Chapter 15

Functional Programming

Topics

- Lists Operations
- Trees
- Lazy Evaluation

Concat

- The function `concat` concatenates a list of lists into one long list.

```
? concat [[1,2],[3,2,1]]
[1,2,3,2,1]
```

- Definition

```
concat           :: [[α]] → [α]
concat []        = []
concat (xs:xss)  = xs ++ concat xss
```

- Basic property:

```
concat (xss ++ yss) = concat xss ++ concat yss
```


Take and drop

- The function `take` and `drop` each take a nonnegative integer `n` and a list `xs` as arguments.
  - The value `take n xs` consists of the first `n` elements of `xs`
  - The value `drop n xs` is what remains

```
? take 3 "functional"  ? take 3 [1,2]
"fun"                  [1,2]
? drop 3 "functional"  ? drop 3 [1,2]
"ctional"              []
```


Take and drop

- Definitions:

```
take             :: Int → [α] → [α]
take 0 xs        = []
take n []        = []
take (n+1)(x:xs) = x:take n xs

drop             :: Int → [α] → [α]
drop 0 xs        = xs
drop n []        = []
drop (n+1)(x:xs) = drop n xs
```


Take and drop

- These definitions use a combination of pattern matching with natural numbers and lists.
- Patterns are disjoint and cover all possible cases.
  - Every natural number is either zero (first equation) or
  - The successor of a natural number
    - Distinguish between an empty list (second equation) and
    - A nonempty list (third equation).

Take and drop

- There are two arguments on which pattern matching is performed
  - Pattern matching is performed on the clauses of a definition in order from the first to the last.
  - Within a clause, pattern matching is performed from left to right.

```
? take 0 ⊥
[]
? take ⊥ []
⊥
```


Take and drop

- The functions take and drop satisfy a number of useful laws:

```
take n xs ++ drop n xs = xs
```

for all (finite) natural numbers `n` and all lists `xs`.

```
take ⊥ xs ++ drop ⊥ xs = ⊥ ++ ⊥ = ⊥
```

not `xs`.

```
take m · take n = take (m min n)
drop m · drop n = drop (m + n)
take m · drop n = drop n · take(m + n)
```


List index

- A list `xs` can be indexed by a natural number `n` to find the element appearing at position `n`.
- This operation is denoted by `xs !! n`

```
? [1,2,3,4]!!2
3
? [1,2,3,4]!!0
1
```

  - Indexing begins at 0.

List index

- Definition

```
(!!)          :: [α] → Int → α
(x:xs)!!0     = x
(x:xs)!!(n+1) = xs!!n
```

- Indexing is an expensive operation since `xs!!n` takes a number of reduction steps proportional to `n`.

Map

- The function *`map`* applies a function to each element of a list.

```
? map square [9,3]
[81,9]

? map (<3) [1,2,3]
[True,True,False]

? map nextLetter "HAL"
"IBM"
```


Map: definition

- The definition is

```
map          :: (α→β)→[α]→[β]
map f []     = []
map f (x:xs) = f x:map f xs
```

- The use of map is illustrated by the following example:
  - *"the sum of the squares of the integers from 1 up to 100"*
  - The function `sum` and `upto` can be defined by

Map: example

```
sum          :: (Num α)⇒[α]→α
sum []       = 0
sum (x:xs)   = x + sum xs
upto         :: (Integral α)⇒α→α→[α]
upto m n     = if m > n then []
               else m:upto(m+1)n

? sum(map square(upto 1 100))
338700

[m..n] = upto m n
[m..] = from m
```


Map: laws

`map id = id`

- Applying the identity function to every element of a list leaves the list unchanged.
  - The two occurrences of `id` have different types; on the left `id :: α → α`, and on the right `id :: [α] → [α]`

`map (f · g) = map f · map g`

- Applying `g` to every element of a list, and the applying `f` to each element of the result gives the same result as applying `f · g` to the original list.

Map: laws

```
f · head = head · map f
map f · tail = tail · map f
map f · reverse = reverse · map f
map f · concat = concat · map(map f)
map f (xs ++ ys) = map f xs ++ map f ys
```

- The common theme behind each of these equations concern the types of the functions involved:

```
head      :: [α] → α
tail      :: [α] → [α]
reverse   :: [α] → [α]
concat    :: [[α]] → [α]
```


Map: laws

- Those functions do not depend in any way on the nature of the list elements.
  - They are simply combinators that shuffle, rearrange, or extract elements from lists.
  - This is why they have polymorphic types.
- We can either 'rename' the list elements (via `map f`) and then do the operation, or do the operation and then rename the elements.

Filter

- The function `filter` takes a boolean function `p` and a list `xs` and return that sublist of `xs` whose elements satisfy `p`.

```
? filter even [1,2,4,5,32]
[2,4,32]

? (sum · map square · filter even) [1..10]
220
```

  - The sum of the squares of the even integers in the range 1 to 10

Filter: definition

```
filter           :: (α→Bool)→[α]→[α]
filter p []      = []
filter p (x:xs)  = if p x then x:filter p xs
                   else filter p xs
```

- Some laws

```
filter p · filter q = filter (p and q)
Filter p · concat = concat · map(filter p)
```

Zip

- The function `zip` takes two lists and returns a list of pairs of corresponding elements.

```
? zip [0..4] “hello”
[(0,’h’),(1,’e’),(2,’l’),(3,’l’),(4,’o’)]

? zip [0,1] “hello”
[(0,’h’),(1,’e’)]
```


Zip: definition

- If two lists do not have the same length, then the length of the zipped list is the shorter of the lengths of the two arguments.

```
zip              :: [α]→[β]→[(α,β)]
zip [] ys        = []
zip xs []        = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

  - What would happen if we just defined `zip [] []` instead of the two basic cases.

Unzip

- The function unzip takes a list of pairs and unzips it into two lists.

```
? unzip [(1,True),(2,True),(3,False)]
([1,2,3],[True,True,False])
```

- Definition

```
unzip  :: [(α,β)]→([α],[β])
unzip  = pair(map fst, map snd)
```


Unzip

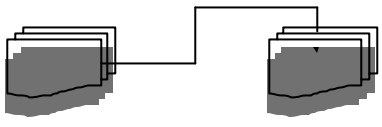
- Two basic functions on pairs are `fst` and `snd`, defined by:

```
fst          :: (α,β) → α
fst (x,y)    = x
snd          :: (α,β) → β
snd (x,y)    = y
```

- A basic function that takes pairs of functions as arguments:

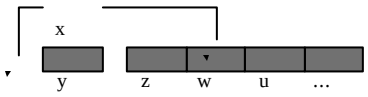
```
pair         :: (α→β,α→γ)→α→(β,γ)
pair (f,g) x  = (f x, g x)
```


Insertion Sort



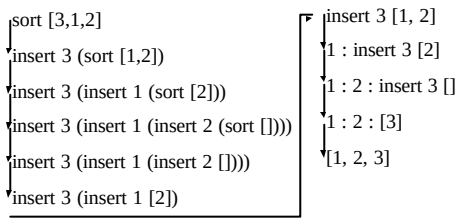
```
sort [] = []
sort (x : xs) = insert x (sort xs)
```


Insertion




```
insert x (y : ys)
            | x<=y      = x : y : ys
            | x>y       = y : insert x ys
insert x [] = [x]
```

Sorting: example



The Type of Sort

What is the type of sort?

```
sort :: [a] -> [a]
```

Can sort many different types of data.

But not all!

Consider a list of functions, for example...

The Correct Type of Sort

```
sort :: Ord a => [a] -> [a]
```

If a has an ordering...

…then sort has this type.

Sort has this type because

```
(<=) :: Ord a => a -> a -> Bool
```

Overloaded, rather than polymorphic.

Polymorphism vs. Overloading

- A *polymorphic* function works in the same way for every type
  - Example: `length, ++`
- An *overloaded* function works in different ways for different types
  - Example: `==, <=`

A Better Way of Sorting

- Divide the list into two roughly equal halves.
- Sort each half.
- Merge the sorted halves together.

Merge Sort: definition

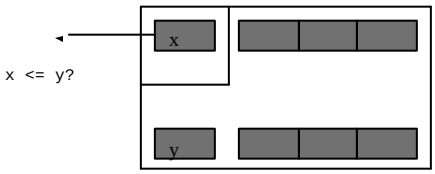
```
mergeSort xs = merge (mergeSort front)
                     (mergeSort back)
          where size = length xs `div` 2
                front = take size xs
                back = drop size xs
```

- But when are front and back smaller than `xs`?

MergeSort with Base Cases

```
mergeSort []             = []
mergeSort [x]            = [x]
mergeSort xs | size > 0  =
                  merge (mergeSort front)
                  (mergeSort back)
         where size = length xs `div` 2
               front = take size xs
               back = drop size xs
```


Merging: example




```
merge [1, 3] [2, 4]  → 1 : merge [3] [2, 4]
                     → 1 : 2 : merge [3] [4]
                     → 1 : 2 : 3 : merge [] [4]
                     → 1 : 2 : 3 : [4]  → [1,2,3,4]
```


Defining Merge

Requires an ordering.

```
merge :: Ord a => [a] -> [a] -> [a]
merge (x : xs) (y : ys)
      | x <= y      = x : merge xs (y : ys)
      | x > y       = y : merge (x : xs) ys
merge [] ys         = ys
merge xs []         = xs
```

One list gets smaller.

Two possible base cases.

The Cost of Sorting

Insertion Sort
Sorting n elements takes n*n/2 comparisons.

Merge Sort
Sorting n elements takes n*log2 n comparisons.

Num elements	Cost by insertion	Cost by merging
10	50	40
1000	500000	10000
1000000	500000000000	20000000

Summary: List Recursion

- Recursive case: expresses the results in terms of the same function on a shorter list.
  - `f (x:xs) = … f xs …`
- Base case(s): handles the shortest possible list.
  - `f [] = …`

Example: Counting Words

Input

A string representing a text containing many words. For example

```
"hello clouds hello sky"
```

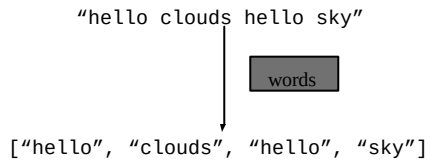
Output

A string listing the words in order, along with how many times each word occurred.

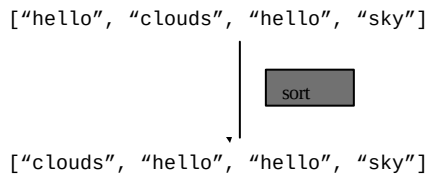
```
"clouds: 1\nhello: 2\nsky: 1"
```

```
clouds: 1
hello: 2
sky: 1
```

Step 1: Breaking Input into Words



Step 2: Sorting the Words



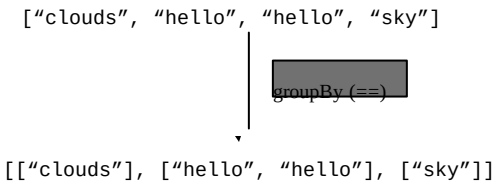
The groupBy Function

```
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

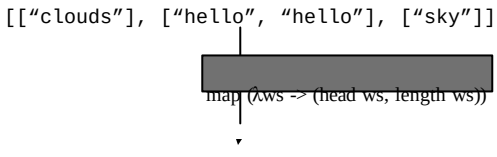
groupBy p xs -- breaks xs into segments [x1,x2…], such that p xi is True for each xi in the segment.

```
groupBy (<) [3,2,4,1,5] = [[3], [2,4], [1,5]]
groupBy (==) "hello" = ["h", "e", "ll", "o"]
```

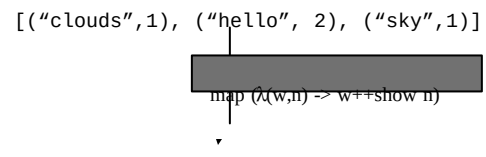

Step 3: Grouping Equal Words



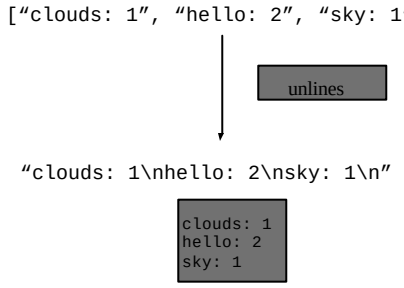
Step 4: Counting Each Group



Step 5: Formatting Each Group



Step 6: Combining the Lines



The Complete Definition

```
countWords :: String -> String
countWords s =
      unlines .
      map (λ(w,n) -> w++show n) .
      map (λws -> (head ws, length ws)) .
      groupBy (==) .
      sort .
      words s
```


Trees

- Any recursive data type that exhibits a nonlinear structure is generically called a tree.
- The syntactic structure of arithmetic or functional expressions can also be modeled by a tree.
- There are numerous species and subspecies of tree.

Trees

- Trees can be classified according to
  - The precise form of the branching structure
  - The location of information within the tree
  - The relationship between the information stored in different parts of the tree

Binary Trees

- A binary tree is a tree with a simple two-way branching structure.

`data Btree α = Leaf α | Fork(Btree α)(Btree α)`

  - A value of `Btree α` is either a *leaf node*, which contains a value of type α, or a *fork node*, which consists of two further trees, called the *left* and *right subtrees* of the node.
  - A leaf is sometimes called an external node, or tip, and a fork node is sometimes called an internal node.

Binary Trees

- Example:

`Fork(Leaf 1)(Fork(Leaf 2)(Leaf 3))`

  - Consists of a node with a left subtree `Leaf 1` and a right subtree which consists of a left subtree `Leaf 2` and a right subtree `Leaf 3`.

`Fork(Fork(Leaf 1)(Leaf 2))(Leaf 3)`

  - Contains the same sequence of numbers in its leaves but the way the information is organized is different.
  - The two expressions denote different values.

Trees: size

- The `size` of a tree is the number of its leaf nodes.

```
size              :: Btree α → Int
size (Leaf x)     = 1
size (Fork xt yt) = size xt + size yt
```

  - The function `size` plays the same role for trees as `length` does for lists.

`size = length · flatten` , where

```
Flatten             :: Btree α → [α]
Flatten (Leaf x)    = [x]
Flatten (Fork xt yt) = flatten xt ++ flatten yt
```


Trees: height

- The `height` of a tree measures how far away the furthest leaf is.

```
height              :: Btree α → Int
height (Leaf x)     = 0
height (Fork xt yt) = 1 +
                      (height xt max height yt)
```


Reductions

- Reduction sequence: `square (3+4)`
- Two reduction policies
  - *Innermost reduction*: a reduction that contains no other reduction.
  - Outermost reduction: a reduction that is contained in no other reduction.
- Other example: `fst (square 4, square 2)`

Outermost Reduction

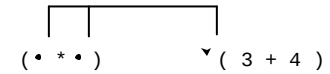
- Sometimes outermost reduction will give an answer when innermost fails to terminate.
- If both methods terminate, then they give the same result.
- Outermost reduction has the important property that if an expression has a normal form then the outermost reduction will compute it.

Outermost Reduction

- Is outermost reduction a better choice than innermost reduction?
- Problem: outermost reduction can sometimes require most steps than innermost reductions.
  - The problem arises with any function whose definition contains repeated occurrences of an argument.

Outermost Reduction

- The problem can be solved by representing expressions as graphs rather than trees.
  - Unlike trees, graphs can share subexpressions.
- Example: `the expression (3+4) * (3+4)`



  - Each occurrence of 3+4 is represented by an arrow, called a pointer, to a single instance of `(3+4)`

Outermost Reduction

- Using outermost graph reduction has only three steps.
  - The representation of expressions as graphs means that duplicated subexpressions can be shared and reduced at most once.
- With graph reduction, outermost reduction never takes more steps than innermost reduction.

Lazy vs. Eager Evaluation

- Outermost graph reduction is called *lazy evaluation*.
- Innermost graph reduction is called *eager evaluation*.
- Lazy evaluation is adopted by Haskell:
  1. It terminates whenever any reduction order terminates.
  2. It requires no more (and possibly fewer) steps than eager evaluation.