Extending Expr Further

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

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

Dictionary-based Evaluation (I)

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

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

Dictionary-based Evaluation (II)

Given a variable, we simply look it up:

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

fromJust (Just x) = x
```

Dictionary-based Evaluation (III)

Given a `Def`, we

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

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

`Expr`: taking stock

- We have introduced a datatype `Expr` for expressions
- We have a lookup table that associates datum values with keys
- We can simplify (`simp`) the expressions (to some degree)
- We can evaluate (`eval`) the expressions (to some degree)
- We can print (`iprint`) out the expressions in a (reasonably) nice manner

`Expr`: Issues (1)

- We extended this before — perhaps we might want to do this again?
- What happens if a variable is not in the dictionary?
- What happens if we divide by zero?
- A lot of very similar looking code ("boilerplate") !

`Expr`: Issues (2)

- We need proper error handling
- We need to reduce the amount of boilerplate
  - This is important if we hope to extend the expression type in any way.
- Three mechanisms are available to help:
  - The type system — we can define types that help with error handling
  - Abstraction — we can capture common boilerplate patterns as functions.
  - Classes — we can capture common boilerplate control patterns as classes.

Using Maybe to handle errors

Remember the Maybe type:

```
data Maybe t = Nothing | Just t
```

We can revise our `eval` function to return a value of type `Maybe Float`, using `Nothing` to signal an error:

```
eval :: Dict -> Expr -> Maybe Float
eval _ (Val x) = Just x
eval d (Var i) = find d i -- returns a Maybe type anyway!
```

Now lets look at some other cases.

Evaluating Mul using Maybe

```
eval d (Mul x y) = Just ( (eval d x) * (eval d y) )
```

!!! Won't work — `eval` no longer returns a `Float`, but a `Maybe Float`!
We have to pattern-match against the recursive `eval` outcomes to see what to do next:

```
eval d (Mul x y)
 = case (eval d x,eval d y) of
     (Just m,Just n)  ->  Just (m*n)
     _                ->  Nothing
```

Evaluating Dvd

Here we can now properly handle division by zero!

```
eval d (Dvd x y)
 = case (eval d x, eval d y) of
     (Just m,Just n)
        ->  if n==0.0 then Nothing else Just (m/n)
     _  ->  Nothing
```

More boilerplate !

Evaluating Def

```
eval d (Def x e1 e2)
 = case eval d e1 of
     Nothing  ->  Nothing
     Just v1  ->  eval (define d x v1) e2
```

More boilerplate !
Error handling seems expensive!
This is why most languages support exceptions.

Closing Observations

- We can add explicit error handling using `Maybe` (or `Either`).
- Exceptions are available, but only in an `IO` context[1]
- However we can still do a lot better, with higher-order abstractions and classes.

[1] ??? - we'll get to this...

Turning Expressions into Functions

Consider the following expression:

```
a * b + 2 - c
```

There are at least four ways we can turn this into a function of one numeric argument

```
f a where  f x = x * b + 2 - c
f b where  f x = a * x + 2 - c
f c where  f x = a * b + 2 - x
f 2 where  f x = a * b + x - c
```

This process of converting expressions into functions is called *abstraction*.

Abstracting Functions

Consider the following function definitions:

```
f a b = sqr a + sqrt b
g x y = sqrt x * sqr y
h p q = log p - abs q
```

They all have the same general form:

```
fname arg1 arg2 = someF arg1 `someOp` anotherF arg2
```

We can abstract this by adding parameters to represent the "bits" of the general form:

```
absF someF anotherF someOp arg1 arg2
                   = someF arg1 `someOp` anotherF arg2
```

Now `f`, `g` and `h` can be defined using absF

```
f a b = absF sqr sqrt (+) a b
g x y = absF sqrt sqr (*) x y
h = absF log abs (-) -- we can use partial application !
```

The "shape" of `eval` using `Maybe`

A typical binary operation case in `eval` looks like

```
eval d (Sub x y)
 = let  r = eval d x ; s = eval d y
   in case (r,s) of
     (Just m,Just n)  ->  Just (m-n)
     _                ->  Nothing
```

We just need to process the two sub-expressions, with a binary operator for the result, so we come up with:

```
evalOp d op x y
 = let  r = eval d x ; s = eval d y
   in case (r,s) of
     (Just m,Just n)  ->  Just (m `op` n)
     _                ->  Nothing
```

This works for `Add`, `Mul` and `Sub`, but not `Dvd` (why not?)

Revised `eval`

The following cases get simplified:

```
eval d (Add x y) = evalOp d (+) x y
eval d (Mul x y) = evalOp d (*) x y
eval d (Sub x y) = evalOp d (-) x y
```

We can't do `Dvd`,
because it will need to return `Nothing` if `y` evaluates to `0`.
At least those operators that cannot raise errors are now easy to code.

Simplifying `simp` (I)

We have code as follows (let's use `Sub` again):

```
simp (Sub e1 e2)
 = let e1' = simp e1
       e2' = simp e2
   in case (e1',e2') of
        (e,Val 0.0)  ->  e
        _            ->  Sub e1' e2'
```

We can't abstract to the same degree as for `eval`, because there is a lot of irregularity in the simplifications.

Simplifying `simp` (II)

We can at least isolate the simplifications out:

```
simpOp (opsimp e1 e2)
 = let e1' = simp e1
       e2' = simp e2
   in opsimp e1' e2'
```

`simp` itself is simpler:

```
simp (Add e1 e2) = simpOp addSimp e1 e2
simp (Mul e1 e2) = simpOp mulSimp e1 e2
simp (Sub e1 e2) = simpOp subSimp e1 e2
simp (Dvd e1 e2) = simpOp dvdSimp e1 e2
```

Simplifying `simp` (III)

Each operator simplifier has its own case-analysis, e.g.:

```
mulSimp (Val 1.0) e  =  e
mulSimp e (Val 1.0)  =  e
mulSimp e1 e2        =  Mul e1 e2
```

Still boilerpate, but perhaps it is clearer this way (no explicit use of `case`).

Some operators are "nice"

- Some operators have nice properties, like having unit values e.g., $0 + a = a = a + 0$ and $1 * a = a = a * 1$
- We can code a simplifier for these as follows:

```
uopSimp cons u (Val v) e | v == u   =  e
uopSimp cons u e (Val v) | v == u   =  e
uopSimp cons u e1 e2                =  cons e1 e2
```

  What is `cons` here?
- Usage:

```
simp (Add e1 e2) = uopSimp Add 0.0 e1 e2
simp (Mul e1 e2) = uopSimp Mul 1.0 e1 e2
```

Data Constructors are Functions (I)

The data constructors of `Expr`, are in fact functions, whose types are as follows:

```
Val :: Double -> Expr
Var :: Id -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
Sub :: Expr -> Expr -> Expr
Dvd :: Expr -> Expr -> Expr
Def :: Id -> Expr -> Expr -> Expr
```

So, `cons` on the previous slide needs to have type `Expr -> Expr -> Expr`, which is why `Add` and `Mul` are suitable arguments to pass into `uopSimp`.

Data Constructors are Functions (II)

- given declaration

```
data MyType  = ... |  MyCons T1 T2 ... Tn | ...
```

  then data constructor `MyCons` is a function of type

```
MyCons :: T1 -> T2 -> ... -> Tn -> MyType
```

- Partial applications of `MyCons` are also valid

```
(MyCons x1 x2) :: T3 -> ... -> Tn -> MyType
```

- Data constructors are the only functions that can occur in patterns.

Abstraction: Summary

- Abstraction is the process of turning expressions into functions
- If done intelligently, it greatly increases code re-use and reduces boilerplate.
- We saw it applied to `eval` and `simp`.
- A lot of the higher-order functions in the Prelude are examples of abstraction of common programming shapes encountered in functional programs (e.g., `map` and `folds`).