Watch out for the arrows
Mapping via list comprehension

```haskell
doubleAll :: [Int] -> [Int]
doubleAll lst = [ 2*x | x <- lst ]

addPairs :: [(Int,Int)] -> [[Int]]
addPairs mns = [[m+n] | (m,n) <- mns ]

multAll :: Int -> [Int] -> [Int]
multAll x ys = [ x*y | y <- ys ]
```

More generally for any function \( f :: a \rightarrow b \), we can define a function

```haskell
apply_f :: [a] -> [b]
apply_f xs = [f x | x <- xs]
```
A start on higher types: Mapping, 2

Mapping via structural recursion over lists

```haskell
doubleAll' :: [Int] -> [Int]
doubleAll' [] = []
doubleAll' (x:xs) = (2*x):doubleAll xs

doublePairs' :: [(Int,Int)] -> [[Int]]
doublePairs' [] = []
doublePairs' ((m,n):mns) = [m+n]:doublePairs mns

multiplyAll' :: Int -> [Int] -> [Int]
multiplyAll' x [] = []
multiplyAll' x (y:ys) = (x*y):(multiplyAll' x ys)
```

More generally for any function \( f :: a \to b \), we can define a function

```haskell
apply_f' :: [a] -> [b]
apply_f' [] = []
apply_f' (x:xs) = (f x):apply_f' xs
```
A start on higher types: Mapping, 3

Mapping via map

Let us define a generic function to do mapping:

\[
\text{map} :: (a \to b) \to [a] \to [b] \\
\text{map } f \text{ lst } = [ f x \mid x \leftarrow \text{lst} ]
\]

—or—

\[
\text{map'} :: (a \to b) \to [a] \to [b] \\
\text{map' } f \text{ [] } = [] \\
\text{map' } f \text{ (x:xs) } = (f \text{ x}):\text{map'} f \text{ xs}
\]

map is higher order, it accepts a function as an argument. E.g.,

\[
\text{map } \text{fst } \text{[(1,\text{False}), (3,\text{True}), (-5,\text{False}), (34,\text{False})]} \leadsto [1,3,-5,34] \\
\text{map } \text{length } \text{[[1,5,6], [3,5], [], [3..10]]} \leadsto [3,2,0,8] \\
\text{map } \text{sum } \text{[[1,5,6], [3,5], [], [3..10]]} \leadsto [12,8,0,52]
\]
Filtering elements from a list via list comprehensions

```haskell
lessThan10 :: [Int] -> [Int]
lessThan10 xs = [ x | x <- xs, x<10 ]

offDiagonal :: [(Int,Int)] -> [(Int,Int)]
offDiagonal mns = [(m,n) | (m,n) <- mns , m/=n]
```
Here is a generic way of doing filtering:

\[
\text{filter} :: \text{(a $\rightarrow$ Bool)} \rightarrow [\text{a}] \rightarrow [\text{a}]
\]
\[
\text{filter} \ p \ \text{lst} = [x \mid x \leftarrow \text{lst}, \ p \ x]
\]
— or —

\[
\text{filter'} :: \text{(a $\rightarrow$ Bool)} \rightarrow [\text{a}] \rightarrow [\text{a}]
\]
\[
\text{filter'} \ p \ [\text{]} \ = \ [\text{}]
\]
\[
\text{filter'} \ p \ (x:xs) \mid p \ x \ = \ x:(\text{filter'} \ p \ xs)
\]
\[
\text{otherwise} = \text{filter'} \ p \ xs
\]

So

\[
\text{isOffDiag} :: \text{(Int,Int)} \rightarrow \text{Bool}
\]
\[
\text{isOffDiag} \ (m,n) = (m/=n)
\]

\[
\text{filter isOffDiag} \ [(3,4),(5,5),(10,-2),(99,99)] \ \leadsto [(3,4),(10,-2)]
\]
\[
\text{filter isDigit} \ "a37bZ9?" \ \leadsto \ "379?"
\]
\[
\text{filter not} \ [\text{True, False, False, True}] \ \leadsto [\text{False, False}]
\]
Functions as First-Class Values

In functional languages (generally), functions are first-class values, i.e. are treated just like any other value. So functions can be

- passed as arguments to functions
- returned as results from functions
- bound to variables
- expressed without being given a name ($\lambda$-expressions)
- elements of list (and other data structures)
- ...

A function that

(i) accepts functions as arguments or
(ii) returns a function as a value or
(iii) both (i) and (ii)

is higher order. E.g., map and filter.
dropWhile, takeWhile
:: (a -> Bool) -> [a] -> [a]

dropWhile p [] = []
dropWhile p (x:xs)
    | p x       = dropWhile p xs
    | otherwise  = x:xs

takeWhile p [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise  = []

Q: What is (<10) doing?
Q: What is “.” doing??

For example:

takeWhile (<10) [0,3..20]    \Rightarrow [0,3,6,9]
dropWhile (<10) [0,3..20]    \Rightarrow [12,15,18]
dropWhile isSpace " hi there "    \Rightarrow "hi there "
takeWhile (not . isSpace) "hi there "    \Rightarrow "hi"
dropWhile (not . isSpace) "hi there "    \Rightarrow " there "
Sections

\[
10 + 3 \equiv (+) 10 3 \equiv (10 +) 3 \equiv (+ 3) 10 \\
10 == 3 \equiv (==) 10 3 \equiv (10 ==) 3 \equiv (== 3) 10 \\
10 \ 'div' \ 3 \equiv \ div \ 10 \ 3 \equiv (10 \ 'div') \ 3 \equiv ('div' \ 3) \ 10
\]

(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)

Example: Define a function \texttt{trim} that deletes leading and trailing white space from a string

\[
\text{trimFront str} = \text{dropWhile isSpace str} \\
\text{trim str} = \text{reverse (trimFront (reverse (trimFront str)))} \\
\ -- \ or \ better \ yet \\
\text{trim'} = \text{reverse} \ . \ \text{trimFront} \ . \ \text{reverse} \ . \ \text{trimFront}
\]
Higher-type goodies, 2

\[
\text{span} :: (a \to \text{Bool}) \to [a] \to ([a],[a])
\]

\[
\begin{align*}
\text{span} \ p \ [] & = ([],[]) \\
\text{span} \ p \ xs@(x:xs') & = \begin{cases} \\
\text{p} \ x & = (x:ys,zs) \\
\text{otherwise} & = ([],xs) \\
\end{cases} \\
\text{where} \ (ys,zs) & = \text{span} \ p \ xs'
\end{align*}
\]

For example:

\[
\begin{align*}
\text{span} \ (<10) \ [0,3..20] & \leadsto ([0,3,6,9],[12,15,18]) \\
\text{span} \ \text{isSpace} \ " \ hi \ there \ " & \leadsto (" ","hi \ there \ ")
\end{align*}
\]

\textbf{Q:} What is the @ doing in “\text{span} \ p \ xs@(x:xs’)”?
Higher-type goodies, 3

\[
\text{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c]
\]

\[
\text{zipWith'} _ [] _ = []
\]
\[
\text{zipWith'} _ _ [] = []
\]
\[
\text{zipWith'} f (x:xs) (y:ys) = f x y : \text{zipWith'} f xs ys
\]

For example:

\[
\text{sum } \text{zipWith} (*) [2, 5, 3] [1.75, 3.45, 0.25]
\]
\[
\leadsto \text{sum } [3.5, 17.25, 0.75]
\]
\[
\leadsto 21.50
\]

\[
\text{zipWith } (\lambda a b \to (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
\]
\[
\leadsto [153.0,61.5,31.0,15.75,6.6]
\]

Q: What is the "\$" doing??

Q: What is the (\lambda a b \to (a * 30 + 3) / b) doing?
Digression: The application operator

\[
($) :: (a \to b) \to a \to b
\]
\[
f \, \$_x = f \, x \quad -- \$ \text{ has low, right-associative binding precedence}
\]

So

\[
\text{sum } \$_\text{filter (> 10)} \$_\text{map (*2) [2..10]}
\]
\[
\equiv
\]
\[
\text{sum (filter (> 10) (map (*2) [2..10]))}
\]
The following definitions are equivalent

\[
\text{munge, munge'} :: \text{Int} \to \text{Int} \\
munge \; x = 3x+1 \\
munge' = \ x \to 3x+1
\]

So the following expressions are equivalent

\[
\text{map munge} \; [2..8] \\
\text{map munge'} \; [2..8] \\
\text{map (} \ x \to 3x+1 \text{) } [2..8]
\]

So, \( (x \to 3x+1) \) defines a “nameless” function.

We can use \( (\ x\to\ ) \) to return functional results. E.g.,

\[
\text{addNum :: Int} \to \text{(Int->Int)} \\
\text{addNum} \; n = \ x\to(x+n)
\]
Consider some structural recursion on lists:

\[
\text{sum'} \; [] = 0
\]
\[
\text{sum'} \; (x:xs) = x + \text{sum'} \; xs \quad \text{--} \quad (+) \; x \; (\text{sum'} \; xs)
\]

\[
\text{concat'} \; [] = []
\]
\[
\text{concat'} \; (xs:xss) = xs ++ \text{concat'} \; xss \quad \text{--} \quad (++) \; xs \; (\text{concat'} \; xxs)
\]

\[
\text{unzip'} \; [] = ([],[])
\]
\[
\text{unzip'} \; ((x,y):xys) = (x:xs,y:ys) \quad \text{--} \quad \text{where} \; f \; (a,b) \; (as,bs)
\]
\[
\text{where} \; (xs,ys) = \text{unzip'} \; xys \quad \text{--} \quad (a:as,b:bs)
\]

These all have the general form:

\[
\text{someFun} \; [] = z
\]
\[
\text{someFun} \; (x:xs) = f \; x \; (\text{someFun} \; xs)
\]

So we can encapsulate this by:

\[
\text{foldr} \; :: \; (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]
\[
\text{foldr} \; f \; z \; [] = z
\]
\[
\text{foldr} \; f \; z \; (x:xs) = f \; x \; (\text{foldr} \; f \; z \; xs)
\]
Higher-types, structural recursion on lists, 2

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

Original

sum' [] = 0
sum' (x:xs) = x + sum' xs

concat' [] = []
concat' (xs:xss) = xs ++ concat' xss

unzip' [] = ([],[])
unzip' ((x,y):xys) = (x:xs,y:ys)
where (xs,ys) = unzip' xys

As a foldr

sum'' xs = foldr (+) 0 xs

concat'' xss = foldr (++) [] xss

unzip'' xys = foldr f ([],[]) xys
where
  f (x,y) (xs,ys) = (x:xs,y:ys)
Higher-types, structural recursion on lists, 3

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
foldr f z [x1, x2, ..., xn]
   == x1 'f' (x2 'f' ... (xn 'f' z)...)
Higher-types, structural recursion on lists, 4

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
Foldr’s cousins, 1: foldl

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

\[
\begin{align*}
\text{foldr } f \ z \ [] & = z \\
\text{foldr } f \ z \ (x:xs) & = f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}
\]

\[
\text{foldr } f \ z \ [x_1, x_2, \ldots, x_n] = x_1 \ 'f' \ (x_2 \ 'f' \ \ldots \ (x_n \ 'f' \ z)\ldots)
\]

\[
\text{foldl} :: (b \to a \to b) \to b \to [a] \to b
\]

\[
\begin{align*}
\text{foldl } f \ z \ [] & = z \\
\text{foldl } f \ z \ (x:xs) & = \text{foldl } f \ (f \ z \ x) \ xs
\end{align*}
\]

\[
\text{foldl } f \ z \ [x_1, x_2, \ldots, x_n] = (\ldots((z \ 'f' \ x_1) \ 'f' \ x_2) \ 'f' \ldots) \ 'f' \ x_n
\]
foldr vs. foldl
Puzzles

**Puzzle 1**

```
foldr (:) [] [1,2,3] = ??
```

**Puzzle 2**

```
foldl (flip (:)) [] [1,2,3] = ??
```

---

**flip**

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```
Puzzles

**Puzzle 1**

```
foldr (:) [] [1,2,3] = ??
```

**Puzzle 2**

```
foldl (flip (:)) [] [1,2,3] = ??
```

**flip**

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

**Pro Tip:** It is almost always better to use `foldl'` than `foldl`. See [https://wiki.haskell.org/Foldr_Foldl_Foldl%27](https://wiki.haskell.org/Foldr_Foldl_Foldl%27) for the gory details.
Foldr’s cousins, 2

For folds: look here
For scans: look here
Foldr’s cousins, 3

foldl

```
f   f   f
  |
  v
a0  a1  ...
```

```
f
  |
  v
b0
```

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

foldr

```
f   f   f
  |
  v
a0  a1  ...
```

```
f
  |
  v
b0
```

```
f
  |
  v
b0
```

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

foldl1

```
f   f   f
  |
  v
a0  a1  ...
```

```
f
  |
  v
a0
```

```
f
  |
  v
a0
```

```
foldl1 :: (a -> a -> a) 
        -> [a]  ->  a
```

foldr1

```
f   f   f
  |
  v
a0  a1  ...
```

```
f
  |
  v
a0
```

```
f
  |
  v
a0
```

```
f
  |
  v
a0
```

```
f
  |
  v
a0
```

```
foldr1 :: (a -> a -> a) 
        -> [a]  ->  a
```

Higher types

---

Foldr’s cousins, 3

Try:

- foldr1 max [1,4,8,4,9,4]
- foldl1 max [1,4,8,4,9,4]
- scanr1 max [1,4,8,4,9,4]
- scanl1 max [1,4,8,4,9,4]
Foldr’s cousin’s: alternatively, 2

\[
\text{scanl1 (+) [1..5] } \sim \ [1, 3, 6, 10, 15] \\
\text{scanr1 (+) [1..5] } \sim \ [15, 14, 12, 9, 5]
\]
Higher types

Foldr’s cousin’s: alternatively, 2

For folds:
http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:11

For scans:
http://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#g:16
1. Use foldr to define \( n \mapsto 1^2 + 2^2 + 3^2 + \cdots + n^2 \).

2. Use foldr and foldl to define length.

3. Use foldr and foldl to define and and or.

4. Use foldr or foldl to define reverse.

5. Use scanr or scanl to define \( n \mapsto [1!, 2!, 3!, \ldots, n!] \).
sumSq \ n = foldr (\ x r \rightarrow x*x+r) 0 [1..n]

length1 \ xs = foldr (\ x r \rightarrow 1+r) 0 \ xs
length2 \ xs = foldl (\ r x \rightarrow 1+r) 0 \ xs

and1 \ bs = foldr (\ x r \rightarrow x && r) True \ bs
and2 \ bs = foldl (\ r x \rightarrow x && r) True \ bs

or1 \ bs = foldr (\ x r \rightarrow x || r) False \ bs
or2 \ bs = foldl (\ r x \rightarrow x || r) False \ bs

reverse' \ xs = foldl (flip(:)) [] \ xs

facts \ n = scanl (*) 1 [2..n]
We can introduce a “natural number data type” by:

\[
\text{data Nat} = \text{Zero} \mid \text{Succ Nat}
\]

where \(\text{Zero}\) stands for 0 and \(\text{Succ}\) stands for the function \(x \mapsto x + 1\).

A structural recursion over \(\text{Nat}'\)s is a function of the form:

\[
\text{fun} :: \text{Nat} \rightarrow a
\]

\[
\text{fun Zero} = z
\]

\[
\text{fun (Succ n)} = f \left( \text{fun n} \right)
\]

where \(z::a\) and \(f::a \rightarrow a\). So if you expand things out, you see that

\[
\text{fun} \left( \text{Succ} \left( \text{Succ} \left( \ldots \text{Zero} \right) \right) \right) = \left( f \left( f \left( \ldots z \right) \right) \right)
\]

\(k\) many \(\text{Succ}'\)s \(k\) many \(f'\)s

We can define a fold for \(\text{Nat}'\)s by:

\[
\text{foldn} :: (a \rightarrow a) \rightarrow a \rightarrow \text{Nat} \rightarrow a
\]

\[
\text{foldn f z Zero} = z
\]

\[
\text{foldn f z (Succ n)} = f \left( \text{foldn f z n} \right)
\]
Aside: Structural Recursions on Natural Numbers, 2

Using

```
data Nat = Zero | Succ Nat

foldn :: (a->a) -> a -> Nat -> a
foldn f z Zero = z
foldn f z (Succ n) = f (foldn f z n)
```

we can bootstrap arithmetic by:

```
add m n = foldn Succ n m
times m n = foldn ('add' n) Zero m
```

*etc.*
Functions and types

In Haskell every function

- takes exactly one argument and
- returns exactly one value.

For example: \( f :: \text{Int} \to \text{Bool} \)

In general: \( g :: t_1 \to t_2 \)

Examples:

- \( (\text{Int} \to \text{Bool}) \to \text{Char} \)
- \( \text{Int} \to (\text{Bool} \to \text{Char}) \equiv \text{Int} \to \text{Bool} \to \text{Char} \)

\( \to \) associates to the right

\[ t_1 \to t_2 \to \cdots \to t_n \to t \equiv t_1 \to (t_2 \to \cdots (t_n \to t) \cdots) \]
Associations

Convention: \( \rightarrow \) associates to the right

\[ t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots \rightarrow t_n \rightarrow t \equiv t_1 \rightarrow (t_2 \rightarrow (t_3 \rightarrow (\ldots (t_n \rightarrow t) \ldots ))) \]

Convention: application associates to the left

\[ f \ x_1 \ x_2 \ x_3 \ \cdots \ x_n \equiv (\ldots (((f \ x_1) \ x_2) \ x_3) \ldots \ x_n) \]

WHY?

Suppose

\begin{align*}
  f &:: t_1 \to t_2 \to t_3 \to t \\
  e_1 &:: t_1 \\
  e_2 &:: t_2 \\
  e_3 &:: t_3
\end{align*}

Then

\begin{align*}
  f \ e_1 &:: t_2 \to t_3 \to t \\
  f \ e_1 \ e_2 &:: t_3 \to t \\
  f \ e_1 \ e_2 \ e_3 &:: t
\end{align*}
Currying and Uncurrying

Consider

\[
\begin{align*}
\text{comp1} & : \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\
\text{comp1 } x \ y &= (x < y) \\
\text{comp2} & : (\text{Int,Int}) \rightarrow \text{Bool} \\
\text{comp2 } (x,y) &= (x < y)
\end{align*}
\]

Every \( f :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \) has a corresponding \( f' :: (t_1,t_2,\ldots,t_n) \rightarrow t \) and vise versa.

In fact

\[
\begin{align*}
\text{curry2} & :: ((a,b)\rightarrow c) \rightarrow a \rightarrow b \rightarrow c \\
\text{curry2 } g &= \ \backslash x \ y \rightarrow g(x,y) \\
\text{uncurry2} & :: (a\rightarrow b\rightarrow c) \rightarrow (a,b) \rightarrow c \\
\text{uncurry2 } f &= \ \backslash (x,y) \rightarrow f \ x \ y
\end{align*}
\]

Mathematically: This is just a fancier version of:

\[
(c^b)^a = c^{a \times b}
\]

from High School math.
Was that so bad?