Uniting our examples

In the lab you experimented and with an evaluator. Now let’s extend it to include variable bindings...

First we extend the expression language:

```haskell
data Expression = Val Float
  | Add Expression Expression
  | Multiply Expression Expression
  | Subtract Expression Expression
  | Divide Expression Expression
  | Var Ident
  | Def Ident Expression Expression

deriving Show

type Ident = String
```

How to model a lookup dictionary?

A dictionary type mapping values to numbers (type introduces a simple type synonymn instead of an algebraic data type):

```haskell
type Dictionary a b = [(a, b)]

find :: Eq a => Dictionary a b -> a -> Maybe b
find [] _ = Nothing
find ( (s,v) : ds ) name | name == s = Just v
                         | otherwise = find ds name

define :: Dictionary a b -> a -> b -> Dictionary a b
define d s v = (s,v):d
```

```haskell
> define [] "speed" 20.0
[ ("speed", 20.0) ]
> find (define [] "speed" 20.0) "speed"
Just 20.0
> find empty "speed"
Nothing
```
Extending the evaluator

```haskell
eval :: Expression -> Dictionary Ident Float -> Float
eval (Val x) _ = x
eval (Add x y) d = eval x d + eval y d
eval (Multiply x y) d = eval x d * eval y d
eval (Subtract x y) d = eval x d - eval y d
eval (Divide x y) d = eval x d / eval y d

eval (Var i) d = fromJust (find d i)
eval (Def i e1 e2) d = eval e2 d'
  where d' = define i (eval e1 d) d

fromJust (Just a) = a
```

Higher Order Functions

What is the difference between these two functions?

```haskell
add x y = x+y
add2 (x,y) = x+y
```

We can see it in the types; add is *curried*, taking one argument at a time.

```haskell
add :: Integer -> Integer -> Integer
add2 :: (Integer,Integer) -> Integer
```

Any type `a->a->a` can also be written `a->(a->a)`. The function type arrow associates to the right.

Is there any significance to this?

Just as an exercise...

```haskell
toString :: Expression -> String
toString (Val x) = show x
toString (Add x y) = "("++(toString x)++"++toString y++)"
toString (Multiply x y) = "("++(toString x)++"*"++toString y++)"
toString (Subtract x y) = "("++(toString x)++"-"++toString y++)"
toString (Divide x y) = "("++(toString x)++"/"++toString y++)"
toString (Var x) = x
toString (Let v x y) = "let "++v++" = "++(toString x) ++" in"++toString y

> toString testExp
let a = ( 2.0 * 3.0 ) in let b = (8.0-1.0) in ((a*b)-1.0)
```

In Haskell functions are *first class citizens*. In other words, they occupy the same status in the language as values: you can pass them as arguments, make them part of data structures, compute them as the result of functions...

```haskell
add3 :: (Integer -> (Integer -> Integer))
add3 = add

> add3 1 2
3
```

```haskell
(add3) 1 2
===> add 1 2
===> 1 + 2
```

Notice that there are no parameters in the definition of `add3`. 
A function with multiple arguments can be viewed as a function of one argument, which computes a new function.

\[
\text{add 3 4} \\
\quad \Rightarrow (\text{add 3}) \ 4 \\
\quad \Rightarrow ((+) \ 3) \ 4
\]

The first place you might encounter this is the notion of partial application:

```haskell
addone :: Integer \rightarrow Integer
increment x = add 1
```

If the type of `add` is `Integer\rightarrow Integer\rightarrow Integer`, and the type of `add 1 2` is `Integer`, then the type of `add 1` is?

It is `Integer \rightarrow Integer`

Some more examples of partial application:

```haskell
second :: [a] \rightarrow a 
second = head . tail

> second [1,2,3] 
2
```

An infix operator can be partially applied by taking a section:

```haskell
increment = (1 +) -- or (+ 1) 
addnewline = (++"\n")
```

```haskell
double :: Integer \rightarrow Integer 
double = (*2)

> [ double x | x <- [1..10] ] 
[2,4,6,8,10,12,14,16,18,20]
```

Functions can be taken as parameters as well.

```haskell
twice :: (a \rightarrow a) \rightarrow a \rightarrow a 
twice f x = f (f x)
```

```haskell
addtwo = twice increment
```

This could also be written:

```haskell
twice f = f . f
```

In fact, we can define composition using this technique:

```haskell
compose :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b 
compose f g x = f (g x)
```

```haskell
twice f = f `compose` f
```

Super-bonus hack. Haskell permits the definition of infix functions:

```haskell
(f ! g) x = f (g x) 
twice f = f!f
```