Polymorphic types

- Some functions have a type definition involving only type names:

```
and :: [Bool] -> Bool
and = foldr (&&) True
```

  These functions are monomorphic.

- It is however useful sometimes to write functions that can work on data of more than one type. These are polymorphic functions.

```
length :: [a] -> Int  -- for any type a, length :: [a]->Int
map :: (a -> b) -> [a] -> [b]
```

- Restricted polymorphism: What is the most general type of a function that sorts a list of values, and why?

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++
               qsort [y | y <- xs, y > x]
```


Type synonyms

- A function that adds two polynomials (see assignments) with floating point coefficients: `polyAdd :: [Float] -> [Float] -> [Float]`
- `polyAdd :: Poly -> Poly -> Poly` would have been nicer though...
  - You can do this if you define "`Poly`" as a type synonym for `[Float]`:
    `type Poly = [Float]`
  - Type synonyms can also be parameterized:

```
type Stack a = [a]

newstack :: Stack a
newstack = []

push :: a -> Stack a -> Stack a
push x xs = x:xs

aCharStack :: Stack Char
aCharStack = push 'a' (push 'b' newstack)
```

```
Main> aCharStack
"ab"
Main> push 'x' aCharStack
"xab"
Main> :t aCharStack
aCharStack :: Stack Char
```


Algebraic types

- Remember when we defined functions using induction (aka recursion)?
- Types can be defined in a similar manner (the general form of mathematical induction is called structural induction): Take for example natural numbers:

```
data Nat = Zero | Succ Nat
    deriving Show

-- Operations:
addNat,mulNat :: Nat -> Nat -> Nat
addNat m Zero = m
addNat m (Succ n) = Succ (addNat m n)
mulNat m Zero = Zero
mulNat m (Succ n) = addnat (mulNat m n) m
```

- Again, type definitions can be parameterized:

```
data List a  = Nil | Cons a (List a)
--  data [a] = []  | a : [a]
data BinTree a = Null | Node a (BinTree a) (BinTree a)
```


Type classes

- Each type may belong to a type class that define general operations. This also offers a mechanism for overloading.
  - Type classes in Haskell are similar with abstract classes in Java.

```
data Nat = Zero | Succ Nat
           deriving (Eq,Show)

instance Ord Nat where
    Zero <= x = True
    x <= Zero = False
    (Succ x) <= (Succ y) = x <= y

one,two,three :: Nat
one = Succ Zero
two = Succ one
three = Succ two
```

```
Main> one
Succ Zero
Main> two
Succ (Succ Zero)
Main> three
Succ (Succ (Succ Zero))
Main> one > two
False
Main> one > Zero
True
Main> two < three
True
Main>
```

Ord Nat works because...

- Class Ord is defined in the standard prelude as follows:

```
class (Eq a) => Ord a where
    compare                :: a -> a -> Ordering
    (<), (<=), (>=), (>)   :: a -> a -> Bool
    max, min               :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x >= y      = x
              | otherwise   = y
    min x y   | x <= y      = x
              | otherwise   = y
```


Type classes (cont'd)

```
data Nat = Zero | Succ Nat
           deriving (Eq,Ord,Show)

instance Num Nat where
    m + Zero = m
    m + (Succ n) = Succ (m + n)
    m * Zero = Zero
    m * (Succ n) = (m * n) + m

one,two,three :: Nat
one = Succ Zero
two = Succ one
three = Succ two
```

```
Main> one + two
Succ (Succ (Succ Zero))
Main> two * three
Succ (Succ (Succ (Succ
 (Succ (Succ Zero)))))
Main> one + two == three
True
Main> two * three == one
False
Main> three - two

ERROR - Control stack
        overflow
```


Definition of Num

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
    fromInt        :: Int -> a

    -- Minimal complete definition: All, except negate or (-)
    x - y           = x + negate y
    fromInt         = fromIntegral
    negate x        = 0 - x
```


Subtraction

```
instance Num Nat where
    m + Zero = m
    m + (Succ n) = Succ (m + n)
    m * Zero = Zero
    m * (Succ n) = (m * n) + m
    m - Zero = m
    (Succ m) - (Succ n) = m - n

Nat> one - one
Zero
Nat> two - one
Succ Zero
Nat> two - three

Program error: pattern match failure:
               instNum_v1563_v1577 Nat_Zero one
```

Other interesting type classes

```
class Enum a where
    succ, pred           :: a -> a
    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]

    -- Minimal complete definition: toEnum, fromEnum
    succ                  = toEnum . (1+)       . fromEnum
    pred                  = toEnum . subtract 1 . fromEnum
    enumFrom x            = map toEnum [ fromEnum x ..]
    enumFromTo x y        = map toEnum [ fromEnum x .. fromEnum y ]
    enumFromThen x y      = map toEnum [ fromEnum x, fromEnum y ..]
    enumFromThenTo x y z  = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```

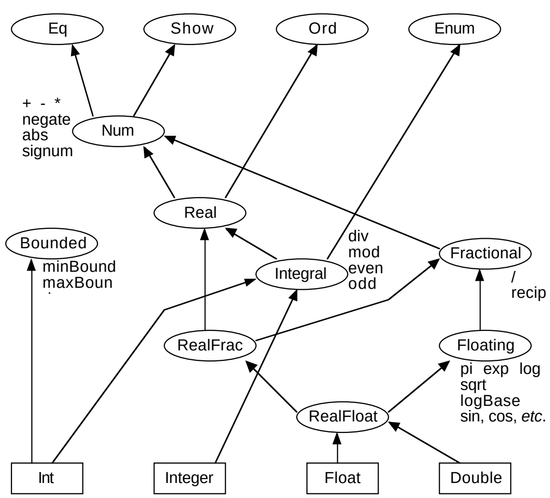

Other interesting type classes (cont'd)

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

    -- Minimal complete definition: show or showsPrec
    show x          = showsPrec 0 x ""
    showsPrec _ x s = show x ++ s
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
                      where showl []     = showChar ']'
                            showl (x:xs) = showChar ',' .
                                           shows x . showl xs

instance Show Nat where
    show Zero = "0"
    show (Succ n) = "1 + " ++ show n
```


Example of type classes



Abstract data types and modules

- An abstract data type specifies the allowed operations and consistency constraints for some type, without specifying an actual implementation.

ADT $Stack$

type $Stack(Elm)$

Operators:

$\underline{empty} : \emptyset \to Stack$
$\underline{push} : Elm \times Stack \to Stack$
$\underline{pop} : Stack \to Stack$
$top : Stack \to Elm$
$isEmpty : Stack \to Bool$

Axioms:

$pop(push(e, S)) = S$
$top(push(e, S)) = e$
$isEmpty(empty) = true$
$isEmpty(push(e, S)) = false$

Restrictions:

$pop(empty)$
$top(empty)$

□

Haskell implementation

```
module Stack (Stack, empty,
              push, pop, top, isEmpty) where
data Stack a = Empty | Push a (Stack a)
               deriving Show

empty :: Stack a
push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
isEmpty :: Stack a -> Bool
empty = Empty
push a s = Push a s
pop (Push e s) = s
pop (Empty) = error "pop (empty)."
top (Push e s) = e
top (Empty) = error "top (empty)"
isEmpty (Push e s) = False
isEmpty Empty = True
```

ABSTRACT DATA TYPES AND MODULES (CONT'D)

```
-- main.hs (Stack module in Stack.hs)
module Main where
import Stack

aStack = push 0 (push 1 (push 2 empty))


Main> aStack
Push 0 (Push 1 (Push 2 Empty))
Main> :type aStack
aStack :: Stack Integer
Main> isEmpty aStack
False
Main> pop aStack
Push 1 (Push 2 Empty)
Main> let aStack = push 0 empty in pop (pop aStack)

Program error: pop (empty).

Main>
```


SIMPLIFIED STACK

```
-- Stack.hs
module Stack where

data Stack a = Empty | Push a (Stack a)
               deriving Show

pop :: Stack a -> Stack a
top :: Stack a -> a
isEmpty :: Stack a -> Bool

pop (Push e s) = s
pop (Empty) = error "pop (empty)."
top (Push e s) = e
top (Empty) = error "top (empty)"
isEmpty (Push e s) = False
isEmpty Empty = True


-- main.hs
module Main where
import Stack

aStack = Push 0 (Push 1 (Push 2 Empty))
```

```
Main> aStack
Push 0 (Push 1 (Push 2 Empty))
Main> :r
Main> aStack
Push 0 (Push 1 (Push 2 Empty))
Main> :type aStack
aStack :: Stack Integer
Main> isEmpty aStack
False
Main> pop aStack
Push 1 (Push 2 Empty)
Main> let aStack = Push 0 Empty
         in pop (pop aStack)

Program error: pop (empty).
```


TYPE INFERENCE

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments:
  - Inspect the argument patterns if any:
  - parse definitions top to bottom, right to left:
    * "q : []" $\Rightarrow$
    * "(f q x)" $\Rightarrow$
    * "scanl f (f q x) xs" $\Rightarrow$
    * "q : scanl f (f q x) xs" $\Rightarrow$
  - So the overall type is

TYPE INFERENCE

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    * "q : []" $\Rightarrow$ no extra information
    * "(f q x)" $\Rightarrow$ f must be a function i.e. $(\beta \to \gamma \to \eta) \to \beta \to [\gamma] \to \delta$
    * "scanl f (f q x) xs" $\Rightarrow$ $\beta = \delta$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to \delta$
    * "q : scanl f (f q x) xs" $\Rightarrow$ $\delta = [\beta]$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to [\beta]$
  - So the overall type is `scanl :: (a -> b -> a) -> a -> [b] -> [a]`