

Lists - The Full Story

Today

How are lists constructed?

Pattern matching on lists.

Recursion over lists.

Examples: sorting, selection, transforming, ...

2

Handling lists in Haskell

Lists are the standard **collection type** in Haskell.

How have we dealt with them so far? **Toolbox**

- Predefined lists `[a .. b]` and strings `"list"`
- List comprehensions: `[2*a | (a,b)<-list, a>b]`
- Library functions: `replicate`, `nub`, `length`, ...

3

How are lists constructed?

A list is either empty, `[]`, ...

... or non-empty, e.g. `[4,1,6]`.

In that case we can think of it being built by adding `4` to the front of `[1,6]`.

`4:[1,6] = [4,1,6]`

The **cons** operator, `:`, adds an item to the front of a list:

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

4

Constructing lists

Questions for you. How are these lists built?

`[3]`
`3:[]`

`[2,1,3]`
`2:[1,3]`
`2:(1:[3])`
`2:(1:(3:[]))`
`2:1:3:[]`

`:` is right
associative

5

Building lists

Lists are built up by repeatedly applying `:` ... starting from `[]`.

... just like we can think of numbers begin build up from `0` by adding `1`.

6

Pattern matching lists

Every list is either empty, [], or not (x:xs).

```
isNull :: [a] -> Bool
isNull []      = True
isNull (x:xs) = False
```

It is a convention to call variables for lists **xs**, **ys**, etc. They are read 'exes', etc.

Pattern matching to give different **cases**.

7

More pattern matching

```
head :: [a] -> a
head []      = error "head of []"
head (x:_)  = x

mystery :: [Int] -> Int
mystery []   = 0
mystery [x]  = x
mystery (x:y:zs) = x+y
```

Pattern matching to **extract parts** of a list.

8

Pattern matching illustrated

Pattern matching to **extract parts** of a list.

```
mystery (x:y:zs) = x+y
mystery [2,1,7,3] = 2+1

isNull (x:xs) = False
isNull [3] = False
```

9

Recursion and calculation

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs

sum [2,7,5]
-> 2 + sum [7,5]
-> 2 + (7 + sum [5])
-> 2 + (7 + (5 + sum []))
-> 2 + (7 + (5 + 0))
-> 14
```

10

Primitive recursion for lists

A template for **primitive recursive** definitions:

```
fun []      = ....
fun (x:xs)  = .... fun xs ....
```

All primitive recursive functions terminate!

Examples ...

11

Example: concatenating lists

```
concat :: [[a]] -> [a]
concat [x1, x2, ..., xn] = x1 ++ x2 ++ ... ++ xn
```

The template for **primitive recursive** definitions:

```
concat []      = []
concat (x:xs)  = .... concat xs ....
```

```
concat [[4,3],[],[1]] = [4,3,1]
concat [[2,3],[4,3],[],[1]] = [2,3,4,3,1]
```

```
concat (x:xs) = x ++ concat xs
```

12

Example: list membership

```
elem :: Int -> [Int] -> Bool
elem 1 [2,1] = True
elem 1 [2]   = False
```

Nothing is an element of [] ...

```
elem n [] = False
```

... to be an element of (x:xs) you're either equal to x or an element of xs.

```
elem n (x:xs) = (n==x) || elem n xs
```

13

Example: double all list items

```
doubleAll :: [Int] -> [Int]
doubleAll [2,1] = [4,2]
```

The empty list [] has no elements to double ...

```
doubleAll [] = []
```

... to double everything in (x:xs) double x and double all elements of xs.

```
doubleAll (x:xs) = 2*x : doubleAll xs
```

14

A calculation using doubleAll

```
doubleAll [] = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

```
doubleAll [2,1,3]
~> 2*2 : doubleAll [1,3]
~> 4 : (2*1 : doubleAll [3])
~> 4 : (2 : (2*3 : doubleAll []))
~> 4 : (2 : (6 : []))
~> [4,2,6]
```

15

Example: select just the even items

```
selectEven :: [Int] -> [Int]
selectEven [4,2,1] = [4,2]
selectEven [3,2,1] = [2]
```

```
selectEven [] = []
```

Select the evens in xs ... x is included only if it's even.

```
selectEven (x:xs)
| isEven x = x : selectEven xs
| otherwise = selectEven xs
```

16

Example: sorting a list.

```
iSort :: [Int] -> [Int]
iSort [3,9,2] = [2,3,9]
iSort [7,3,9,2] = [2,3,7,9]
```

```
iSort [] = []
```

Sort xs and insert x in the correct place.

```
iSort (x:xs) = insert x (iSort xs)
```

A perfect example of top-down design ... now we have to implement insert!

17

Example: insert into a sorted list

```
insert :: Int -> [Int] -> [Int]
insert 1 [2,3,9] = [1,2,3,9]
insert 7 [2,3,9] = [2,3,7,9]
```

```
insert n [] = [n]
```

```
insert n (x:xs)
```

If n is smaller than x, it goes at the front ...

```
| n<=x = n:(x:xs)
```

otherwise x goes at the front and n inserted into xs.

```
| otherwise = x : insert n xs
```

18

Removing all duplicates: nub

```
removeEl :: Int -> [Int] -> [Int]
removeEl n xs = [ x | x <-xs, x/=n ]
```

```
nub [] = []
nub (x:xs) = x : nub (removeEl x xs)
```

Primitive recursive?

19

More general recursion patterns

We said that primitive recursion took the form

```
fun [] = ...
fun (x:xs) = ... fun xs ...
```

This will always give a result.

That is also true for other forms of recursion:

```
fun xs = ... fun xs1 ...
```

As long as xs_1 is smaller than xs , then the recursion will make progress to the base case.

20

Zippping lists

Go along two lists simultaneously; example

```
zip [True,False] [2,4,7] = [(True,2), (False,4)]
```

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
zip (x:xs) [] = []
```

```
zip [] (y:ys) = []
```

```
zip [] [] = []
```

Can replace the last three lines by

```
zip xs ys = []
```

because to reach this line at least one list must be [].

21

Quicksort of a list: idea

```
[4,2,7,1,4,6,5]
```

Split the list into small and large. Take the first element as splitting point and split the rest...

```
[2,1,4] [7,6,5]
```

sort each half ...

```
[1,2,4] [5,6,7]
```

and reassemble ...

```
[1,2,4] ++ [4] ++ [5,6,7]
```

to give ...

```
[1,2,4,4,5,6,7]
```

22

Quicksort of a list: program

```
qSort :: [Int] -> [Int]
```

```
qSort [] = []
```

```
qSort (x:xs) = qSort [y | y<-xs, y<=x] ++
               [x] ++
               qSort [y | y<-xs, y>x]
```

23

Recursion as part of the toolbox

Lots of other examples in the book.

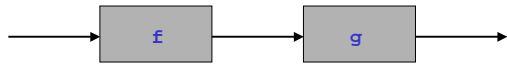
In the remainder of the lecture, build up a larger program: indexing, from Section 10.8.

First one more component of the toolbox.

24

Joining functions together

```
f >.> g
```



```
(f >.> g) x = g (f x)
```

We can build a function as a **pipeline** of functions joined together by `>.>`.

25

Indexing a document

```
cat dog cat
bat dog cat
cat
```

```
"cat dog cat\nbat dog cat\ncat"
```

```
bat    2
cat    1,1,2,3
dog    1,2
```

26

Indexing a document: example

```
"cat dog cat\nbat dog cat\ncat"
```

```
["cat dog cat", "bat dog cat", "cat"]
```

```
[(1,"cat dog cat"), (2,"bat dog cat"), (3,"cat")]
```

```
[(1,"cat"), (1,"dog"), (1,"cat"), (2,"bat"),  
(2,"dog"), (2,"cat"), (3,"cat")]
```

27

Indexing a document: example cont.

```
[(1,"cat"), (1,"dog"), (1,"cat"), (2,"bat"),  
(2,"dog"), (2,"cat"), (3,"cat")]
```

```
[(2,"bat"), (1,"cat"), (1,"cat"), (2,"cat"),  
(3,"cat"), (1,"dog"), (2,"dog")]
```

```
[[2],"bat"), ([1],"cat"), ([1],"cat"),  
[2],"cat"), ([3],"cat"), ([1],"dog"), ([2],"dog")]
```

```
[[2],"bat"), ([1,1,2,3],"cat"), ([1,2],"dog")]
```

28

Indexing: top-level

```
makeIndex :: String -> [ ([Int],String) ]
```

```
makeIndex  
  = lines >.>  
    numLines >.>  
    allNumWords >.>  
    sortLs >.>  
    makeLists >.>  
    amalgamate >.>  
    shorten
```

29

Amalgamating

```
amalgamate :: [([Int],String)] -> [([Int],String)]
```

```
amalgamate [(2),"bat"), ([1],"cat"), ([1],"cat")]  
  = [(2),"bat"), ([1,1],"cat")]
```

```
amalgamate [] = []
```

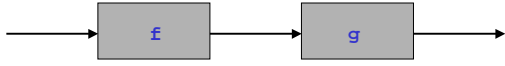
```
amalgamate [p] = [p]
```

```
amalgamate ((l1,w1):(l2,w2):rest)  
  | w1/=w2    = (l1,w1) : amalgamate ((l2,w2):rest)  
  | otherwise = amalgamate ((l1++l2,w2):rest)
```

30

Predefined function composition

$g . f$



$(g . f) x = g (f x)$