□ List Comprehensions

Convenient syntax for creating lists.

```
[f x | x <- xs]
```

The list of all `f x` such that `x` is drawn from `xs`.

```
[(x,y) | x <- xs, y <- ys]
```

The cartesian product of the lists `xs` and `ys`.

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

□ Homework 3 (more details)

Here are some different datatypes for implementing the lambda calculus:

Named variables:

```
data Exp = Var String | Lam String Exp | App Exp Exp
```

DeBruijn Indices:

```
data Exp' = Var' Int | Lam' Exp' | App' Exp' Exp'
```

Write a normalization function for each of these representations of the lambda calculus.

Also make each representation a member of the `Show` type class; have the show functions generate Haskell syntax.

```
show $ Lam "x" (App (Var "x") (Lam "y" (App (Var "x") (Var "z"))))
↪ "(\x -> (x (\y -> (x z))))"
```

Have the printer for DeBruijn terms convert to named syntax:

```
show $ Lam' (App' (Var' 0) (Lam' (App' (Var' 1) (Var' 2))))
↪ "(\x0 -> (x0 (\x1 -> (x0 x2))))"
```

Extra credit: Do the same for the following

Higher-Order Abstract Syntax:

```
data Exp = Lam (Exp -> Exp) | App Exp Exp
```

□ Ad hoc Polymorphism

Some polymorphic functions behave in a type specific manner.

`==`   `<`   `>`   `+`   `show`

Assigning == the type `a -> a -> Bool` is wrong since that implies == works for all types in a uniform manner.

There should be a separate implementation of == for each type for which equality is defined.

Haskell provides a clean, structured way to do this.

□ Type Classes

Type classes define a class of types for which particular functions are defined.

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

A specific instance of each function in the class must be given for each type in the class.

```
instance Eq Int where
    x == y = x ‘integerEq‘ y
```

Now we can write the type of equality as

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

`Eq a` expresses a constraint on the type, and is called a context.

□ Using Type Classes

Regular functions can have constrained types.

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

elem :: Eq a => a -> [a] -> Bool
```

Recursive types can be members of a type class.

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

Instance declarations can also have contexts.

□ Defaults and Extensions

Type classes can have default methods.

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

Classes can extend other classes.

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

`Eq` is a superclass of `Ord`.

□ Higher-order Type Classes

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

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

instance Functor Tree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
```

The `Functor` class is composed of type functions (e.g. `List`, `Tree`).

`instance Functor Int where ...` would result in a *kind* error.

Kinds are types for types.

□ Derived Instances

Equality for trees

```
instance Eq a => Eq (Tree a) where
    Leaf x     == Leaf y       = x == y
    Branch l r == Branch l' r' = l == l' && r == r'
    _          == _            = False
```

is quite similar to that for any other datatype.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Eq
```

causes Haskell to automatically generate the equality instance.

Some classes can be derived for all types (e.g. Eq, Show).

Some classes have restrictions on the types they can be derived for (e.g. Enum).

□ Showing Trees

```
class Show a where
    show :: a -> String
```

A class for translating to a string.

A pretty printer for trees

```
showTree :: (Show a) => Tree a -> String
showTree (Leaf x) = show x
showTree (Branch l r) =
    "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

(++), i.e. append, makes this function inefficient.

Note this is different from the derived show function for trees.

□ A Faster Algorithm

```
shows :: (Show a) => a -> String -> String
show x = shows x ""
```

`shows` is part of `Show` class.

```
showsTree :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) s = shows x s
showsTree (Branch l r) s =
    '<' : showsTree l ('|' : showsTree r ('>' : s))

showTree x = showsTree x ""
```

`showsTree` is linear in size of given tree.

□ A Nicer Presentation

```
type ShowS =  String -> String
```

Haskell allows type synonyms.

```
showsTree :: (Show a) => Tree a -> ShowS
showsTree (Leaf x) = shows x
showsTree (Branch l r) =
    ('<' :) . showsTree l . ('|' :) . showsTree r . ('>' :)
```

Note:

. is function composition: `f . g` ≡ `\x -> f (g x)`

strings are lists of Chars and `('<' :)` ≡ `\x -> '<' : x`

This is a “point free” presentation.

□ Power of Type Classes

- Structured method for dealing with ad hoc polymorphism.
- Contexts allow for constrained polymorphic types.
- Can be simulated with extra explicit machinery, but:
  - type classes allow for nice code presentation
  - type classes allow better code re-use