Contents

Adding Variables to Expressions **1**

Simplification Again 1

Evaluating Exprs with Variables 2

How to Model a Lookup Dictionary **2**

Maybe (Prelude) 3

Dictionary at Work 3

Extendin the Evaluator **3**

Expr Pretty-Printing **4**

Adding Variables to Expressions

Now lets extend our expression datatype to include variables. First we extend the expression language:

```
data Expr = Val Float
          | 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)
```

```
  = ...

simp e = e -- catches all remaining cases (Val, Var)
```

Evaluating Exprs with Variables

We can't fully evaluate these without some way of knowing wat values and of the variables (`Var`) have.

We can imagine that `eval` should have a signature like this:

```
eval :: Dictionary Id Float -> Expr -> Float
```

It now has a new (first) argument, a `Dictionary` that associates `Float` (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 Dictionary k d = [ (k, d) ]
```

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

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

- Lookup simply searches along the list

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

Maybe (Prelude)

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

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

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
```

Dictionary at Work

Building a simple dictionary 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
```

Extendin the Evaluator

```
eval :: Dictionary Id Float -> Expr -> Float
eval _ (Val x) = x
eval d (Var i) = fromJust (find d i)
eval d (Add x y) = eval d x + eval d y
```

```
eval d (Multiply x y) = eval d x * eval d y
eval d (Subtract x y) = eval d x - eval d y
eval d (Divide x y) = eval d x / eval d y

fromJust (Just a) = a
```

Expr Pretty-Printing

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

```
print :: Expr -> String
print (Val x) = show x
print (Var x) = x
print (Add x y) = "("++(print x)++"+"++print y++")"
print (Multiply x y) = "("++(print x)++"*"++print y++")"
print (Subtract x y) = "("++(print x)++"-"++print y++")"
print (Divide x y) = "("++(print x)++"/"++print y++")"
```

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