higher order functions

- take functions as arguments
- return functions as results
- or both

\[
\begin{align*}
\text{doubleL} :: [\text{Int}] \to [\text{Int}] \\
\text{double} \ x = [2 \times x | x \leftarrow \text{x}] \\
\text{doubleL} \ [\ ] &= [\ ] \\
\text{doubleL} \ [x : \text{xs}] &= [2 \times x : \text{doubleL} \ \text{xs}]
\end{align*}
\]

\[
\begin{align*}
\text{trebleL} :: [\text{Int}] \to [\text{Int}] \\
\text{treble} \ x = [3 \times x | x \leftarrow \text{x}] \\
\text{trebleL} \ [\ ] &= [\ ] \\
\text{trebleL} \ [x : \text{xs}] &= [3 \times x : \text{trebleL} \ \text{xs}]
\end{align*}
\]
map \ f \ xs = [ f \ x \mid x \leftarrow xs ]

map \ f [ ] = [ ]
map \ f [ x : xs ] = [ f \ x : map \ f \ xs ]

doubleL \ xs = map \ twice \ xs
where twice \ x = 2 \cdot x

map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]

values for which the function can be applied

the type of values after applying the function

map \ • \ apply some function to every element of a list
thus yielding another list

doubleL \ xs = map \ twice \ xs
where twice \ x = 2 \cdot x

7 3 9 2

14 6 18 4
properties as functions

getDigits :: [Char] -> [Char]
getDigits s = [c | c <- s, isDigit c]

isDigit :: Char -> Bool
isDigit 'a' = True
isDigit 'b' = True
isDigit 'c' = True
isDigit 'd' = True

x has a property f if (f x) = True

property f over type t  t -> Bool

isEven :: Int -> Bool
isEven n = (mod n 2 == 0)

isSorted :: [Int] -> Bool
isSorted xs = (xs == qSort xs)
filtering

\[
\text{filter } f = \begin{cases} 
\text{[]} & \text{if } f \text{ is false} \\
\text{filter } f (x : xs) = x \text{ filter } f \text{ xs} \\
\text{otherwise} & \text{filter } f \text{ xs}
\end{cases}
\]

\[
\text{filter isSorted } [[2,3,4,5], [], [7,3,6]] \rightarrow [[2,3,4,5], []]
\]

folding

\[
\text{fold1} \xi \begin{cases} 
\text{[e, e, e, ... , e]} & \text{=} \\
\text{[e, \xi ((e, \xi (...) , e) \ldots )]} & \text{=} \\
\text{[e, \xi (fold1 \xi \text{[e, e, e, ... , e])]} & \text{=}
\end{cases}
\]

\[
\text{fold1} (\ast) \text{[e, e, e]} = e_1 (\ast) (\text{fold1} (\ast) \text{[e, e]}) = e_1 (\ast) e_2 (\ast) e_3
\]
higher order functions

- take functions as arguments
- return functions as results
- or both

sequence of processes:

for every \( \text{process}_i : \text{P} \), \( \text{OUT-process}_{i+1} \rightarrow \text{IN-process}_{i+1} \)
function composition

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

\[
\text{type of } f \quad \text{type of } g \quad \text{type of } f \cdot g
\]

```
Prelude> and [(5 == 5), (3 > 5)]
False
Prelude> (not . and) [(5 == 5), (3 > 5)]
True
Prelude> cos (sin pi)
1.0
Prelude> (cos . sin) pi
1.0
Prelude>
```

```
f . g
\[\text{type of } f \rightarrow \text{type of } g \rightarrow \text{type of } f \cdot g\]
```

```
Main> succ 110
111
Main> succ (succ 110)
112
Main> (twice succ) 110
112
Main>
```

```
twice -- function on function

\[
\text{twice} :: (a -> a) \rightarrow (a -> a)
\]
```

```
Main> twice 110
110
Main> twice (110 + 1)
112
Main> twice (twice succ) 110
112
Main>
```

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

```
Main> succ 110
111
Main> succ (succ 110)
112
Main> (twice succ) 110
112
Main>
```

```
Main> twice 110
110
Main> twice (110 + 1)
112
Main> twice (twice succ) 110
112
Main>
```

```
Main> succ 110
111
Main> succ (succ 110)
112
Main> (twice succ) 110
112
Main>
```

```
f . g
\[\text{type of } f \rightarrow \text{type of } g \rightarrow \text{type of } f \cdot g\]
```

```
Main> succ 110
111
Main> succ (succ 110)
112
Main> (twice succ) 110
112
Main>
```

```
Main> twice 110
110
Main> twice (110 + 1)
112
Main> twice (twice succ) 110
112
Main>
```

```
Main> succ 110
111
Main> succ (succ 110)
112
Main> (twice succ) 110
112
Main>
```

```
Main> twice 110
110
Main> twice (110 + 1)
112
Main> twice (twice succ) 110
112
Main>
```

```
Main> succ 110
111
Main> succ (succ 110)
112
Main> (twice succ) 110
112
Main>
```

```
Main> twice 110
110
Main> twice (110 + 1)
112
Main> twice (twice succ) 110
112
Main>
```
... thrice, four-times, ..., n-times

```haskell
ntimes :: Int -> (a -> a) -> (a -> a)
ntimes n f
| n > 0 = f . ntimes (n-1) f
| otherwise = id

Main> twice succ 110
112
Main> ntimes 2 succ 110
112
Main> ntimes 1 succ 110
111
Main> ntimes 0 succ 110
110
Main> ntimes 5 succ 110
115
Main>
```

The `ntimes` function takes an `Int` and a function, and applies the function `n` times. It uses pattern matching to determine whether it should apply the function once (using `f . ntimes (n-1) f`) or repeatedly (using `id`).

---

**Type classes**

```haskell
isinBList :: Bool -> [Bool] -> Bool
isinBList x [] = False
isinBList x (y : ys) = (x == y) || isinBList x ys
```

The `isinBList` function checks whether an element `x` is in a list of type `[Bool]`. It uses pattern matching to handle the case where the list is empty (using `False`) and the case where it is not empty (using `(x == y) || isinBList x ys`).
if the list was of type [Int]

\[
\text{isinList} :: \text{Int} \rightarrow \text{[Int]} \rightarrow \text{Bool}
\]

\[
\text{isinList} \ x \ \text{[]} = \text{False}
\]

\[
\text{isinList} \ x \ (y : \text{ys}) = (x == \text{Int} \ y) \mid| \text{isinList} \ x \ \text{ys}
\]

generically

\[
\text{isinList} :: \text{a} \rightarrow \text{[a]} \rightarrow \text{Bool}
\]

and restrict a to only those types that have equality defined over them

overloading

there are two kinds of functions that work over more than one class

- **polymorphic** - single definition which works over all its types

\[
\text{length} :: \text{[a]} \rightarrow \text{Int}
\]

\[
\text{length} \ \text{[]} = 0
\]

\[
\text{length} \ (x : \text{xs}) = 1 + \text{length} \ \text{xs}
\]

- **overloaded** - (e.g. equality, \ .show) that can be used for many types but have different definitions for different types
**type classes - collection of types**

**equality type class (Eq)**

```
class Eq where
  (==) :: a -> a -> Bool
```

- `Int`
- `Float`
- `Bool`
- `Char`
- `(Int, Bool)`
- `([Char])`

**instance of Eq**

```
same3 :: Int -> Int -> Int -> Bool
same3 m n p = (m == n) && (n == p)
```

```
isInList :: Eq a => a -> [a] -> Bool
isInList x [ ] = False
isInList x (y:ys) = (x == y) || isInList x ys
```

**in the context of**

```
same3 :: Eq a => a -> a -> a -> Bool
same3 m n p = (m == n) && (n == p)
```

**thus restricting a to types such as:**

- `Char`
- `Int`
- `(Int, Bool)`
- `Float`
- etc.
**definition of Eq**

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

**signature**

**derived class Ord**

```haskell
class Eq a => Ord where
  (<), (<=), (>, >=) :: a -> a -> Bool
  max, min :: a -> a -> a
  compare :: Ordering
  compare x y |
                x == y     = EQ
                x <= y     = LT
                otherwise = GT
```

class **Ord** inherits the operations of **Eq**

**class Enum**

```haskell
class Ord a => Enum a where
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

```haskell
[n...]
[n..m]
[n..m]
[n..m']
```
class Bounded a where
    minBound, maxBound :: a

type ShowS = String -> String

class Show a where
    showPrec :: Int -> a -> ShowS
    show     :: a -> String
    showList :: [a] -> ShowS

most types belong to Show

### numeric types in Haskel

- **Int**: fixed precision integers
- **Integer**: all integers represented accurately
- **Float**: floating point numbers
- **Double**: Float in double precision
- **Rational**:

the basic class to which all numeric types belong is **Num**
class (Eq a Show a) a => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
  fromInt :: Int -> a
  x - y = x + negate y
  fromInt = fromIntegral

Integer types belong to the class `Integral` whose signature include:
  
  quot, rem :: a -> a -> a
  div, mod :: a -> a -> a

Algebraic types

- Base types
  - Int
  - Float
  - Bool
  - Char

- Composite types
  - Tuples
  - Lists
  - Functions

  - Type of months
  - Alternative
  - Trees

  January, ..., December
  E.g., elements can be either strings or numbers
enumerated types

data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

defines 7 new constants called constructors

dayval :: Day -> Int

dayval Sun = 0
dayval Mon = 1
...........

dayval Sat = 6

product types

data People = Student Id Grade

Student "BS02143" 86
Student "MS02187" 67

showStdnt :: People -> String
showStdnt (Student x y) = show x ++ " " ++ show y
**product versus tuple types**

The previous example could be defined as

```
type Student = (Id, Grade)
```

**Product types**

- Each object of the type has an explicit label of the purpose of the object (meaning).
- Each object must be explicitly constructed by using the predefined constructors.
- Type error will be identified in the compiler/interpreter diagnostics.

**Tuple types**

- Shorter definitions, more familiar notation.
- Many Prelude polymorphic functions exist (and thus can be 'inherited'). Especially for pairs.

**Alternative types**

```
data GeomS = Circle Float | Square Float | Rect Float Float
```

```
area :: GeomS -> Float
area (Circle r) = pi * r ^ 2
area (Square a) = a ^ 2
area (Rect a b) = a * b
```
deriving instances of classes

built-in classes
- **Eq** equality, inequality
- **Ord** ordering of elements
- **Enum** allows the type to be enumerated \([n \ldots m]\) style
- **Show** elements of the type to be turned into text form
- **Read** values can be read from strings

data `Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat`
deriving (**Eq, Ord, Enum, Show**)

which let us do
comparisons
represent via

Mon == Mon, Mon /= Tue
[ Mon ... Fri ]

binary trees

data `Tree a`
= `Nil` |
Node `a` (Tree `a`) (Tree `a`)
deriving (**Eq, Ord, Show, Read**)

```
depth :: Tree `a` -> Int
depth `Nil` = 0
depth (Node `n` `t1` `t2`) = 1 + max (depth `t1`) (depth `t2`)
```

```
traverse :: Tree `a` -> [`a`]
traverse `Nil` = []
traverse (Node `x` `t1` `t2`) = traverse `t1` ++ [`x`] ++ traverse `t2`
```

... (Node 17 (Node 14 `Nil` `Nil`) (Node 20 `Nil` `Nil`)) ...

June 2009
FP for DB
More HUGS 20
### binary trees

**left, right ::** Tree a -> Tree a

left (Node x ys zs) = ys  
right (Node x ys zs) = zs

**isinT ::** Eq a => a -> Tree a -> Bool

isinT p Nil = false  
isinT p (Node x ys zs) = (p == x) || isinT p ys || isinT p zs

**mirrorT ::** Tree a -> Tree a

mirror T Nil = Nil  
mirrorT (Node x ys zs) = (Node x zs ys)

---

### evaluation

**square (4 + 2)**  
= square 6  
= 6 * 6  
= 36

**applicative-order evaluation**

`reduce func expr`

* reduce `expr` as far as possible  
* expand definition of `func` and continue reducing

**simple but may not terminate**

`fst (42, inf)` where `inf = 1 / inf`
**evaluation**

<table>
<thead>
<tr>
<th>Normal-order evaluation</th>
<th>Square ((4 + 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(= (4 + 2) \times (4 + 2))</td>
<td>(= 6 \times (4 + 2))</td>
</tr>
<tr>
<td>(= 6 \times 6)</td>
<td>(= 36)</td>
</tr>
</tbody>
</table>

| Avoids non-termination | \(\text{fst}(42, \inf) = 42\) |

<table>
<thead>
<tr>
<th>Lazy evaluation</th>
<th>(\text{square}(4 + 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(= \text{square} \quad \text{where} \quad = (4 + 2))</td>
<td>(= \ast \quad \text{where} \quad = (4 + 2))</td>
</tr>
<tr>
<td>(= \ast \quad \text{where} \quad = 6)</td>
<td>(= 36)</td>
</tr>
</tbody>
</table>

**Lazy evaluation**

<table>
<thead>
<tr>
<th>As normal-order evaluation...</th>
<th>(\text{reduce } \text{func expr})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{expand definition of } \text{func, substituting } \text{expr as necessary})</td>
<td>(\text{reduce result})</td>
</tr>
</tbody>
</table>

**Lazy evaluation**

- Evaluate argument unless it is needed (normal order)
- Evaluate argument more than once (applicative order)

**Lazy evaluation**

Wait with all computation for as long as possible
an example

\[
\text{sumSq } n = \text{sum (map } (^2) [1 .. n])
\]

= \text{sumSq } 100
= \text{sum (map } (^2) [1 .. 100])
= \text{sum (map } (^2) [2 .. 100])
= 1^2 + \text{sum (map } (^2) [2 .. 100])
= 1 + \text{sum (map } (^2) [2 .. 100])
= ...
= 1 + (4 + \text{sum (map } (^2) [3 .. 100])
= ...

in this evaluation never the whole list [1 .. 100] is in existence

infinite lists

\[
\text{ones = 1 : ones}
\]

would generate [1, 1, 1, 1, 1, 1\text{[Interrupted]}

if they were to be evaluated fully an infinite amount of time would have been needed - but we can compute with a part of rather than the whole object

\[
\text{head ones} \rightarrow 1
\]
\[
\text{take 4 (map } (^2) [1 .. ]} \rightarrow [1, 4, 9, 16]
\]
some infinite lists

- \([n ..] = [n, n+1, n+2, \ldots]\)
- \([n, m ..] = [n, n + (m - n), n + 2 \times (m - n), \ldots]\)
- \(\text{repeat } n = n : \text{repeat } n\)
- \(\text{fibs } = 0 : \text{zipWith (+) fibs (tail fibs)}\)
- \(\text{iterate }: (a -> a) -> a -> [a]\)
  \(\text{iterate } f x = x : \text{iterate } f (f x)\)
- \(\text{primes } = [n | n \in [2 ..], \text{divisors } n == [1, n]
  \text{where divisors } n = [d | d \in [1 .. n], (\text{mod } d n) == 0]\)
  \(\text{getNprimes } n = \text{takeWhile } (\leq n) \text{ primes}\)

more infinite lists

- \(\text{repeat }: a -> [a]\)
  \(\text{repeat } n = n : \text{repeat } n\)
- \(\text{twos } = [\text{Int}]\)
  \(\text{twos } = \text{repeat } 2\)
- \(\text{iterate }: (a -> a) \rightarrow a -> [a]\)
  \(\text{iterate } f x = x : \text{iterate } f (f x)\)
  \(\text{Main> take 20 twos}\)
  \([2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]\)
  \(\text{Main> take 10 (iterate (+2)} 0)\)
  \([0,2,4,6,8,10,12,14,16,18]\)
  \(\text{Main> take 10 (iterate (+2)) 1}\)
  \([1,3,5,7,9,11,13,15,17,19]\)
  \(\text{Main> take 10 (iterate (+3)} 1)\)
  \([1,4,7,10,13,16,19,22,25,28]\)
  \(\text{Main> take 10 (iterate (+3)) 5}\)
  \([5,8,11,14,17,20,23,26,29,32]\)
  \(\text{Main>}\)
modules

Abcd.hs
module Abcd where
    types
    functions
    calculateA = .....  

Bxyz.hs
module Bxyz where
    import Abcd
    types
    functions
    computeB = .....  

Cpqr.hs
module Cpqr where
    import Bxyz
        .....  

visible definitions of Abcd are visible now in Bxyz

visible definitions of Abcd not visible in Cpqr

modules - EXPORT CONTROL

• stating explicitly which definitions are exported

module Bxyz (computeSum, Abcd (...), calculateA) where ...

  constructors of the type are exported with the type itself

  names of defined objects

• all visible definitions of the specified modules are exported

module Bxyz (module Bxyz, module Abcd) where ...

visible definitions of Abcd are visible now in Bxyz
modules - IMPORT CONTROL

• stating explicitly which definitions are to be imported

\[ \text{import Abcd (specification of what is to be imported)} \]

• stating explicitly which definitions are to be hidden

\[ \text{import Abcd hiding (specification of what is to be concealed)} \]

• stating explicitly the need for qualification of names from Abcd

\[ \text{import qualified Abcd means that objects defined in Abcd must be used as Abcd.object-name} \]

ADTs as modules

module Queue (Queue, emptyQ, isEmptyQ, addQ, delQ) where

emptyQ :: Queue a
isEmptyQ :: Queue a -> Bool
addQ :: a -> Queue a -> Queue a
delQ :: Queue a -> Queue a

newtype Queue a = Q [a]  as data but will not permit the use of the Prelude list functions

emptyQ = Q []
isEmptyQ (Q []) = True
isEmptyQ _ = False
addQ x (Q xs) = Q (xs ++ [x])
delQ (Q _) (xs) = Q xs
delQ (Q []) = error "cannot remove from empty Q"
module Queue (Queue, emptyQ, isEmptyQ, addQ, delQ) where

emptyQ :: Queue a
isEmptyQ :: Queue a -> Bool
addQ :: a -> Queue a -> Queue a
delQ :: Queue a -> Queue a

newtype Queue a = Q ([a], [a])

emptyQ = Q ([], [])
isEmptyQ (Q ([], [])) = True
isEmptyQ _ = False

addQ x (Q ([], [])) = Q ([x], [])
addQ y (Q (xs, ys)) = Q (xs, y:ys)
delQ (Q ([], ys)) = error "cannot remove from empty Q"
delQ (Q (x:xs, ys)) = Q (xs, ys)
module Queue (Queue, emptyQ, isEmptyQ, addQ, delQ) where

emptyQ :: Queue a
isEmptyQ :: Queue a -> Bool
addQ :: a -> Queue a -> Queue a
delQ :: Queue a -> Queue a

newtype Queue a = Q ([a], [a])

emptyQ = Q ([], [])
isEmptyQ (Q ([], [])) = True
isEmtpyQ _ = False

addQ x (Q ([], [])) = Q ([x], [])
addQ y (Q (xs, ys)) = Q (xs, y:ys)

delQ (Q ([], [])) = error "cannot remove from empty Q"
delQ (Q ([], ys)) = Q (tail (reverse ys), [])
delQ (Q (x : xs, ys)) = Q (xs, ys)

queue via two lists

first part
second part

same signature
different implementation
set as unordered list with duplicates

```
module Set (Set, emptyS, isEmptyS, inS, addS, delS) where

    emptyS :: Set a
    isEmptyS :: Set a -> Bool
    inS :: (Eq a) => a -> Set a -> Bool
    addS :: (Eq a) => a -> Set a -> Set a
    delS :: (Eq a) => a -> Set a -> Set a

    newtype Set a = S [a]
    emptyS = S []
    isEmptyS (S []) = True
    isEmptyS _     = False
    inS x (S xs)   = elem x xs
    addS x (S a)   = S (x : a)
    delS x (S xs)  = S (filter (/= x) xs)
```

```