This week:
- Defining infix operators
  - Fixity declarations and precedence
- Field labels
- Type classes!
- More on I/O
Defining infix operators

- Infix operators are really just functions
  - Can be defined like other functions e.g.
    
    ```hs
    (+++) :: [a] -> [a] -> [a]  
    [] ++ ys = ys  
    (x:xs) ++ ys = x : (xs ++ ys)
    ```

    ```hs
    (.) :: (b->c) -> (a->b) -> (a->c)  
    f . g = \x -> f (g x)
    ```
  
  - (These are both in the Prelude)
Fixity (precedence) declarations

- Ten operator precedence levels exist in Haskell
  - can't add any more
  - can assign new operators to any precedence

- **infixr 5 ++**
- **infixr 9 .**

- **infixr**: right-associative
- **infixl**: left-associative
- **infix**: non-associative
Field labels (1)

- Might want to define a record-like data structures

```haskell
data Point = Pt Float Float

pointx :: Point -> Float
pointx (Pt x _) = x

pointy :: Point -> Float
pointy (Pt _ y) = y
```
Field labels (2)

- Short form:
  
  ```haskell
  data Point = Pt { pointx :: Float,  
                   pointy :: Float }
  ```

- or:
  
  ```haskell
  data Point = Pt { pointx, pointy :: Float }  
  :t pointx
  pointx :: Point -> Float
  :t pointy
  pointy :: Point -> Float
  ```
Field labels (3)

- Can use field labels to construct new values:
  \[ \text{Pt} \{\text{pointx} = 1, \text{pointy} = 2\} \]

- Equivalent to:
  \[ \text{Pt} \ 1 \ 2 \]

- Can pattern match on labels:
  \[
  \text{absPoint} :: \text{Point} \to \text{Float} \\
  \text{absPoint} (\text{Pt} \{\text{pointx} = x, \text{pointy} = y\}) = \sqrt{x^2 + y^2}
  \]
Type classes (1)

- Some operations can be defined for many different data types
  - `== /=` defined for many types
  - `< <= > >=` defined for many types
  - `+ - *` defined for numeric types
- Causes problems for most languages
  - does `+` mean "add integers" or "add floats"?
- Most languages resolve using variable type decls
- Some define separate operators (`+` vs `+.`)
Type classes (2)

- Problems:
  - May want to overload operators for new data types
  - Want to resolve all types at compile time
  - Don't want to break type inference

- Solution:
  - Declare certain groups of operations as a "type class"
**Type classes (3)**

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

- This declares `Eq` as a type class with two operations `==` and `/=` of type `a -> a -> Bool`
- Provides default definition for `/=` in terms of `==`
- Defined in Prelude
Defining class instances (1)

- Make pre-existing classes instances of type class:

```haskell
instance Eq Integer where
    x == y = x `integerEq` y
instance Eq Float where
    x == y = x `floatEq` y
```

- (assumes `integerEq` and `floatEq` functions exist)
Defining class instances (2)

- Do same for user-defined classes:

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

instance (Eq a) => Eq (Tree a) where
    Leaf x == Leaf y = x == y
    (Branch l1 r1) == (Branch l2 r2) =
                                 (l1==l2) && (r1==r2)
    _ == _ = False
```

- Note context: `(Eq a) => ...`
Other useful classes

- Comparable types:
  \[\text{Ord} \rightarrow \text{<} \; \text{<=} \; \text{>} \; \text{>=}\]

- Printable types:
  \[\text{Show} \rightarrow \text{show where}\]
  \[
  \text{show} :: \text{a} \rightarrow \text{String}
  \]

- Numeric types:
  \[\text{Num} \rightarrow + - * \text{ negate abs etc.}\]
Using type classes

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    quicksort lt ++ [x] ++ quicksort ge
    where
        lt = [y | y <- xs, y < x]
        ge = [y | y <- xs, y >= x]

- Any type not defining < or >= can't be quicksorted using this definition
Sometimes instance definition is obvious:

```haskell
data Color = Red | Green | Blue
instance Show (Color) where
    show Red    = "Red"
    show Green  = "Green"
    show Blue   = "Blue"
```
Deriving type classes (2)

- Shorter:
  ```haskell
  data Color = Red | Green | Blue
  deriving Show
  ```

- Now `instance` definition not needed
- Often used for classes whose definition is trivial
- e.g. `Eq`, `Show`

```haskell
data Color = Red | Green | Blue
  deriving (Eq, Show)
```

- Only a few classes can be derived
Input / Output (I/O)

- Input/output is modeled in Haskell as "actions" or "computations"
- Represented by types of form: `IO a`
- A type `IO a` is a type which does some input and/or output and "returns" a value of type `a`
- Entire program is a value of type `IO ()`
  - where `()` is the unit (no value) type
  - This is the type of the `main` function
Simple I/O actions

- Take a string, print it, return nothing:
  \[
  \text{putStr} :: \text{String} \rightarrow \text{IO} ()
  \]

- Take a string, print it + newline, return nothing:
  \[
  \text{putStrLn} :: \text{String} \rightarrow \text{IO} ()
  \]

- Get a string ending in a newline and return it
  \[
  \text{getLine} :: \text{IO String}
  \]
Combining I/O actions (1)

- I/O would be unusable if couldn't combine I/O actions to make more complex actions

Two basic functions:

```haskell
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- `(>>=)` is called "bind"
  - (real types are more general than this)

- These are also the characteristic functions of the `Monad` type class
Combining I/O actions (2)

- `return x` converts a value into an action that returns that value
- `(>>=)` combines
  - an action returning type `a`
  - a function that takes a value of type `a` and returns an action returning type `b`
- ... to get an action returning type `b`
Combining I/O actions (3)

\[(\triangleright\triangleright\triangleright)\, :: \, \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b\]

- Consider:
  \[f_1 \triangleright\triangleright\triangleright \, \backslash x \rightarrow f_2 \, x \quad -- \text{or: } f_1 \triangleright\triangleright\triangleright \, f_2\]
  
- \(f_1\) has type \(\text{IO } a\)

- \(a\) value from \(\text{IO } a\) is "unpacked" into \(x\)

- \(x\) is passed to function of type \((a \rightarrow \text{IO } b)\)

- result: value of type \(\text{IO } b\)

- This is the only way to use the \(\text{IO}\) value!
Example (in ghci)

Prelude> return "hello!"
[nothing happens]
Prelude> return "hello!" >>= putStrLn
hello!

- Alternate notation:
  Prelude> do s <- return "hello!";
         putStrLn s

- Called "do notation"
May want to use several I/O actions in a function

```haskell
getTwoLines :: IO String
getTwoLines = getLine >>= \a ->
    getLine >>= \b ->
    return (a ++ b)
```

Yuck!

So common that special syntactic sugar exists to make it easier to use

Works for any monad (including \texttt{IO} monad)
do notation (2)

- Short form:

```haskell
getTwoLines :: IO String
getTwoLines = do a <- getLine
                b <- getLine
                return (a ++ b)
```

- Looks like imperative code
- Acts like imperative code
- but is purely functional!
Other IO operators (1)

- `(>>=)` operator sequences actions, passing result of one action to another action
- Sometimes need to sequence actions but don't care about the result
- `(>>)`) operator used in that case
- Definition:
  
  
  `(>>) :: IO a -> IO b -> IO b`
  
  `a >> b = a >>= _ -> b`
  
  (type is actually more general than this)
Other IO operators (2)

- **fail** function used when something goes wrong

\[
\text{fail :: String} \rightarrow \text{IO a}
\]

- fail can produce a result of any IO type
- Weak form of error handling
- Allows you to break out of a computation that cannot succeed
- **String** is the error message you give
More functions (1)

- Take a sequence of IO actions, do them one after the other, return list of all results
  \[
  \text{sequence} \:: [\text{IO}\ a] \rightarrow \text{IO}\ [a]
  \]

- Take a sequence of IO actions, do them one after the other, return nothing
  \[
  \text{sequence}_\_ \:: [\text{IO}\ a] \rightarrow \text{IO}\ ()
  \]

- Map a function generating IO actions over a list
  \[
  \text{mapM} \:: (a \rightarrow \text{IO}\ b) \rightarrow [a] \rightarrow \text{IO}\ [b]
  \]
  \[
  \text{mapM}\_\_ \:: (a \rightarrow \text{IO}\ b) \rightarrow [a] \rightarrow \text{IO}\ ()
  \]
More functions (2)

Prelude> mapM return [1, 2, 3]
Prelude> mapM return [1, 2, 3] >>= print
[1,2,3]
Prelude> mapM_ return [1, 2, 3] >>= print
()
Prelude> mapM_ putStrLn ["foo", "bar", "baz"]
foo
bar
baz
Reference

- A good (more advanced) tutorial on Haskell I/O:
  http://haskell.org/haskellwiki/IO_inside
- Explains in more detail how I/O actually is implemented and how it works
Next week

- Monads
  - see how this stuff really works
  - generalize to many other situations