Monads revisited

- Monads combine pure functional computation with side effects.
- Sequencing and order of computation are important for the side effects.
- With monads it is much more clear which parts contain side effects/are dependent on order and which are purely functional.
- For I/O, graphics etc it is “obvious” that we need sequential computation.
- It is less obvious how sequentiality is described formally, especially when combined with a lazy language.
- Using the formal description one can use monads to describe more types of computation than just sequential.
- A monad contains (at least) two operators
  - \textit{bind} or \texttt{>>=} used for sequential operation and (optional) binding of name
  - \textit{return} for extracting values from the monad
Monad Axioms

- Return acts a “neutral” (or identity; \(x + 0 = 0 + x = x\)) element of \(\text{bind}\)

\[
\begin{align*}
(\text{return } x) & \gg= f \equiv f \; x \\
m & \gg= \text{return} \equiv m
\end{align*}
\]

- Binding two functions in succession is the same as binding one function that can be determined from them

  - This is a composition property

\[
\begin{align*}
(m \gg= f) & \gg= g \equiv m \gg= (\lambda x \rightarrow (f \; x \gg= g))
\end{align*}
\]

- Optional zero value (\(x \; 0 = 0 \; x = 0\))

\[
\begin{align*}
mzero & \gg= f \equiv mzero \\
m & \gg= (\lambda x \rightarrow mzero) \equiv mzero
\end{align*}
\]
Maybe Monad

- Computations might fail (in general)
- Can we express and propagate failed computations in an easy way?
  - Compare to exceptions; skip rest of computation if one fails

```haskell
data Maybee a = The a | Nil deriving Show

instance Monad Maybee where
  (The x) >>= f = f x
  Nil >>= _ = Nil
  return x = The x

madd x y =
  do
    x1 <- x
    y1 <- y
    return (x1 + y1)

*Main> :type madd
madd :: (Monad m, Num b) => m b -> m b -> m b
*Main> madd (The 2) (The 3)
The 5
*Main> madd (The 2) Nil
Nil
```

Tisdag 11 december 2012
Maybe Monad

- Be aware the sequencing/bind “unpacks” values inside monads
- The following two definitions of monadic add are not equivalent
  - observe the very different types!

```haskell
madd x y =
  do
    x1 <- x
    y1 <- y
    return (x1 + y1)

madd2 x y =
  do return (x + y)

*Main> :type madd
madd :: (Monad m, Num b) => m b -> m b -> m b
*Main> :type madd2
madd2 :: (Monad m, Num a) => a -> a -> m a
```
List Monad

- Use a monad for a set of possible results
- Note the it makes sense to read “<-” as “belongs to” or “in”
- Very much like list comprehension

instance Monad [] where
   -- single possible result
   return a = [a]
   -- apply to each element and flatten results
   xs >>= f = concat (map f xs)
   -- failure is no results
   fail s = []

Prelude> do x<-[1,2,3]; y<-[4,5]; return (x*y)
[4,5,8,10,12,15]
Prelude> do x<-[1,2,3]; y<-[4,5]; if x > 1 then return (x*y) else fail ""
[8,10,12,15]
Other Monads

- State monad
  - our previous cell example was a state monad
- Environment Monad
  - compute with values from a (non local) context
- Writer monad
  - perform computations while writing “output” on the side
- See Wikipedia page for more details.