MORE
Haskell through HUGS

higher order functions

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

```
doubleL :: [ Int ] -> [ Int ]
double  xs = [ 2 * x | x <- xs ]
```

```
doubleL :: [ Int ] -> [ Int ]
doubleL [ ]        = [ ]
doubleL [ x : xs ] = [ 2 * x : doubleL xs ]
```

```
trebleL :: [ Int ] -> [ Int ]
trebleL  xs = [ 3 * x | x <- xs ]
```

```
trebleL :: [ Int ] -> [ Int ]
trebleL [ ]        = [ ]
trebleL [ x : xs ] = [ 3 * x : trebleL xs ]
```

⋮ ⋮

sin x
ord x
……

```
map f  xs = [ f x | x <- xs ]
```

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

```
doubleL  xs = map  twice xs
               where twice x = 2 * x
```


function IN OUT

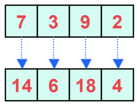
```
map    :: (a → b) → [a] → [b]
```

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 * x
```



```
-- lambda notation for local function defn
doubleLambda  xs = map  (\x -> 2 * x) xs
```

```
Main> doubleLambda [2, 7, 3, 12]
[4,14,6,24]
Main> doubleLambda [ ]
[ ]
Main> doubleLambda [1]
[2]
Main>
```

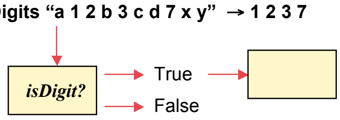
```
cnvrtC :: [ Char ] -> [ Int ]
cnvrtC xs = map ord xs
```

```
Main> cnvrtC "Stefan"
[83,116,101,102,97,110]
Main> cnvrtC ['a', 'b', 'c', 'd']
[97,98,99,100]
```


properties as functions

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

getDigits “a 1 2 b 3 c d 7 x y” → 1 2 3 7



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

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

```
filter  f  xs = [ x  | x <- xs, f  x ]
```

filter isSorted [[2,3,4,5], [ ], [7,3,6]] → [[2,3,4,5], [ ]]

folding

foldr1 ξ [e_1, e_2, e_3, ..., e_n] =
=[e_1 ξ (e_2 ξ (... ξ e_n ...)
= [e_1 ξ (foldr1 ξ [e_1, e_2, e_3, ..., e_n])

foldr1 (+) [e_1, e_2, e_3]
= e_1 (+) (foldr1 (+) [e_2, e_3])
= e_1 (+) e_2 (+) e_3

binary function over type a

result

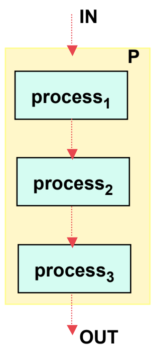
```
foldr1 :: (a -> a ->a) -> [a] -> a
foldr1 f [x]  = x
foldr1 f (x : xs) = f x (foldr1  f  xs)
-- at least one element in the list x
```

```
Main> foldr1 (+) [1,2,3,4]
10
Main> foldr1 (+) [1]
1
Main> foldr1 (+) [ ]
Program error: {foldr1 (instNum_v30 Num_+) [ ]}
Main> foldr1 (||) [True, False, False]
True
Main> foldr1 (++) ["Dark", "side", "  ", "of" ]
"Darkside  of"
Main> foldr1 (*) [1 .. 7]
5040
```


higher order functions

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

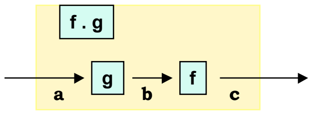
```
unx > date
Tue Apr 07 05:37:22 BST 2009
unx > f | grep p00 | cut -c48-58
 Mon 10:18
 Mon 16:23
 Sat   14:32
 Tue  14:38
 Mon 10:30
unx >
```



sequence of processes:

for every process$_i \in$ P , OUT-process$_i$ → IN-process$_{i+1}$

function composition



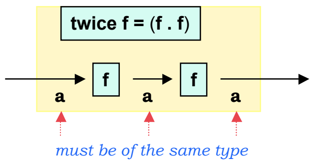
(f . g) x = f (g x)

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

*type of **f***   *type of **g***   *type of **f . 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>
```


twice -- function on function



(twice) :: (a -> a) -> (a -> a)

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

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

... thrice, four-times, ..., n-times

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

↑ identity

```
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>
```


type classes

is this *an element of this list (of type, say, Bool) ?*

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

if the list was of type [Int]

```
isinIList  :: Int -> [Int] -> Bool
isinIList x [ ]          = False
isinIList  x ( y : ys ] =  (x == Int y) || isinIList  x ys
```

generically

```
isinList  :: a -> [a] -> Bool
```

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

overloading

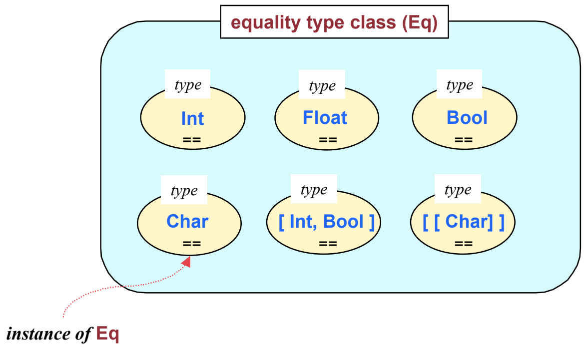
there are two kinds of functions that work over more than one class

- polymorphic - single definition which works over all its types

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

length [ ]          =  0
length (x : xs)   =  1 + length 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




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



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

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.

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

a -
- Bool
- Char
- Int
- (Int, Int)

definition of Eq

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

signature

derived class Ord

```
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

```
class Ord a => Enum a where
     toEnum           ::  Int  -> a
     fromEnum         ::  a  -> Int
     enumFrom         ::  a -> [a]           [n ..]
     enumFromThen     ::  a -> a -> [a]      [n, m .. ]
     enumFromTo       ::  a -> a -> [a]      [n .. m]
     enumFromThenTo   ::  a -> a -> a -> [a] [n, n' .. m]
```

```
class Bounded a where
    minBound, maxBound  ::  a
```

types
Int, Char, Bool, Ordering

```
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	composite types
Int	tuples
Float	lists
Bool	function
Char	

- type of months — January, ..., December
- alternative — e.g. elements can be either strings or numbers
- trees

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

type name

constructor name

```
data People = Student  Id  Grade
```

```
type Id        =   String
type  Grade  =   Int
```

```
Student  "BS02143"  86
Student  "MS02187"  67
```

```
showStdnt   :: People -> String
showStdnt    (Student   x   y) = show  x  ++  " "  ++  show  y
```

product versus tuple types

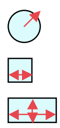
the previous example could be defined as

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

product types	tuple types
each object of the type has an explicit label of the purpose of the object (meaning)	shorter definitions, more familiar notation
each object must be explicitly constructed by using the predefined constructors	many Prelude polymorphic functions exist (and thus can be 'inherited'). especially for pairs
type error will be identified in the compiler/interpreter diagnostics	

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 classess

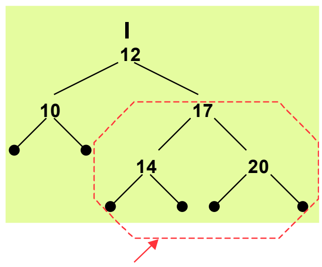
Eq	equality, inequality
Ord	ordering of elements
Enum	allows the type to be enumerated [n .. m] style
Show	elements of the type to be turned into text form
Read	values can be read from strings

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

which let us do
comparisons Mon == Mon, Mon /= Tue
represent via [Mon ... Fri]

binary trees

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

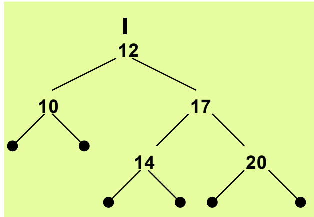


... (Node 17 (Node 14 Nil Nil) (Node 20 Nil Nil)) ...

```
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
```

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

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

normal-order evaluation

reduce func expr

- expand definition of func, substituting expr as necessary
- reduce result

avoids non termination	fst (42, inf) = 42
may involve repeating work as in	(4 + 2) * (4 + 2)

lazy evaluation

```
square (4 + 2)
  = square  x  where x = (4 + 2)
  = x * x where  x = (4 + 2)
  = x * x where  x  = 6
  = 36
```

as normal-order evaluation ...

reduce func expr

- expand definition of func, substituting expr as necessary
- reduce result

but instead of copying arguments, make pointers and share them

does not
- 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

```
sumSq  n  = sum (map (^2)  [1 .. n])

 = sumSq 100
 = sum (map (^2)  [1 .. 100])
 = sum (map (^2) (1: [2 .. 100]))
 = sum (1^2 : map (^2)  [2 .. 100])
 = 1^2 + sum (map (^2)  [2 .. 100])
 = 1 + sum (map (^2)  [2 .. 100])
 = ...
 = 1 + (4 + sum (map (^2)  [3 .. 100])
 = ...
```

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

infinite lists

```
ones = 1 : ones
would generate [1, 1, 1, 1, 1, 1^C{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

```
head ones   →   1
take 4 (map (^2) [1 ..])  →   [1, 4, 9, 16]
```

some infinite lists

- [n ..] = [n, n+1, n+2, ...]
- [n, m ..] = [n, n + (m - n), n + 2 * (m - n), ...]
- repeat n = n : repeat n
- fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
- iterate :: (a -> a) -> a -> [a]
  iterate f x = x : iterate f (f x)
- primes = [n | n <- [2 ..], divisors n == [1, n]
  where divisors n = [d | d <- [1 .. n], (mod d n) == 0]

  getNprimes n = takeWhile (<= n) primes

more infinite lists

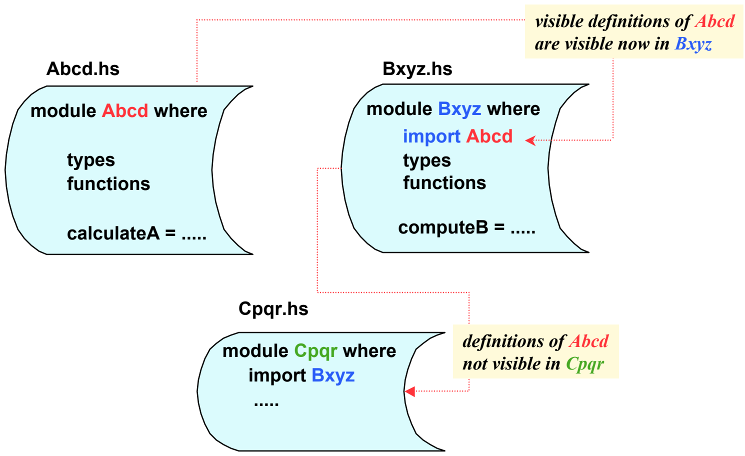
```
repeat :: a -> [a]
repeat n = n : repeat n

twos :: [Int]
twos = repeat 2
```

```
Main> take 20 twos
[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2]
Main>
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f ( f x)
```

```
Main> take 10 (iterate (+2) 0)
[0,2,4,6,8,10,12,14,16,18]
Main> take 10 (iterate (+2) 1)
[1,3,5,7,9,11,13,15,17,19]
Main> take 10 (iterate (+3) 1)
[1,4,7,10,13,16,19,22,25,28]
Main> take 10 (iterate (+3) 5)
[5,8,11,14,17,20,23,26,29,32]
Main>
```

modules



modules - EXPORT CONTROL

• stating explicitly which definitions are exported

constructors of the type are exported with the type itself

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

names of defined objects

• all visible definitions of the specified modules are exported

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

modules - IMPORT CONTROL

- stating explicitly which definitions are to be imported

  import Abcd (*specificaltion of what is to be imported***)**

- stating explicitly which definitions are to be hidden

  import Abcd hiding (*specificaltion what is to be concealed***)**

- stating explicitly the need for qualification of names from Abcd

  import qualified Abcd — *means that objects defined in* **Abcd** *must be used as* **Abcd.**object-name

ADTs as modules

signature:

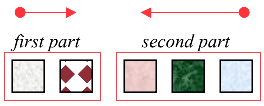
```
module Queue (Queue, emptyQ, isEmptyQ, addQ, delQ) where
 emptyQ    ::  Queue a
 isEmptyQ  ::  Queue  a  ->  Bool
 addQ      ::  a -> Queue a -> Queue a
 delQ      ::  Queue a -> Queue a
```

implementation:

```
 newtype Queue a  = Q  [a]

 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"
```

as **data** *but will not permit the use of the Prelude list functions*

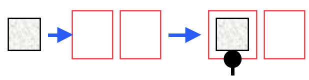
queue via two lists



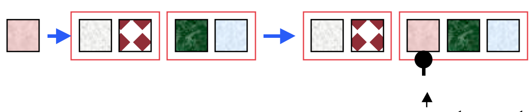

```
module Queue (Queue, emptyQ, isEmptyQ, addQ, delQ) where
 emptyQ     ::   Queue a
 isEmptyQ  ::  Queue  a  ->  Bool
 addQ        ::  a -> Queue a -> Queue a
 delQ         ::  Queue a -> Queue a
```

same signature

different implementation

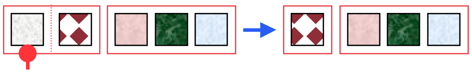


addQ x (Q ([ ], [ ])) = Q ([x], [ ])

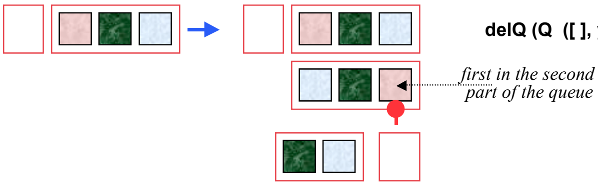


addQ y (Q (xs, ys)) = Q (xs, y:ys)

most recent addition

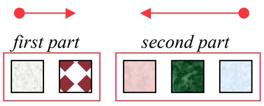


delQ (Q (x : xs, ys)) = Q (xs, ys)



delQ (Q ([ ], ys)) = Q (tail (reverse ys), [ ])

queue via two lists




```
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
```

same signature

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

different implementation

set as unordered list with duplicates

```
module Set (Set, emptyS, isEmptyQ, 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
 delQ      ::  (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)
```

```
elem  x  [ ]  = False
elem  x  (y : ys)
           | x == y       = True
           | otherwise = elem  x  ys
```