# Lecture 5 – Type Class

- Type class is a form of *ad-hoc* polymorphism
  - A type class specifies a set of overloaded operators.
  - An instance of a type class implements the operators in the type class.
  - A type class may be extended with subclasses.
- A program using type classes can be translated to a program without type classes.

# Why type class

- *Ad hoc* polymorphism with finite overloading is limited

```
square x = x * x

squares (x, y, z) = (square x, square y, square z)
```

- If square can apply to either Int or Float.

```
square :: Int -> Int
square :: Float -> Float
```

- Then the type of squares has 8 possibilities

```
squares :: (Int, Int, Int) -> (Int, Int, Int)
squares :: (Int, Int, Float) -> (Int, Int, Float)
squares :: (Int, Float, Float) -> (Int, Float, Float)
squares :: (Int, Float, Int) -> (Int, Float, Int)
squares :: (Float, Float, Float) -> (Float, Float, Float)
squares :: (Float, Float, Int) -> (Float, Float, Int)
squares :: (Float, Int, Int) -> (Float, Int, Int)
squares :: (Float, Int, Float) -> (Float, Int, Float)
```

# Why type class

- Equality type is another problem with *ad hoc* polymorphism.

```
member [] y = False
member (x:xs) y = (x == y) || elem xs y
```

- The type of member cannot be just `[a] -> a -> Bool` since not all a typed values can be compared with `==`.
- SML uses a type variable ‘‘a that can only be instantiated with types that can be compared with equality operator.

```
member : [``a] -> ``a -> bool
```

# Num type class

Num class defines arithmetic operators for numbers: Int or Float.

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a

instance Num Int where
  (+)    = addInt
  (*)    = multInt
  negate = negInt

instance Num Float where
  (+)    = addFloat
  (*)    = mulFloat
  negate = negFloat
```

# Num type class

The square and squares functions now constrain their type parameters to be in Num class.

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a

square :: Num a => a -> a
square x = x * x

squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
squares (x, y, z) = (square x, square y, square z)
```

# Eq type class

Eq class defines equality operator on types with equality.

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

instance Eq Int where
  (==) = eqInt

instance Eq Char where
  (==) = eqChar

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

# Eq type class

Eq can be extended to tuples and lists, where type parameters of tuple and list constructors are restricted to be in Eq class.

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

instance (Eq a, Eq b) => Eq (a,b)
  (u, v) == (x, y) = (u == x) && (v == y)

instance Eq a => Eq [a] where
  [] == []      = True
  [] == y:ys    = False
  x:xs == []    = False
  x:xs == y:ys = (x == y) && (xs == ys)
```

# Eq type class

Eq can be extended to sets as well. Two sets are equal if each member of a set is a member of the other set.

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

data Set a = MkSet [a]

instance Eq a => Eq (Set a) where
  MkSet xs == MkSet ys =
        and (map (member xs) ys) &&
        and (map (member ys) xs)
```

The function and is a conjunction of a list of Booleans.

# Define a ToString class

Define a class that supports a toString function for its instances.

```
class ToString a where
  toString :: a -> String

data WeekDay = Tuesday | OpenDay | Thursday | MeetingDay | BlurDay
               deriving (Show, Read, Eq, Ord)

instance ToString WeekDay where
  toString Tuesday = "Tu"
  toString OpenDay = "Wed"
  toString Thursday = "Th"
  toString MeetingDay = "Fr"
  toString BlurDays = "#!?"
```

# Instances of ToString class

We still auto-derive Show class for these data types for serialization purpose.

```haskell
data DayTime = Morning | Afternoon | Night
               deriving (Show, Read, Eq, Ord)

instance ToString DayTime where
  toString Morning = "AM"
  toString Afternoon = "PM"
  toString Night = "zzz"
```

# Instances of ToString class

```haskell
data WeekNum = Week Int deriving (Show, Read, Eq, Ord)

instance ToString WeekNum where
  toString (Week n) = "Wk-" ++ show n

data MeetTime = Meet WeekNum WeekDay DayTime
                deriving (Show, Read, Eq, Ord)

instance ToString MeetTime where
  toString (Meet w d t) =
     toString w ++ ": " ++ toString d ++ "/" ++ toString t
```

# Show and Read

Auto-derive the Show and Read instance for Event type so that a calendar can be converted to and from a string.

```
type Calendar = [Event]

data Event = Tag String [Event]
           | One MeetTime          deriving (Show, Read)

:t show
show :: Show a => a -> String

:t read
read :: Read a => String -> a
```

Read class has a read function that takes a string and returns a data, opposite to the show function. Every type used by Calendar has to be an instance of Show and Read.

# Print calendar

```haskell
-- mconcat does the job of foldl (++) []
showCalendar _ [] = ""
showCalendar tabs events = mconcat $ e : map ('\n':) es
  where (e:es) = map (showEvent tabs) events

showEvent tabs (One m) = tabs ++ toString m
showEvent tabs (Tag tag lst) =
  tabs ++ tag ++ "\n" ++ showCalendar ('\t':tabs) lst

instance ToString Calendar where
  toString c = showCalendar "" c
```

Defining instance for a type synonym like Calendar needs to set some flags

```
:set -XTypeSynonymInstances
:set -XFlexibleInstances
```

# File IO

An ADT can be serialized to a string using its show function and then be saved to a file using writeFile function.

```
:t writeFile
writeFile :: FilePath -> String -> IO ()

:i FilePath
type FilePath = String
```

writeFile takes a path and write a string to the file of the path. writeFile returns an IO monad $\Rightarrow$ use writeFile inside a do block.

```
writeFile "calendar.txt" $ show c3
```

Write the calendar c3 as a string to the file calendar.txt.

# File IO

We can read a file using readFile function.

```
:t readFile
readFile :: FilePath -> IO String
```

readFile takes a path and return an IO monad ⇒
use <- to extract the returned string of readFile inside a do block.

```
x <- readFile "calendar.txt"
let c3' = read x
```

Read the content of calendar.txt and convert it to a calendar c3'.

# File IO

We can read a file using readFile function.

```
main = do
          let m1 = Meet (Week 1) Tuesday Afternoon
          let m2 = Meet (Week 1) OpenDay Morning
          let m3 = Meet (Week 1) Thursday Morning
          let (c1, _) = addEvent [] ["Research", "Adam"] m1
          let (c2, _) = addEvent c1 ["Research", "Alan"] m2
          let (c3, _) = addEvent c2 ["Courses", "790", "Bob"] m3
          writeFile "calendar.txt" $ show c3

          x <- readFile "calendar.txt"
          putStrLn x
          let c3' = read x

          let (c4, _) = addEvent c3' ["Courses", "431", "Carol"] m3
          putStrLn $ toString c4
          writeFile "calendar.txt" $ show c4
```

# File IO

Run main to print the file calendar.txt that corresponds to c3 (serialized version) and the formatted c4.

```
main

[Tag "Research" [Tag "Adam" [One (Meet (Week 1) OpenDay Afternoon)
                 Tag "Alan" [One (Meet (Week 1) OpenDay Morning)]]
 Tag "Courses" [Tag "790" [Tag "Bob" [One (Meet (Week 1) Thursday

Research
        Adam
                Wk-1: Wed/PM
        Alan
                Wk-1: Wed/AM
Courses
        790
                Bob
                        Wk-1: Th/AM
        431
                Carol
                        Wk-2: Tu/AM
```

# Case expression

Use case expression to do pattern matching inside a function.

```
case exp of
  pattern1 -> exp1
  pattern2 -> exp2
  ...

head [] = error "empty list"
head (a:_) = a

head' xs = case xs of [] -> error "empty list"
                      (a:_) -> a
```

# Polymorphic ADT

Maybe type is either Just a value or Nothing

```
data Maybe a = Just a | Nothing

data Grade = A | B | C | D | F Int | W deriving (Eq)

gpa A = 4
gpa B = 3
gpa C = 2
gpa D = 1
gpa (F _) = 0

grades = [A, B, A, C, D, W, F 16, A, B]

points = map (\x -> case x of W -> Nothing
                              _ -> Just $ gpa x) grades
print points

-- [Just 4,Just 3,Just 4,Just 2,Just 1,
--  Nothing,Just 0,Just 4,Just 3]
```

# Polymorphic ADT

```haskell
data Maybe a = Just a | Nothing

data Grade = A | B | C | D | F Int | W deriving (Eq)

grades = [A, B, A, C, D, W, F 16, A, B]

points = map (\x -> case x of W -> Nothing
                              _ -> Just $ gpa x) grades

actuals = foldr (\e c -> case e of Just y -> y : c
                                   Nothing -> c) [] points
print actuals

-- [4,3,4,2,1,0,4,3]
```

# Polymorphic ADT

`checkMeeting` function checks whether there is a meeting at a particular meet time. The result is a maybe value.

```
type Calendar = [Event]
data Event = Tag String Calendar | One MeetTime

checkMeeting :: MeetTime [Event] = Maybe ([String], MeetTime)

checkMeeting m events = checks [] events
  where checks _ [] = Nothing
        checks tags (e:es) = case check tags e
                                of Just x -> Just x
                                   Nothing -> checks tags es
        check tags (One m')
            | m == m' = Just (tags, m)
            | otherwise = Nothing
        check tags (Tag tag events) = checks (tags++[tag]) events
```

# Polymorphic ADT

`checkMeeting` function checks whether there is a meeting at a particular meet time. The result is a maybe value.

```
main = do
          let m1 = Meet (Week 1) Tuesday Afternoon
          let m2 = Meet (Week 1) OpenDay Morning
          let m3 = Meet (Week 1) Thursday Morning
          let (c1, m1') = addEvent [] ["Research", "Adam"] m1
          let (c2, m2') = addEvent c1 ["Research", "Alan"] m2
          let (c3, m3') = addEvent c2 ["Courses", "790", "Bob"] m3
          let (c4, _) = addEvent c3 ["Courses", "431", "Carol"] m3
          print $ checkMeeting m1 c4
          print $ checkMeeting m1' c4

-- Nothing
-- Just (["Research","Adam"],Meet (Week 1) OpenDay Afternoon)
```

# Polymorphic ADT

```haskell
data Maybe a = Just a | Nothing

data Grade = A | B | C | D | F Int | W deriving (Eq, Ord)

grades = [A, B, A, C, D, W, F 16, A, B]

-- points = map (\x -> case x of W -> Nothing
--                               _ -> Just $ gpa x) grades

maybe_gpa x
  | x < W = Just $ gpa x
  | otherwise = Nothing

points = map maybe_gpa grades
print points

-- [Just 4,Just 3,Just 4,Just 2,Just 1,
--  Nothing,Just 0,Just 4,Just 3]
```

# Polymorphic ADT

```haskell
data Maybe a = Just a | Nothing

data Grade = A | B | C | D | F Int | W deriving (Eq, Ord)

grades = [A, B, A, C, D, W, F 16, A, B]

bump_gpa = fmap f where
    f x
      | x < 4 = x + 1
      | otherwise = x

points = map maybe_gpa grades
print $ map bump_gpa points

-- [Just 4,Just 4,Just 4,Just 3,Just 2,
--  Nothing,Just 1,Just 4,Just 4]
```

# What is fmap?

fmap transforms one function `h :: a -> b` to another function `h' :: f a -> f b` where `f` is a type constructor that is an instance of the Functor type class.

```haskell
data Maybe a = Just a | Nothing

data Grade = A | B | C | D | F Int | W deriving (Eq, Ord)

grades = [A, B, A, C, D, W, F 16, A, B]

bump_gpa = fmap f where
   f x
    | x < 4 = x + 1
    | otherwise = x

:t fmap
-- fmap :: Functor f => (a -> b) -> f a -> f b
```

# Functor type class

The Functor type class categorizes data type constructor $f$ of the *kind* `* -> *` with one method `fmap` that acts as an adapter that transforms an ordinary function `a -> b` to another function `f a -> f b`.

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

Or we can say that every instance $f$ of the Functor class has a functor `fmap` that transforms an ordinary function from a to b type to another from `f a` to `f b` type.

# A word about kind

*Kind* is the type of types. An ordinary type has the kind $*$. A type constructor with one parameter has the kind `* -> *`.

```
Int :: *
Int -> Int :: *
Maybe Int :: *
[Int] :: *

Maybe :: * -> *
(->) :: * -> * -> *
(,) :: * -> * -> *
[] :: * -> *
```

# Functor type class

*Maybe* is an instance of the Functor class.

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just $ f x
```

# Functor type class

*List* is an instance of the Functor class.

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap _ [] = []
  fmap f (a::b) = f a :: fmap f b
```

or

```
instance Functor [] where
  fmap = map
```

# Binary tree

A binary tree type is defined as either a leaf or a node with two subtrees and a value.

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show)
```

A value of type a in *Ord* class can be inserted into a tree of type a.

```
insertTree :: (Ord a) => Tree a -> a -> Tree a

insertTree Leaf x = Node Leaf x Leaf
insertTree (Node left y right) x
   | x < y      = Node (insertTree left x) y right
   | otherwise  = Node left y (insertTree right x)
```

InsertTree function ensures a binary search tree.

# Binary tree

listToTree function builds a binary tree from a list.

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show)

insertTree Leaf x = Node Leaf x Leaf
insertTree (Node left y right) x
    | x < y      = Node (insertTree left x) y right
    | otherwise = Node left y (insertTree right x)

listToTree :: (Ord a) => [a] -> Tree a
listToTree [] = Leaf
listToTree (a:b) = insertTree (listToTree b) a

listToTree [3,1,2]
-- Node (Node Leaf 1 Leaf) 2 (Node Leaf 3 Leaf)
```

# Tree as a Functor instance

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show)

instance Functor Tree where
  fmap _ Leaf = Leaf

  fmap f (Node left x right) =
    Node (fmap f left) (f x) (fmap f right)

inc = \x -> x + 1

incTree = fmap inc

t = listToTree [3,1,2]

incTree t
-- Node (Node Leaf 2 Leaf) 3 (Node Leaf 4 Leaf)
```

# Tree as a Functor instance

```haskell
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show)

instance Functor Tree where
  fmap _ Leaf = Leaf

  fmap f (Node left x right) =
    Node (fmap f left) (f x) (fmap f right)

evenOrOdd = \x -> mod x 2 == 0

t = listToTree [3,1,2]

fmap evenOrOdd t
-- Node (Node Leaf False Leaf) True (Node Leaf False Leaf)
```

# Functor laws

- fmap should map identity function to identity function.

```
id = \x -> x

-- fmap id = id
```

- fmap should map the composition of functions to the composition of fmap'ed functions.

```
-- fmap (f . g) = (fmap f) . (fmap g)
```

# Functor laws – preserve id

```haskell
id = \x -> x

-- fmap id = id

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just $ f x

-- fmap id Nothing = Nothing

-- id Nothing       = Nothing

-- fmap id (Just x) = Just $ id x
--                  = Just x

-- id (Just x)      = Just x
```

# Functor laws – preserve composition

```haskell
-- fmap (f . g) = (fmap f) . (fmap g)

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just $ f x

-- fmap (f . g) Nothing = Nothing

-- ((fmap f) . (fmap g)) Nothing
--                        = (fmap f) (fmap g Nothing)
--                        = (fmap f) Nothing
--                        = Nothing

-- fmap (f . g) (Just x) = Just $ (f . g) x
--                        = Just $ f (g x)

-- ((fmap f) . (fmap g)) $ Just x
--                         = (fmap f) (fmap g $ Just x)
--                         = (fmap f) (Just $ g x)
--                         = Just $ f (g x)
```

# Tree as an instance of Foldable class

```haskell
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show)

instance Foldable Tree where

  foldr f c Leaf = c

  foldr f c (Node left x right) = foldr f c' left

    where c' = f x right'

          right' = foldr f c right
```

# Tree as an instance of Foldable class

```haskell
data Tree a = Leaf
            | Node (Tree a) a (Tree a) deriving (Show)

instance Foldable Tree where
  foldr f c Leaf = c
  foldr f c (Node left x right) = foldr f c' left
    where c' = f x right'
          right' = foldr f c right

t = listToTree [3,1,2]

foldr (:) [] t
-- [1,2,3]

foldl (\c e -> c ++ [e]) [] t
-- [1,2,3]

foldl (flip (:)) [] t
-- [3,2,1]
```