algebraic types

base types
- Int
- Float
- Bool
- Char

algebraic types
- type of months
- alternative
- trees

composite types
- tuples
- lists
- function

Jan, ..., Dec
- elements can be either strings or numbers

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 Tue = 2
dayval Wed = 3
dayval Thu = 4
dayval Fri = 5
dayval Sat = 6
```haskell
data People = Student Id Grade

type Id = String
type Grade = Int

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

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

---

**product versus tuple types**

The previous example could be defined as:

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

<table>
<thead>
<tr>
<th>Product types</th>
<th>Tuple types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each object of the type has an explicit label of the purpose of the object (meaning)</td>
<td>Shorter definitions, more familiar notation</td>
</tr>
<tr>
<td>Each object must be explicitly constructed by using the predefined constructors</td>
<td>Many Prelude polymorphic functions exist (and thus can be ‘inherited’), especially for pairs</td>
</tr>
<tr>
<td>Type error will be identified in the compiler/interpreter diagnostics</td>
<td></td>
</tr>
</tbody>
</table>

---

October 2005

Functional Programming for DB

Xtras
alternative types

```haskell
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: \([n .. m]\) style
- **Show**: elements of the type to be turned into text form
- **Read**: values can be read from strings

```haskell
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

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

binary trees

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

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

```haskell
left, right :: Tree a -> Tree a
left (Node x ys zs) = ys
right (Node x ys zs) = zs

```haskell
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

```haskell
mirrorT :: Tree a -> Tree a
mirror T Nil = Nil
mirrorT (Node x ys zs) = (Node z s y)
```
evaluate $\text{square}(4 + 2)$

- $\text{square}(6)$
- $6 \times 6$
- $36$

**applicative-order evaluation**

- reduce $\text{func expr}$
  - reduce $\text{expr}$ as far as possible
  - expand definition of $\text{func}$ and continue reducing

Simple but may not terminate

```
\text{fst}(42,\text{inf})\quad\text{where inf=1+inf}
```
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 .. 100])
= sum (1*2 : map (^2) [2 .. 100])
= 1^2 + sum (map (^2) [2 .. 100])
= ... 
= 1 + sum (map (^2) [2 .. 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...][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

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

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> 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]

modules

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

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

module Cpqr where
  import Bxyz
  ..... 

visible definitions of Abcd are visible now in Bxyz

definitions of Abcd not visible in Cpqr
modules - EXPORT CONTROL

• stating explicitly which definitions are exported

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

• 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 (specification of what is to be imported)

• stating explicitly which definitions are to be hidden

import Abcd hiding (specification 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

newtype Queue a = Q [a]

emptyQ = Q []
isEmptyQ (Q []) = True
isEmptyQ _ = False
addQ x (Q xs) = Q (xs ++ [x])
delQ (Q (_ : xs)) = Q (tail (reverse ys), [x])
delQ (Q []) = error "cannot remove from empty Q"

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

newtype Queue a = Q (xs, ys)

emptyQ = Q ([], [])
isEmptyQ (Q ([], [])) = True
isEmptyQ _ = False
addQ x (Q (xs, ys)) = Q (xs, [x] : ys)
delQ (Q (xs, ys)) = Q (tail (reverse ys), [x])
delQ (Q ([], ys)) = Q (tail (reverse ys), [x])

as data but will not permit the use of the Prelude list functions
addQ \ x \ Q ([], []) = Q ([x], [])

most recent addition

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

first in the second part of the queue

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

delQ \ Q ([], ys) = Q (tail (reverse ys), [])
Functional Programming for DB

**Queue via two lists**

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

**Set as unordered list with duplicates**

```haskell
module Set (Set, emptyS, isEmptyQ, inS, addS, delS) where

emptyS :: Set a
isEmptyQ :: 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)
```