algebraic types

base types	composite types
Int	tuples
Float	lists
Bool	function
Char	

algebraic types
- 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

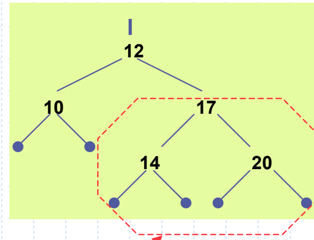
```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
        deriving (Eq, Ord, Enum, Show)
```

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)
```



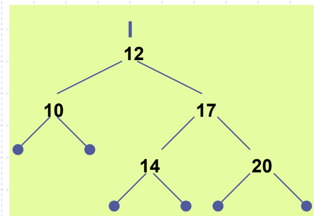
... (Node 17 (Node 14 Nil Nil) (Node 20 Nil Nil)) ...

```
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

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

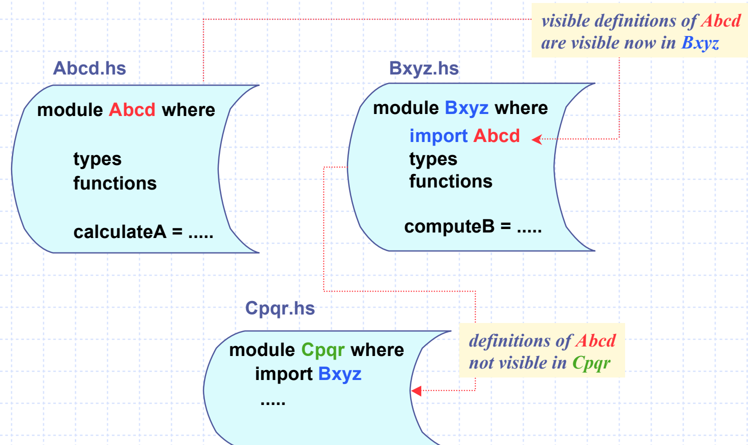
```
repeat :: a -> [a]
repeat n  =  n : repeat n

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

```
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
```

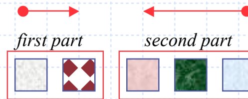
signature

```
 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*

implementation

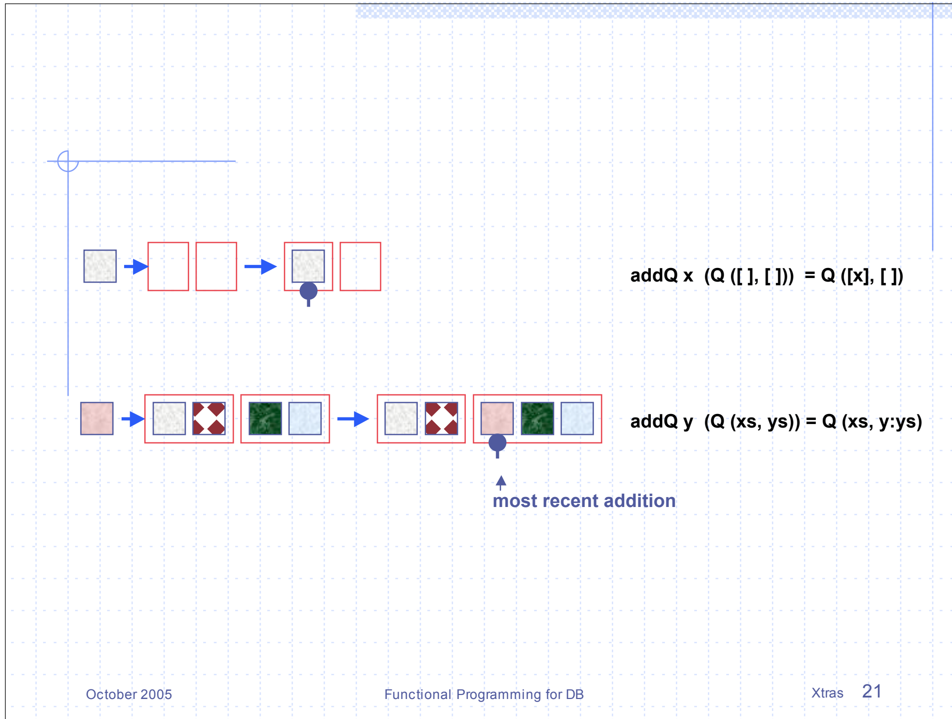
queue via two lists

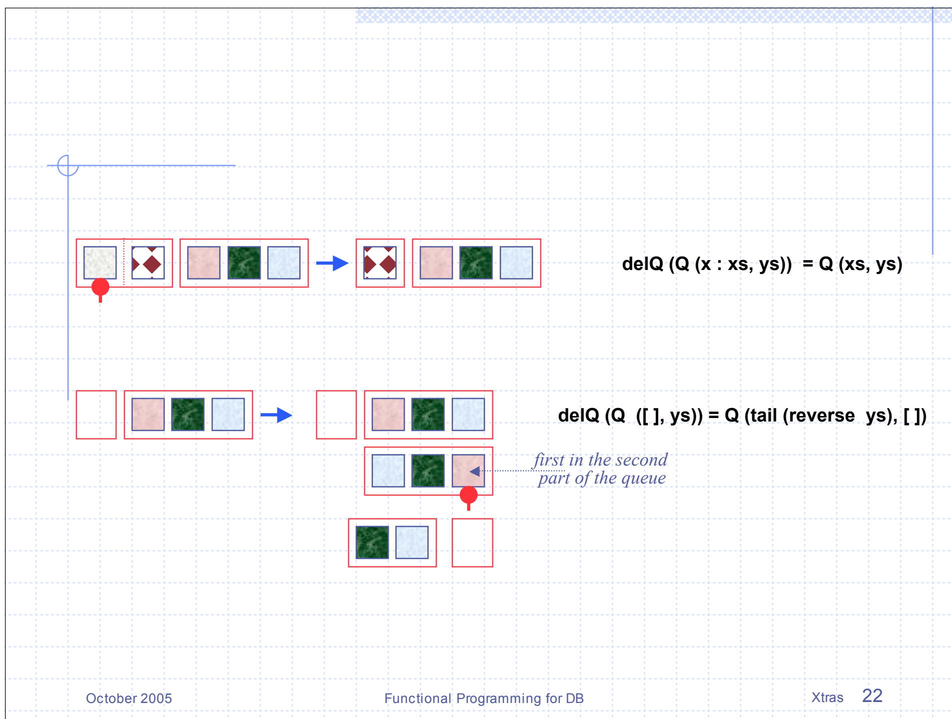



```
module Queue (Queue, emptyQ, isEmptyQ, addQ, delQ) where
 emptyQ    ::  Queue a
 isEmptyQ  ::  Queue a -> Bool
 addQ      ::  a -> Queue a -> Queue a
 delQ      ::  Queue a -> Queue a
```

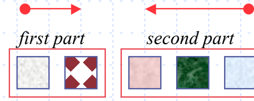
same signature

different implementation


addQ x (Q ([ ], [ ])) = Q ([x], [ ])
addQ y (Q (xs, ys)) = Q (xs, y:ys)
most recent addition
October 2005
Functional Programming for DB
Xtras 21


delQ (Q (x : xs, ys)) = Q (xs, ys)
delQ (Q ([ ], ys)) = Q (tail (reverse ys), [ ])
first in the second
part of the queue
October 2005
Functional Programming for DB
Xtras 22

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
```