Higher Order Functions

A function is called <u>higher-order</u> if it takes a function as an argument or returns a function as a result.

```
twice     :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

- <u>Common programming idioms</u>, such as applying a function twice, can be encapsulated as general purpose higher-order functions;

- <u>Special purpose languages</u> can be defined using higher-order functions, such as for list processing, interaction, or parsing;

- <u>Algebraic properties</u> of higher-order functions can be used to reason about programs.

The higher-order library function called map applies a function to every element of a list.

```
map :: (a → b) → [a] → [b]
```

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

Alternatively, using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs]
```

```
  map (+1) [1,3,5,7]
=
 [(+1) x | x <- [1,3,5,7]]
=
 [(+1) 1] ++ [(+1) 3] ++ [(+1) 5] ++ [(+1) 7]
=
 [2] ++ [4] ++ [6] ++ [8]
=
 [2,4,6,8]
```

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

```
  map (+1) [1,3,5,7]
=
 map (+1) (1 : (3 : (5 : (7 : [ ]))))
=
 (+1) 1 : map (+1) (3 : (5 : (7 : [ ])))
=
 (+1) 1 : ((+1) 3 : map (+1) (5 : (7 : [ ])))
=
 (+1) 1 : ((+1) 3 : ((+1) 5 : map (+1) (7 : [ ])))
=
 (+1) 1 : ((+1) 3 : ((+1) 5 : ((+1) 7 : map (+1) [ ])))
=
 (+1) 1 : ((+1) 3 : ((+1) 5 : ((+1) 7 :[ ])))
=
 2 :(4 : (6 : (8 : [ ])))
=
 [2,4,6,8]
```

The higher-order library function <u>filter</u> selects every element from a list that satisfies a predicate.

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p []     = []
filter p (x:xs)
   | p x        = x : filter p xs
   | otherwise  = filter p xs
```

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to a value v, and any non-empty list to a function ⊕ applied to its head and f of its tail.

For example:

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

$v = 0$

$\oplus = +$

```
product []     = 1
product (x:xs) = x * product xs
```

$v = 1$

$\oplus = *$

```
and []     = True
and (x:xs) = x && and xs
```

$v = \text{True}$

$\oplus = \&\&$

The higher-order library function <u>foldr</u> (“fold right”) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

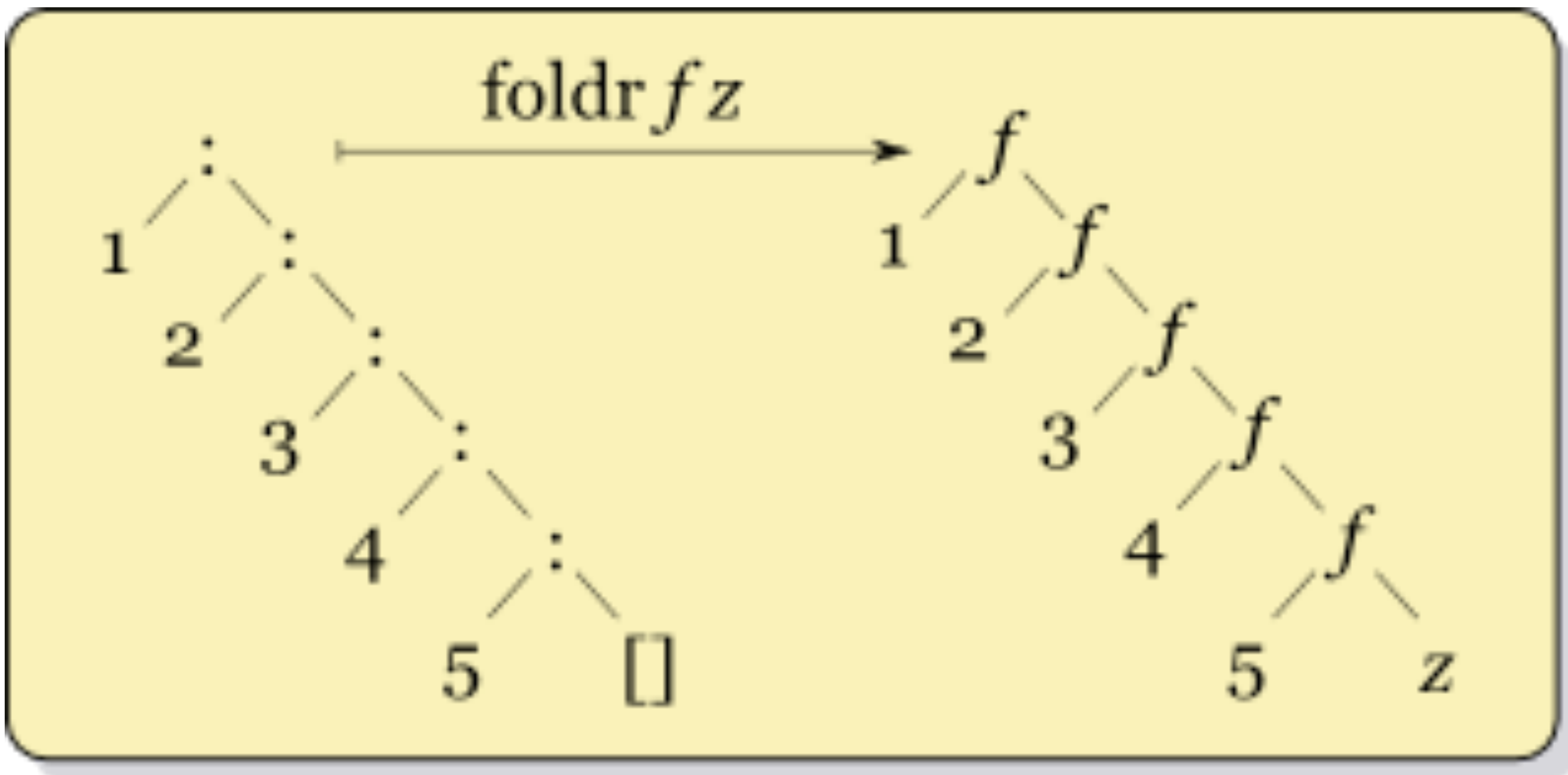
For example:

```
sum     = foldr (+) 0

product = foldr (*) 1

and     = foldr (&&) True
```

Foldr defined using recursion:

```
foldr (⊕) v []     = v
foldr (⊕) v (x:xs) =
   x ⊕ foldr (⊕) v xs
```

Think of `foldr` <u>non-recursively</u>, as simultaneously replacing each cons in a list by an infix function, and `[]` by a value.



foldr f z
:
1
2
3
4
5
[]
f
z

For example:

```
   sum [1,2,3]
=
=  foldr (+) 0
   foldr (+) 0 (1:(2:(3:[])))
=
   1+(2+(3+0))
=
   6
```

Replace each cons by + and [] by 0.

sum using foldr: how it works

```
  sum [1,2]
=
  foldr 0 [1,2]
=
  foldr (+) 0 (1 : (2 : []))
=
  1 + (foldr (+) 0 (2 : []))
=
  1 + (2 + (foldr (+) 0 []))
=
  1 + (2 + 0)
=
  3
```

what is the type of foldr?

Type of foldr

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

Combining

Sum of squares of positive integers in a list

list comprehension

```
f :: [Int] -> Int
f xs = sum [x*x | x<-xs, x>0]
```

recursion

```
f :: [Int] -> Int
f [] = 0
f (x:xs) | x>0 = (x*x) + f xs
         | otherwise = f xs
```

Higher order functions

```
f :: [Int] -> Int
f xs  =  foldr (+) 0 (map sqr (filter pos xs))
   where
   sqr x = x*x
   pos x = x > 0
```

find the largest number under 100,000 that's divisible by 3829

```
largestDivisible :: (Integral a) => a
largestDivisible = head (filter p [100000,99999..])

    where p x = x `mod` 3829 == 0
```

What do these evaluate to?

```
foldr(:) [ ] xs

foldr(:) ys xs
```

foldr(:) [ ] xs
Replaces ":" by ":", and [] by [] --no change!
The result is equal to xs.

foldr(:) ys(a:(b:(c:[ ])))
= a:(b:(c:ys))
The result is xs++ys!

Folding an operation into a non-empty list

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

foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Operator: function composition

"do something, do something else"

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

Note: `f . g x` not the same as `f (g x)`

not all pairs can be composed
output type of `g` must be input type of `f`

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

input of `f` and output of `g` are of the same type: `b`

`f . g` has input type `a`, same as `g`,

and output type `c`, same as `f`

composition is associative

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

forward composition: >.>

order of composition is significant
`(f . g)` means first apply `g` then apply `f`

Can define an operator that
applies functions in the opposite order

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

the application operator: $

application of function `f` to argument `e`

```
f e
```

can also be explicit

```
f $ e
```

can use as an alternative to parentheses

```
flipV (flipH (rotate horse))
```

becomes

```
flipV $ flipH $ rotate horse
```

can use as a function

```
zipWith ($) [sum, product] [[1,2], [3,4]]
```

Application and composition

suppose `f` has type `Integer -> Bool`

`f . x` means `f` composed with `x`
so `x` must have type `s -> Integer` for some type `s`

`f x` means `f` applied to `x` so `x` must be of type `Integer`

`f $ x` also means `f` applied to `x` so `x` has type `Integer`

Exercises

let id be the polymorphic identity function

`id x = x`

explain the behaviour of the following expressions:

`(id . f)`   `(f . id)`   `id f`

define a function `composeList` which composes a list of functions into a single function type

give the type

explain why the function has this type

what is the effect on an empty list?

what is the type of the application operator, `$`?

what is the result of
`zipWith ($) [sum, product] [[1,2], [3,4]]`?

explain the behaviour of the expressions
`(id $ f)`   `(f $ id)`   `id ($)`

lambda abstractions

`addOne x = x + 1`

could save the definition overhead and just use

`\x -> x + 1`

e.g. `map (\x -> x + 1) [3,4,5]`

```
mapFuns fs x = map (\f -> f x) fs
```

```
comp2 :: (a -> b) -> (b -> b -> c) -> (a -> a -> c)
comp2 f g = (\x y -> g (f x) (f y))
```

e.g.

`comp2 sq add 3 4`

Exercises

define a function `total`

`total :: (Integer -> Integer) -> (Integer -> Integer)`

so that total f is the function which at value n gives

`f $ 0 + f $ 1 + . . . f $ n`

given a function `f` of type `a -> b -> c`
write down a lambda abstraction that describes
a function of type `b -> a -> c` which behaves
like `f` but takes its arguments in the other order

define

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

which reverses the order in which a function takes its arguments

partial application

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

`multiply 2` given a number `y` returns `2 * y`

any function taking two or more arguments
can be partially applied to one or more arguments

```
doubleAll :: [Int] -> [Int]
doubleAll = map (multiply 2)
```

partial applications can specialise general functions

Exercise

standard operators in Haskell are called operator sections when partially applied

Find operator sections `sec1` and `sec2` so that

`map sec1 . filter sec2`

has the same effect as

`filter (>0) . map (+1)`

currying and uncurrying

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

neater, permits partial application

```
multiplyUC :: (Int, Int) -> Int
multiplyUC (x, y) = x * y

curry :: ((a, b) -> c) -> (a -> b -> c)
curry g x y = g (x, y)
```

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

curry and **uncurry** are inverses of each other

inverse of

`unzip :: [(a, b)] -> ([a], [b])`

is not `zip`

`zip :: [a] -> [b] -> [(a, b)]`

but

`uncurry zip :: ([a], [b]) -> [(a, b)]`

so

`prop_zip xs = uncurry zip $ unzip xs == xs`

Exercises

what is the effect of `uncurry ($)`? what is its type?
what about `uncurry (:)`, `uncurry (.)`?

define functions

```
curry3 :: ((a,b,c) -> d) -> (a -> b -> c -> d)
uncurry3 :: (a -> b -> c -> d) -> ((a,b,c) -> d)
```