Real World Functional Programming

天涯古巷 出品

Lecture 1

Administrative Details and Introduction

1. Content

- Lazy functional programming
- Purely functional data structures
- Key libraries
- Functional design patterns
- Concurrency
- GUIs

2. Assessment

- 50 % unseen written examination (1.5 h)
- 50 % coursework. 2 parts:
  - Part I: Basics; 15 h
    - Release: Wednesday 16 October
    - Deadline: Wednesday 6 November
  - Part II: Advanced topics and applications; 35 h
    - Release: Wednesday 6 November
    - Deadline: Wednesday 11 December

3. Real-world Functional Programming Project

- Release of project criteria: Wednesday 13 November
- Pitch deadline: Wednesday 4 December (but earlier is better)
- Submission deadline (code and report): 15 January

4. 命令式（Imperative）与声明式（Declarative）语言的区别

Imperative Language（命令式语言）：

- Implicit state（隐含状态）.
- Computation essentially a sequence of side-effecting actions.
- Examples: Procedural and OO languages（面向对象语言）

Declarative Language（声明式语言）：

- No implicit state.
- A program can be regarded as a theory.
- Computation can be seen as deduction from this theory.
- Examples: Logic and Functional Languages.

5. Another perspective:

- Algorithm = Logic + Control
- Strategy needed for providing the "how":

- Declarative programming emphasises the logic ("what") rather than the control ("how"). 告诉"机器"你想要的是什么(what)，让机器想出如何去做(how)
- Resolution (logic programming languages)
- Lazy evaluation (some functional and logic programming languages)
- (Lazy) narrowing: (functional logic programming languages)

6. 声明式语言的控制

Declarative languages for practical use tend to be only weakly declarative; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed. （方程在函数式语言中直接表达）
- Order of patterns often matters for pattern matching.
- Constructs for taking control over the order of evaluation.

7. 声明式语言的 Relinquishing（放弃） Control

Relinquishing control by exploiting lazy evaluation（通惰性计算来放弃控制）.

- Evaluation orders
- Strict vs. Non-strict semantics
- Lazy evaluation
- Applications of lazy evaluation:
  - Programming with infinite structures
  - Circular programming
  - Dynamic programming
  - Attribute grammars

8. Evaluation Orders（求值顺序）

Reducible Expression or **redex**（可化简表达式）：any expression that can be evaluated（计算） or reduced（化简） by using the equations（方程式） as rewrite rules.

```
sqr x = x * x
dbl x = x + x
main = sqr (dbl (2 + 3))
```

（**1**）**Applicative Order Reduction (AOR)**：Innermost, leftmost redex first (自左往右、从里往外化简)

```
main ⇒ sqr (dbl (2 + 3))
⇒ sqr (dbl 5)
⇒ sqr (5 + 5)
⇒ sqr 10
⇒ 10 * 10
⇒ 100
```

This is **Call-By-Value**（引值调用）：在调用一个参数前，需要先把参数的值求出.

（**2**）**Normal Order Reduction (NOR，常序化简）**：Outermost, leftmost redex first (自左往右、从外往里化简)

```
main ⇒ sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * dbl (2 + 3)
⇒ ((2 + 3) + (2 + 3)) * dbl (2 + 3)
⇒ (5 + (2 + 3)) * dbl (2 + 3)
⇒ (5 + 5) * dbl (2 + 3)
⇒ 10 * dbl (2 + 3)
⇒ ... ⇒ 10 * 10
⇒ 100
```

This is Demand-driven evaluation or **Call-By-Need**（按名调用）：在应用函数时，不必计算参数的值，而是将参数直接替换到函数体当中，如果这个参数在函数体中没有参与计算，那么它就不会被计算。

9. NOR 的作用

• Best possible termination properties.

A pure functional language is just the λ-calculus in disguise. Two central theorems:

- Church-Rosser Theorem I: No term has more than one normal form.
- Church-Rosser Theorem II: If a term has a normal form, then NOR will find it.

• More expressive power;

e.g.: - "Infinite" data structures

- Circular programming

• More declarative code as control aspects (order of evaluation) left implicit.

10. Strict vs. Non-strict Semantics

• ⊥, or "bottom", the undefined value, representing errors and non-termination.

A function f is strict iff（iff 指当且仅当）: f ⊥ = ⊥

For example:

+ is strict in both its arguments:

(0/0)+1 =⊥+1 =⊥

1+(0/0) = 1+⊥=⊥

Again, consider:

f x =1 g x = g x. (表示 g x 为 undefined,即 g = ⊥)

What is the value of f (0/0)? Or of f (g 0)?

• AOR: f(0/0)⇒⊥; f(g 0)⇒⊥ Conceptually, f ⊥ = ⊥; i.e., f is strict.

• NOR: f(0/0)⇒1; f(g 0)⇒1 Conceptually, foo ⊥ = 1; i.e., foo is non-strict.

Thus, **NOR results in non-strict semantics**.

11. Lazy Evaluation (惰性求值)

Lazy evaluation is a technique for implementing NOR more efficiently:

- A redex is evaluated only if needed.
- Sharing employed to avoid duplicating redexes.
- Once evaluated, a redex is updated with the result to avoid evaluating it more than once.

As a result, under lazy evaluation, any one redex is evaluated at most once.

Recall:

```
sqr x = x * x
dbl x= x+x
main = sqr (dbl (2 + 3))
```

Lazy Evaluation:

```
sqr(dbl(2+3))
⇒dbl(2+3) * (•)
⇒((2+3) +(•)) * (•)
⇒ ( 5 + (•)) * (•)
⇒10 *(•)
⇒ 100
```

注意：

"Evaluated at most once" referes to **individual redex instances**.

For example:

- (1 + 2) * (1 + 2)

  1 + 2 evaluated twice as not the same redex.

- f x = x + y where y = 6 * 7

  6 * 7 evaluated whenever f is called.

A good compiler will rearrange such computations to avoid duplication of effort, but this has nothing to do with laziness.

Memoization（性能优化） means caching function results to avoid re-computing them. Also distinct from laziness.

Exercise

Evaluate main using AOR, NOR, and lazy evaluation:

```
f x y z = x*z
g x = f (x*x) (x*2) x
main = g (1 + 2)
```

How many reduction steps in each case?

Answer: 7, 8, 6 respectively

AOR:

starting from main:

```
main
⇒ g (1 + 2)
⇒ g (3)
⇒ f (3*3) (3*2) 3
⇒ f 9 (3*2) 3
⇒ f 9 6 3
⇒ 9*3
⇒ 27                                   7 steps in total.
```

NOR:

starting from main:

```
main
⇒ g (1 + 2)
⇒ f ((1+2)*(1+2)) ((1+2)*2) (1+2)
⇒ ((1+2)*(1+2))*(1+2)
⇒ (3*(1+2)) * (1+2)
⇒ (3*3) * (1+2)
⇒ (3*3) * 3
⇒ 9 * 3
⇒ 27                                   8 steps in total.
```

Lazy Evaluation:

starting from main:

main

⇒ g (1 + 2)

⇒f ((1+2)*(•)) (•)*2 (•)

⇒ ((1+2)*(•))*(•)

⇒ (3*(•))*(•)

⇒ 9*(•)

⇒ 27 6 steps in total.

Lecture 2&3

Pure Functional Programming: Exploiting Laziness

1. Infinite Data Structures 无穷数据结构

1.1 Prelude 中递归函数 take 的定义：

```
take 0 _ = []                          -- 定义递归基本条件
take n [] = []                         -- 定义递归基本条件
take n (x:xs) = x : take (n-1) xs      -- 定义递归步骤
```

作用：从列表 xs 中，取出从 0 开始的前 n 个元素构成一个新的列表

递归步骤：如 take 3 [1,2,3,4,5]

```
  take 3 [1,2,3,4,5]
= take 3 (1:[2,3,4,5])          --由(:)的定义得
= 1:(take 2 [2,3,4,5])          --由递归步骤定义得
= 1:(take 2 (2:[3,4,5]))
= 1:(2:(take 1 [3,4,5]))
= 1:(2:(take 1 (3:[4,5])))
= 1:(2:(3:(take 0 [4,5])))
= 1:(2:(3:[]))
= 1:(2:[3])
= 1:[2,3]
= [1,2,3]
```

1.2 定义构造无穷长的列表的 from 函数：

```
from n = n : from (n+1)
```

递归步骤：如 from 0

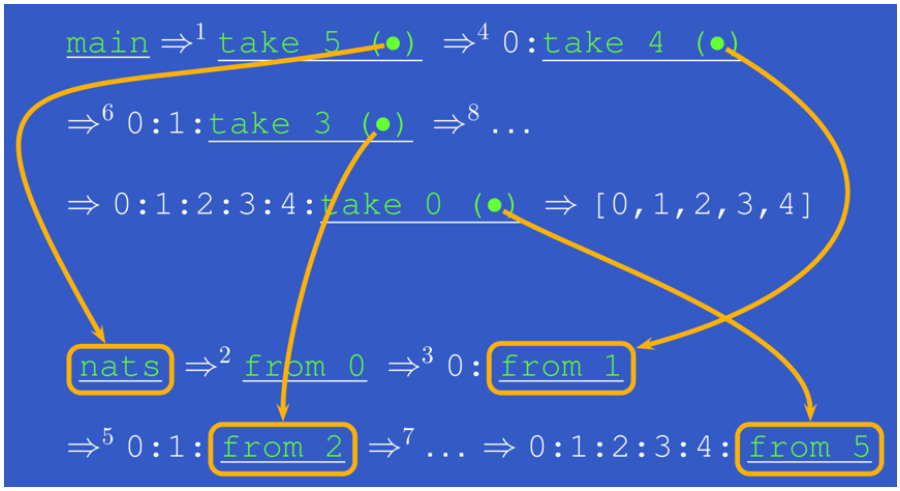
```
  from 0 = 0:from 1 = 0:(1:from 2)
= 0:(1:(2:from 3)) = ...
= [0,1,2,3,4..]
```

1.3 无穷数据结构中的惰性计算

定义无穷长自然数（nature）列表 nats 与主函数 main：

```
nats = from 0
main = take 5 nats
```

惰性计算：



2. Circular Data Structures 循环数据结构

2.1 定义数字 1 循环的无穷长的列表的 ones 函数：

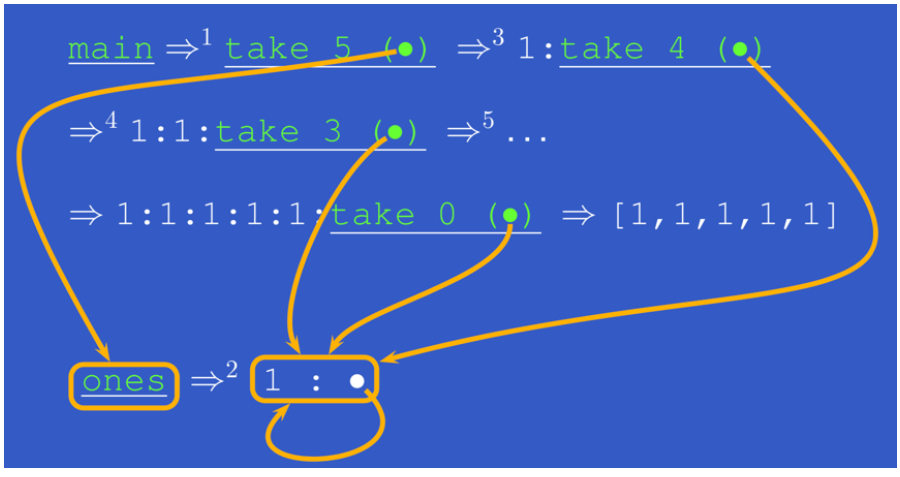
```
ones = 1 : ones
```

2.2 循环数据结构中的惰性计算

定义主函数 main：

```
main = take 5 ones
```

惰性计算：



3. 数据类型定义

Given the following tree type:

```
data Tree = Empty | Node Tree Int Tree
```

使用关键字 data 通过枚举递归定义来定义一个名为 Tree 的数据类型，该类型为杂合定义类型。该树的类型可以是基本形式——“空”，也可以是一个递归的形式，即一个节点，节点有自己的权值（Int），节点后跟两颗子树：左子树（Tree）和右子树（Tree）。其中“Empty”和“Node”为数据构造器：

```
Empty :: Tree
Node :: Tree -> Int -> Tree -> Tree
```

Define:

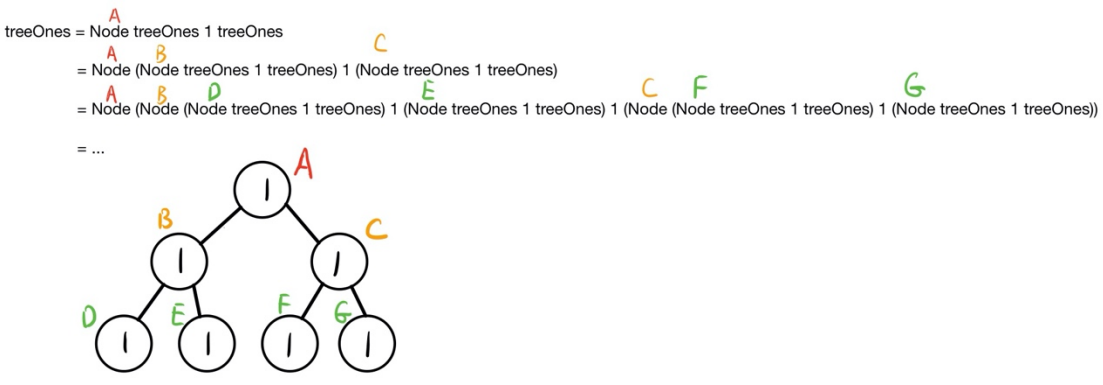
(1) An infinite tree where every node is labelled by 1.

(2) An infinite tree where every node is labelled by its depth from the root node.

Answer:

(1) 通过定义无参递归函数（treeOnes）和数据构造器（Node 构造函数）来递归生成：

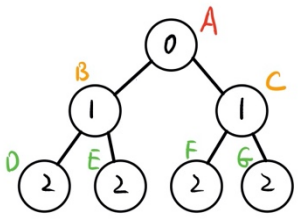
```
treeOnes = Node treeOnes 1 treeOnes
```



(2)

```
treeFrom n = Node (treeFrom (n + 1)) n (treeFrom (n + 1))
treeDepths = treeFrom 0
```

```
treeDepths
= treeForm 0
= Node (treeForm 1) 0 (treeFrom 1)
= Node (Node (treeFrom 2) 1 (treeFrom 2)) 0 (Node (treeFrom 2) 1 (treeFrom 2))
= Node (Node (Node (treeFrom 3) 2 (treeFrom 3)) 1 (Node (treeFrom 3) 2 (treeFrom 3)) 0 (Node (Node (treeFrom 3) 2 (treeFrom 3)) 1 (Node (treeFrom 3) 2 (treeFrom 3))
= ...
```



4. Circular Programming 循环编程

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

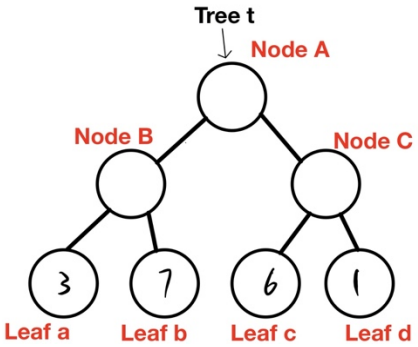
非空树数据类型，该树的类型可以是基本形式——“叶子”，也可以是一个递归的形式，即一个节点，除叶子有权值（Int）外，其他节点没有权值（Int），节点后跟两颗子树：左子树（Tree）和右子树（Tree）。其中“Leaf”和“Node”为数据构造器。

Suppose we would like to write a function that replaces each leaf integer in a given tree with the smallest integer（记为 m） in that tree.

How many passes(步骤) over the tree are needed?

Answer： One

假设已知一棵树 Tree t 如下：



Write a function that replaces all leaf integers by a given integer, and returns the new tree along with the smallest integer of the given tree:

```
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) =  (Node tl' tr', min ml mr) where
    (tl', ml) = fmr m tl
    (tr', mr) = fmr m tr
```

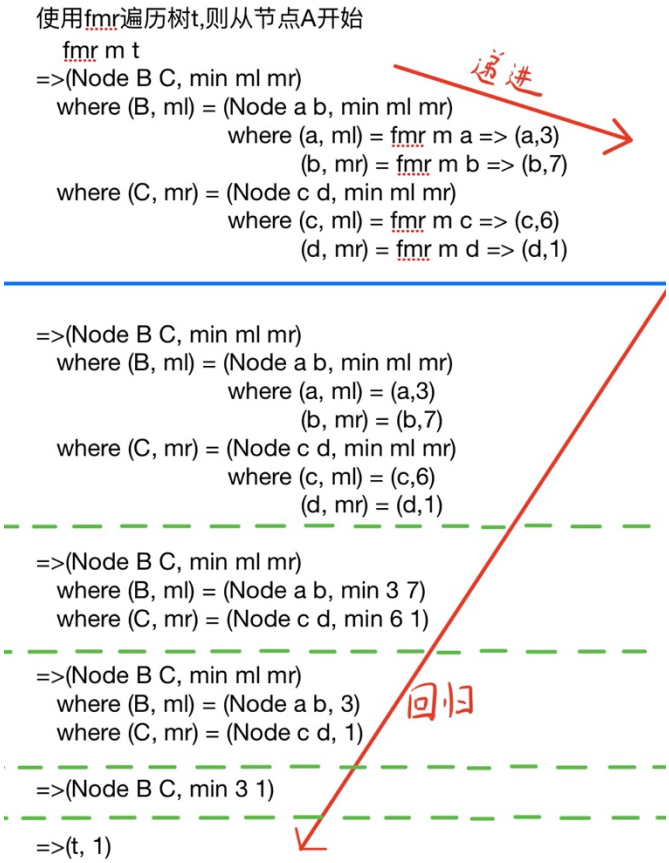
For a given tree t, the desired tree is now obtained as the solution to the equation:

```
{- 只需要这一步就可以实现，故 only one pass needed -}
(t', m) = fmr m t
```

Thus:

```
findMinReplace t = t' where (t', m) = fmr m t
```

递归过程如下：



Intuitively, this works because fmr can compute its result without needing to know the value of m. 由于 Haskell 中的惰性计算，fmr 函数可以在不知道最小值 m 的情况下运行。

5. A Simple Spreadsheet Evaluator 表格计算

5.1 Data.Array 模块

Array 属于类型的 kind,需要填入两个类型才可以使用

```
Prelude Data.Array> :k Array
Array :: * -> * -> *
```

5.2 表格计算

	a	b	c
1	c3 + c2		a3
2	a3 * b2	2	a2 + b2
3	7		a2 + b3

s

	a	b	c
1	37		7
2	14	2	16
3	7		21

s'

- 用 a1、a2、a3、b1…等 char 与 Int 的二元组来表示每个格子的位置
- 格子里的表达式共有三种情况：值(Lit)、索引(Ref)、索引表示的待计算式(App)

(1) 数据结构定义

```
import Data.Array
--将二元组重命名为 CellRef(CellReference)
type CellRef = (Char, Int)
--将 Array CellRef a 类型重命名为 Sheet a
type Sheet a = Array CellRef a
--定义表格的操作类型
data BinOp = Add | Sub | Mul | Div
--定义表达式类型
data Exp = Lit Double                  --表达式为一个确定的值
         | Ref CellRef                 --表达式为一个索引
         | App BinOp Exp Exp.          --表达式为一个索引表示的待计算式
```

(2) 计算单个格子

```
--定义表格中计算的数据类型为 Double
evalCell :: Sheet Double -> Exp -> Double
--当匹配的表达式为一个数时，直接返回这个数
evalCell _ (Lit v) = v
--当匹配的表达式为一个索引时，用 Data.Array 中的(!)运算符求出索引 r 的值
evalCell s (Ref r) = s ! r
--当匹配的表达式为一个索引表示的待计算式时，计算该表达式
evalCell s (App op e1 e2) = (evalOp op) (evalCell s e1) (evalCell s e2)
evalOp :: BinOp -> (Double -> Double -> Double)
evalOp Add = (+)
evalOp Sub = (-)
evalOp Mul = (*)
evalOp Div = (/)
```

(3) 计算整个表格

```
evalSheet :: Sheet Exp -> Sheet Double
--使用 array 函数构造一个新的数组（表格）
evalSheet s = s'
    where
        s' = array (bounds s) [ (r, evalCell s' (s ! r)) | r <- indices s ]
```

(4) 测试表格计算

给定一个正常的测试表格：

```
testSheet :: Sheet Exp
testSheet = array (('a', 1), ('c', 3))
                  [ (('a', 1), Lit 1.0),
                    (('a', 2), Ref ('b', 1)),
                    (('a', 3), App Add (Ref ('a',1)) (Ref ('a',2))),
                    (('b', 1), App Add (Ref ('c', 2)) (Ref ('b', 2))),
                    (('b', 2), Lit 2.0),
```

```
                  (('b', 3), App Add (Ref ('b',1)) (Ref ('b',2))),
                  (('c', 1), Lit 3.0),
                  (('c', 2), Lit 4.0),
                  (('c', 3), App Mul (Ref ('a',1)) (Ref ('b',3)))
                ]
```

测试结果：

```
Prelude Data.Array> evalSheet testSheet
array (('a',1),('c',3)) [(('a',1),1.0),(('a',2),6.0)
,(('a',3),7.0),(('b',1),6.0),(('b',2),2.0),(('b',3),
8.0),(('c',1),3.0),(('c',2),4.0),(('c',3),8.0)]
```

给定一个 bad 测试表格：

```
badSheet :: Sheet Exp
badSheet   = array (('a', 1), ('c', 3))
                [ (('a', 1), Lit 1.0),
                  (('a', 2), Ref ('b', 1)),
                  (('a', 3), App Add (Ref ('a',1)) (Ref ('a',2))),
                  (('b', 1), App Add (Ref ('c', 2)) (Ref ('b', 2))),
                  (('b', 2), Lit 2.0),
                  (('b', 3), App Add (Ref ('b',1)) (Ref ('b',2))),
                  (('c', 1), Ref ('c', 2)),
                  (('c', 2), Ref ('c', 1)),
                  (('c', 3), App Mul (Ref ('a',1)) (Ref ('b',3)))
                ]
```

测试结果：

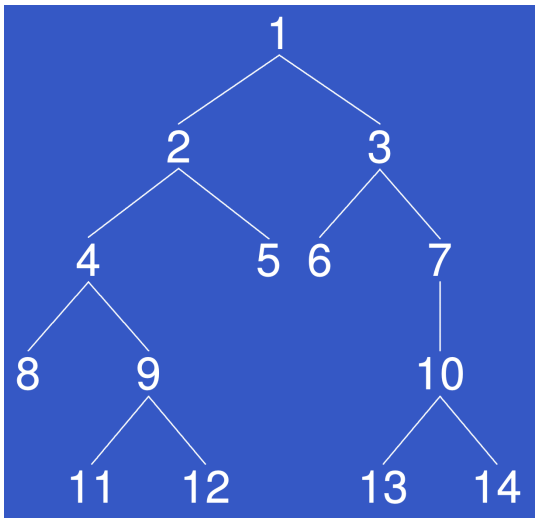
```
Prelude Data.Array> evalSheet badSheet
array (('a',1),('c',3)) [(('a',1),1.0),(('a',2),
```

程序一直在计算，无法停止也无法得出结果

6. Breadth-first Numbering 广度优先编号

6.1 树的广度优先算法（先根遍历）

Consider the problem of numbering a possibly infinitely deep tree in breadth-first order: 给一颗无穷树按广度优先算法编号



(1)定义树的数据结构

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

(2)定义编号函数

定义 width 函数：width t i，返回第 i 层的节点个数(即：第 i 层树的宽度)

```
--The width of a tree t at level i (0 origin).
width t i
```

定义 label 函数：label t i j，返回第 i 层中第 j 个标签的编号

```
--The jth label at level i of a tree t (0 origin).
label t 0 0 = 1   --根节点编号为 1
--每一层第一个节点的编号 = 上一层第一个节点的编号 + 上一层的宽度
label t (i+1) 0 = label t i 0 + width t i
--每一层第 j 的节点的编号 = 该层前一个节点的编号 + 1
label t i (j+1) = label t i j + 1
```

(3)编号过程

第 0 层：i=0

只有一个节点，故 j=0： label t 0 0 = 1 --编号为 1

----------第 1 层编号结束----------

第 1 层：i=1

第 0 个节点：j=0：label t 1 0

= label t (0+1) 0

= label t 0 0 + width t 0

--第 0 层树的宽度为 0，故 width t 0 = 1

= 1 + 1 = 2 --编号为 2

第 1 个节点：j=1：label t 1 1

= label t 1 (0+1)

= label t 1 0 + 1

= 2 + 1 = 3 --编号为 3

----------第 2 层编号结束----------

第 2 层：i=2

第 0 个节点：j=0：label t 2 0

= label t (1+1) 0

= label t 1 0 + width t 1

--第 1 层树的宽度为 2，故 width t 0 = 2

= 2 + 2 = 4 --编号为 4

第 1 个节点：j=1：label t 2 1

= label t 2 (0+1)

= label t 2 0 + 1

= 4 + 1 = 5 --编号为 5

第 2 个节点：j=2：label t 2 2

= label t 2 (1+1)

= label t 2 1 + 1

= 5 + 1 = 6 --编号为 6

第 3 个节点：j=3：label t 2 3

= label t 2 (2+1)

= label t 2 2 + 1

= 6 + 1 = 7 --编号为 7

----------第 3 层编号结束----------

第 3 层：i=3

第 0 个节点：j=0：label t 3 0

= label t (2+1) 0

= label t 2 0 + width t 2

--第 2 层树的宽度为 4，故 width t 2 = 4

= 4 + 4 = 4 --编号为 8

……(以下编号过程省略)

6.2 通过惰性计算来实现广度优先编号

The code that follows sets up the defining system of equations: 以下代码建立方程式的定义系统：

• Streams (infinite lists) of labels are used as a mediating data structure to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next. 标签流（即：无限列表）用作中间数据结构，以允许在层级内的相邻节点之间以及一个层级的最后一个节点与下一个层级的第一个节点之间建立方程式。

• Idea: the tree numbering function for a subtree takes a stream of labels for the first node at each level, and returns a stream of labels for the node after the last node at each level. 想法：子树的编号函数接收的标签流给每层的第一个节点，并返回每层的最后一个节点之后的节点的标签流。

• As there manifestly are no cyclic dependences among the equations, we can entrust the details of solving them to the lazy

evaluation machinery in the safe knowledge that a solution will be found. 由于方程之间显然没有循环依赖关系，因此在可以找到解决方案的安全认知下，我们可以将求解它们的细节委托给惰性计算机制。

(1)定义广度优先编号函数 bfn

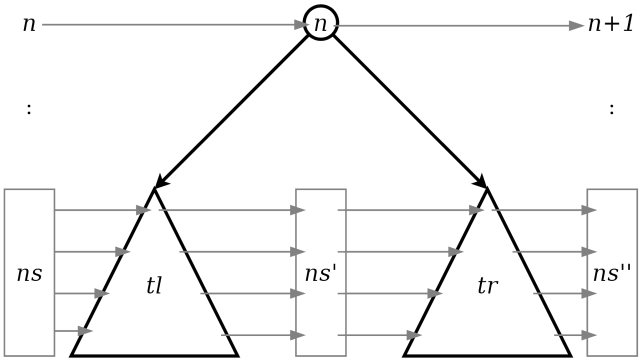
```
bfn :: Tree a -> Tree Integer
bfn t = t'
    where
        (ns, t') = bfnAux (1 : ns) t
```

(2)定义广度优先编号辅助函数 bfnAux

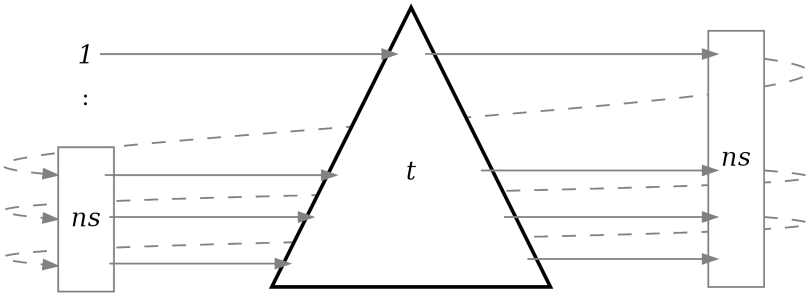
```
bfnAux :: [Integer] -> Tree a -> ([Integer], Tree Integer)
bfnAux ns Empty = (ns, Empty)
bfnAux (n:ns) (Node tl _ tr)=((n+1):ns'', Node tl' n tr')
    where
        (ns', tl') = bfnAux ns tl
        (ns'', tr') = bfnAux ns' tr
```

(3)递归图解

对于每个节点的编号：



对于整棵树的编号：



(4)编号过程

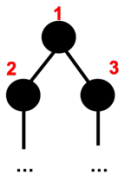
算法思想：

①定义一个用来装树的每个节点编号(label)的无穷列表 ns，即[1,2,3...].

②将这个列表 ns 以及需要编号的树 t 作为参数，通过编号辅助函数 bfnAux 来完成编号，bfnAux 返回一个二元组，这个二元组里装有无穷列表 ns 经过编号操作之后的得到的列表 ns''（ns 在经过给左子树编号后得到 ns'，ns'再给右子树编号后得到 ns''）以及编号之后的树 t'.

③对每个节点的编号：首先把 ns 的第一个元素 n 取出，将节点编号为元素 n 的值。

e.g. 以下图为例：



遍历根节点(n=1)及其左右子树：在给根节点编号时，bfnAux 将 ns [1,2,3,4,5...]写为 1:[2,3,4,5..]，取出第一个元素 1 编号给根节点，剩下的列表[2,3,4,5...]写为 2:[3,4,5..]，取出第一个元素 2 编号给根节点的左子树的根节点，剩下的列表记为 ns' [3,4,5..]，写为 3:[4,5..]，取出第一个元素 3 编号给根节点的右子树的根节点，剩下的列表记为 ns'' [4,5..]，此时 bfnAux 返回一个二元组，这个二元组的中新的列表 ns 为(n+1):ns''，即新列表 ns 为 [2,4,5..]

遍历第二个根节点(n=2)及其左右子树：在对第一层编号完成之后，bfnAux 把编号为 2 的左子树作为新的根节点进行编号，将新的列表 ns [2,4,5]写为 2:[4,5]，取出第一个元素 2 对已经编过号且编号为 2 的根节点再次编号为 2，剩下的列表[4,5..]写为 4:[5..]，取出第一个元素 4 编号给左子树，剩下 ns' [5..]，取出 5 编号给右子树，剩下 ns'' [6..]... 返回新的列表 ns [3,6..]

以此类推...

7. Dynamic Programming 动态规划

(1)定义

Dynamic Programming:

- Create a table of all subproblems that ever will have to be solved. 创建所有必须解决的子问题列表。
- Fill in table without regard to whether the solution to that particular subproblem will be needed. 填写表格，而不考虑是否需要解决该特定子问题的解决方案
- Combine solutions to form overall solution. 合并解决方案以形成整体解决方案

Lazy Evaluation is perfect match: no need to worry about finding a suitable evaluation order.惰性计算的特点非常适用于动态规划，不用担心找到合适的计算顺序。

In effect, using laziness to implement limited form of memoization. 实际上，通过惰性计算来实现有限形式的存储。

(2)惰性计算适合动态规划的原因

①The equations that define the solution can be transliterated directly into the language without any need to be explicit about the order in which the subproblems are solved, or indeed exactly which subproblems need to be solved. 定义解决方案的方程式可以直接转换成语言，而无需明确说明子问题的解决顺序，或者确切地说需要解决哪些子问题。

②The demand-driven aspect of lazy evaluation ensures that the solutions are computed as and only if needed, at most once, in an appropriate order. Having the order induced automatically by demand is a major advantage compared with typical solutions in strict languages (imperative or not). 具有需求驱动特点的懒惰计算可确保仅在需要时才按适当顺序最多计算一次解决方案。与严格的语言（强制性或非强制性）中的典型解决方案相比，由需求自动生成顺序是一个主要优势。

③However, note that it still is necessary to tabulate the solutions: the "evaluate at most once" aspect of lazy evaluation applies to individual, physical redexes. It does not mean that only one of a number of physically distinct but identical redexes are evaluated, and the others automatically updated with the result. 但是，请注意，仍然有必要列出解决方案的表格："懒惰计算"至多计算一次"的特点适用于单个的物理 redex。这并不意味着仅计算许多物理上不同但完全相同的 redex 中的一个，而其他的则根据结果自动更新。

④That effect can be achieved by memoization of functions. Such a facility can be implemented in typical lazy language implementations, but generally would have to be invoked explicitly for functions of interest. It is not the default.通过记忆功能可以达到这种效果。可以在典型的惰性语言实现中实现这种功能，但是通常必须针对感兴趣的功能显式调用此功能。这不是默认值。

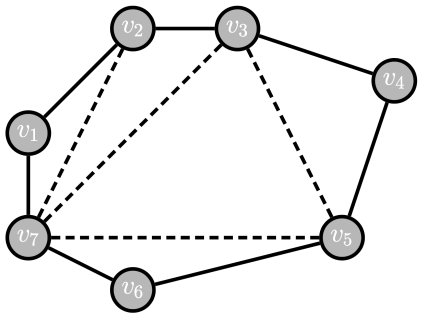
8. The Triangulation Problem (动态规划问题)

(1)问题描述

Select a set of chords that divides a convex polygon into triangles such that: 选择一组将凸多边形分成三角形的弦

- no two chords cross each other 任何两条弦都不交叉
- the sum of their length is minimal. 弦的长度和最小

We will only consider computing the minimal length.



(2)算法

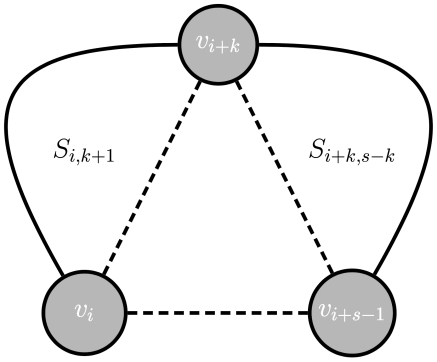
• Let S_{is} denote the subproblem of size s starting at vertex v_i of finding the minimum triangulation of the polygon $v_i, v_{i+1}, \ldots, v_{i+s-1}$ (counting modulo the number of vertices).

• Subproblems of size less than 4 are trivial.

• Solving S_{is} is done by solving $S_{i,k+1}$ and $S_{i+k,s-k}$ for all $k, 1 \le k \le s-2$

• The obvious recursive formulation results in 3^{s-4} (non-trivial) calls

• But for $n \ge 4$ vertices there are only n(n − 3) non-trivial subproblems!



• Let C_{is} denote the minimal triangulation cost of S_{is}.

• Let $D(v_p, v_q)$ denote the length of a chord between v_p and v_q (length is 0 for non-chords; i.e. adjacent v_p and v_q).

• For $s \ge 4$:

$$C_{is} = \min_{k \in [1,s-2]} \{C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1})\}$$

• For $s<4$, $S_{is} = 0$

(3)代码实现

These equations can be transliterated straight into Haskell:

```
triCost :: Polygon -> Double
triCost p = cost ! (0,n) where
    cost = array ((0,0), (n-1,n))
                 ([ ((i,s),
                    minimum [cost!(i, k+1)
                             + cost!((i+k) `mod` n, s-k)
                             + dist p i ((i+k) `mod` n)
                             + dist p ((i+k) `mod` n)
                                      ((i+s-1) `mod` n)
                             |k <- [1..s-2] ])
                | i <- [0..n-1], s <- [4..n] ] ++
                [ ((i,s), 0.0) | i <- [0..n-1], s <- [0..3] ])
    n = snd (bounds b) + 1
```

9. Attribute Grammars 属性文法

Lazy evaluation is also very useful for evaluation of Attribute Grammars:

- The attribution function is defined recursively over the tree:
  - takes inherited attributes as extra arguments;
  - returns a tuple of all synthesised attributes.
- As long as there exists some possible attribution order, lazy evaluation will take care of the attribute evaluation.

Lecture 4

Purely Functional Data Structures

1. Purely Functional Data structures 纯函数式数据结构

1.1 为什么需要纯函数式数据结构

Standard implementations of many data structures rely on imperative update. But: 许多数据结构的标准实现都依赖命令式更新，但是

- In a pure functional setting, we need pure alternatives. 在纯函数式编程中，我们需要另寻纯函数式的数据结构方法。
- In concurrent or distributed settings, side effects are not your friends. Purely functional structures can thus be very helpful! 在并发或分布式的设定中，我们需要尽量避免 side effects。 因此，使用纯函数数据结构非常有帮助。

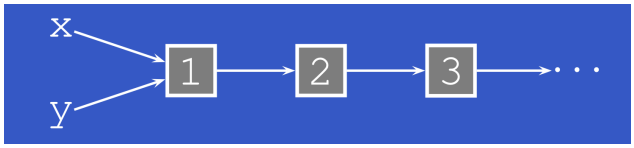
1.2 纯函数式数据结构与命令式语言的数据结构的区别

(1)主要区别

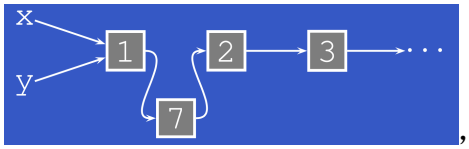
- Imperative data structures are ephemeral: a single copy gets mutated whenever the structure is updated. 命令性数据结构是短暂的：每当更新结构时，单个副本都会发生改变。
- Purely functional data structures are persistent: a new copy is created whenever the structure is updated, leaving old copies intact. (Common sub-parts can be shared.) 纯功能数据结构是持久性的与不变性（immutablity）的：每当结构更新时都会创建一个新副本，而完整保留旧副本。（公共子部分可以共享。）

(2)区别举例

比如，有个链表 x 如下：当我们向头节点插入一个节点 7

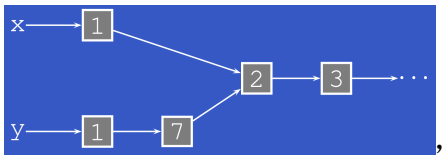


①在命令式语言的数据结构中：

链表变为： 

，原链表 x[1,2,3]丢失，只剩下更新后的链表[1,7,2,3].

②在纯函数式数据结构中：

链表变为： 

，原链表 x[1,2,3]依然存在，相反，我们会创建一个全新的链表 y[1,7,2,3].

(3)不变性 mmutablity 的好处

不可变的数据结构能让我们：

①保持老的数据结构不变的情况下，更新数据结构

②线程安全：一个线程对数据结构的更改不会影响另外一个线程

2. Numerical Representations 数值表示

(1)列表的数据结构：

```
data List a = Nil | Cons a (List a)
tail (Cons _ xs) = xs   --求去掉列表第一个元素后剩下的列表
append Nil ys = ys   --往列表 xs 添加另一个列表 ys 的元素
append (Cons x xs) ys = Cons x (append xs ys)
```

(2)自然数的数据结构：

```
data Nat = Zero | Succ Nat
pred (Succ n) = n --求前一个数
plus Zero n = n
plus (Succ m) n = Succ (plus m n)
```

This analogy can be taken further for designing container structures because: 在设计容器结构时，可以进一步类比，因为

- inserting an element resembles incrementing a number 插入一个元素类似于递增一个数字

• combining two containers resembles adding two numbers etc.
组合两个容器就像两个数字相加，等等。

Thus, representations of natural numbers with certain properties induce container types with similar properties. Called Numerical Representations. 具有一些属性的自然数 induce 了容器类型一些类似属性

3. Random Access Lists 随机存取列表

(1)随机存取列表数据结构

```
data RList a
```

(2)该列表相关函数的类型签名

```
empty :: RList a
isEmpty:: RList a -> Bool
cons :: a -> RList a -> RList a --往列表开头添加一个元素
head :: RList a -> a --返回列表的第一个元素
tail :: :: RList a -> RList a --返回列表除去第一个元素后剩下的列表
lookup :: Int -> RList a -> a
update :: Int -> a -> RList a -> RList a
```

4. Positional Number Systems 进位制

• A number is written as a sequence of digits $b_0b_1 \ldots b_{m-1}$, where $b_i \in D_i$ for a fixed family of digit sets given by the positional system.

• b_0 is the least significant digit, b_{m-1} the most significant digit (note the ordering 注意排序：b_0 为最低位，b_{m-1} 为最高位).

• Each digit b_i has a weight w_i. Thus:

$$\text{value}(b_0b_1 \ldots b_{m-1}) = \sum_{0}^{m-1} b_i w_i$$

where the fixed sequence of weights w_i is given by the positional system.

• A number is written in base B if $w_i = B^i$ and $D_i = \{0,...,B-1\}$.

• The sequence w_i is usually, but not necessarily, increasing.

• A number system is redundant(多余的) if there is more than one way to represent some numbers (disallowing trailing zeroes 不允许后面跟零).

• A representation of a positional number system can be dense, meaning including zeroes, or sparse, eliding zeroes. 进位制的表示可以是密集的，意味着包括零或稀疏的零。

【exercise 1】

Suppose $w_i = 2^i$ and $D_i = \{0, 1, 2\}$. Give three different ways to represent 17.

答：• 10001, since value(10001) = $1 \cdot 2^0 + 1 \cdot 2^4$

• 1002, since value(1002) = $1 \cdot 2^0 + 2 \cdot 2^3$

• 1021, since value(1021) = $1 \cdot 2^0 + 2 \cdot 2^2 + 1 \cdot 2^3$

• 1211, since value(1211) = $1 \cdot 2^0 + 2 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$

5. From Positional Number Systems to Container

(1)进位制与数值容器的关联

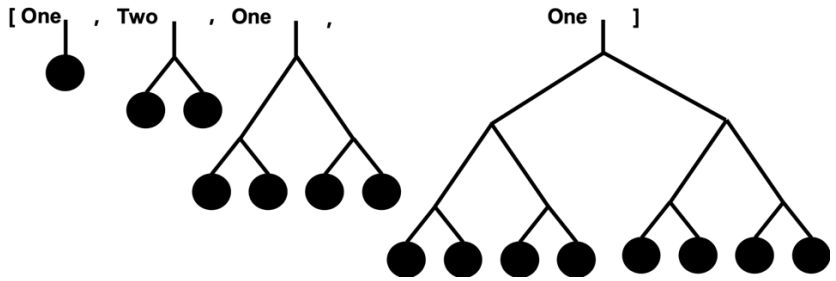
Given a positional system, a numerical representation may be derived as follows:给定一种进位制，数值的容器可表示如下

• for a container of size n, consider a representation $b_0 b_1 . . . b_{m-1}$ of n, 对于大小为 n 的容器，用 $b_0 b_1 . . . b_{m-1}$ 表示 n

• represent the collection of n elements by a sequence of trees of size w_i such that there are b_i trees of that size. 用一组大小为 w_i 的有序的树的集合来表示 n，这样每棵树就能够分别表示每个数位 b_i。

For example, given the positional system of exercise 1, a container of size 17 might be represented by 1 tree of size 1, 2 trees of size 2, 1 tree of size 4, and 1 tree of size 8 (即数值 17 的容器可以表示为

1211，通过用有序的四棵树的集合来表示：一颗大小为 1 的树+两颗大小为 2 的树+一颗大小为 4 的树+一颗大小为 8 的树).



(2)树的类型的选择

The kind of tree should be chosen depending on needed sizes and properties. Two possibilities:

- Complete Binary Leaf Trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

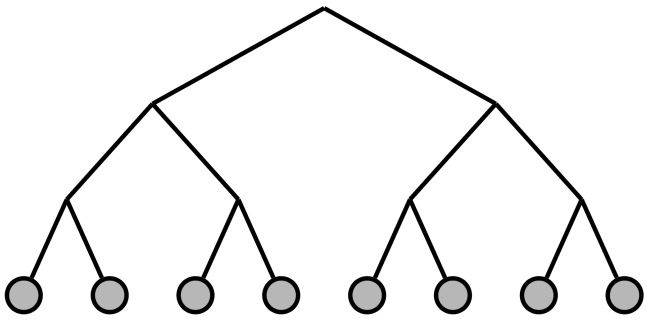
Sizes: 2^n, $n \geq 0$

- Complete Binary Trees

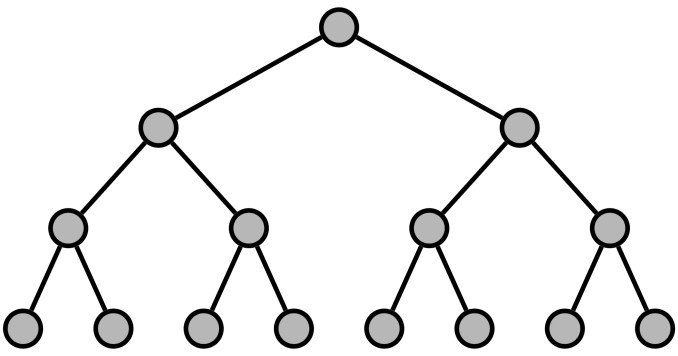
```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

Sizes: $2^{n+1} - 1$, $n \geq 0$

若这个数表示为 2^n(如 $8=2^3$),则选择 Complete Binary Leaf Trees：



若这个数表示为 2^{n+1}-1(如 $15=2^4$-1),则选择 Complete Binary Trees：



6. Binary Random Access Lists 二进制随机访问列表

Binary Random Access Lists are induced by

- the usual binary representation, i.e. $w_i = 2^i$, $D_i = \{0,1\}$
- complete binary leaf trees

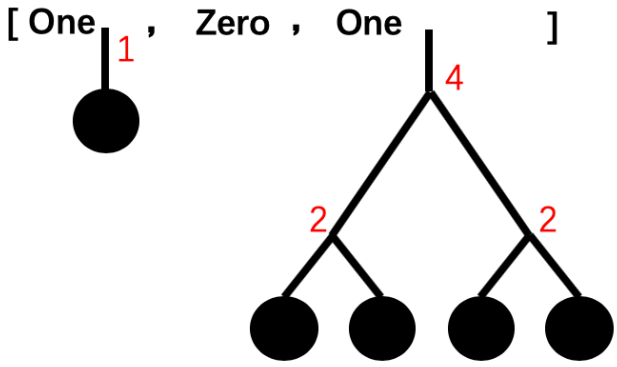
(1)数据结构定义

```
data Tree a = Leaf a | Node Int (Tree a) (Tree a)
data Digit a = Zero | One (Tree a)
type RList a = [Digit a]
```

The Int field keeps track of tree size for speed. Int 字段用来记录树的节点个数

【例】

Binary Random Access List of size 5:



(2)递增函数

The increment function on dense binary numbers:密集二进制数的递增

```
inc [] = [One]
inc (Zero : ds) = One : ds
inc (One :ds) = Zero : inc ds
```

(3)插入元素

The utility function link joins two equally sized trees:link 函数把两个长度一样的树合并

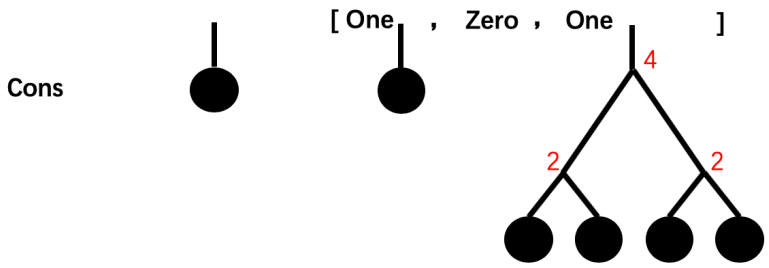
```
link t1 t2 = Node (2 * size t1) t1 t2
```

Inserting an element first in a binary random access list is analogous(类似) to inc: 往二进制随机访问列表的第一位插入新元素的过程与 lic 函数类似

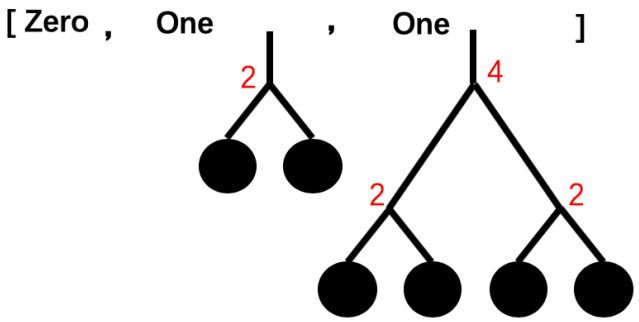
```
cons :: a -> RList a -> RList a
cons x ts = consTree (Leaf x) ts
consTree :: Tree a -> RList a -> RList a
consTree t [] = [One t]
consTree t (Zero : ts) = (One t : ts)
consTree t (One t' : ts) = Zero : consTree (link t t') ts
```

【例】

Result of consing element onto list of size 5:将一个元素合并到长度为 5 的列表后的结果



Result:



(4) Time complexity 时间复杂度

- cons, head, tail, perform O(1) work per digit, thus O(log n) worst case.
- lookup and update take O(log n) to find the right tree, and then O(log n) to find the right element in that tree, so O(log n) worst case overall.

7. Skew Binary Numbers 斜二进制数

(1)斜二进制数定义

- $w_i = 2^{i+1} - 1$ (rather than 2^i)
- $D_i = \{0,1,2\}$

Representation is redundant. But we obtain a canonical form if we insist that only the least significant non-zero digit may be 2. 斜二进制数的表示是多余的，但是如果我们规定最低位的非零数字只能为 2 的话，我们就可以得到规范形式。

Note：The weights correspond to the sizes of complete binary trees. 注：权重对应于完整二叉树的大小。

Theorem: Every natural number n has a unique skew binary canonical form. 定理：每个自然数 n 具有唯一的斜二进制规范形式。

(2)Case

① Base case

the case for 0 is direct.

② Inductive case

Assume n has a unique skew binary representation $b_0b_1...b_{m-1}$ 假设 n 有一个唯一的斜二进制表示形式

- If the least significant non-zero digit is smaller than 2, then n + 1 has a unique skew binary representation obtained by adding 1 to the least significant digit b_0.如果 n 的最低非零位不是 2，那么将通过将 n 的最低位 b_0 加 1 后的 n+1 具有唯一的斜二进制表达式
- If the least significant non-zero digit b_i is 2, then note that $1+2(2^{i+1}-1)=2^{i+2}-1$.

Thus n + 1 has a unique skew binary representation obtained by setting b_i to 0 and adding 1 to b_{i+1}.

【exercise 2】

Give the canonical skew binary representation for 31, 30, 29, and 28.

注意：斜二项数的权重依次为 1,3,7,15,31,...计算时先拆成这些数字的和

Solution: 00001, 0002, 0021, 0211

$31 = 2^{4+1} - 1 = 00001$

$30 = 2 * 15 = 2(2^{3+1} -1) = 0002$

$29 = 2 * 7 + 15 = 2(2^{2+1} -1) + (2^{3+1} -1) = 0021$

$28 = 2 * 3 + 7 + 15= 2(2^{1+1} -1) + (2^{2+1} -1) + (2^{3+1} -1)$

$= 0211$

(3)递增函数

Assume a sparse skew binary representation of the natural numbers type Nat = [Int], where the integers represent the weight of each non-zero digit, in increasing order, except that the first two may be equal indicating smallest non-zero digit is 2. 用稀疏斜二进制的形式表示自然数（类型为 Nat = [Int]）：其中每个整数表示每个非零位的权重（以递增顺序表示），除了前两位整数可以相等，用于表示最小的非零数字为 2，其余整数不相等。

```
inc :: Nat -> Nat
inc (w1 : w2 : ws) | w1 == w2 = w1 * 2 + 1 : ws
inc ws = 1 : ws
```

8. Skew Binary Random Access Lists 斜二进制随机列表

(1)数据结构定义

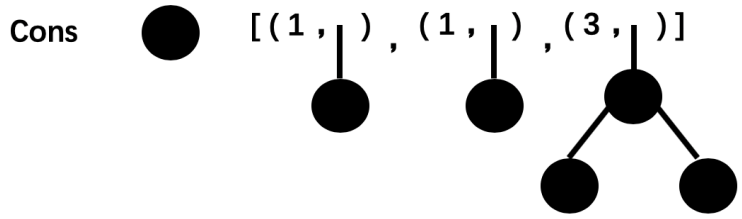
```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
type RList a = [(Int, Tree a)]
```

(2)cons 函数

```
empty :: RList a
empty = []
cons :: a -> RList a -> RList a
-- 若列表前两项相等
cons x ((w1, t1) : (w2, t2) : wts) | w1 == w2 =
            (w1 * 2 + 1, Node t1 x t2) : wts
-- 若列表前两项不相等
cons x wts = ((1, Leaf x) : wts)
```

【例 1】

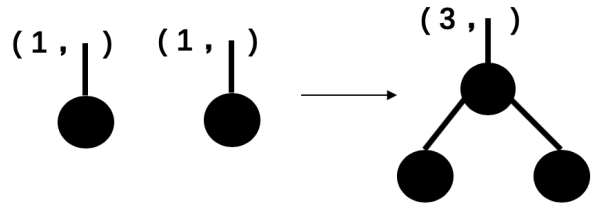
Consing onto list of size 5:



Result：

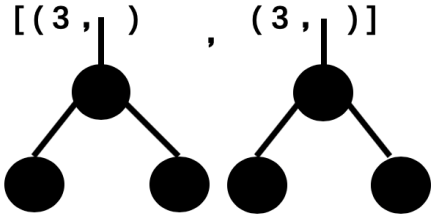
①首先通过模式匹配的值，若列表的前两项相等，将前两项合并：

(w1 * 2 + 1, Node t1 x t2)



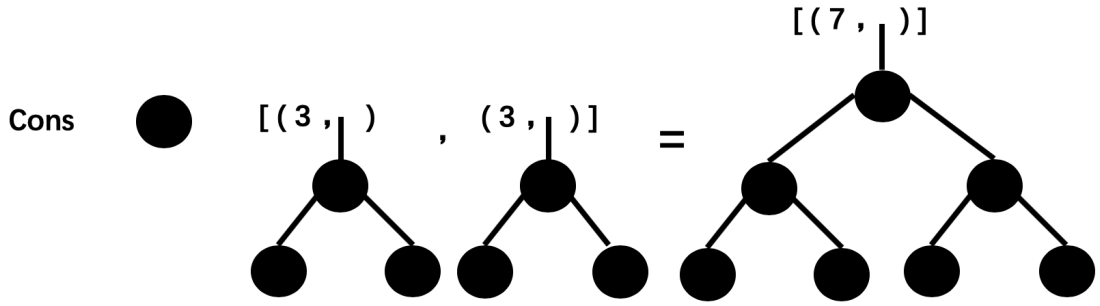
②将合并后的元素加入列表

(w1 * 2 + 1, Node t1 x t2) : wts



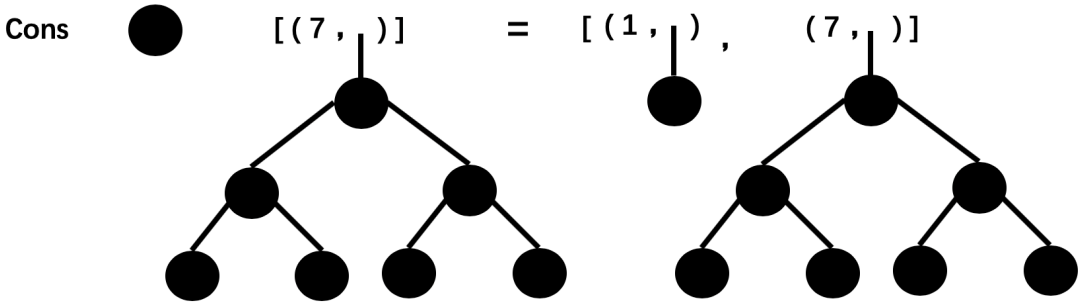
【例 2】

Consing onto list of size 6:



【例 3】

Consing onto list of size 7:



(3)head 与 tail 函数

```
head :: RList a -> a
head ((_, Leaf x) : _) = x
head ((_, Node _ x _) : _) = x

tail :: RList a -> RList a
tail ((_, Leaf _): wts) = wts
tail ((w, Node t1 _ t2) : wts) = (w', t1) : (w', t2) : wts
    where
        w' = w `div` 2
```

(4)lookup 函数

```
lookup :: Int -> RList a -> a
lookup i ((w, t) : wts) | i < w = lookupTree i w t
                        | otherwise = lookup (i - w) wts
lookupTree :: Int -> Int -> Tree a -> a
lookupTree _ _ (Leaf x) = x
lookupTree i w (Node t1 x t2)
        |i==0 =x
        |i<=w' =lookupTree(i-1)w't1
        | otherwise = lookupTree (i - w' - 1) w' t2
            where
                w' = w `div` 2
```

(5) Time complexity 时间复杂度

- cons, head, tail: O(1).
- lookup and update take O(log n) to find the right tree, and then O(log n) to find the right element in that tree, so O(log n) worst case overall.

Lecture 5 Type Classes

1. Type Class 类型类

- Type classes is one of the distinguishing fetures of Haskell
- Introduced to make ad hoc(特定的) polymorphism(多态), or overloading, less ad hoc
- Promotes reuse, making code more readable
- Central to elimination of all kinds of "boiler-plate" code(重复代码) and sophisticated datatype-generic(通用数据类型) programming.

2. Haskell 重载

(1) "=="

the following both work:

```
1 == 2
'a' == 'b'
```

I.e., (==) can be used to compare both numbers and characters. But (==) cannot work uniformly(统一) for arbitrary(任意的) types!

(2) "id 函数"

A function like the identity function

```
id :: a → a
id x = x
```

is polymorphic precisely because it works uniformly for all types: there is no need to "inspect" the argument.

In contrast, to compare two "things" for equality, they very much have to be inspected, and an appropriate method of comparison needs to be used.

Moreover, some types do not in general admit a decidable equality. E.g. functions (when their domain is infinite).

(3)idea

• Introduce the notion(概念) of a type class: a set of types that support certain related operations.

• Constrain(限制) those operations to only work for types belonging to the corresponding class.

• Allow a type to be made an instance of (added to) a type class by providing type-specific implementations of the operations of the class.

(4)Haskell 与面向对象语言的重载区别

A method, or overloaded function, may thus be understood as a family of functions where the right one is chosen depending on the types. 方法或重载函数可以理解为一系列函数，其中根据对应类型选择了对应的函数。

A bit like OO languages like Java. But the underlying mechanism is quite different and much more general. Consider read:

```
read :: (Read a) => String → a
```

Note: overloaded on the result type. A method that converts from a string to any other type in class Read.

```
>let xs =[1,2,3]::[Int]
> let ys = [1,2,3] :: [Double]
>xs
[1, 2, 3]
>ys
[ 1.0, 2.0, 3.0 ]
> (read "42": xs)
[42, 1, 2, 3]
> (read "42": ys)
[ 42.0, 1.0, 2.0, 3.0 ]
```

3. Eq 类型类

3.1 定义

```
class Eq a where
    (==) :: a → a → Bool
```

(==) is not a function, but a method of the type class Eq.

Its type signature is:

```
(==)::Eq a ⇒a →a →Bool
```

Eq a is a class constraint(约束). It says that that the equality method works for any type belonging to the type class Eq.

3.2 实例化

Various types can be made instances of a type class like Eq by providing implementations of the class methods for the type in question:

```
instance Eq Int where
    x ==y =primEqInt x y     -- primEqInt 是 haskell 的原始函数
instance Eq Char where
    x == y = primEqChar x y   -- primEqChar 是 haskell 的原始函数
```

(1)Answer

Suppose we have a data type:

```
data Answer = Yes | No | Unknown
```

We can make Answer an instance of Eq as follows:

```
instance Eq Answer where
    Yes == Yes = True
    No == No = True
    Unknown == Unknown = True
    _ == _ = False
```

(2)Tree

Make Tree an instance of Eq

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

for any type a that is already an instance of Eq:

```
instance (Eq a) ⇒ Eq (Tree a) where
    Leaf a1 == Leaf a2 = a1 == a2
    Node t1l t1r == Node t2l t2r = t1l == t2l && t1r == t2r
    _ == _ = False
```

Note that (==) is used at type a (whatever that is) when comparing a1 and a2 , while the use of (==) for comparing subtrees is a recursive call.

4. Derived Instances

Instance declarations are often obvious and mechanical. Thus, for certain built-in classes (notably 尤其是 Eq, Ord, Show), Haskell provides a way to automatically derive instances, as long as

- the data type is sufficiently simple
- we are happy with the standard definitions

Thus, we can do:

```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Eq
```

GHC provides many additional possibilities. With the extension -XGeneralizedNewtypeDeriving, a new type defined using newtype can "inherit" any of the instances of the representation type:

```
newtype Time = Time Int deriving Num
```

With the extension -XStandaloneDeriving, instances can be derived separately from a type definition (even in a separate module):

```
deriving instance Eq Time
deriving instance Eq a => Eq (Tree a)
```

5. Class Hierarchy 类型等级

Type classes form a hierarchy. E.g.:

```
class Eq a ⇒ Ord a where
    (<=) :: a → a → Bool
    ...
```

Eq is a superclass of Ord; i.e., any type in Ord must also be in Eq.

6. 重载的实现

The class constraints represent extra implicit arguments that are filled in by the compiler. These arguments are (roughly) the functions to use.类约束表示由编译器填充的额外隐式参数。这些参数（大致）是要使用的函数。

Thus, internally 内部的 (==) is a higher order function 高阶函数 with three arguments:

```
(==) eqF x y = eqF x y.   --eqF 表示判断相等的函数
```

An expression like 1 == 2 is essentially translated into

```
(==) primEqInt 1 2
```

So one way of understanding a type like

```
(==)::Eq a ⇒ a →a →Bool
```

is that Eq a corresponds to an extra implicit argument.

The implicit argument corresponds to a so called directory, or tuple/record of functions, one for each method of the type class in question.

7. Haskell 基础类型类

(1)Eq 类型类

```
class Eq a where
    (==), (/ =) :: a → a → Bool
class (Eq a) ⇒ Ord a where
    compare :: a → a → Ordering
    (<), (<=), (>=), (>) :: a → a → Bool
    max , min :: a → a → a
```

(2)Show 类型类

```
class Show a where
    show :: a → String
```

(3)Num 类型类

```
class Num a where
    (+),(−),(∗)::a →a →a
    negate :: a → a
    abs, signum :: a → a
    fromInteger :: Integer → a
```

```
class Num a ⇒ Fractional a where
    (/) ::a→a→a
    recip :: a → a
    fromRational :: Rational → a
```

【Quiz】

What is the type of a numeric literal like 42? What about 1.23? Why?

Haskell's numeric literals are overloaded:

- 42 means fromInteger 42
- 1.23 means fromRational (133 % 100)

Thus: 42 ::Num a ⇒ a

1.23 :: Fractional a ⇒ a

8. Multi-parameter Type Classes 多类型参数

GHC supports an extension to allow a class to have more than one parameter; i.e., defininig a relation on types rather than just a predicate: GHC 支持扩展，以允许一个类型类具有多个类型参数。即，定义类型的关系，而不仅仅是谓词：

```
class C a b where
    ...
```

This often leads to type inference ambiguities. Can be addressed through functional dependencies: 这通常会导致类型推断歧义。 能够通过功能依赖性解决：

```
class StateMonad s m | m → s where
    ...
```

This enforces that all instances will be such that m uniquely determines s. 这样可以强制所有实例都由 m 唯一地确定 s。

9. Application: Automatic Differentiation

- Automatic Differentiation: method for augmenting code so that derivative(s) computed along with main result. 自动微分：用于扩展代码的方法，以便随主结果一起计算导数。
- Purely algebraic method: arbitrary code can be handled 纯代数方法：可以处理任意代码
- Exact results 确切结果
- But no separate, self-contained representation of the derivative. 但是没有单独的，自包含的导数表示。

Consider a code fragment(片段):

```
z1 = x + y
z2 = x * z1
```

Suppose x' and y' are the derivatives of x and y w.r.t.(关于) a common variable. Then the code can be augmented to compute the derivatives of z1 and z2 :

```
z1 =x + y
z1' =x' + y'
z2 =x*z1
z2' = x' * z1 + x * z1'
```

(1)方法 Approaches

- Source-to-source translation

• Overloading of arithmetic operators and mathematical functions 算术运算符和数学函数的重载

Infinite list of derivatives allows derivatives of arbitrary order to be computed.导数的无限列表允许计算任意顺序的导数。

(2)实现

Introduce a new numeric type C : value of a continuously differentiable function at a point along with all derivatives at that point: 引入一个新的类型 C：在一点连续可微函数的值，以及该点上的所有导数：

```
data C = C Double C
valC (C a _) = a
derC (C _ x′) = x′
```

Constants and the variable of differentiation: 常数和微分变量

```
zeroC :: C
zeroC = C 0.0 zeroC
constC :: Double → C constC a = C a zeroC
dVarC :: Double → C
dVarC a = C a (constC 1.0)
```

Part of numerical instance:

```
instance Num C where
    (C a x′) + (C b y′) = C (a + b) (x′ + y′)
    (C a x′) − (C b y′) = C (a − b) (x′ − y′) x@(C a x′) ∗ y@(C b y′)
= C (a ∗ b) (x′ ∗ y + x ∗ y′)
    fromInteger n = constC (fromInteger n)
```

Computation of $y = 3t^2 + 7$ at t = 2:

```
t = dVarC 2
y=3∗t∗t+7
```

We can now get whichever derivatives we need:

```
valC y                          ⇒ 19.0
valC (derC y )                  ⇒ 12.0
valC (derC (derC y ))           ⇒ 6.0
valC (derC (derC (derC y))) ⇒ 0.0
```

Of course, we're not limited to picking just one point. Let tvals be a list of points of interest:

```
[3 ∗ t ∗ t + 7 | tval ← tvals, let t = dVarC tval ]
```

Or we can define a function:

```
y :: Double → C
y tval = 3 ∗ t ∗ t + 7
    where
        t = dVarC tval
```

Lecture 6

Functional Programming Patterns: Functor, Foldable, and Friends

1. Type Classes and Patterns

• In Haskell, many functional programming patterns are captured through specific type classes.在 Haskell 中，很多模式的匹配是通过特定的类型类来完成的

• Additionally, the type class mechanism itself and the fact that overloading is prevalent(普遍的) in Haskell give raise to other programming patterns.另外，类型类机制本身以及 Haskell 中重载很普遍的事实引发了其他编程模式。

2. Semigroups and Monoids 半群与单位半群

Semigroups and monoids are algebraic(代数) structures:

• Semigroup: a set (type) S with an associative binary operation $\cdot : S \times S \to S$: 半群是有结合性质的二元运算·的集合 S 构成的代数结构

$$\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$$

• Monoid: a semigroup with an identity element: 单位半群是具有单位元 e 的半群

$$\exists e \in S, \forall a \in S : e \cdot a = a \cdot e = a$$

2.1 Semigroups 半群 （有二元运算的集合）

Class definition (most important methods):

```
class Semigroup a where
  (<>) :: a → a → a                --(<>)为二元运算
  sconcat :: NonEmpty a → a        --将非空列表的元素取出并运用<>
```

sconcat：Take a nonempty list of type a and apply the <> operation to all of them to get a single result.

NonEmpty is the non-empty list type:

```
data NonEmpty a = a :| [a]
```

:|是一个数据构造器函数，将一个值 a 和装有值 a 的列表[a]返回成一个 NonEmpty 类型类的值 a:|[a]。

Instances of Semigroup 半群的实例

(1)列表

A list [a] is a semigroup (for any type a):

```
instance Semigroup [a] where
    (<>) = (++)
```

(2)Maybe 类型类

Maybe a is a semigroup if a is one:

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> y = y
    x <> Nothing = x
    Just x <> Just y = x <> y
```

(3)基于 Num 类型类的容器

Addition and multiplication are associative; a numeric type with either operation forms a semigroup.加法与乘法是有结合型的，Num 的集合和这些二元运算可以构成半群

- Sum a: the semigroup (a, (+))
- Product a: the semigroup (a, (∗))

Semigroup instances for Sum a and Product a:

```
instance Num a ⇒ Semigroup (Sum a) where
    (<>) = (+)
instance Num a ⇒ Semigroup (Product a) where
    (<>) = (∗)
```

Similarly, any type with a total ordering forms a semigroup with maximum or minimum as the associative operation: 类似地，任何具有总排序的类型都将形成一个以 maximum 或 minimum 为二元运算的半群

- Max a: the semigroup (a, max)
- Min a: the semigroup (a, min)

Semigroup instances:

```
instance Ord a ⇒ Semigroup (Max a) where
    (<>) = max
instance Ord a ⇒ Semigroup (Min a) where
    (<>) = min
```

(4)半群的乘积

All products of semigroups are semigroups; e.g.:

```
instance (Semigroup a, Semigroup b)=> Semigroup (a, b) where
    (x,y) <> (x’,y’) = (x <> x’,y <> y’)
```

a → b is a semigroup if the range b is a semigroup:

```
instance Semigroup b => Semigroup (a -> b) where
    f <> g = \x → f x <> g x
```

【Exercise】

What is the value of the following expressions?

（1）[1,3,7] <> [2,4]

（2）Sum 3 <> Sum 1 <> Sum 5

（3）Just (Max 42) <> Nothing <> Just (Max 3)

（4）sconcat (Product 2 :| [Product 3, Product 4])

（5）([1], Product 2) <> ([2, 3], Product 3)

（6）((1:) <> tail) [4, 5, 6]

答案：

（1）在列表中[]，<>相当于(++)，故结果为：[1,3,7,2,4]

（2）在 Sum 中,<>相当于(+)，故结果为：Sum 9

（3）在 Max 中，<>相当于取最大，故结果为：Just (Max 42)

（4）sconcat 将 Product 2 :| [Product 3, Product 4]这个 NonEmpty 类型类的列表的元素依次取出并运用<>，即 Product 2 <> Product 3 <> Product 4，故结果为 Product 24

（5）([1,2,3],Product 6)

（6）[1,4,5,6] <> tail [4,5,6] = [1,4,5,6] <> [5,6] = [1,4,5,6,5,6]

2.2 Monoid 单位半群（有二元运算和单位元的集合）

(1)定义

```
class Semigroup a ⇒ Monoid a where
    mempty :: a                          --单位元
    mappend:: a→a→a                      --二元运算
    mappend = (<>)
    mconcat :: [a]→a
    mconcat = foldr mappend mempty       --用二元运算折叠成一个值
```

(2)实例

A list [a] is the archetypical(原生的) example of a monoid:

```
instance Monoid [a] where
    mempty = [ ]
```

Any semigroup can be turned into a monoid by adjoining an identity element:

```
instance Semigroup a => Monoid (Maybe a) where
    mempty = Nothing
```

Monoid instances for Sum a and Product a:

```
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
instance Num a => Monoid (Product a) where
    mempty = Product 1
```

Monoid instances for Min a and Max a:

```
instance (Ord a,Bounded a) => Monoid (Min a) where
    mempty = maxBound
instance (Ord a,Bounded a) => Monoid (Max a) where
    mempty = minBound
```

All products of monoids are monoids; e.g.:

```
instance (Monoid a, Monoid b) => Monoid (a, b) where
    mempty = (mempty,mempty)
```

a → b is a monoid if the range b is a monoid:

```
instance Monoid b ⇒ Monoid (a -> b) where
    mempty _ = mempty
```

3. Functors 函子 （将普通函数放盒子里计算）

A Functor is a notion that originated in a branch of mathematics called Category Theory. 函子是起源于范畴论的数学分支的概念。

However, for our purposes, we can think of functors as type constructors T (of arity 1) for which a function map can be defined: 我们可以将函子视为可以定义函数映射的类型构造器 T（类型 1）

```
map ::(a→b)→T a→T b
```

that satisfies the following laws:

```
map id = id
map(f ◦ g) = map f ◦ map g
```

Common examples of functors include (but are not limited to) container types like lists:列表和树这样的容器类型一般都属于函子类型类

3.1 函子的定义

Of course, the notion of a functor is captured by a type class in Haskell:

```
class Functor f where
    fmap::(a →b)→f a →f b
    (<$) :: a → f b → f a
    (<$) = fmap const
(<$>) :: Funtor f => (a -> b) -> f a->f b
(<$>) = fmap          --将一个普通函数放到 f 盒子里运算
```

In general, the programmer is responsible for ensuring that an instance respects all laws associated with a type class. 通常，程序员需要保证实例满足与类型类关联的所有定律。

Note that the type of fmap can be read: (a → b) → (f a → f b)

That is, we can see fmap as promoting a function to work in a different context.

3.2 函子的实例

(1)列表类型

```
mapList :: (a → b) → [a] → [b]
mapList _ [] =[]
mapList f (x:xs) = f x : map List f xs
```

As noted, list is a functor:

```
instance Functor [ ] where
    fmap = listMap
```

(2)Maybe 类型

Maybe is also a functor:

```
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

(3)函数类型

The type of functions from a given domain is a functor with function composition as the map:给定定义域的函数((->) a)也是一个函子

```
instance Functor ((->) a) where
    fmap = (.)
```

(4)树类型

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
mapTree :: (a → b) → Tree a → Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node l x r) = Node (mapTree f l) (f x) (mapTree f r)
```

Indeed, there is a GHC extension for deriving Functor instances. For example, the functor instance for our tree type can be derived:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving Functor
```

4. Foldable 折叠类型类 （可将容器的所有值折叠成容器外的一个值）

(1)折叠类型类定义

Class of data structures that can be folded to a summary value. Foldable 类型类是一种能将容器类的多个数据通过计算得出一个数据的数据结构。

Many methods; minimal instance foldMap, foldr: Foldable 有很多种方法，但至少实现 foldMap 或 foldr 中的一个。

```
class Foldable t where
    fold ::Monoid m =>t m →m
    foldMap::Monoid m ⇒(a →m)→t a →m
-- 接收能将 foldable 容器 t 中 a 类型值变成 monoid(m)类型值的函数，一个 foldable 盒子，将这个函数运用到盒子里的每个值，并依据 Monoid 类型里定义的二元操作<>将各个值折叠成一个值
e.g. foldMap (\x -> Sum x) [1,2,3]，结果为Sum 6
    foldr :: (a → b → b) → b → t a → b
-- foldr 接收一个二元操作，初始值，foldable 盒子，将初始值运用到二元操作右边生成单个值
e.g. foldr (-) 0 [1,2,3] 相当于1-(2-(3-0))，结果为2
    foldr’ :: (a → b → b) → b → t a → b
    foldl :: (b → a → b) → b → t a → b
-- foldl 接收一个二元操作，初始值，foldable 盒子，将初始值运用到二元操作左边生成单个值
e.g. foldl (-) 0 [1,2,3] 相当于(((0-1)-2)-3)，结果为 -6
    foldl’ :: (b → a → b) → b → t a → b
    foldr1 ::(a →a →a)→t a →a
    foldl1 :: (a → a → a) → t a → a
    toList :: t a → [ a ]
    null ::t a→Bool
    length :: t a → Int
```

```
elem ::Eq a ⇒a →t a →Bool
maximum :: Ord a ⇒ t a → a
minimum :: Ord a ⇒ t a → a
sum ::Num a ⇒t a →a
product ::Num a ⇒ t a→ a
```

foldl typically incurs a large space overhead due to laziness. The version with strict application of the operator, foldl' is typically preferable. foldl 通常由于惰性计算而导致大的空间开销。对于操作符严格应用时，应采用 foldl'。

(2)实例

All expected instances, e.g.:

- instance Foldable [ ] where . . .
- instance Foldable Maybe where . . .

And GHC extension allows deriving instances in many cases; e.g.

```
data Tree a = . . . deriving Foldable
```

But there are also some instances that are less expected, e.g.:

- instance Foldable (Either a) where . . .
- instance Foldable ((,) a) where . . .

This has some arguably odd consequences: 对于二元组、Either 这些实例中，fold 函数操作的是第二个元素:

```
length (1, 2) ⇒ 1          --操作 2，故 2 的长度为 1
sum (1, 2) ⇒ 2             --操作 2，故 2 的和为 2
length (Left 1) ⇒ 0        --对于 Left 1，因为没有 Right，所以长度为 0
length (Right 2) ⇒ 1       --对于 Right 2，长度为 1
```

(3)将树类型折叠

假设树的数据结构如下：

```
data Tree a = Empty | Node (Tree a) a (Tree a) deriving (Show,Eq)
```

Let us make it an instance of Foldable:

```
instance Foldable Tree where
    foldMap f Empty = mempty
    foldMap f (Node l a r) = foldMap f l <> f a <> foldMap f r
```

We wish to compute the sum and max over a tree of Int. One way: 我们希望计算出树的各节点处值的和以及最大值

```
sumMax :: Tree Int → (Int , Int )
sumMax t = (foldl (+) 0 t, foldl max minBound t)
```

Another way, with a single traversal:

```
sumMax :: Tree Int -> (Int , Int )
sumMax t = (sm,mx) where
    (Sum sm, Max mx) = foldMap (\n → (Sum n, Max n)) t
```

The latter can be generalized to e.g. computing the sum, product, min, and max in a single traversal: 后者可以推广到在一次遍历中计算总和，乘积，最小值和最大值：

```
foldMap (\n → (Sum n, Product n, Min n, Max n)) t
```

(3)对“折叠”一词的争议

Note that the kind of “folding” captured by the class Foldable in general makes it impossible to recover the structure over which the “folding” takes place. 通常 Foldable 类型类在 “折叠”后无法恢复发生“折叠”前的结构。

Such an operation is also known as “reduce” or “crush”, and some authors prefer to reserve the term “fold” for catamorphisms, where a separate combining function is given for each constructor, making it possible to recover the structure. 这样的操作也称为“减少”或“压碎”，并且一些作者更喜欢保留术语“折叠”用于同构形式，其中为每个构造函数指定了单独的组合函数，从而可以恢复结构。

One might thus argue that Reducible or Crushable would have been a more precise name.因此，可能有人争辩说，Reducible 或 Crushable 会是一个更精确的名称。

Lecture 7 Introduction to Monads

1. 纯函数式数据结构存在的问题

1.1 A Blessing and a Curse 纯函数式的利与弊

- In Haskell, many functional programming patterns are captured through specific type classes.在 Haskell 中，很多模式的匹配是通过特定的类型类来完成的
- In pure functional programming, "everything is explicit" 函数式编程中，所有的一切明确的

纯函数的优点是安全可靠。函数输出完全取决于输入，不存在任何隐式依赖。它的存在如同数学公式般完美无缺。可有时越是完美的东西就越没有用处，比如纯函数，因为隔绝了外部环境，纯函数连基本的输入输出都做不了。

1.2 Conundrum 难题

（1）pure or impure 纯与不纯

内部环境的数据可以认为是“纯数据”，具有明确性，不会导致 side effect.

外部环境的数据（e.g.输入与输出数据、系统的状态等）则被认为“不纯数据”，这些数据具有很大的不确定性，会产生 side effect.

如果只追求“纯”，则好多像简单输入输出的功能将无法实现。

如果直接引入“不纯”，则 Haskell 将失去 pure functional 的特征，带来 side effect.

纯函数安全可靠但无用，是个无趣的好男人；普通函数能力强但 Bug 多，对程序来说却是必不可少的，如同危险而有魅力的坏男孩。如何同时拥有两者，让它们合作无间各自发挥特点而不互相打架呢？这就是 Monad 的作用了。

（2）Answer to Conundrum 解决方案

- Monads bridges the gap: allow effectful programming in a pure setting. 为了解决“纯”与“不纯”的问题，Haskell 引入了 Monad 的概念，Monad 相当于“纯”与“不纯”之间的一个桥梁。

• Key idea: Computational types: an object of type M A denotes a computation of an object of type A. 核心方案就是构造一种计算类型 M a，这个类型相当于一个盒子 M，里面装了 a 类型的数据。Monad 将带有副作用的不纯数据以一种可控的方式引入到 Haskell 中。当需要用不纯的 a 类型数据计算时，将其放入盒子 M 中计算，计算后的结果不能用于盒子外部。盒子内部是不纯的，而盒子外面是纯的。这样就避免了“不纯”对“纯”的污染，同时又实现了“不纯”的功能。

（3）Monad 带来的好处

• promote disciplined use of effects since the type reflects which effects can occur; 对副作用数据的规范化使用，Monad 标志着可能发生副作用

• allow great flexibility in tailoring the effect structure to precise needs;

• support changes to the effect structure with minimal impact on the overall program structure; 最小化副作用对程序的影响

• allow integration into a pure setting of real effects such as

- I/O 输入输出（IO Monad）
- mutable state. 可变状态 （State Monad）

Lecture 8 Monad

1. Applicative （将盒子里的函数与盒子里的数计算）

（1）applicative 定义

An applicative functor is a functor with application, providing operations to:

- embed pure expressions (pure), and
- sequence computations and combine their results (<*>)

```
class Functor f ⇒ Applicative f where
    pure :: a→f a                    --将一个值或函数放到盒子 f 里
    (<*>):: f (a→b)→f a→f b          --将盒子里的函数与盒子里的值做运算
    (*>) :: f a→f b→f b
    (<*) :: f a→f b→f a
    u *> v = pure (const id) <*> u <*> v
    u <* v = pure const <*> u <*> v
```

（2）applicative 定律

- 单位元定律 pure id <*> v = v

  v 是装了值的盒子，如：

```
> pure id <*> Just 5
Just 5
```

- 复合定律 pure (∘) <*> u <*> v <*> w = u <*> (v <*> w)

  ∘ 是函数符合运算，u 和 v 是装了函数的盒子，w 是装了值的盒子，如：

```
> pure (.) <*> (Just not) <*> (Just even) <*> (Just 5)
Just True
--相当于(Just not) <*> (((Just even) <*> (Just 5)))
--(Just not) <*> (Just False)
--Just True
```

- 同态定律 pure f <*> pure x = pure (f x)

  f 是盒子外的函数，x 是盒子外的数，如：

```
> Just (+1) <*> pure 5
Just 6
```

- 互换定律 u <∗> pure y = pure ($ y) <∗> u

  u 是装了函数的盒子，y 是盒子外的数，

```
> Just (+1) <*> pure 5
Just 6
> pure ($ 5) <*> Just (+1)
Just 6
-- 等于 Just ($ 5) <*> Just (+1)
-- 相当于盒子里的(+1)函数做了盒子里($ 5)函数的参数
```

（3）Instance of applicative 可应用函子的实例

列表实例：

```
instance Applicative [ ] where
    pure x = [x ]
    fs <∗> xs = [ f x | f ← f s , x ← x s ]
    -- fs 是装了函数的列表，xs 是装了值的列表
```

如：

```
> [(+1),(+2),(+3)] <*> [1,2,3]
[2,3,4,3,4,5,4,5,6]
```

Maybe 实例：

```
instance Applicative Maybe where
    pure = Just
    Just f <∗> m = fmap f m
    Nothing <∗> _ = Nothing
```

2. Alternative（从两个装了东西的盒子中选一个盒子）

（1）alternative 定义

The class Alternative is a monoid on applicative functors:

```
-- :m +Control.Applicative 使用时要导包
class Applicative f ⇒ Alternative f where
    empty :: f a
    (<|>) :: f a → f a → f a
    some :: f a → f [a]
    many :: f a → f [a]
```

<|> can be understood as “one or the other” (<|>两个选一个合适的)

some can be understood as “at least one” ()

many can be understood as “zero or more”.

（2）instance of alternative 选择可应用函子实例

Maybe 实例：

```
instance Alternative Maybe where
    empty = Nothing
    Nothing <|> p = p
    Just x <|> _ = Just x
```

如：

```
> Nothing <|> Just 3
Just 3
```

列表实例：

```
instance Alternative [ ] where
    empty = [ ]
    (<|>) = (++)
```

如：

```
> [] <|> [1,2]
[1,2]
```

3. Monad （将不纯数据放盒子计算）

（1）Monad 定义

```
class Monad m where
    return :: a → m a                    --将值或函数放到盒子 m 里
    (>>=) :: m a→(a → m b)→m b
--将盒子里的值拿出来，带入一个能将盒子外的值计算后放入盒子内的函数中
```

（2）Instance of Monad 实例

Maybe Monad：

```
instance Monad Maybe where
    return = Just
    Nothing >>=  _ = Nothing
    (Just x) >>= f = f x
```

如：

```
> (Just 'c') >>= (\ _ -> (Just 7))
Just 7      --Just 盒子原本装了 Char 类型，做了 bind 后盒子装了 int 类型
```

（3）The Monad Type Class Hierachy

Monads are mathematically related to two other notions:

- Functors
- Applicative Functors (or just Applicatives)

Every monad is an applicative functor, and every applicative functor (and thus monad) is a functor.

Class hierarchy:

class Functor f where . . .

class Functor f ⇒ Applicative f where . . .

class Applicative m ⇒ Monad m where . . .

即：applicative 实现了 functor，monad 实现了 applicative

map can be defined in terms of >= and return, demonstrating that a monad is a functor:

```
fmap f m = m >>= (\x -> return (f x))
```

（4）State Monad

I. State s a 是一种用来**封装状态处理函数**：\s -> (a,s')的数据结构，因为 State 封装的是一个函数，而不是状态 s 本身，所以称 State 为 State 类型(type)是不准确的，应该把 State s a 称为**状态处理器**(State processor)。

```
newtype State s a = State { runState :: s -> (a,s) }
-- 即：newtype State s a = State (s -> (a,s))
```

【注意】(\s -> (a,s')) 是状态处理函数，封装了状态处理函数的 State 容器叫做状态处理器，不要混淆了。

• 尽管将 State 称为类型不准确，但毕竟 State s a 是通过 newtype 定义的，也就是对现有类型的封装。我们可以不准确地将其称为 State 数据类型，该类型有两个类型参数：表示状态的类型参数 s 以及表示结果的类型参数 a。虽然写作 State s a，但在理解上，我们应该把 State 看成是一个装了状态处理函数的容器 State (s -> (a, s))，即状态处理器。

• State s a 的实质是 State (s -> (a, s))，State s a 与 State (s -> (a, s))完全等价。

• 通过 runState 访问函数可以从 State 类型中取出这个封装在里面的状态处理函数：\s -> (a,s')，该函数接收一个状态参数 s，经过计算（转换）之后返回一个由**状态 s 下对应的返回结果 a** 以及**新的状态 s'** 的二元组 (a, s')。runState 可以直接当作状态处理函数的函数名，他相当于把 State 容器中的状态处理函数从 State 容器中取出来。

II. State s 的 Monad 实例

```
instance Monad (State s) where
    return :: x -> State s x
    return x = state (\s -> (x, s))      --注意这里 state 的 s 是小写
    -- return x = State $ \s -> (x,s)   --注意这里的 State 的 S 是大写
    (>>=) :: State s a -> (a -> State s b) -> State s b
    pr >>= f = state $ \ st ->
        let (x, st') = runState pr st      -- Run first processor on st
         in runState (f x) st'             -- Run second processor on st'
```

• state :: (s -> (a, s)) -> State s a 即：把一个状态处理函数封装到 State 容器里

• return 函数把结果 x 封装成一个 State s x 即 State (s -> (x, s))的状态处理器

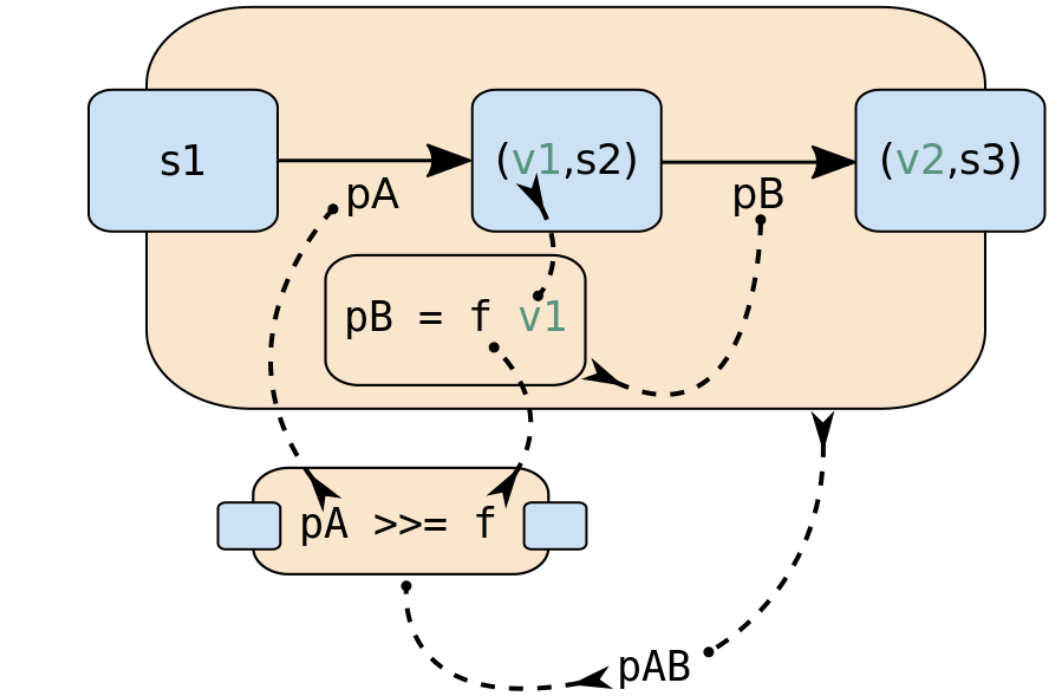
• bind 函数的类型签名为：

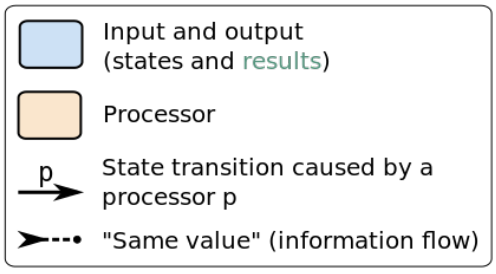
```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

大致相当于 State (s -> (a, s)) -> (a -> State (s -> (b, s))) -> State (s -> (b, s))

• 函数(a -> State s b)相当于一个生成状态处理器 State (s -> (b, s))的函数，称为状态处理器生成函数 f

• 整个 bind 函数最后返回的结果为一个状态处理器 State(s -> (b, s))，即用装入了状态处理函数的 State 容器






```
pAB = pA >>= f
    = state $ \ s1 -> (v2, s3)
        where
        (v1, s2) = runState pA s1
        pB = f v1
        (v2, s3) = runState pB s2
```

III. 对 State Monad 的理解

<1> 对 bind 函数(>>=)的理解：

• bind 运算的左侧为一个状态处理器 pA (写为 State s a 类型，理解为装了一个状态转换函数的 State 容器 State (\s1 -> (v1, s2)))

• bind 运算的右侧为一个状态处理器生成函数 f （即： \v1 -> State (\s2 -> (v2, s3))），该函数接受一个结果 v1，返回一个状态处理器 pB（即：State (\s2 -> (v2, s3))）

• 我们将 pA >>= f 产生 pB 的这一个过程记为 pAB，即 pAB = pA >>= f，pAB 为一个封装了由 s1 到(v2, s3)的 State 容器

【注意】pB 与 pAB 不是一样的，pB 是封装了(\s2 -> (v2, s3)) 的 State 容器，而 pAB 是封装了(\s1 -> (v2, s3)) 的 State 容器

<2> 数据流向：

• 首先给定一个初始状态 s1(常称为 seed)和一个状态处理器 pA，在这个状态 s1 下，我们通过 runState 把状态处理器 pA 中的状态处理函数取出来，并向该函数传入状态 s1，得到一个包含状态 s1 下对应的返回结果 v1 以及新状态 s2 的二元组(v1，s2)

• 通过状态处理器生成函数 f，我们将得到的二元组(v1，s2)中的 v1 取出传入 f 中，得到一个新的状态处理器 pB。之所以要把 v1 取出传给函数 f 是因为 v1 是 State s 这个 Monad 中 State s a 的 a，而 f 需要接收一个 State s a 中的 a，从而生成一个 State s b（即 pB）

• 通过 runState 函数将 pB 中的状态处理函数取出来，传入新的状态 s2，得到一个包含状态 s2 下对应的返回结果 v2 以及新状态 s3 的二元组(v2，s3)

<3> 做 bind 的意义

在计算 pA >> f 的过程中，我们产生了两个不同状态(s1、 s2)下对应的两个结果(v1、v2)，并将状态更新为了 s3

这就像我们在做伪随机投骰子一样，得到了两次投掷结果 v1、v2，并将状态跟新了，以便下次能继续根据新状态 s3 产生投掷结果

我们可以把状态处理器 pA、pB 理解成骰子，状态处理器生成函数 f 理解为骰子生成器。相同状态下骰子产生的结果是唯一的。

• 首先我们有一个状态 s1 和一个骰子 pA，骰子 pA 根据状态 s1 投出了结果 v1，并且将状态更新为 s2

• 然后骰子生成器 f 接收到 v1，得知骰子 pA 已经被使用，随后生成一个新的骰子 pB，新的骰子 pB 根据新的状态 s2 投出了新的结果 v2，并将状态更新为 s3

所以：p >>= f 可以理解为投了两次骰子得到了两个结果

p >>= f >>= f 可以理解为投了三次骰子得到三个结果

以此类推…

IV. State Monad 中的相关函数

<1> 设置与读取 State

在前面，我们讨论了给定一个初始状态 s 和一个状态处理器 p，我们可以得到一个返回结果 a 和一个新的状态 s'。初始状态的话我们可以直接设定，那么初始状态处理器是怎么得到的呢?

通过 put 函数生成状态处理器：

```
put newState = State $ \_ -> ((), newState)
```

• put 函数首先接收一个我们给定的初始状态 newState，然后生成一个状态处理器 State (_ -> ((), newState))。这个状态处理器忽略它接收到的任何 state 参数，然后返回一个二元组，这个二元组里包含一个空结果()和输入给 put 函数的初始状态 newState

• 由于我们不需要这个初始状态 newState 下返回的结果，我们只是需要一个状态来替换，所以元组的第一个元素是()，一个占位的值(universal placeholder value)

<2> 获取值与状态

通过 evalState 函数，可以获取状态处理器 p 在传入状态 s 下，返回的结果 a

```
evalState :: State s a -> s -> a
evalState p s = fst (runState p s)
```

通过 execState 函数，可获取状态处理器 p 传入状态 s 后，生成的新状态 s’

```
execState :: State s a -> s -> s
execState p s = snd (runState p s)
```

（5）Maybe Monad

```
instance Monad Maybe where
    return = Just
    Nothing >>= _   = Nothing
     (Just x) >>= f = f x
```

（6）List Monad

Computation with many possible results, “nondeterminism”:

```
instance Monad [ ] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = [ ]
-- concat :: Foldable t => t [a] -> [a]
```

列表 Monad 代表一系列结果可能为多个或者可能失败的计算。当计算失败时，返回空列表[]，定义 xs >> f 时，如果列表中有多个元素，那么每个元素都参与 f 函数的计算，这样返回的结果会有多种情况，因此计算的结果是不确定的，可能失败，也可能有多种结果。

如：

```
plus :: Num b => [b] -> [b] -> [b]
plus xs ys = xs >>= \x -> (ys >>= (\y -> return (x + y)))
```

运行结果：

```
> plus [1,2,3] [4,5,6]
[5,6,7,6,7,8,7,8,9]
```

计算过程：

```
xs >>= \x -> (ys >>= (\y -> return (x + y)))
concat (map (\x -> (ys >>= (\y -> return (x + y)))) xs)
concat (map (\x -> (concat (map (\y -> return (x + y))) ys)) xs)
```

如：

```
f :: Monad m => m a -> m b -> m (a, b)
f xs ys = xs >>= \x -> (ys >>= (\y -> return (x,y)))
```

运算结果：

```
> f [1,2] ['a','b']
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

4. The do-notation 语法糖

Haskell provides convenient syntax for programming with monads:

（1）Monadic binding 语法糖

```
do
    a <- exp1
    b <- exp2
    return exp3
```

is syntactic sugar for

```
exp1 >>= \a ->
exp2 >>= \b ->
return exp3
```

<- 表示把 monad 里面的值拿出来

exp1,exp2 是装了值的 Monad 盒子　　exp3 是对从盒子取出的值做运算

return 将计算后的值装回盒子里

例如：

```
plus xs ys = xs >>= \x -> (ys >>= (\y -> return (x + y)))
```

若想去掉括号，可以写为：

```
plus xs ys = xs >>= \x ->
             ys >>= \y ->
             return (x + y)
```

等价于：

```
plus xs ys = do
    x <- xs             -- xs 装了值的盒子，x 是 xs 盒子里的值
    y <- ys             -- ys 装了值的盒子，y 是 ys 盒子里的值
    return (x+y)
```

（2）Sequencing 语法糖

```
do
    exp1
    exp2
    return exp3
```

is syntactic sugar for

```
exp1 >>= \ _ ->
exp2 >>= \ _ ->
return exp3
```

例如：

```
hxx= do
    "hello"
    return "world"
```

相当于：

```
hxx = "hello" >>= \ _ -> return "world"
```

也相当于：

```
hxx = "hello" >> return "world"
```

>>与>>=类似，区别在于>>不需要传入一个(a -> m b)的函数，只要传入另一个 monad 即可。

>> 用于提取“x >> f”中 x 的上下文，即 x 的 monad 盒子的信息

运行结果：

```
> hxx
["world","world","world","world","world"]
> :t hxx
hxx :: [[Char]]
```

• 结果为["world","world","world","world","world"]的原因：

在"hello" >> return "world"中，>>提取到上下文的信息为"hello"，即一个装了五个元素（'h', 'e', 'l', 'l', 'o'）的[]列表盒子，>>将盒子里的内容进行忽略，保留盒子信息，告诉给后面的 return 函数，return 函数知道了上下文后，知道要把"world"放到[]列表盒子中，且盒子里需要有五个元素，所以最后的结果是["world","world","world","world","world"]

（3）Let binding 语法糖

```
do
    let a = exp1
        b = exp2
    return exp3
```

is equivalent to

```
do
    a <- return exp1
    b <- return exp1
    return exp3
```

【Exercise】

Recall that a type Int → (a,Int) can be viewed as a state monad.

Haskell 2010 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

```
newtype S a = S {unS :: (Int → (a, Int))}
```

Thus:

```
unS :: S a → (Int → (a , Int ))
```

Provide a Functor, Applicative, and Monad instance for S.

答案：

Functor:

```
instance Functor S where
    fmap f sa = S $ \s ->
    let
        (a,s') = unS sa s
    in
        (f a,s')
```

Applicative:

```
instance Applicative S where
    pure a = S $ \s -> (a, s)
    sf <*> sa = S $ \s ->
    let
        (f,s')=unS sf s
    in
        unS (fmap f sa) s'
```

Monad:

```
instance Monad S where
    m >>= f = S $ \s ->
        let (a,s') = unS m s
        in unS (f a) s'
```

(Using the default definition return = pure)

Lecture 9 Concurrency

1. Concurrency Monad

（1）Thread 线程定义

A Thread represents a (branching) process: a stream of primitive atomic operations:一个线程代表一个进程的分支，即一组原始的原子操作

```
data Thread = Print Char Thread | Fork Thread Thread | End
```

【注意】

- a Thread represents the entire rest of a computation.
- a Thread can spawn other Threads (so we get a tree, if you prefer). 一个线程可以生成其他线程

Introduce a monad representing "interleavable computations". At this stage, this amounts to little more than a convenient way to construct threads by sequential composition. 引入一个表示"可交叉计算"的 monad。 在此阶段，这仅是通过顺序组合构造线程的便捷方式。

How can Threads be constructed sequentially? The only way is to parameterize thread prefixes on the rest of the Thread. This leads directly to continuations. 如何按顺序构造线程？ 唯一的方法是在其余线程上参数化线程前缀。 这直接实现连续。

（2）定义 Concurrency Monad

```
newtype CM a = CM ((a -> Thread) -> Thread)
fromCM :: CM a -> ((a -> Thread) -> Thread)
fromCM (CM x) = x
thread :: CM a -> Thread
thread m = fromCM m (const End)
instance Monad CM where
    return x = CM (\k -> k x)
    m >>= f = CM $ \k -> fromCM m (\x -> fromCM (f x) k)
```

定义了一个叫 CM 的 Monad 盒子，盒子里装了一个以函数(a -> Thread)为参数，返回一个线程 Thread 的高阶函数

fromCM 函数用于将 CM 盒子里的类型参数 a 的值取出来

（3）Atomic operations 定义原子操作

```
cPrint :: Char -> CM ()
cPrint c =CM (\k -> Print c (k ()))
cFork :: CM a -> CM ()
cFork m = CM (\k -> Fork (thread m) (k ()))
cEnd :: CM a
cEnd = CM (\_ -> End)
```

（4）定义并发计算中的类型

```
type Output = [Char]
type ThreadQueue = [Thread]
type State = (Output, ThreadQueue)
```

（5）定义调度函数

Selects next Thread to run, if any.

```
schedule :: State → Either (State , Thread) Output
schedule (o, [ ]) = Right o
schedule (o, t : ts) = Left ((o, ts), t)
```

（6）定义分发函数

Dispatch on the operation of the currently running Thread. Then call the scheduler.

```
dispatch :: State → Thread → Either (State,Thread) Output
dispatch (o,rq) (Print c t) = schedule (o + [ c ], rq + [ t ])
dispatch (o,rq) (Fork t1 t2) = schedule (o,rq +[t1,t2])
dispatch (o,rq) End = schedule (o,rq)
```

（7）Running a Concurrent Computation 运行并发计算

```
runCM :: CM a → Output
runCM m = runHlp ("", [ ]) (thread m) where
    runHlp s t = case dispatch s t of
                    Left (s',t) → runHlp s' t
                    Right o → o
```

2. Concurrent Programming

（1）轻量级线程

Primitives for concurrent programming provided as operations of the IO monad. They are in the module Control.Concurrent.

```
forkIO :: IO () → IO ThreadId
killThread :: ThreadId → IO ()
threadDelay :: Int → IO ()
```

forkIO 用于创建一个新的线程，它会返回一个线程的 ID，ThreadId 类型可以看作是一个 int 类型

（2）基本线程通信 MVar

The fundamental synchronisation mechanism is the MVar. 最基本的线程通信是通过使用一个 MVar(Mutable variable)容器类型来实现。

An MVar is a "one-item box" that may be empty or full. MVar 只能储存一个值，他有两个状态，为空或者为满。

```
newMVar :: a → IO (MVar a)
-- 新建一个放入值的 MVar 容器
newEmptyMVar :: IO (MVar a)
-- 新建一个空的 MVar 容器
putMVar :: MVar a → a → IO ( )
-- putMVar 将一个值放入 MVar 容器中，若容器已满，则该线程将会等待
takeMVar :: MVar a → IO a
-- takeMvar 将 MVar 容器中值取出，若容器为空，则该线程将会等待
```

Reading (takeMVar) and writing (putMVar) are atomic operations:

- Writing to an empty MVar makes it full.
- Writing to a full MVar blocks.
- Reading from an empty MVar blocks.
- Reading from a full MVar makes it empty.

（3）Basic Synchronization

```
import Control.Monad
import Control.Concurrent
import System.IO
countFromTo :: Int -> Int -> IO ()
countFromTo m n | m > n = return ()
                | otherwise = do
                    putStrLn (show m)
                    countFromTo (m + 1) n
main::IO()
main = do
    start <- newEmptyMVar
    done <- newEmptyMVar
    forkIO $ do
        takeMVar start
        countFromTo 1 10
        putMVar done ()
    putStrLn "Go!"
    putMVar start ()
    takeMVar done
    countFromTo 11 20
    putStrLn "Done!"
```

3. Software Transactional Memory

3.1 STM 特点

• Operations on shared mutable variables grouped into transactions.

• A transaction either succeeds or fails in its entirety. I.e., atomic w.r.t.(关于) other transactions.

• Failed transactions are automatically retried until they succeed.

• Transaction logs, which records reading and writing of shared variables, maintained to enable transactions to be validated, partial transactions to be rolled back, and to determine when worth trying a transaction again.

• Basic consistency requirement: The effects of reading and writing within a transaction must be indistinguishable from the transaction having been carried out in isolation.

• No locks! (At the application level.)

3.2 STM and Pure Declarative Languages

• STM perfect match for purely declarative languages:

- reading and writing of shared mutable variables explicit and relatively rare;

- most computations are pure and need not be logged.

• Disciplined use of effects through monads a huge payoff: easy to ensure that only effects that can be undone can go inside a transaction.

3.3 Composition

STM operations can be robustly composed. That’s the reason for making readBuffer and writeBuffer STM operations, and leaving it to client code to decide the scope of atomic blocks.

3.4 STM 的使用

软件事务内存管理提供了一个容器类型 TVar a (Transactional variable) 来存储共享数据。STM 实现了 Monad 类型类。使用时需要先导入 STM 库（Control.Concurrent.STM）

（1）操作 TVar 的相关函数

```
newTVar :: a -> STM (TVar a)
-- 新建一个 TVar 容器，并且内部存储一个值
readTVar :: TVar a -> STM a
-- 从一个 TVar 容器中读取一个值
writeTVar :: TVar a -> a -> STM ()
-- 向一个 TVar 容器中写入一个值
```

（2）原子执行 atomically

STM Monad 代表了事务内存的效应，使用 automically :: STM a -> IO a 可以将一次事务原子执行，如：

```
transfer :: Account -> Account -> Int -> IO ()
transfer a1 a2 amount = atomically $ do
                                withdraw a1 amount
                                deposit.   a2 amount
```

Lecture 11-12 The Threepenny GUI Toolkit

1 What is Threepenny

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML Document Object Model (DOM) to be manipulated
  - handles JavaScript events in Haskell
- Works by sending JavaScript code to the client.
- Frequent communication between browser and server: Threepenny is best used running on localhost or over the local network.

2 Rich API

- Full set of widgets (buttons, menus, etc.)
- Drag and Drop
- HTML elements
- Support for CSS
- Canvas for general drawing
- Functional Reactive Programming (FRP)

3 Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.
- Set up event handlers to act on events from the elements.
- Knowing a bit of HTML helps.

Lecture 15 Property-based Testing

1 Quick Check

（1）QuickCheck 的用途

- Framework for property-based testing 基于属性的测试框架
- Flexible language for stating properties
- Random test cases generated automatically based on type of argument(s) to properties. 自动生成测试的 case
- Highly configurable:
  - Number, size of test cases can easily be specified
  - Additional types for more fine-grained control of test case generation
  - Customised test case generators
- Support for checking test coverage
- Counterexample produced when test case fails
- Counterexamples automatically shrunk in attempt to find minimal counterexample

（2）测试举例

对于两个整数的和，奇偶性满足 even (x + y) == (even x == even y)，如：

even (4 + 5) => even 9 => False

even 4 == even 5 => True = False => False

可以使用 QuickCheck 来测试这个性质：

I. 安装 QuickCheck

```
$ stack install QuickCheck
```

II. 导入 QuickCheck 库

```
import Test.QuickCheck
```

III. 编写测试属性函数

```
prop_even x y = even (x + y) == (even x == even y)
```

IV. 使用 quickCheck 来测试这个属性是否成立

```
> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
> quickCheck prop_even
+++ OK, passed 100 tests.
```

quickCheck 自动生成了 100 个测试案列进行了测试，并且这 100 个案列都通过了。通过使用 quickCheck，用户就不需要靠自己一个一个设计并输入测试案例，这些工作全部有 quickCheck 完成了。

另一个测试举例：

测试属性 reverse (xs ++ ys) == reverse ys ++ reverse xs

编写测试属性函数：

```
prop_rev1 xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

使用 quickCheck 测试，在测试时可指定测试自动生成的案列的类型

```
> quickCheck (prop_rev1 :: [Int] -> [Int] -> Bool)
+++ OK, passed 100 tests.
> quickCheck (prop_rev1 :: [Char] -> [Char] -> Bool)
+++ OK, passed 100 tests.
```

测试属性 reverse (reverse xs) == xs

编写测试属性函数：

```
prop_rev2 xs = reverse (reverse xs) == xs
```

使用 quickCheck 测试，在测试时可指定测试自动生成的案列的类型

```
> quickCheck (prop_rev2 :: [Int] -> Bool)
+++ OK, passed 100 tests.
```

如果想要将 prop_rev1 与 prop_rev2 同时测试可以用 .&&. 连接：

```
> quickCheck (prop_rev1 .&&. prop_rev2)
+++ OK, passed 100 tests.
```

使用 verboseCheck 查看具体测试的 case

```
> verboseCheck (prop_rev2 :: [Int] -> Bool)
```

结果：

```
Prelude Test.QuickCheck> verboseCheck (prop_rev2 :: [Int]
-> Bool)
Passed:
[]

Passed:
[1]

Passed:
[-1]

Passed:
[-1,-1,-3]

Passed:
[-1,-2,-3]

Passed:
[-2,0,4,2,-3]
```

……

```
Passed:
[-17,-69,13,-56,-52,-32,-38,-16,-74,79,25,-75,97,-41,-93,-
72,22,-69,-10,-51,-93]

Passed:
[31,75,-39,-34,53,78,76,58,32,33,-25,73,16,-33,-84,61,-91,
73,-39,71,-10,-79,-90,-39,-95,-58,-82,-30,-94,82,-15,52,-3
6,-27,-56,-48,93,-57,70,34,-86,-75,-30,-59,-42,35,-59,-75,
65,-74,-45,-53,-13,43,95,-98,45,-28,-99,34,-61,-84,66,-14,
-57,-50,-35,31,-55,-87,82]

+++ OK, passed 100 tests.
```

（3）Testable 类型类

Testable and some instances:

```
class Testable prop where
    property :: prop -> Property
    exhaustive :: prop -> Bool
instance Testable Bool
instance Testable Property
instance (Arbitrary a, Show a, Testable prop) =>
            Testable (a -> prop)
```

（4）Controlling test cases 控制案例属性 (e.g. number and size)

```
quickCheckWith :: Testable prop ⇒ Args → prop → IO ()
quickCheckWith (stdArgs {maxSize = 10,
                         maxSuccess = 1000}) prop_XXX
```

（5）Labelling and Coverage

label attaches a label to a test case:

```
label :: Testable prop ⇒ String → prop → Property
```

如：

```
prop_RevRev :: [Int ] ->Property
prop_RevRev xs = label ("length is " ++ show (length xs ))
                       $ reverse (reverse xs) == xs
```

测试结果：

```
> quickCheck prop_RevRev
+++ OK, passed 100 tests:
 6% length is 3
 5% length is 1
 5% length is 14
 4% length is 2
 4% length is 4
...
```

2 测试数据生成器

QuickCheck 中定义了 Arbitrary 类型类，其中的 arbitrary 函数定义了如何生成测试数据。

（1）Arbitrary 类型类

```
class Arbitrary a where
    arbitrary :: Gen a
    shrink :: a → [ a ]
```

（2）generate 函数

```
generate :: Gen a → IO a
```

generate 函数用于生成一组指定类型的值

```
> generate (arbitrary :: Gen [ Int ])
[24,9,-7,-18,-8,-18,7,27,19,22,-7,30,17,25,4,23,-6,-23,-21,22,-
28,9,-3,-24,-16]
> generate (arbitrary :: Gen [ Int ])
[-7]
> generate (arbitrary :: Gen [ Int ])
[5,6,29,-15,-13,-15,-13,-30,-3,24]
```

3 Stating Properties 属性声明

在对函数测试时，如果需要参数满足特有的条件，可以使用 ==> 来做限定， ==> 的 typesignature 如下：

```
(==>) :: Testable prop ⇒ Bool -> prop -> Property
```

Universal quantification:

```
forAll :: (Show a, Testable prop) ⇒ Gen a → (a → prop) → Property
```

例如：

```
prop_Index :: Eq a => [a] -> Property
prop_Index xs = length xs > 0 ==>
                forAll (choose (0, length xs − 1)) $ \i ->
                xs !! i == head (drop i xs)
```

Lecture 16 Optics

1 What is Optics(透镜组)

- Optics are functional references: focusing on one part of a structure for access an update.
- Examples of "optics" include Lens, Prism, Iso, Traversable.
- Different kinds of "optics" allow different number of focal points and may or may not be invertible.
- Today, we'll look at lenses. Lenses compose very nicely, allowing focusing on the target step-by-step.

2 Some other useful lenses

The Lens package defines lots of optics for standard types. In particular, it defines lenses for all field of tuples up to size 19. For example:

```
> view _2 (3,4,5)
4
> view _2 (5,6,7,8)
6
> set _3 9 (5,6,7,8)
(5,6,9,8)
> over _3 (*2) (3, 4, 5)
(3, 4, 10)
```

Guest Lecture

Functional Programming in Real World

Case 1: Shop.com Merchant System

Shop.com is an e-commerce site offering access to vendors organised in a taxonomy. Driving the Shop.com web site is Shop.com Merchant Management, a database storing shopping categories and merchant links. The database itself is stored as a Haskell source file. The codebase generates SQL insert commands which are then piped into a sql database server. Shell scripts generate the site from the database server. (The original version generated the site in Haskell direct from the Haskell database.)

Case 2: Bluespec

Bluespec, Inc. provides tools for chip design that draw much inspiration from Term Rewriting Systems and Haskell. Bluespec also uses Haskell to implement many of its tools.
Bluespec's products include synthesis, simulation and other tools for two languages: BSV and ESE.

BSV incorporates Haskell-style polymorphism and overloading (typeclasses) into SystemVerilog's type system. BSV also treats modules, interfaces, rules, etc. as first-class objects, permitting very powerful static elaboration (including recursion).