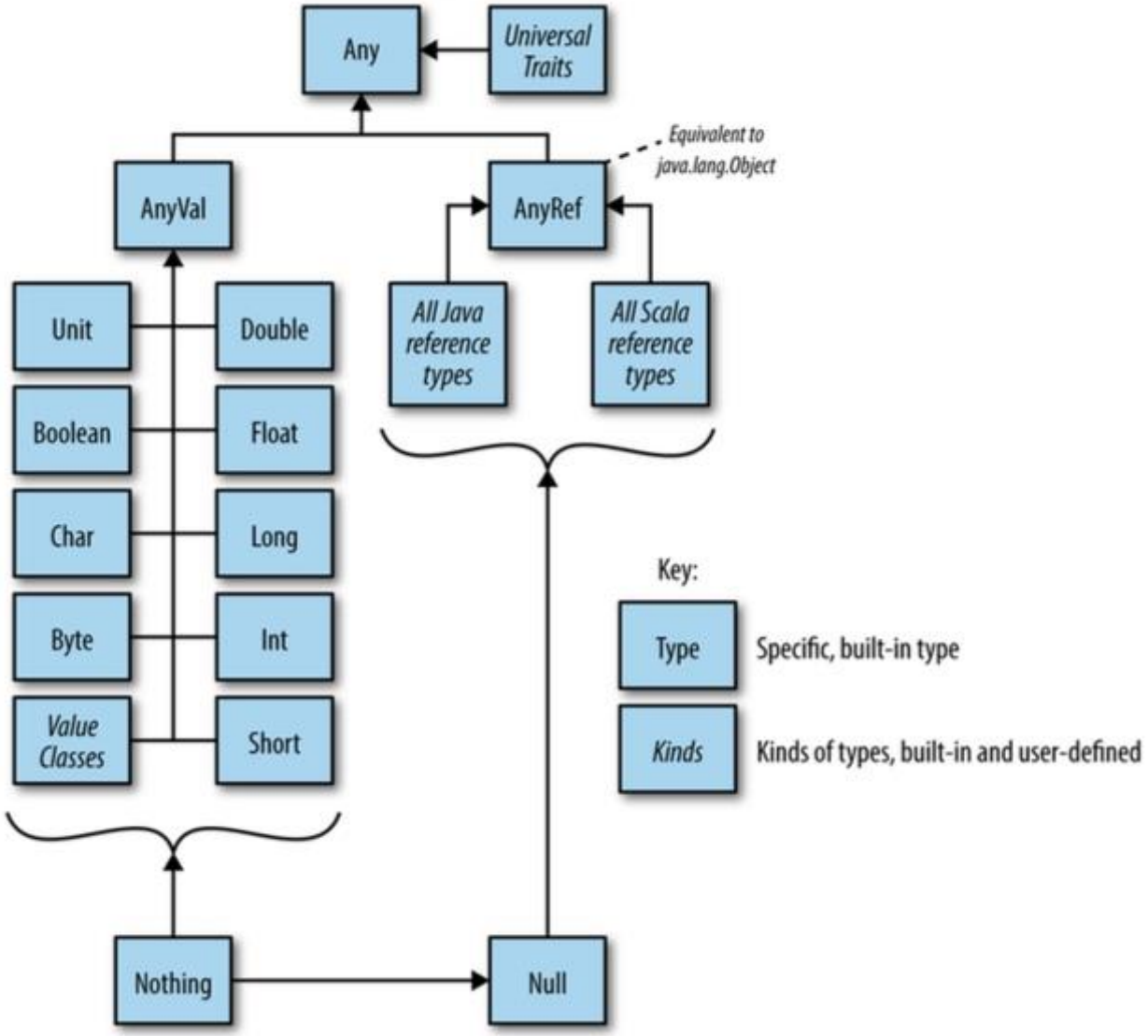


（2）继承

Nothing是所有类型的子类型，也称为底部类型（bottom types）。没有一个值是Nothing类型的。它的用途之一是给出非正常终止的信号，如抛出异常、程序退出或者一个无限循环（可以理解为它是一个不对值进行定义的表达式的类型，或者是一个不能正常返回的方法）。

Null是所有引用类型的子类型。它有一个单例值由关键字null所定义。Null主要是使得Scala满足和其他JVM语言的互操作性，但是几乎不应该在Scala代码中使用。

Scala类结构



（2）继承

Scala Trait(特征)：

- Scala Trait(特征) 相当于 Java 的接口，实际上它比接口的功能强大；
- 与接口不同的是，它可以定义属性和方法的实现；
- 一般情况下Scala的类只能够继承单一父类，但如果是 Trait(特征) 的话就可以继承多个，从结果来看就是实现了多继承。

3. 函数式

在scala中，一切函数都是值，函数也可作为另一个函数的参数，因此scala是一门函数式语言。

这一点又和python类似，因此我们说scala集成了java和python的特性。

方法与函数:

- Scala 有方法与函数，二者在语义上的区别很小。Scala 方法是类的一部分，而函数是一个对象可以赋值给一个变量。换句话来说在类中定义的函数即是方法。
- Scala 中的方法跟 Java 的类似，方法是组成类的一部分。
- Scala 中的函数则是一个完整的对象，Scala 中的函数其实就是继承了 Trait 的类的对象。
- Scala 中使用 val 语句可以定义函数，def 语句定义方法。

高阶函数：

高阶函数是指使用其他函数作为参数、或者返回一个函数作为结果的函数。在Scala中函数是“头等程序对象”，所以允许定义高阶函数。

最常见的一个例子是Scala集合类（collections）的高阶函数map（map方法接受一个函数参数，将它应用到数组中的所有值，然后返回结果的数组）：

```
val salaries = Seq(20000, 70000, 40000)
val doubleSalary = (x: Int) => x * 2
val newSalaries = salaries.map(doubleSalary) // List(40000, 140000, 80000)
```

高阶函数：

为了简化压缩代码，我们可以使用匿名函数，直接作为参数传递给map:

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(x => x * 2)
```

注意在上述示例中x没有被显式声明为Int类型，这是因为编译器能够根据map函数期望的类型推断出x的类型。对于上述代码，一种更惯用的写法为：

```
val salaries = Seq(20000, 70000, 40000)
val newSalaries = salaries.map(_ * 2)
```

考察：1.以下简写方式是否有效?

```
val fun = 3 * _
```

```
val fun = 3 * (_: Double)
```

2.以下代码输出结果?

```
(1 to 9) .map("*" * _).foreach(println _)
```

练习：1. *val fun = 3 * _ //错误：无法推断出类型*

*val fun = 3 * (_: Double) //OK*

2.输出结果：

```
*
**
***
****
*****
******
*******
********
*********
```

高阶函数(接收函数作为参数的函数):

使用高阶函数的一个原因是减少冗余的代码。如下例所示，promotion有两个参数，薪资列表和一个类型为Double => Double的函数（参数和返回值类型均为Double），返回薪资提升的结果。

```
object SalaryRaiser {

  private def promotion(salaries: List[Double], promotionFunction: Double => Double): List[Double] =
    salaries.map(promotionFunction)

  def smallPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * 1.1)

  def bigPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * salary)
}
```

高阶函数(返回函数的函数):

有一些情况我们希望生成一个函数， 比如:

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) // "https://www.example.com/users?id=1": String
```

注意urlBuilder的返回类型是(String, String) => String，这意味着返回的匿名函数有两个String参数，返回一个String。在这个例子中，返回的匿名函数是:

(endpoint: String, query: String) => s"https://www.example.com/$endpoint?$query"

嵌套函数:

在Scala中可以嵌套定义方法。例如以下对象提供了一个factorial方法来计算给定数值的阶乘:

```
def factorial(x: Int): Int = {
   def fact(x: Int, accumulator: Int): Int = {
     if (x <= 1) accumulator
     else fact(x - 1, x * accumulator)
   }
   fact(x, 1)
}

println("Factorial of 2: " + factorial(2))
println("Factorial of 3: " + factorial(3))
```

程序输出:

```
Factorial of 2: 2
Factorial of 3: 6
```

案例类（Case classes）

案例类（Case classes）和普通类差不多，只有几点关键差别。案例类非常适合用于不可变的数据。

一个最简单的案例类定义由关键字case class，类名，参数列表（可为空）组成。案例类实例化时，不用使用关键字new，这是因为案例类有一个默认的apply方法来负责对象的创建。当你创建包含参数的案例类时，这些参数是公开（public）的val。

```
case class Book(isbn: String)
val frankenstein = Book("978-0486282114")
```

案例类（Case classes）

案例类在比较的时候是**按值比较而非按引用比较**：

```
case class Message(sender: String, recipient: String, body: String)

val message2 = Message("jorge@catalonia.es", "guillaume@quebec.ca", "Com va?")
val message3 = Message("jorge@catalonia.es", "guillaume@quebec.ca", "Com va?")
val messagesAreTheSame = message2 == message3  // true
```

尽管message2和message3引用不同的对象，但是他们的值是相等的，所以message2 == message3为true。

模式匹配

模式匹配是检查某个值（value）是否匹配某一个模式的机制，一个成功的匹配同时会将匹配值解构为其组成部分。它是Java中的switch语句的升级版，同样可以用于替代一系列的 if/else 语句。

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
matchTest(3)  // other
matchTest(1)  // one
```

一个模式匹配语句包括一个待匹配的值，match关键字，以及至少一个case语句。

案例类（case classes）的匹配

案例类非常适合用于模式匹配。

Notification 是一个虚基类，它有三个具体的子类Email, SMS和VoiceRecording，我们可以在这些案例类(Case Class)上像这样使用模式匹配：

```
abstract class Notification
case class Email(sender: String, title: String, body: String) extends Notification
case class SMS(caller: String, message: String) extends Notification
case class VoiceRecording(contactName: String, link: String) extends Notification

def showNotification(notification: Notification): String = {
  notification match {
    case Email(sender, title, _) =>
      s"You got an email from $sender with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"you received a Voice Recording from $name! Click the link to hear it: $link"
  }
}
val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))
// prints You got an SMS from 12345! Message: Are you there?

println(showNotification(someVoiceRecording))
// you received a Voice Recording from Tom! Click the link to hear it: voicerecording.org/id/123
```

4. 映射与元组

映射是键/值对偶的集合。

Sala有一个通用的叫法——元组——n 个对象的聚集，并不一定要相同类型的。

对偶不过是一个n=2的元组。

元组对于那种需要将两个或更多值聚集在一起时特别有用。

构造映射

val scores = Map(“Alice” -> 10,“Bob” -> 3,”Cindy” -> 8)

上述代码构造出一个不可变的Map[String, Int].其值不能被改变。如果你想要一个可变映射，则用

val scores = scala.collecton.mutable.Map("Alice -> 10, "Bob" -> 3, "Cindy" -> 8)

如果想从一个空的映射开始，你需要选定一个映射实现并给出类型参数:

val scores = new scala.collection .mutable. HashMap[String, Int]

在Scala中，映射是对偶的集合。对偶简单地说就是两个值构成的组，这两个值并不一定是同一个类型的，比如("Alice”, 10)。

元组

映射是键/值对偶的集合。对偶是元组（tuple）的最简单形态——元组是不同类型的值的聚集。元组的值是通过将单个的值包含含在圆括号中构成的。例如:

(1, 3.14, "Fred")

是一个元组，类型为:

Tuple3[Int, Double, java.lang string]

类型定义也可以写为:

(Int, Double, java.lang .string)

元组

和数组或字符串中的位置不同，元组的各组元从1开始，而不是0。可以用方法_1、_2、_3访问元组组元， 比如:

```
val t = (1, 3.14, "Fred")
val second = t._2 //将second设为3.14
```

说明:你可以把t._2写为t _2(用空格而不是句点)，但不能写成t_2。

通常，使用模式匹配来获取元组的组元，例如:

```
val (first, second, third) = t //将first设为1. second设为3.14. third设为"Fred”
```

如果并不是所有的部件都需要，那么可以在不需要的部件位置上使用_:

```
val (first, second,_ ) = t
```

元组

元组可以用于两数需要返回不止一个值的情况。举例来说，StringOps的 partition 方法返回的是一对字符串， 分别包含了满足某个条件和不满足该条件的字符:

```
"New York". partition(_.isUpper) //输出对偶("NY"， "ew ork")
```

拉链操作

使用元组的原因之一是把多个值绑在一 起，以便它们能够被一起处理， 这通常可以用zip方法来完成。举例来说，下面的代码:

```
val symbols = Array("<", “-” , ”>")
val counts = Array(2, 10, 2)
val pairs = symbols.zip (counts)
```

输出对偶的数组:

```
Array(("<", 2)，("-", 10)，(">"， 2))
```

然后这些对偶就可以被一起处理:

```
for ((s,n) <- pairs) Console.print(s * n) //会打印<<---------->>
```

5. 类型参数

在Scala中，可以用类型参数来实现类和函数，这样的类和函数可以用于多种类型。举例来说，Array[T]可以存放任意类型T的元素。基本的想法很简单，但细节可能会很复杂。有时，需要对类型做出限制。例如，要对元素排序，T必须提供一定的顺序定义。并且，如果参数类型变化了，那么参数化的类型应该如何应对这个变化呢? 举例来说，当一个函数预期Array[Any]时， 你能不能传入一个ArrayString呢?在Scala中，你可以指定你的类型如何根据它们的类型参数的变化而变化。

泛型类

和Java或C++样，类和特质可以带类型参数。在Scala中， 用方括号来定义类型参数，例如

class Pair[T, S](val first:T, val second: S)

以上将定义一个带有两个类型参数T和S的类。在类的定义中，可以用类型参数来定义变量、方法参数，以及返回值的类型。

带有一个或多个类型参数的类是泛型的。如果把类型参数替换成实际的类型，将得到一个普通的类，比如Pair[Int, String]。

泛型类

Scala会从构造参数推断出实际类型:

```
val p = new Pair(42, "String") //这是一个Pair[Int, String]
```

也可以自己指定类型:

```
val p2 = new Pair(Any,Any)(42, "String")
```

泛型函数

函数和方法也可以带类型参数。以下是一个简单的示例:

```
def getMiddle[T] (a:Array[T]) = a(a.length / 2)
```

和泛型类样，需要把类型参数放在方法名之后。

Scala会从调用该方法使用的实际参数来推断出类型。

```
getMiddle("Mary, "had", "a", "little", "lamb") // 将会调用getMiddle[String]
```

可以自己指定类型:

```
val p2 = new Pair[Any,Any](42,“ String”)
```

```
def  printArgs (args: Array[String]): Unit ={
  var i=0
  while ( i < args.length) {
      i+= 1
  }
}
```

```
def  printArgs (args: Array[String]): Unit ={
  for ( arg <- args)
     println(arg)
}
```

```
def  printArgs (args: Array[String]): Unit ={
   args. foreach ( println )
}
```

```
def  formatArgs (args: Array[String]) ={args.mkString( “\n” )}
```

R语言函数式编程

函数的构成：

1. 主体 body()，函数内部的代码。
2. 形式参数 formals()，控制如何调用函数的参数列表。
3. 环境 environment()，函数变量的位置映射。

```
> f <- function(x) x^2
> f
function(x) x^2
> formals(f)
$x
```

```
> body(f)
x^2
> environment(f)
<environment: R_GlobalEnv>
```

"To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call."
  - John Chambers

```
> class(`+`)
[1] "function"
> class(`<-`)
[1] "function"
```

函数可以作为

- manipulated
- stored in a variable
- lambda
- stored in a list
- arguments of a function
- returned by a function

1. functional 泛函：
   - 函数可以作为函数参数传入，这些特性可以设计map/reduce/accumulate/filter等操作；

2. function factory函数工厂：
   - 利用函数的闭包性质，把函数的一些参数固定下来，形成新的函数；

3. function operator函数操作符：
   - 利用函数的闭包性质，在函数的前或后增加前置操作或后置操作，形成新的函数。

三种类型的高阶函数

1. functional 泛函
2. function factory函数工厂
3. function operator函数操作符

In \ Out	Vector	Function
Vector		Function factory
Function	Functional	Function operator

I. R语言中的泛函是指：包含函数作为输入，返回值为向量的一类高阶函数。

- 随机生成一批标准正态分布随机数
- 用待定函数来处理这批数据
  - mean：平均数
  - sd：标准差

```
> myfun <- function(f) f(rnorm(100))
> myfun(mean)
[1] 0.1579459
> myfun(sd)
[1] 0.8930742
```

I. 匿名函数

– lapply (list, function)：对每个列表对象应用函数；

- 创建20个长度为[1, 10]范围内的随机数数组的列表；
- 使用for循环依次求每个数组的长度；

```
> l<-replicate(20,rnorm(sample(1:10,1)))
> out<-vector("list",length(l))
> for(i in seq_along(l)){ out[[i]]<-length(l[[i]])}
> unlist(out)
 [1] 10  8  1  8  8 10  5 10  1  3  9  2  7  1  4  2  8 10  9 10
>
```

- 使用匿名函数

```
> unlist(lapply(l, length))
 [1] 10  8  1  8  8 10  5 10  1  3  9  2  7  1  4  2  8 10  9 10
>
```

1. 替代for循环
   - 泛函一般用来替代for循环；

2. 减少出错风险
   - 常用泛函已经封装好，比起for循环，Bug会更少；

3. 提高效率
   - 常用的泛函，底层是用C写的，提高了运行效率；

1. “An object is data with functions. A closure is a function with data.” — John D. Cook

2. 闭包：返回函数的函数；
   - 闭包 = 环境变量 + 函数
   - 匿名函数最常见的用途是：创建闭包；

1. 闭包实例：

```
> power<-function(exponent){
+ function(x){
+ x^exponent
+ }
+ }
> square<-power(2)
> square(2)
[1] 4
> square(4)
[1] 16
> cube<-power(3)
> cube(2)
[1] 8
> cube(4)
[1] 64
```

创建两个子函数square( )和cube( )

```
> library('pryr')
> unenclose(square)
function (x)
{
    x^2
}
> unenclose(cube)
function (x)
{
    x^3
}
```

查看两个函数的环境内容

1. 函数工厂：其功能是产生新的函数；其适用场景：

- 调用层次更加复杂，有多个参数和复杂的函数主体；
- 有些工作只需要在生成函数时执行一次；

1. 函数可以存储在列表中，这使得处理相关函数更加容易。

```
> compute_mean <- list(
+   base = function(x) mean(x),
+   sum = function(x) sum(x) / length(x),
+   manual = function(x) {
+     total <- 0
+     n <- length(x)
+     for (i in seq_along(x)) {
+       total <- total + x[i] / n
+     }
+     total
+   }
+ )
```

```
> x <- runif(1e5)
> system.time(compute_mean$base(x))
   user  system elapsed
  0.001   0.000   0.002
> system.time(compute_mean[[2]](x))
   user  system elapsed
  0.000   0.002   0.001
> system.time(compute_mean[["manual"]](x))
   user  system elapsed
  0.006   0.000   0.005
```

从列表中调用函数

1. 函数可以存储在列表中，这使得处理相关函数更加容易。

```
> lapply(compute_mean, function(f) f(x))
$base
[1] 0.4985025

$sum
[1] 0.4985025

$manual
[1] 0.4985025
```

```
> call_fun <- function(f, ...) f(...)
> lapply(compute_mean, call_fun, x)
$base
[1] 0.4985025

$sum
[1] 0.4985025

$manual
[1] 0.4985025
```

调用每个函数

1. 除了操作单个函数外，函数运算符还可以接受多个函数作为输入。

2. 一个简单的例子是plyr::each()。它获取一个向量化的函数列表，并将它们组合成一个单独的函数。

```
> summaries <- plyr::each(mean, sd, median)
> summaries(1:10)
    mean       sd   median
5.50000 3.02765 5.50000
>
```

I. 另一种函数组合的形式：f(g(x))

- compose <- function(f, g) {
- function(...) f(g(...))
- }

```
> compose <- function(f, g) {
+   function(...) f(g(...))
+ }
> x<-list(1:10)
> sapply(x,compose(sqrt, sum))
[1] 7.416198
```

- 另一种表达形式：
- (f o g) (x)

```
> compose <- function(f, g) {
+   function(...) f(g(...))
+ }
>
> compose(sqrt, sum)(-4,8)
[1] 2
```

I. 逻辑谓词和布尔代数的组合函数

```
> and <- function(f1, f2) {
+   force(f1); force(f2)
  }
+   function(...) {
}
+     f1(...) && f2(...)
+   }
+ }
  function(...) {
    f1(...) || f2(...)
  }
>
> or <- function(f1, f2) {
+   force(f1); force(f2)
+   function(...) {
+     f1(...) || f2(...)
+   }
+ }
  force(f)
>
  function(...) {
> not <- function(f) {
+   force(f)
+   function(...) {
+     !f(...)
+   }
}+ }
```

- Filter(function(x)
  is.character(x) ||
  is.factor(x), iris)
- Filter(or(is.character,
  is.factor), iris)
- Filter(not(is.numeric), iris)

基于S3的面向对象编程

- 是一种泛型函数的实现方式；
- 泛型函数是一种特殊的函数，根据传入对象的类型决定调研哪个具体的方法；
- 基于S3的面向对象编程，是一种动态函数调用的模拟实现。

基于泛函的S3面向对象

1. 对象模型
   - 类的定义
     - 在基于S3的面向对象系统中，类是一种类型，任何一个对象，都可以增加这样一个类型。并且一个对象可以有多种类型；
     - 定义的格式如下：
       - *attr*(对象名, "*class*")<-"类名“
       - 对象名 <- structure(属性, class = 类名)

```
> a<-list(1,23)
> attr(a,"class")<-"foo"
```

```
> x<-1
> y<-structure(x,class="foo")
```

基于泛函的S3面向对象

1. 对象模型
   - 类的定义：方法的定义
     - 泛函函数的定义，格式如下：
       - 泛型函数名 <- *function*(*x*, ...)
         UseMethod(参数)
     - 泛型函数的分派
       - 泛函函数名.类名<-*function*(参数) 表达式
     - 给类添加行为，格式如下：

```
> teacher<-function(x,...)UseMethod("teacher")
> teacher.lecture<-function(x)print("讲课")
> teacher.assginment<-function(x)print("布置作业")
> a<-'teacher'
> attr(a,'class')<-'lecture'
> teacher(a)
[1] "讲课"
```

基于泛函的S3面向对象

1. **对象模型：继承**
   - 继承方式：提供哪些继承方式？
     - R语言基于S3的面向对象系统有一种非常简单的继承方式，用NextMethod()函数来实现 ；
     - 其具有单继承和多重继承两种继承方式；

```
> node <- function(x) UseMethod("node", x)
> node.default <- function(x) "Default node"
> node.father <- function(x) c("father")
> node.son <- function(x) c("son", NextMethod())
> n1 <- structure(1, class = c("father"))
> n2 <- structure(1, class = c("son", "father"))
> node(n2)
[1] "son"    "father"
```

单继承

通过对*node*()函数传入n_2的参数，*node.son*()先被执行，然后通过NextMethod()函数继续执行了*node.father*()函数。
这样其实就模拟了，子函数调用父函数的过程，实现了面向对象编程中的继承。

基于泛函的S3面向对象

1. **对象模型：继承**
   - 继承方式：提供哪些继承方式？
     - R语言基于S3的面向对象系统有一种非常简单的继承方式，用NextMethod()函数来实现 ；
     - 其具有单继承和多重继承两种继承方式；

```
> node <- function(x) UseMethod("node", x)
> node.father <- function(x) c("father")
> node.son <- function(x) c("son", NextMethod())
> node.grandson <- function(x) c("grandson",NextMethod())
> n1 <- structure(1, class = c("father"))
> n2 <- structure(1, class = c("son", "father"))
> n3 <- structure(1, class = c("grandson","son", "father"))
> node(n3)
[1] "grandson" "son"      "father"
```

多重继承

基于泛函的S3面向对象

1. 对象模型：多态
   - 类属：方法覆盖

通过泛型函数，实现方法的多态机制

```
> node <- function(x) UseMethod("node", x)
> node.father <- function(x,y) c("father")
> node.son <- function(x) c("son", NextMethod())
> node.grandson<- function(x) c("grandson", NextMethod())
> n1 <- structure(1, class = c("father"))
> n2 <- structure(1, class = c("son","father"))
> node(n1)
[1] "father"
> node(n2)
[1] "son"    "father"
> n3 <- structure(1, class = c("grandson","son","father"))
> node(n3)
[1] "grandson" "son"      "father"
```

5.6 问题与讨论

(1) 模拟状态不易

(2) 效率还是问题

- 其原因是:
  - 函数是第一类对象， 局部于它的数据一般要在堆(heap)上分配，为了避免悬挂引用，要有自动重配的检查。
  - 无类型(如LISP)要在运行中检查类型，即使是强类型的(如ML， Miranda)减少了类型动态检查， 但函数式语言天然匹配选择模式的途径也是运行低效原因。
  - 懒求值开销大
  - 中间复合值一多费时费空间。
  - 无限表动态生成，计算一次增长一个元素！效率也很低。

(3) 并发性

- 函数式语言被认为是非常适用于处理并发性问题的工具， 共享值不需加特殊保护， 因为他们不会被更新、并行进程之间不会互相干扰。
- 问题：在将表达式求值分配给不同的处理器这一点上就有隐藏的额外开销： 用于求表达式值的数据必须从一个处理器传到另一个处理器， 而表达式的计算结果还得被传回来。

纯函数式语言：Haskell 语言

Haskell是一种标准化的，通用纯函数式编程语言，有非限定性语义和强静态类型。它的命名源自美国逻辑学家Haskell Brooks Curry，他在数学逻辑方面的工作使得函数式编程语言有了广泛的基础。在Haskell中，函数是头等程序对象。

作为函数式编程语言，主要控制结构是函数。Haskell语言是1990年在编程语言Miranda的基础上标准化的，并且以λ演算（Lambda-Calculus)为基础发展而来。具有“证明即程序、结论公式即程序类型”的特征。这也是Haskell语言以希腊字母「λ」（Lambda）作为自己标志的原因。

Haskell语言的应用：主要应用于服务器端的开发，用于构建并发高速网络服务程序；

混合型的函数式语言

- Scala语言的特点
  - 一种结合了面向对象和函数式编程的语言；
  - 在众多函数式编程语言里，借鉴ML和Haskell最多；
- clojure语言的特点
  - 保持了函数式语言的主要特点，还能够非常方便的调用Java类库的API，和Java类库进行良好的整合；
- Scala和clojure语言的应用
  - Scala主要用于Spark流式大数据处理框架；
  - clojure主要用于Storm流式大数据处理框架；

Clojure 语言

Clojure 是一种运行在 Java™ 平台上的 Lisp 方言，它的出现彻底颠覆了我们在Java虚拟机上并发编程的思考方式改变了这一现状。如今，在任何具备 Java 虚拟机的地方，您都可以利用 Lisp 的强大功能

作为Lisp方言，Clojure或许拥有最灵活的编程模型，因此绝不缺乏号召力。与其他Lisp方言不同的是，它不会带那么多括号，还有众多Java库和在各平台上的广泛部署作为坚强后盾

Scheme 语言

Scheme 语言是 Lisp 的一个现代变种、方言，诞生于1975年，由 MIT 的 Gerald J. Sussman and Guy L. Steele Jr. 完成。与其他lisp不同的是，scheme是可以编译成机器码的 。

Scheme语言的规范很短，总共只有50页，甚至连Common Lisp 规范的索引的长度都不到，但是却被称为是现代编程语言王国的皇后。它与以前和以后的 Lisp 实现版本都存在一些差异，但是却易学易用。

具备部分函数式语言的特性的语言

- Python语言的特点
  - Python中函数式编程的基本要素包括functionsmap()、reduce()、filter()和lambda算子（operator）；
- Ruby语言的特点
  - Ruby虽然是一个完全面向对象的语言；
  - 函数式编程的要素包括：匿名函数、表达式求值、柯里化、组合等；

总结：函数式语言的优点

- 高模块；
- 可复用；
- 简化并发编程；