An running example: Expressions

We are going to write functions that manipulate expressions in a variety of ways

```
data Expr
 = Val Double
 | Add Expr Expr
 | Mul Expr Expr
 | Sub Expr Expr
 | Dvd Expr Expr
 deriving Show -- makes it possible to see values (DEMO!)
```

$(10 + 5) * 90$ becomes

```
Mul (Add (Val 10) (Val 5)) (Val 90)
```

$10 + (5 * 90)$ becomes

```
Add (Val 10) (Mul (Val 5) (Val 90))
```

An evaluator

We can write a function to calculate the result of these expressions:

```
eval :: Expr -> Double
eval (Val x) = x
eval (Add x y) = eval x + eval y
eval (Mul x y) = ... -- similar to above
-- similarly for Sub and Dvd
```

```
> eval (Add (Val 10) (Mul (Val 5) (Val 90)))
460.0
```

A simplifier

We can write a function to simplify expressions:

```
simp :: Expr -> Expr
simp (Val x) = (Val x)
-- we can use pattern matching in let-expressions!
simp (Add e1 e2) = let (Val x) = simp e1
                       (Val y) = simp e2
                   in Val (x+y)
simp (Dvd x y) = ... -- similar to above
-- similar for Mul and Sub
```

```
> simp (Add (Val 10) (Mul (Val 5) (Val 90)))
Val 460.0
```

Adding Variables to Expressions

Now let's extend our expression datatype to include variables. First we extend the expression language:

```
data Expr = Val Double
          | Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
          | Dvd Expr Expr
          | Var Id
     deriving Show

type Id = String
```

Simplification again

```
simp (Add e1 e2)
 = let e1' = simp e1
       e2' = simp e2
   in case (e1',e2') of
     (Val 0.0,e)  ->  e
     (e,Val 0.0)  ->  e
     _            ->  Add e1' e2'
simp (Mul e1 e2) = ... -- similar (and Sub,Dvd)
simp e = e -- catches all remaining cases (Val, Var)
```

Evaluating Exprs with Variables

Remember our extended expression language:

```
type Id = String
data Expr = Val Double
          | Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
          | Dvd Expr Expr
          | Var Id
    deriving Show
```

We can't fully evaluate these without some way of knowing what values any of the variables (`Var`) have.
We can imagine that `eval` should have a signature like this:

```
eval :: Dict Id Double -> Expr -> Double
```

It now has a new (first) argument, a `Dict` that associates `Double` (datum values) with `Id` (key values).

How to model a lookup Dictionary?

A *Dictionary* maps keys to datum values

- An obvious approach is to use a list of key/datum pairs:

  ```
  type Dict k d = [ (k, d) ]
  ```

- Defining a link between key and datum is simply cons-ing such a pair onto the start of the list.

  ```
  define :: Dict k d -> k -> d -> Dict k d
  define d s v = (s,v):d
  ```

- Lookup simply searches along the list:

  ```
  find :: Eq k => Dict k d -> k -> Maybe d
  find [] _ = Nothing
  find ( (s,v) : ds ) name | name == s = Just v
                           | otherwise = find ds name
  ```

  We need to handle the case when the key is not present. This is the role of the `Maybe` type.

Maybe (Prelude)

```
data Maybe a
  =  Nothing
  |  Just a
  deriving (Eq, Ord, Read, Show)

maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x
```

Maybe (Data.Maybe)

```
import Data.Maybe -- need to explicitly import this

isJust                  :: Maybe a -> Bool
isJust (Just a)         =  True
isJust Nothing          =  False

isNothing               :: Maybe a -> Bool
isNothing               =  not . isJust

fromJust                :: Maybe a -> a
fromJust (Just a)       =  a
fromJust Nothing        =  error "Maybe.fromJust: Nothing"

fromMaybe               :: a -> Maybe a -> a
fromMaybe d Nothing     =  d
fromMaybe d (Just a)    =  a
```

Dict at work

Building a simple Dict that maps key "speed" to datum 20.0.

```
> define [] "speed" 20.0
[ ("speed", 20.0) ]
> find (define [] "speed" 20.0) "speed"
Just 20.0
> find [] "speed"
Nothing
```

Extending the evaluator

```
eval :: Dict Id Double -> Expr -> Double
eval _ (Val x) = x
eval d (Var i) = fromJust (find d i)
eval d (Add e1 e2) = eval d e1 + eval d e2
-- similar for Add, Mul, Dvd
fromJust (Just a) = a
```

We are back to simpler code (no need for `case ... of ...`)

Expr Pretty-Printing

We can write something to print the expression in a more "friendly" infix style:

```
iprint :: Expr -> String
iprint (Val x) = show x
iprint (Var x) = x
iprint (Dvd x y) = "("++(iprint x)++"/"++iprint y++")"
-- similar for Add, Mul, Sub
```

There are many ways in which this could be made much prettier.

Extending Expr Further

We can augment the expression type to allow expressions with local variable declarations:

```
data Expr = Val Double
          | Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
          | Dvd Expr Expr
          | Var Id
          | Def Id Expr Expr
```

The intended meaning of `Def x e1 e2` is: `x` is in scope in `e2`, but not in `e1`; compute value of `e1`, and assign value to `x`; then evaluate `e2` as overall result.

Def example

A sample expression in this form could look like this:

```
testExpr = Def "a" (Mul (Val 2) (Val 3)) (
             Def "b" (Sub (Val 8) (Val 1)) (
               Sub (Mul (Var "a") (Var "b"))
                        (Val 1)))
```

A nice way to print this out might be:

```
let a = 2 * 3
 in let b = 8 - 1
     in (a * b ) - 1
```

Dict-based Evaluation (I)

For the non-identifier parts of expressions we simply pass the Dict around, but otherwise ignore it.

```
eval :: Dict Id Double -> Expr -> Double
eval d (Val v) = v
eval d (Add e1 e2)  =  (eval d e1) + (eval d e2)
-- others similarly
```

Dict-based Evaluation (II)

Given a variable, we simply look it up:

```
eval d (Var n) = fromJust (find d n)

fromJust (Just x) = x
```

Dict-based Evaluation (III)

Given a `Def`, we

1. evaluate the first expression in the given Dict;
2. add a binding from the defined variable to the resulting value, and then
3. evaluate the second expression with the updated Dict:

```
eval d (Def x e1 e2) = eval (define d x (eval d e1 ) ) e2
```