Functional programming
Lecture 04 – High-order functions

(version: 2025-11-03–17:03:06)

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

High-order functions

High-order functions

- A function that takes a function as an argument or returns a function as a result is called a high-order function.
- Because the term curried already exists for returning functions as results, the ther high-order is often just used for taking functions as arguments.
- Using high-order functions considerably increases the power of Haskell by allowing common programming patterns to be encapsulated as functions within the language itself.

Filtering

`Data.List.filter :: (a -> Bool) -> [a] -> [a]`

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

Filtering

```
Data.List.filter :: (a -> Bool) -> [a] -> [a]
```

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
λ > filter even [1..10]
[2,4,6,8,10]
λ > filter (\x -> x `mod` 2 == 0) [1..10]
[2,4,6,8,10]
λ > filter (\x -> even x && odd x) [1..10]
[]
λ > filter (\_ -> True) [1..10]
[1,2,3,4,5,6,7,8,9,10]
λ > filter (\_ -> False) [1..10]
[]
```

Filtering

```haskell
Data.List.filter :: (a -> Bool) -> [a] -> [a]
```

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
λ > filter (\x -> x > 5) [1,5,2,6,3,7,4,8]
[6,7,8]
λ > filter (> 5) [1,5,2,6,3,7,4,8]
[6,7,8]
λ > filter (\x -> x <= 5) [1,5,2,6,3,7,4,8]
[1,5,2,3,4]
λ > filter (<= 5) [1,5,2,6,3,7,4,8]
[1,5,2,3,4]
```

Filtering

```
Data.List.filter :: (a -> Bool) -> [a] -> [a]
```

`filter` applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
-- recursive
filter1 :: (a -> Bool) -> [a] -> [a]
filter1 _ [] = []
filter1 p (x : xs)
  | p x       = x : filter1 p xs
  | otherwise = filter1 p xs

-- with a list comprehension
filter2 :: (a -> Bool) -> [a] -> [a]
filter2 p xs = [x | x <- xs, p x]
```

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

Mapping

`Data.List.map :: (a -> b) -> [a] -> [b]`

`map f xs` is the list obtained by applying `f` to each element of `xs`.

```
λ > map (*2) [1..5]
[2,4,6,8,10]
λ > map even [1..5]
[False,True,False,True,False]
λ > map (\x -> 2*x) [1..5] -- == map (2*) [1..5]
[2,4,6,8,10]
λ > map (\x -> [x]) [1..5]
[[1],[2],[3],[4],[5]]
λ > map (\x -> replicate x x) [1..5]
[[1],[2,2],[3,3,3],[4,4,4,4],[5,5,5,5,5]]
```

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

```
λ> map (map (* 2)) [[1,2,3],[4,5,6],[7,8,9]]
[[2,4,6],[8,10,12],[14,16,18]]
λ> map (filter even) [[1,2,3],[4,5,6],[7,8,9]]
[[2],[4,6],[8]]
λ> map length [[1,2,3],[4,5,6],[7,8,9]]
[3,3,3]
λ> map (take 2) [[1,2,3],[4,5,6],[7,8,9]]
[[1,2],[4,5],[7,8]]
```

Mapping

```
Data.List.map :: (a -> b) -> [a] -> [b]
```

`map f xs` is the list obtained by applying `f` to each element of `xs`.

```
-- recursive
map1 :: (a -> b) -> [a] -> [b]
map1 _ [] = []
map1 f (x : xs) = f x : map1 f xs

-- with a list comprehension
map2 :: (a -> b) -> [a] -> [b]
map1 f xs = [f x | x <- xs]
```

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad M' = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
m = [[1,2],
     [3,4],
     [5,6]]
```

```
m' = [[1,2,0,0,0,0,0],
      [3,4,0,0,0,0,0],
      [5,6,0,0,0,0,0]]
```

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```
λ > m = [[1,2],[3,4],[5,6]]
λ > addExtraColumns 0 m
[[1,2],[3,4],[5,6]]
λ > addExtraColumns 1 m
[[1,2,0],[3,4,0],[5,6,0]]
λ > addExtraColumns 5 m
[[1,2,0,0,0,0,0],[3,4,0,0,0,0,0],[5,6,0,0,0,0,0]]
```

Mapping – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```haskell
addExtraColumns1 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns1 k xss = map (++ replicate k 0) xss

addExtraColumns2 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns2 k xss = map (++ zs) xss
  where
    zs = replicate k 0
```

Exercice

Define a function `nestedReverse` which takes a list of strings as its argument and reverses each element of the list and then reverses the resulting list.

```
λ> nestedReverse ["in", "the", "end"]
["dne", "eht", "ni"].
```

Exercice

Define a function `atFront :: a -> [[a]] -> [[a]]` which takes an object and a list of lists and sticks the object at the front of every component list.

```
λ > atFront 7 [[1,2], [], [3]]
[[7,1,2], [7], [7,3]]
```

Exercice

the function `filter` can be defined in terms of `concat` and `map`:

```
filter p = concat . map box
  where
    box x = ...
```

Complete this definition of `filter` by defining `box`.

Taking with a predicate

```
Data.List.takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.

Taking with a predicate

```
Data.List.takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.

```
λ > takeWhile (< 10) [1..20]
[1,2,3,4,5,6,7,8,9]
λ > takeWhile odd ([1,3..10] ++ [1..10])
[1,3,5,7,9,1]
λ > takeWhile even [1..10]
[]
λ > takeWhile (> 0) (map (`mod` 5) [1..10])
[1,2,3,4]
```

Taking with a predicate

```
Data.List.takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.

```
takeWhile1 :: (a -> Bool) -> [a] -> [a]
takeWhile1 _ [] = []
takeWhile1 p (x : xs)
  | p x         = x : takeWhile1 p xs
  | otherwise   = []
```

Dropping with a predicate

```
Data.List.dropWhile :: (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`.

Dropping with a predicate

```
Data.List.dropWhile :: (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`.

```
λ> dropWhile (< 10) [1..20]
[10,11,12,13,14,15,16,17,18,19,20]
λ> dropWhile odd ([1,3..10] ++ [1..10])
[2,3,4,5,6,7,8,9,10]
λ> dropWhile even [1..10]
[1,2,3,4,5,6,7,8,9,10]
λ> dropWhile (> 0) (map (`mod` 5) [1..10])
[0,1,2,3,4,0]
λ> dropWhile (< 3) (takeWhile (< 6) [1..10])
[3,4,5]
```

Dropping with a predicate

```
Data.List.dropWhile :: (a -> Bool) -> [a] -> [a]
```

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`.

```
dropWhile1 :: (a -> Bool) -> [a] -> [a]
dropWhile1 _ [] = []
dropWhile1 p (x : xs)
  | p x         = dropWhile1 p xs
  | otherwise   = x : xs

dropWhile2 :: (a -> Bool) -> [a] -> [a]
dropWhile2 _ [] = []
dropWhile2 p xs@(x : xs')
  | p x        = dropWhile2 p xs'
  | otherwise = xs
```

Iterating

```
Data.List.iterate :: (a -> a) -> a -> [a]
```

`iterate` creates an infinite list where the first item is calculated by applying the function on the second argument, the second item by applying the function on the previous result, and so on.

```
λ > iterate (\x -> x+1) 1 -- == iterate (+1) 1
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
^C Interrupted.
λ > take 10 (iterate (\x -> x+1) 1)
[1,2,3,4,5,6,7,8,9,10]
λ > take 10 (iterate (+ 1) 1)
[1,2,3,4,5,6,7,8,9,10]
λ > takeWhile (< 10) (iterate (+ 1) 1)
[1,2,3,4,5,6,7,8,9]
```

Iterating

```haskell
Data.List.iterate :: (a -> a) -> a -> [a]
```

```haskell
iterate1 :: (a -> a) -> a -> [a]
iterate1 f x = let y = f x in x : iterate1 f y
-- iterate1 f x = x : iterate1 f (f x)
```

```haskell
iterate1 f x
  = x : iterate1 (f x)
  = x : f x : iterate1 (f (f x))
  = x : f x : f (f x) : iterate1 (f (f (f x)))
  = ...
```

Iterating

```
Data.List.iterate :: (a -> a) -> a -> [a]

iterate2 :: (a -> a) -> a -> [a]
iterate2 f x = x : [f y | y <- iterate2 f x]

iterate2 f x
  = x : [f y | y <- iterate2 f x]
  = x : f x : [f y | y <- iterate2 f (f x)]
  = x : f x : f (f x) : [f y | y <- iterate2 f (f (f x))]
  = ...
```

Zipping with functions

```
Data.List.zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

`zipWith` generalises `zip` by zipping with the function given as the first argument, instead of a tupling function.

```
λ > zipWith (+) [0..4] [10..14]
[10,12,14,16,18]
λ > zipWith (\x y -> (x,y)) [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
λ > zipWith (,) [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
λ > f x b = if b then x*10 else x
λ > zipWith f [1,2,3,4] [True,False,True,False]
[10,2,30,4]
```

Zipping with functions

```haskell
Data.List.zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith1 :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith1 _ []       _        = []
zipWith1 _ _        []       = []
zipWith1 f (x : xs) (y : ys) = f x y : zipWith1 f xs ys

zip2 :: [a] -> [b] -> [(a,b)]
zip2 = zipWith1 (,)
```

Zipping with functions – In practice

Determine whether a list is in non-decreasing order.

```haskell
nonDec1 :: Ord a => [a] -> Bool
nonDec1 []                = True
nonDec1 [_]               = True
nonDec1 (x1 : x2 : xs) = x1 <= x2 && nonDec1 (x2 : xs)

nonDec2 :: Ord a => [a] -> Bool
nonDec2 []                    = True
nonDec2 [_]                   = True
nonDec2 (x1 : xs@(x2 : _)) = x1 <= x2 && nonDec2 xs

nonDec3 :: Ord a => [a] -> Bool
nonDec3 xs = and $ zipWith (<=) xs (tail xs)
```

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad M' = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

```
m = [[1,2],
     [3,4],
     [5,6]]
```

```
m' = [[1,2,0,0,0,0,0],
      [3,4,0,0,0,0,0],
      [5,6,0,0,0,0,0]]
```

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```
λ > m = [[1,2],[3,4],[5,6]]
λ > addExtraColumns 0 m
[[1,2],[3,4],[5,6]]
λ > addExtraColumns 1 m
[[1,2,0],[3,4,0],[5,6,0]]
λ > addExtraColumns 5 m
[[1,2,0,0,0,0,0],[3,4,0,0,0,0,0],[5,6,0,0,0,0,0]]
```

Zipping with functions – In practice

You are constructing a numeric matrix (as a list of lists), but you want to add extra columns to pad on the right side.

```
addExtraColumns1 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns1 k xss = map (++ zs) xss
  where
    zs = replicate k 0

addExtraColumns2 :: Num a => Int -> [[a]] -> [[a]]
addExtraColumns2 k xss = zipWith (++) xss zss
  where
    zss = repeat (replicate k 0)
```

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

```
approxPi1 k = 4 * sum (take k xs)
  where
    ss = [(-1)^n | n <- [0..]]
    xs = zipWith (*) ss (map (1/) (iterate (+2) 1))

approxPi2 k = 4 * sum (take k xs)
  where
    ss = 1 : [(-1)*s | s <- ss]
    xs = zipWith (*) ss (map (1/) (iterate (+2) 1))
```

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

```
λ > pi
3.141592653589793
λ > let k = 10 in approxPi1 k
3.0418396189294032
λ > let k = 100 in approxPi1 k
3.1315929035585537
λ > let k = 10000 in approxPi1 k
3.1414926535900345
```

Zipping with functions – In practice

The Leibniz formula for π, named after Gottfried Leibniz, states that

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

```
λ > ks = iterate (*10) 1
λ > mapM_ print (take 8 [pi / approxPi1 k | k <- ks])
0.7853981633974483
1.0327936535639899
1.003193183258315
1.0003184111600008
1.0000318320017856
1.0000031831090173
1.0000003183099935
1.00000003183099
```

η-conversion

An eta conversion (also written η-conversion) is adding or dropping of abstraction over a function.

The following two values are equivalent under η-conversion:

```
\x -> someFunction x
```

and

```
someFunction
```

Converting from the first to the second would constitute an η-reduction, and moving from the second to the first would be an *eta*-expansion.

The term η-conversion can refer to the process in either direction.

η-conversion

```haskell
f :: T1 -> T2 -> T3
f t1 t2  = g t1 t2
```

$\Big\downarrow$ η-reduction

```haskell
f :: T1 -> T2 -> T3
f t1 = g t1
```

$\Big\downarrow$ η-reduction

```haskell
f :: T1 -> T2 -> T3
f = g
```

The composition operator

The high-order library operator `.` returns the composition of two function as a single function

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

`f . g`, which is read as `f` composed with `g`, is the function that takes an argument `x`, applies the function `g` to this argument, and applies the function `f` to the result.

The composition operator

Composition can be used to simplify nested function applications, by reducing parentheses ans avoiding the need to explicitly refer to the initial argument.

The composition operator

Composition can be used to simplify nested function applications, by reducing parentheses ans avoiding the need to explicitly refer to the initial argument.

```haskell
odd1 :: Integral a => a -> Bool
odd1 n = not (even n)

odd2 :: Integral a => a -> Bool
odd2 n = (not . even) n
-- i.e., odd2 = \x -> not (even n)

odd3 :: Integral a => a -> Bool
odd3 = not . even
```

The composition operator

Composition can be used to simplify nested function applications, by reducing parentheses ans avoiding the need to explicitly refer to the initial argument.

```
twice1 :: (a -> a) -> a -> a
twice1 f x = f (f x)

twice2 :: (a -> a) -> a -> a
twice2 f x = (f . f) x   -- i.e., twice2 = \x -> f (f x)

twice3 :: (a -> a) -> a -> a
twice3 f = f . f
```

The composition operator

Composition is associative

```
f . (g . h) = f . g . h
```

for any functions `f`, `g` and `h` of the appropriate types.

```
sumSqrEven1 :: Integral a => [a] -> a
sumSqrEven1 xs = sum (map (^2) (filter even xs))

sumSqrEven2 :: Integral a => [a] -> a
sumSqrEven2 xs = (sum . map (^2) . filter even) xs

sumSqrEven3 :: Integral a => [a] -> a
sumSqrEven3 = sum . map (^2) . filter even
```

The composition operator

Composition also has an identity, given by the identity function:

```
id :: a -> a
id = \x -> x
```

For any function `f`:

```
id . f  = f
f  . id = f
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ > f = head . id
λ > f [1,2,3,4]
1
```

```
f = head . id
  = \x -> head (id x)
  = \x -> head x
  = head
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> g = id . head
λ> g [1,2,3,4]
1
```

```
g = id . head
  = \x -> id (head x)
  = \x -> head x
  = head
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> :type take
take :: Int -> [a] -> [a]
λ> f = take . id
λ> f 3 [1..10]
[1,2,3]
```

```
f = take . id
  = \x -> take (id x)
  = \x -> take x -- :: Int -> ([a] -> [a])
  = take
```

The composition operator

Composition also has an identity, given by the identity function:

```
λ> :type take
take :: Int -> [a] -> [a]
λ> g = id . take
λ> g 3 [1..10]
[1,2,3]
```

```
g = id . take
  = \x -> id (take x)
  = \x -> take x -- :: Int -> ([a] -> [a])
  = take
```

Composition

Be sure to understand the following:

```
λ > ((+1) . (+1)) 1
3
λ > ((+) . (+1)) 100 10
111
λ > ((++) . (++ " ")) "hello" "world!"
"hello world!"
```

Composition

Be sure to understand the following:

```
λ> trim = let f = reverse . dropWhile (== ' ') in f . f
λ> :type trim
trim :: [Char] -> [Char]
λ> trim "   ab   cd ef  g hi    "
"ab   cd ef  g hi"
```

Composition (difficult !)

Explain :

```
λ > f = (\x y z -> x) . (\a b -> a*b)
λ > f 1 2 3 4
4
λ > g = (\x y z -> y) . (\a b -> a*b)
λ > g 1 2 3 4
<interactive>: error:
  Could not deduce (Num t0)
  ...
```

The function application operator

The `$` is an operator for function application.

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

All this does is apply a function. So, `f $ x` exactly equivalent to `f x`:

```
λ > head $ [1,2,3,4]
1

λ > tail $ [1,2,3,4]
[2,3,4]

λ > map (+ 1) $ [1,2,3,4]
[2,3,4,5]
```

The function application operator

This seems utterly pointless, until you look beyond the type.

```
λ > :info ($)
($) :: (a -> b) -> a -> b  -- Defined in ‘GHC.Base’
infixr 0 $
```

The function application operator

This seems utterly pointless, until you look beyond the type.

```
λ> :info ($)
($) :: (a -> b) -> a -> b  -- Defined in ‘GHC.Base’
infixr 0 $
```

This little note holds the key to understanding the ubiquity of `($)`: `infixr 0`.

- `infixr` tells us it's an infix operator with right associativity.
- `0` tells us it has the lowest precedence possible.

In contrast, normal function application (via white space)

- is left associative and
- has the highest precedence possible (`10`).

The function application operator

Compare

```
λ > take 10 "Haskell " ++ "rocks!"
"Haskell rocks!"
λ > (take 10  "Haskell ") ++ "rocks!"
"Haskell rocks!"
```

with

```
λ > take 10 $ "Haskell " ++ "rocks!"
"Haskell ro"
λ > take 10 ("Haskell " ++ "rocks!")
"Haskell ro"
```

The function application operator

One pattern where you see the dollar sign used sometimes is between a chain of composed functions and an argument being passed to (the first of) those.

```
λ > sum . drop 3 . take 5 [1..10]
error.
λ > sum . drop 3 . take 5 $ [1..10]
9
λ > (sum . drop 3 . take 5) [1..10]
9
λ > sum . drop 3 $ take 5 [1..10]
9
```

The function application operator

Function application.

```
λ > map (\f -> f 2) [(* i) | i <- [1,2,3,4,5]]
[2,4,6,8,10]
λ > map 2 [(* i) | i <- [1,2,3,4,5]]
error.
λ > map ($ 2) [(* i) | i <- [1,2,3,4,5]]
[2,4,6,8,10]
λ > map ($ 2) [f i | f <- [(*),(+)], i <- [1,2,3,4,5]]
[2,4,6,8,10,3,4,5,6,7]
```

And a curiosity

$ is just an identity function for . . . functions.

```
($) :: (a -> b) -> a -> b
    :: (a -> b) -> (a -> b)

id  :: a -> a
    :: (a -> b) -> (a -> b) -- for a ~ a -> b
```

And a curiosity

$ is just an identity function for ... functions.

```
($) :: (a -> b) -> a -> b
    :: (a -> b) -> (a -> b)

id  :: a -> a
    :: (a -> b) -> (a -> b) -- for a ~ a -> b
```

```
λ> (sum . drop 3 . take 5) [1..10]
9

λ> sum . drop 3 $ take 5 [1..10]
9

λ> (sum . drop 3) `id` take 5 [1..10]
9

λ> id (sum . drop 3) (take 5 [1..10])
9
```

Origami programming

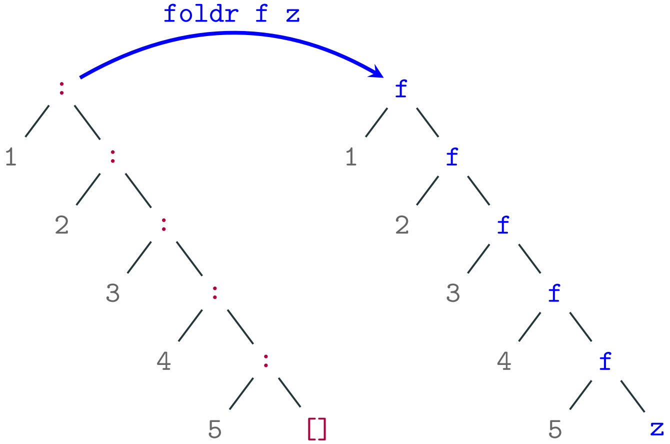
Folding

- In functional programming, `fold` is a family of higher order functions that process a data structure in some order and build a return value.
- This is as opposed to the family of `unfold` functions which take a starting value and apply it to a function to generate a data structure.
- A `fold` deals with two things:
  1. a combining function, and
  2. a data structure.

  The `fold` then proceeds to combine elements of the data structure using the function in some systematic way.

Folding right

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



Folding right

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

  foldr (+) 0 [1,2,3,4]
= (+) 1 (foldr (+) 0 [2,3,4]
= (+) 1 ((+) 2 (foldr (+) 0 [3,4])
= (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [4])
= (+) 1 ((+) 2 ((+) 3 ((+) 4 (foldr (+) 0 [])
= (+) 1 ((+) 2 ((+) 3 ((+) 4 0) -- stop recursion
= (+) 1 ((+) 2 ((+) 3 4)
= (+) 1 ((+) 2 7)
= (+) 1 9
= 10
```

Folding right

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

  foldr (:) [] [1,2,3,4]
= (:) 1 (foldr (:) [] [2,3,4]
= (:) 1 ((:) 2 (foldr (:) [] [3,4])
= (:) 1 ((:) 2 ((:) 3 (foldr (:) [] [4])
= (:) 1 ((:) 2 ((:) 3 ((:) 4 (foldr (:) [] [])
= (:) 1 ((:) 2 ((:) 3 ((:) 4 []) -- stop recursion
= (:) 1 ((:) 2 ((:) 3 4:[])
= (:) 1 ((:) 2 3 : 4 : [])
= (:) 1 (2 : 3 : 4 : [])
= 1 : 2 : 3 : 4 : []                -- [1,2,3,4]
```

Folding right

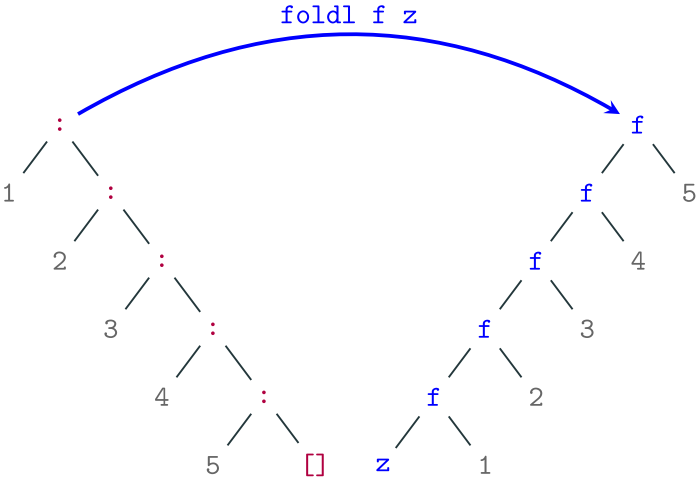
```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []       = z
  foldr f z (x : xs) = f x (foldr f z xs)

  let f x acc = [x] : acc in foldr f [] [1,2,3,4]
= f 1 (foldr f [] [2,3,4]
= f 1 (f 2 (foldr f [] [3,4]))
= f 1 (f 2 (f 3 (foldr f [] [4])))
= f 1 (f 2 (f 3 (f 4 (foldr f [] []))))
= f 1 (f 2 (f 3 (f 4 [])))     -- stop recursion
= f 1 (f 2 (f 3 [4] : []))
= f 1 (f 2 [3] : [4] : [])
= f 1 ([2] : [3] : [4] : [])
= [1] : [2] : [3] : [4] : []  -- [[1],[2],[3],[4]]
```

Folding right

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

  let f x acc = acc ++ [x] in foldr f [] [1,2,3,4]
= f 1 (foldr f [] [2,3,4]
= f 1 (f 2 (foldr f [] [3,4]))
= f 1 (f 2 (f 3 (foldr f [] [4])))
= f 1 (f 2 (f 3 (f 4 (foldr f [] []))))
= f 1 (f 2 (f 3 (f 4 [])))        -- stop recursion
= f 1 (f 2 (f 3 ([] ++ [4])))
= f 1 (f 2 ([] ++ [4] ++ [3]))
= f 1 ([] ++ [4] ++ [3] ++ [2])
= [] ++ [4] ++ [3] ++ [2] ++ [1]  -- [4,3,2,1]
```

Folding left

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



Folding left

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

  foldl (+) 0 [1,2,3,4]
= foldl (+) ((+) 0 1) [2,3,4]
= foldl (+) ((+) ((+) 0 1) 2) [3,4]
= foldl (+) ((+) ((+) ((+) 0 1) 2) 3) [4]
= foldl (+) ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) []
= ((+) ((+) ((+) ((+) 0 1) 2) 3) 4) -- stop recursion
= ((+) ((+) ((+) 1 2) 3) 4)
= ((+) ((+) 3 3) 4)
= ((+) 6 4)
= 10
```

Folding left

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

  let fC acc x = x : acc in foldl fC [] [1,2,3,4]
= foldl fC (fC [] 1) [2,3,4]
= foldl fC (fC (fC [] 1) 2) [3,4]
= foldl fC (fC (fC (fC [] 1) 2) 3) [4]
= foldl fC (fC (fC (fC (fC [] 1) 2) 3) 4) []
= (fC (fC (fC (fC [] 1) 2) 3) 4) -- stop recursion
= (fC (fC (fC 1 : [] 2) 3) 4)
= (fC (fC 2 : 1 : [] 3) 4)
= (fC 3 : 2 : 1 : [] 4)
= 4 : 3 : 2 : 1 : []               -- [4,3,2,1]
```

Folding left

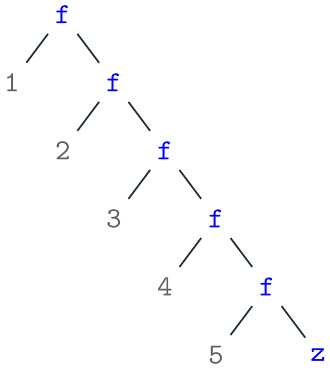
```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f z []       = z
  foldl f z (x : xs) = foldl f (f z x) xs

  let fC acc x = [x]: acc in foldl fC [] [1,2,3,4]
= foldl fC (fC [] 1) [2,3,4]
= foldl fC (fC (fC [] 1) 2) [3,4]
= foldl fC (fC (fC (fC [] 1) 2) 3) [4]
= foldl fC (fC (fC (fC (fC [] 1) 2) 3) 4) []
= (fC (fC (fC (fC [] 1) 2) 3) 4) -- stop recursion
= (fC (fC (fC [1] : [] 2) 3) 4)
= (fC (fC [2] : [1] : [] 3) 4)
= (fC [3] : [2] : [1] : [] 4)
= [4] : [3] : [2] : [1] : []             -- [[4],[3],[2],[
```

Folding left

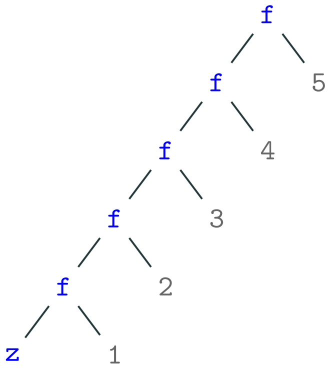
```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f z []       = z
  foldl f z (x : xs) = foldl f (f z x) xs

  let fC acc x = acc ++ [x] in foldl fC [] [1,2,3,4]
= foldl fC (fC [] 1) [2,3,4]
= foldl fC (fC (fC [] 1) 2) [3,4]
= foldl fC (fC (fC (fC [] 1) 2) 3) [4]
= foldl fC (fC (fC (fC (fC [] 1) 2) 3) 4) []
= (fC (fC (fC (fC [] 1) 2) 3) 4)    -- stop recursion
= (fC (fC (fC [] ++ [1] 2) 3) 4)
= (fC (fC [] ++ [1] ++ [2] 3) 4)
= (fC [] ++ [1] ++ [2] ++ [3] 4)
= [] ++ [1] ++ [2] ++ [3] ++ [4]    -- [1,2,3,4]
```

Folding

`foldr f z`



`foldl f z`



Removing duplicates

Exercice

The function `remDups` removes adjacent duplicates from a list.

```
λ > remDups [1,2,2,3,3,3,1,1]
[1,2,3,1]
```

Define `remDups` using `foldr`. Give another definition using `foldl`.

Exercice

The `fold` on integers (let's call it `foldI`) can be defined as follows:

```
foldI :: (a -> a) -> a -> Int -> a
foldI _ q 0 = q
foldI f q i = f . foldI f q $ pred i
```

Define the functions `add`, `mult` and `exp` in terms of `foldI`. Of course, you're not allowed to use `(+)` and `(*)`.

Curried functions & friends

Currying

Currying is the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments, one by one, that the original function would receive in the rest of that tuple.

```haskell
f :: a -> b -> c    -- i.e. f :: a -> (b -> c)
```

is the curried form of

```haskell
g :: (a, b) -> c
```

In Haskell, all functions are considered curried: That is, all functions in Haskell take just one argument.

Currying / uncurrying

```
f :: a -> b -> c    -- i.e. f :: a -> (b -> c)

g :: (a, b) -> c
```

You can convert these two types in either directions with the Prelude functions `curry` and `uncurry`:

```
curry :: ((a, b) -> c) -> a -> b -> c

uncurry :: (a -> b -> c) -> (a, b) -> c
```

We have:

```
f = curry g

g = uncurry f
```

Currying / uncurrying

```haskell
f :: a -> b -> c    -- i.e. f :: a -> (b -> c)

g :: (a, b) -> c
```

You can convert these two types in either directions with the Prelude functions `curry` and `uncurry`:

```haskell
curry :: ((a, b) -> c) -> a -> b -> c

uncurry :: (a -> b -> c) -> (a, b) -> c
```

Both forms are equally expressive. It holds:

```haskell
f x y = g (x,y)
```

Uncurrying

```
λ> :type (+)
(+) :: Num a => a -> a -> a

λ> add1 = (+) 1
λ> :type add1
add1 :: Num a => a -> a

λ> add1 2
3

λ> :type uncurry (+)
uncurry (+) :: Num a => (a, a) -> a

λ> uncurry (+) (1,2)
3

λ> uncurry (+) 1
error.
```

Uncurrying

```
λ> zipWith (+) [0..4] [10..14]
[10,12,14,16,18]

λ> :type (+)
(+) :: Num a => a -> a -> a

λ> :type map
map :: (a -> b) -> [a] -> [b]

λ> zip [0..4] [10..14]
[(0,10),(1,11),(2,12),(3,13),(4,14)]

λ> map (\(x,y) -> x+y) $ zip [0..4] [10..14]
[10,12,14,16,18]

λ> map (uncurry (+)) $ zip [0..4] [10..14]
[10,12,14,16,18]
```

Currying

```
λ > :type fst
fst :: (a, b) -> a

λ > fst (1,2)
1

λ > fst 1
error.

λ > type curry fst
curry fst :: a -> b -> a

λ > f = curry fst 1
λ > :type f
f :: Num a => b -> a

λ > f 2
1
```

Currying

```
λ > add p = fst p + snd p
λ > :type add
add :: Num a => (a, a) -> a

λ > add (1,2)
3

λ > add1 = curry add 1
λ > :type add1
add1 :: Num a => a -> a

λ > add1 2
3
```

Flipping

```haskell
flip :: (a -> b -> c) -> b -> a -> c
```

evaluates the function flipping the order of arguments

```
λ> (/) 1 2
0.5

λ> foldr (++) [] ["A","B","C","D"]
"ABCD"

λ> foldr (flip (++)) [] ["A","B","C","D"]
"DCBA"

λ> foldr (:) [] ['a'..'d']
"abcd"

λ> foldr (flip (:)) [] ['a'..'d']
error.
```

Flipping

```haskell
flip :: (a -> b -> c) -> b -> a -> c
```

evaluates the function flipping the order of arguments

```
λ> (/) 1 2
0.5
λ> foldr (++) [] ["A","B","C","D"]
"ABCD"
λ> foldr (flip (++)) [] ["A","B","C","D"]
"DCBA"
λ> foldr (:) [] ['a'..'d']
"abcd"
λ> foldr (flip (:)) [] ['a'..'d']
error.
```

Flipping

```
flip :: (a -> b -> c) -> b -> a -> c
```

evaluates the function flipping the order of arguments

```
flip1 :: (a -> b -> c) -> b -> a -> c
flip1 f x y = f y x
```

```
flip2 :: (a -> b -> c) -> b -> a -> c
flip2 f = \x -> \y -> f y x
```

Flipping – Use cases

```
λ > foldr (:) [] [1..4]
[1,2,3,4]
λ > foldl (flip (:)) [] [1..4]
[4,3,2,1]

λ > foldl (-) 100 [1..4]        -- (((100-1)-2)-3)-4
90
λ > foldr (-) 100 [1..4]        -- 1-(2-(3-(4-100)))
98
λ > foldl (flip (-)) 100 [1..4] -- 4-(3-(2-(1-100)))
102
λ > foldr (flip (-)) 100 [1..4] -- (((100-4)-3)-2)-1
90
```

Constant

```
const :: a -> b -> a
```

`const x y` always evaluates to `x`, ignoring its second argument.

```
λ > const 1 2
1
λ > const (2/3) (1/0)
0.6666666666666666
λ > const take drop 5 [1..10]
[1,2,3,4,5]
λ > foldr (\_ acc -> 1 + acc) 0 [1..10]
10
λ > foldr (const (1+)) 0 [1..10]
10
```

Constant

```
const :: a -> b -> a
```

`const x y` always evaluates to `x`, ignoring its second argument.

```
const1 :: a -> b -> a
const1 x _ = x
```

```
const2 :: a -> b -> a
const2 = \x -> \_ -> x
```

Fun with flipping and constant

```haskell
curry id = \x y -> id (x, y)  -- def. curry
         = \x y -> (x, y)     -- def. id
         = \x y -> (,) x y    -- desugar
         = \x -> (,) x        -- eta reduction
         = (,)                -- eta reduction
```

```
λ > curry id 1 2
(1,2)
λ > (,) 1 2
(1,2)
```

Fun with flipping and constant

```haskell
uncurry const = \(x, y) -> const x y   -- def. uncurry
              = \(x, y) -> x           -- def. const
              = fst                    -- def. fst
```

```
λ > uncurry const (1, 2)
1
λ > fst (1, 2) -- from Data.Tuple (in Prelude)
1
```

Fun with flipping and constant

```
uncurry (flip const)
      = \(x, y) -> (flip const) x y   -- def. uncurry
      = \(x, y) -> const y x          -- def. flip
      = \(x, y) -> y                  -- def. const
      = snd                           -- def. snd
```

```
λ > uncurry (flip const) (1, 2)
2
λ > snd (1, 2) -- from Data.Tuple (in Prelude)
2
```

Fun with flipping and constant

```
uncurry (flip (,))
        = \(x, y) -> (flip (,)) x y   -- def. uncurry
        = \(x, y) -> (,) y x          -- def. flip
        = \(x, y) -> (y, x)           -- desugar
```

```
λ > uncurry (flip (,)) (1, 2)
(2,1)
λ > import Data.Tuple
λ > :type swap
swap :: (a, b) -> (b, a)
λ > swap (1, 2)
(2,1)
```

Processing lists – revisit

High-order functions

Origami programming

Curried functions & friends

Processing lists – revisit

Rotations – revisit

Produce all rotations of a list.

```
λ > rotate []
[[]]
λ > rotate [1]
[[1]]
λ > rotate [1,2]
[[2,1],[1,2]]
λ > rotate [1,2,3]
[[3,1,2],[2,3,1],[1,2,3]]
λ > rotate [1,2,3,4]
[[4,1,2,3],[3,4,1,2],[2,3,4,1],[1,2,3,4]]
```

Rotations – revisit

Produce all rotations of a list.

```
shift1 :: [a] -> [a]
shift1 []       = []
shift1 (x : xs) = xs ++ [x]

rotate1 :: [a] -> [[a]]
rotate1 [] = [[]]
rotate1 xs@( _ : xs') = foldl f [xs] xs'
  where
    f acc@(xs'' : _) _ = shift1 xs'' : acc
```

Rotations – revisit

Produce all rotations of a list.

```
import Data.List

rotate2 :: [a] -> [[a]]
rotate2 xs = init $ zipWith (++) (tails xs) (inits xs)

-- tails [1,2,3,4] = [[1,2,3,4], [2,3,4], [3,4], [4],
-- inits [1,2,3,4] = [[],        [1],     [1,2], [1,2,3],
```

Rotations – revisit

Produce all rotations of a list.

```
λ > let xs = [1,2,3,4] in rotate1 xs == rotate2 xs
False
λ > rotate1 [1,2,3,4]
[[4,1,2,3],[3,4,1,2],[2,3,4,1],[1,2,3,4]]
λ > rotate2 [1,2,3,4]
[[1,2,3,4],[2,3,4,1],[3,4,1,2],[4,1,2,3]]
```

Finding (revisit)

`Data.List.elem` is the list membership predicate, usually written in infix form, e.g., `x `elem` xs`. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
-- foldr
elem1 :: a -> [a] -> Bool
elem1 x' xs = foldr f False xs
  where
    f x b = x == x' || b
```

Finding (revisit)

`Data.List.elem` is the list membership predicate, usually written in infix form, e.g., `` x `elem` xs ``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
-- eta-reduction
elem2 :: a -> [a] -> Bool
elem2 x' = foldr f False
  where
    f x b = x == x' || b
```

Finding (revisit)

`Data.List.elem` is the list membership predicate, usually written in infix form, e.g., `` x `elem` xs ``. For the result to be `False`, the list must be finite; `True`, however, results from an element equal to `x` found at a finite index of a finite or infinite list.

```
-- using a lambda
elem3 :: a -> [a] -> Bool
elem3 x' = foldr (\x b -> x == x' || b) False
```

Finding

Exercice

Define `elem` using

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

but not

`length :: [a] -> Int`.

Finding

Exercice

Notice that

```
λ > 10 `elem1` [1..]
True
```

```
λ > 10 `elem1` [11..]
^C Interrupted.
```

Explain the two results (keep in mind the implementation of `elem1`).

Filtering (revisit)

`Data.List.filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate.

```
filter3 :: (a -> Bool) -> [a] -> [a]
filter3 p xs = foldr f [] xs
  where
   f x acc
     | p x       = x : acc
     | otherwise = acc
```

Repeating (revisit)

`Data.List.repeat` takes an element and returns an infinite list that just has that element.

```
repeat4 :: a -> [a]
repeat4 x = foldr (\_ acc -> x : acc) [] [1..]
```

Repeating (revisit)

`Data.Foldable.maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.

Repeating (revisit)

`Data.Foldable.maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.

```
maximum4 :: Ord a => [a] -> a
maximum4 []       = error "empty list"
maximum4 (x : xs) = foldr f x xs
  where
    f x m = if x > m then x else m

maximum5 :: Ord a => [a] -> a
maximum5 []       = error "empty list"
maximum5 (x : xs) = foldr max x xs
```

Repeating (revisit)

`Data.Foldable.maximum` returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.

```
maximum6 :: Ord a => [a] -> a
maximum6 [] = error "empty list"
maximum6 xs = foldl1 max xs

maximum7 :: Ord a => [a] -> a
maximum7 [] = error "empty list"
maximum7 xs = foldr1 max xs
```

Remove duplicate

```
Data.List.nub :: Eq a => [a] -> [a]
```

The `nub` function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.

Remove duplicate

```
Data.List.nub :: Eq a => [a] -> [a]
```

The `nub` function removes duplicate elements from a list. In particular, it keeps only the first occurrence of each element.

```
nub1 :: Eq a => [a] -> [a]
nub1 [] = []
nub1 (x : xs) = x : nub1 (filter (\y -> x /= y) xs)

nub2 :: Eq a => [a] -> [a]
nub2 [] = []
nub2 (x : xs) = x : nub2 xs'
  where
    xs' = filter (/= x) xs
```

Remove duplicate

```
Data.List.nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded == function.

Remove duplicate

```haskell
Data.List.nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded == function.

```haskell
nubBy1 :: Eq a => (a -> a -> Bool) -> [a] -> [a]
nubBy1 eq []        = []
nubBy1 eq (x : xs) = x : xs'
  where
    xs' = nubBy1 eq (filter (\y -> not (eq x y)) xs)

nub3 :: Eq a => [a] -> [a]
nub3 = nubBy (==)
```

Remove duplicate

```haskell
Data.List.nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

The nubBy function behaves just like nub, except it uses a user-supplied equality predicate instead of the overloaded == function.

```haskell
elemBy :: (a -> a -> Bool) -> a -> [a] -> Bool
elemBy _  _ []         =  False
elemBy eq y (x : xs) =  x `eq` y || elemBy eq y xs
```

```haskell
nubBy2 :: (a -> a -> Bool) -> [a] -> [a]
nubBy2 eq xs = go xs []
  where
    go []     _         = []
    go (y:ys) xs
      | elemBy eq y xs = go ys xs
      | otherwise      = y : go ys (y : xs)
```