Type classes part 2

LYaH:

- Chapter 3 *Types and Type Classes*
- Chapter 8: *Making Our Own Types and Typeclasses*

Another generalization of classes

1. Done: classes as sets of types that have certain ops defined.
2. Done: classes as sets of *tuples* of types that have certain ops defined.
3. New: classes as sets of **type constructors**.

Notation: `*` stands for the *kind* of all types. Some Haskell versions use `type` for `*`.

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

Notation: `* -> *` is the kind of all one-argument type constructors.

```
[]          :: * -> *    -- for any type a, [a] is a type, i.e. a ↦ [a]
Maybe       :: * -> *    -- a ↦ Maybe a
Tree        :: * -> *    -- where data Tree a = Leaf a | Node (Tree a) (Tree a)
((->) r)    :: * -> *    -- a  ↦  r->a
((,) r)     :: * -> *    -- a  ↦ (r,a)
```

Int -> Int ≡ (->) Int Int

parameterized by r.

The `Functor` class for "container" type constructors

f :: type → type

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

instance Functor [] where
    fmap f l = map f l

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

data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Node t0 t1) = Node (fmap t0) (fmap t1)
```

-- for f = [], fmap = map

a ↦ Tree a

fmap f t1

$a \mapsto (r, a)$

```haskell
class Functor ((,) r) where
    -- fmap :: (a -> b) -> (r,a) -> (r,b)
    fmap f (x,y) = (x, f y)
```

$a \mapsto (r \to a)$

```haskell
class Functor ((->) r) where
    -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = \x -> f (g x) -- = f . g
```

$\in r \to a$

A puzzle for you. What does `fmap fmap fmap` do?

Monads

General definition of *monad* m :: type -> type

```
class Monad m where
  (>>=) :: t a -> (a -> t b) -> t b  -- the "bind" operator -- ???
  (>>) :: t a -> t b -> t b          -- just a special case of the bind operator
  return :: a -> t a                 -- insert a value
```

What does this mean in general? Nothing!

We understand it through particular kinds of instances.

if t : type -> type is a Monad instance

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Just x  >>= f  =  f x
  Nothing >>= f  =  Nothing
  -- return :: a -> Maybe a
  return x = Just x

instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b   = ...
  return :: a -> IO a                    = ...
```

What `do` translates to in Haskell

```
do
    x1 <- e1          e1 >>= (\x1 ->
    x2 <- e2                   e2 >>= (\x2 ->
    ...        ===>                        ...
    xn <- en                                    en >>= (\xn ->
    e                                                     e))...)
```

Can use the constructors directly:

```
printWithIntro :: (Show a) -> String -> a -> IO()
printWithIntro str x = putStr str >> print x
```

```haskell
instance Monad [] where
  -- (>>=) :: [a] -> (a -> [b]) -> [b]
  l >>= f = concat (map f l)
  -- return :: a -> [a]
  return x = [a]
```

x

A puzzle for you. What does the following return?

[(x, 4), (x, 5), (x, 6)]

```haskell
do
  x <- [1,2,3]
  y <- [4,5,6]
  return (x,y)
```

≡ [1, 2, 3] >>= \x -> ([4, 5, 6] >>= \y -> [(x, y)])

~~[(1,4), (1,5~~

Interpreters

Big picture

Theme
Symbolic computing
Metaprogramming
Reflection
Compilers
Interpreters

Common element:

Center on **formalization of syntax**, i.e. data structures representing syntax.

Formalized syntax from assignment 6

"Concrete" ("surface") syntax example:

```
f=plus(times(3.3,23.4),0.0)
g=x
h=f(f(g(x,y)),h(z))
```

Haskell type for "abstract" syntax:

```haskell
data Exp
  = Const Double
  | Var String
  | If Exp Exp Exp
  | App1 Name Exp
  | App2 Name Exp Exp
```

Writing interpreters in Haskell

- representative of a large&important category of applications
- functional programming with pattern matching is ideally suited
- writing interpreters deepens understanding of the interpreted language
- will learn about two important+deep CS topics:
  - term-rewriting
  - denotational semantics

Plan

1. Overview of the Scheme programming language.
2. Formalize the (almost-trivial) syntax of Scheme in Haskell.
3. Develop an interpreter (evaluator) based on term-rewriting.
4. Develop an interpreter (evaluator) based on structural recursion (i.e. recursion with pattern matching).

Scheme: a variant of Lisp, the first FP language

We'll just be using the functional part.

Scheme as compared to Haskell:

- No currying: each function has a fixed number of arguments.
- No types/typechecking.
- No pattern matching or guards.
- Call-by-value: in function call, arguments are evaluated before being passed to function.
- Data is only numbers, strings, `#t/#f` (true/false), `nil` and "conses" (pairs).
- Syntax is by far the simplest of any programming language.

Example Scheme programs

```
;;; Using built-in integers
(define (factorial n)
  (if (eq? n 0)
        1
        (* n (factorial (- n 1)))))
```

```
;;; cons is the only provided way build data structures
(define (zip l0 l1)
  (cond ((empty? l0)
         (list))
        ((empty? l1)
         (list))
        (#t
         (cons (cons (car l0) (car l1)) (zip (cdr l0) (cdr l1))))))
```

l0 eg (list 0 1 2) ~ [0, 1, 2]

can't write in a Scheme program

1st of cons pair

2nd of cons

(list 0 1) ≡ (cons 0 (cons 1 '()))

Nil

A Haskell data type for Scheme expressions

```haskell
data Exp
  = Atom String
  | List [Exp]
  | Number Integer
  | String String
  | Nil
  | Bool Bool
  deriving (Eq, Ord, Show)
```

Parse of "(define (factorial ..."

```
List
    [ Atom "define"
    , List [ Atom "factorial" , Atom "n" ]
    , List
        [ Atom "if"
        , List [ Atom "eq?" , Atom "n" , Number 0 ]
        , Number 1
        , List
            [ Atom "*"
            , Atom "n"
            , List
                [ Atom "factorial"
                , List [ Atom "-" , Atom "n" , Number 1
                    ]
                ]
            ]
        ]
    ]
```

Tree addressing

The result returned by `parseExp :: String -> Exp` is called a

- *parse tree* or an
- *abstract syntax tree*, AST for short

It will be convenient to refer to subtrees using tree *paths*.

Convenient fact: Scheme AST nodes are simply either leaves or a list of subtrees.

```
type Path = [Int]
```

Exercise: locate the expression at the path address [2,3,0] on the previous slide.

It's convenient to package together an expression and a path into it.

```haskell
data Lens = Lens
  { lensExp :: Exp
  , lensPath :: Path
  }
```

Two fundamental operations on lenses:

```haskell
-- Get the subexpression addressed by the path.
get :: Lens -> Exp

-- Replace (can't really "set" anything in Haskell) the addressed subexpression
set :: Lens -> Exp -> Exp
```

Exercise: try `set` and `get` on `lens0` in the lecture's Haskell file.