Monads

Monads part 1

Computations where order matters

A monad orders actions

- An action is any computation that has a natural notion of order. I.e. one thing happens before another.
  - IO is the action of altering the real world.
  - There are many other styles of computation that have a natural notion of order
- A Monad is Haskell's way of specifying which actions come before others.
- The "do" operator provides this control over the order in which computations occur

```
do { var <- location x   -- the first action
   ; write var (b+1)     -- the next action
   }
```

Observations

- Actions are first class.
  - They can be abstracted (parameters of functions)
  - Stored in data structures. -- It is possible to have a list of actions, etc.
- Actions can be composed.
  - They can be built out of smaller actions by glueing them together with `do` and `return`
  - They are sequenced with `do` much like one uses semi-colon in languages like Pascal and C.
- Actions can be performed (run).
  - separation of construction from performance is key to their versatility.
  - IO actions are "run" as the "main" function, or interactively in GHCI
- Actions of type: `Action()` are like statements in imperative languages.
  - They are used only for their side effects.

Do syntactic sugar

```
do { a; b }              = do { _ <- a; b }


do { x <- a
   ; do { y <- b
        ; do { z <- c
             ; d }}}  = do {x <- a; y <- b;
                              z <- c; d }
= do x <- a
     y <- b
     z <- c
     d                   -- uses indentation
                         -- rather than { ; }
```

Do: syntax, types, and order

Int

IO Int

```
do { x <- f
   ; y <- g 7
   ; putChar y
   ; return (x + 4)
   }
```

actions without
`v <- ...` must have
type `(M a)` for some
monad M

last action must
must have type `M a`
which is the type of `do`
cannot have `v <- ...`

semi colons separate actions, I think it is good style to line `;` up with
opening `{` and closing `}`

Monads have Axioms

- Order matters (and is maintained by `Do`)

  - do { x <- do { y <- b; c }
  - ; d } =
  - do { y <- b; x <- c; d }

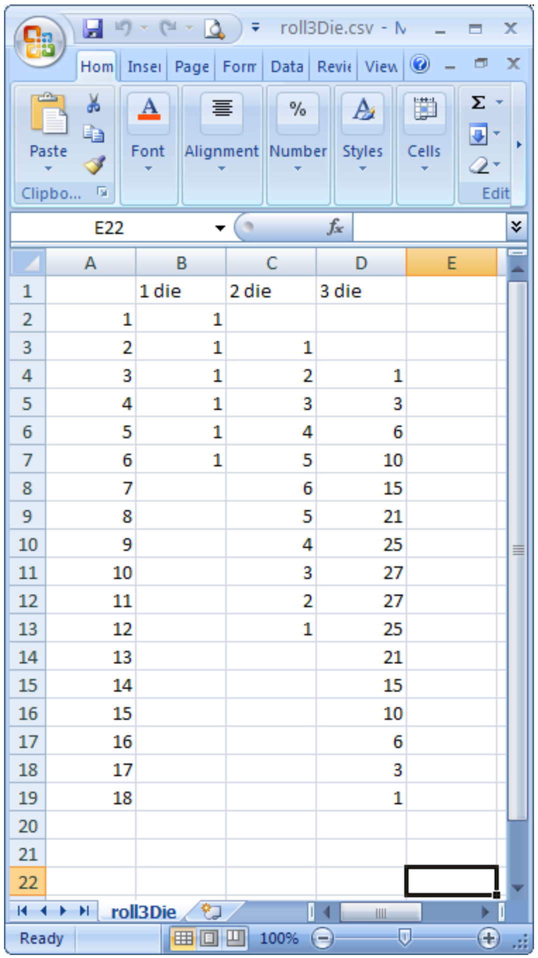
- `Return` introduces no effects

  - do { x <- Return a; e } = e[a/x]
  - do { x <- e; Return x } = e

Sample Monads

- `data Id x = Id x`
- `data Exception x = Ok x | Fail`
- `data Env e x = Env (e -> x)`
- `data Store s x = Store (s -> (x,s))`
- `data Mult x = Mult [x]`
- `data Output x = Output (x,String)`

Sample Problem
Importing Excel tables into Haskell

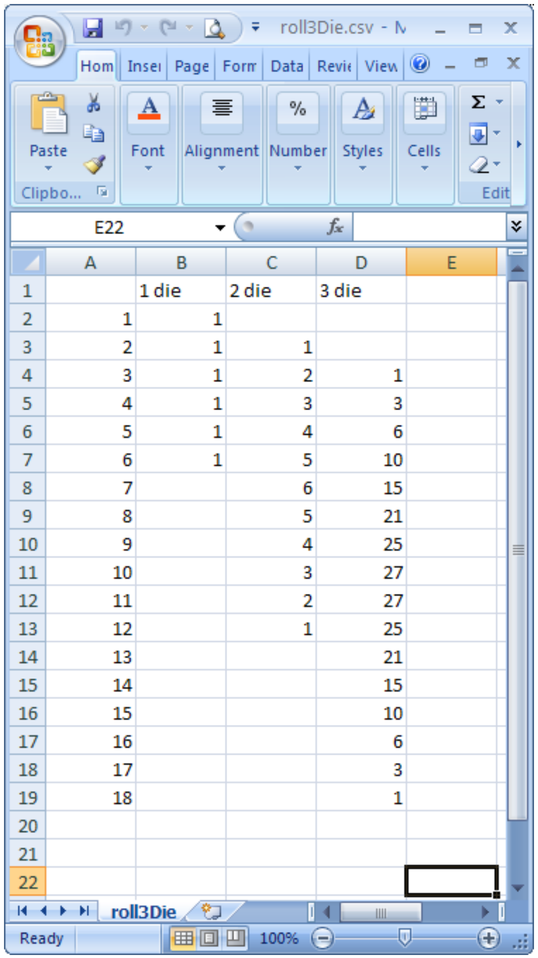


	A	B	C	D	E
1		1 die	2 die	3 die	
2	1	1			
3	2	1	1		
4	3	1	2	1	
5	4	1	3	3	
6	5	1	4	6	
7	6	1	5	10	
8	7		6	15	
9	8		5	21	
10	9		4	25	
11	10		3	27	
12	11		2	27	
13	12		1	25	
14	13			21	
15	14			15	
16	15			10	
17	16			6	
18	17			3	
19	18			1	
20					
21					
22					

```
(["1 die","2 die","3 die"]
,[("1",[Just 1,Nothing,Nothing])
 ,("2",[Just 1,Just 1,Nothing])
 ,("3",[Just 1,Just 2,Just 1])
 ,("4",[Just 1,Just 3,Just 3])
 ,("5",[Just 1,Just 4,Just 6])
 ,("6",[Just 1,Just 5,Just 10])
 ,("7",[Nothing,Just 6,Just 15])
 ,("8",[Nothing,Just 5,Just 21])
 ,("9",[Nothing,Just 4,Just 25])
 ,("10",[Nothing,Just 3,Just 27])
 ,("11",[Nothing,Just 2,Just 27])
 ,("12",[Nothing,Just 1,Just 25])
 ,("13",[Nothing,Nothing,Just 21])
 ,("14",[Nothing,Nothing,Just 15])
 ,("15",[Nothing,Nothing,Just 10])
 ,("16",[Nothing,Nothing,Just 6])
 ,("17",[Nothing,Nothing,Just 3])
 ,("18",[Nothing,Nothing,Just 1])]
)
```

Strategy

- Write Excel table into a comma separated values file.
- Use the CSV library to import the comma separated values file into Haskell as a [[String]]
- Process each sublist as a single line of the Excel table
- Interpret each string in the sublist as the correct form of data. E.g. an Int, or Bool, or list element, etc
- Note that order matters. The first element might be an Int, but the second might be a Bool.



	A	B	C	D	E
1		1 die	2 die	3 die	
2	1	1			
3	2	1	1		
4	3	1	2	1	
5	4	1	3	3	
6	5	1	4	6	
7	6	1	5	10	
8	7		6	15	
9	8		5	21	
10	9		4	25	
11	10		3	27	
12	11		2	27	
13	12		1	25	
14	13			21	
15	14			15	
16	15			10	
17	16			6	
18	17			3	
19	18			1	
20					
21					
22					


```
[["","1 die„
 ,"2 die","3 die"],
 ["1","1","",""],
 ["2","1","1",""],
 ["3","1","2","1"],
 ["4","1","3","3"],
 ["5","1","4","6"],
 ["6","1","5","10"],
 ["7","","6","15"],
 ["8","","5","21"],
 ["9","","4","25"],
 ["10","","3","27"],
 ["11","","2","27"],
 ["12","","1","25"],
 ["13","","","21"],
 ["14","","","15"],
 ["15","","","10"],
 ["16","","","6"],
 ["17","","","3"],
 ["18","","","1"],[""]]
```

```
(["1 die","2 die","3 die"]
,[("1",[Just 1,Nothing,Nothing])
 ,("2",[Just 1,Just 1,Nothing])
 ,("3",[Just 1,Just 2,Just 1])
 ,("4",[Just 1,Just 3,Just 3])
 ,("5",[Just 1,Just 4,Just 6])
 ,("6",[Just 1,Just 5,Just 10])
 ,("7",[Nothing,Just 6,Just 15])
 ,("8",[Nothing,Just 5,Just 21])
 ,("9",[Nothing,Just 4,Just 25])
 ,("10",[Nothing,Just 3,Just 27])
 ,("11",[Nothing,Just 2,Just 27])
 ,("12",[Nothing,Just 1,Just 25])
 ,("13",[Nothing,Nothing,Just 21])
 ,("14",[Nothing,Nothing,Just 15])
 ,("15",[Nothing,Nothing,Just 10])
 ,("16",[Nothing,Nothing,Just 6])
 ,("17",[Nothing,Nothing,Just 3])
 ,("18",[Nothing,Nothing,Just 1])]
)
```

Pattern

There is a pattern to the process (a simple form of parsing)

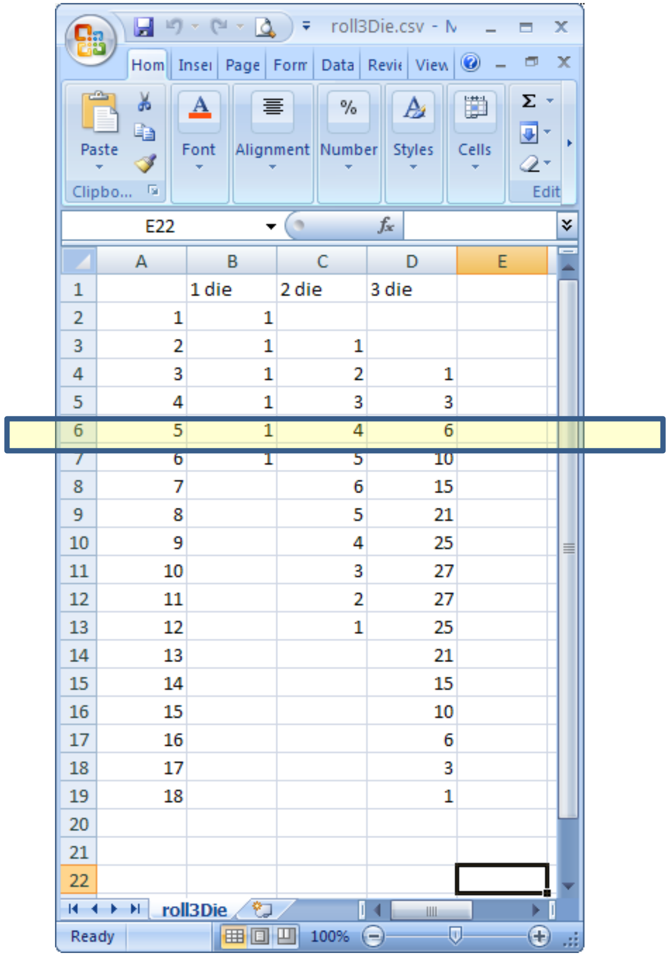
- Take a [String] as input
- Interpret 1 or more elements to produce data
- Return the data and the rest of the strings

```
f:: [String] -> (Result,[String])
```

- Repeat for the next piece of data

Interpretation is different depending upon the data we want to produce.

What's involved



	A	B	C	D	E
1		1 die	2 die	3 die	
2	1	1			
3	2	1	1		
4	3	1	2	1	
5	4	1	3	3	
6	5	1	4	6	
7	6	1	5	10	
8	7		6	15	
9	8		5	21	
10	9		4	25	
11	10		3	27	
12	11		2	27	
13	12		1	25	
14	13			21	
15	14			15	
16	15			10	
17	16			6	
18	17			3	
19	18			1	
20					
21					
22					

Lets observe what happens for the 6th line of the Excel table

1. Read a string
2. Read 3 values to get a [Int]

Note the order involved

Write some code

```
getString:: [String] -> (String,[String])
getString (s:ss) = (s,ss)
getString [] =
  error "No more strings to read a 'String' from"

getInt:: [String] -> (Int,[String])
getInt (s:ss) = (read s,ss)
getInt [] =
   error "No more strings to read an 'Int' from"
```

Interpret as a string. I.e. do nothing

Interpret as an Int. Use read

How can we get a list of Int?

```
getInts:: Int -> [String] -> ([Int],[String])
getInts 0 ss = ([],ss)
getInts n ss =
   case getInt ss of
     (x,ss2) -> case getInts (n-1) ss2 of
                    (xs,ss3) -> (x:xs,ss3)
```

Note that the order is enforced by data dependencies, and we use the case to implement it.

Now get line 6

```
getLine6:: [String] ->
           ((String,[Int]),[String])
getLine6 ss =
   case getString ss of
     (count,ss2) ->
            case getInts 3 ss2 of
              (rolls,ss3) -> ((count,rolls),ss3)
```

Note how the ordering is enforced again. We can do better than this

There are three patterns

- Threading of the list of strings in the function types
  - `getString:: [String] -> (String,[String])`

- Threading in the use of the list

```
(count,ss2) ->
        case getInts 3 ss2 of
```

- Use of the case to create data dependencies that enforce ordering

- This is a Monad

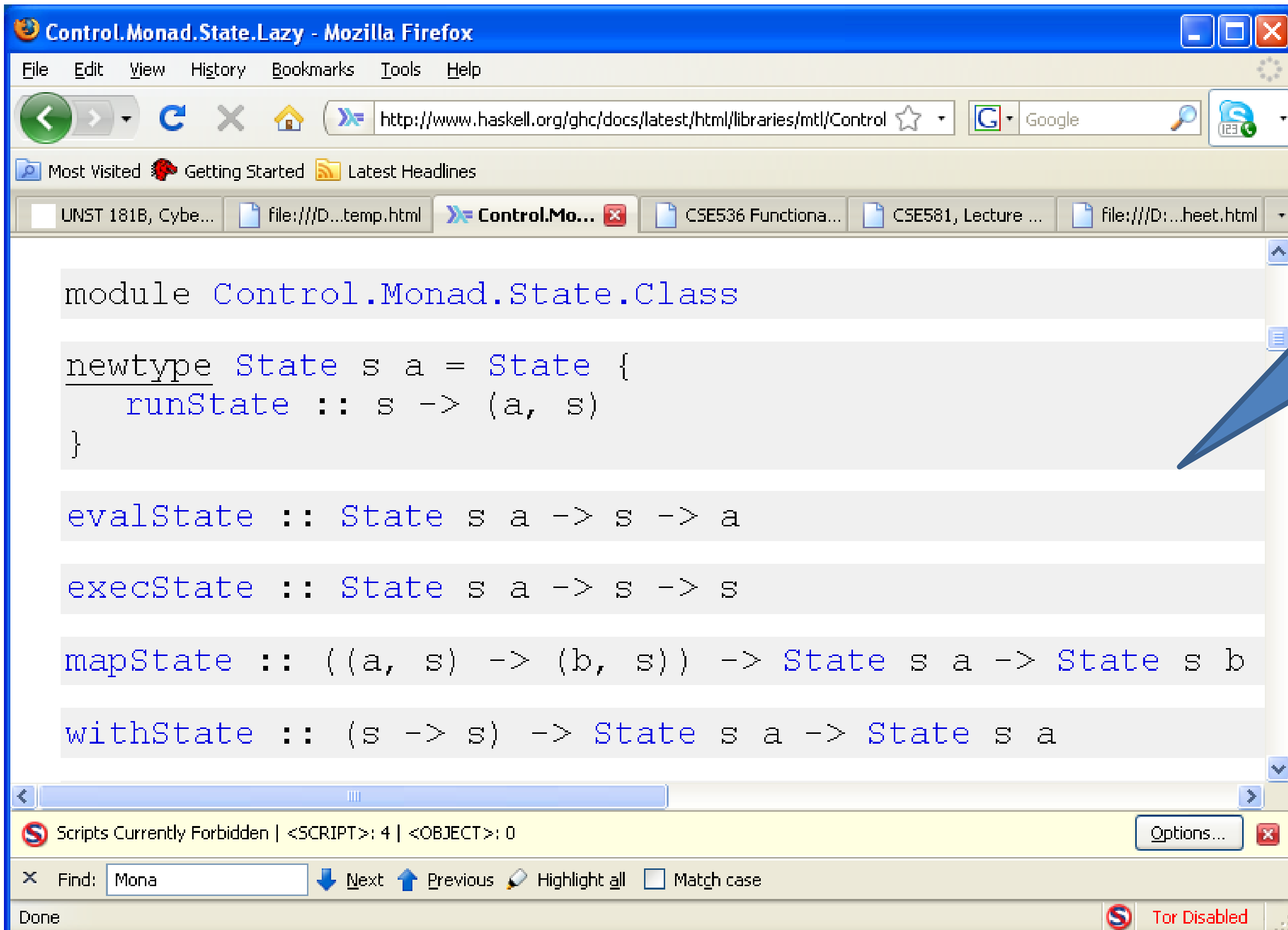
Parts of a Monad

- A Monad encapsulates some hidden structure
- A Monad captures a repeated pattern
- A Monad enforces ordering

The State Monad

- import Control.Monad.State
- Defines the type constructor (State t a)
  - It behaves like
  - data State s a = State (s -> (a,s))
- Use the do notation to compose and order actions (without performing them)
- Use the function evalState to perform actions

One of the standard libraries



Use these functions, plus do and return to solve problems

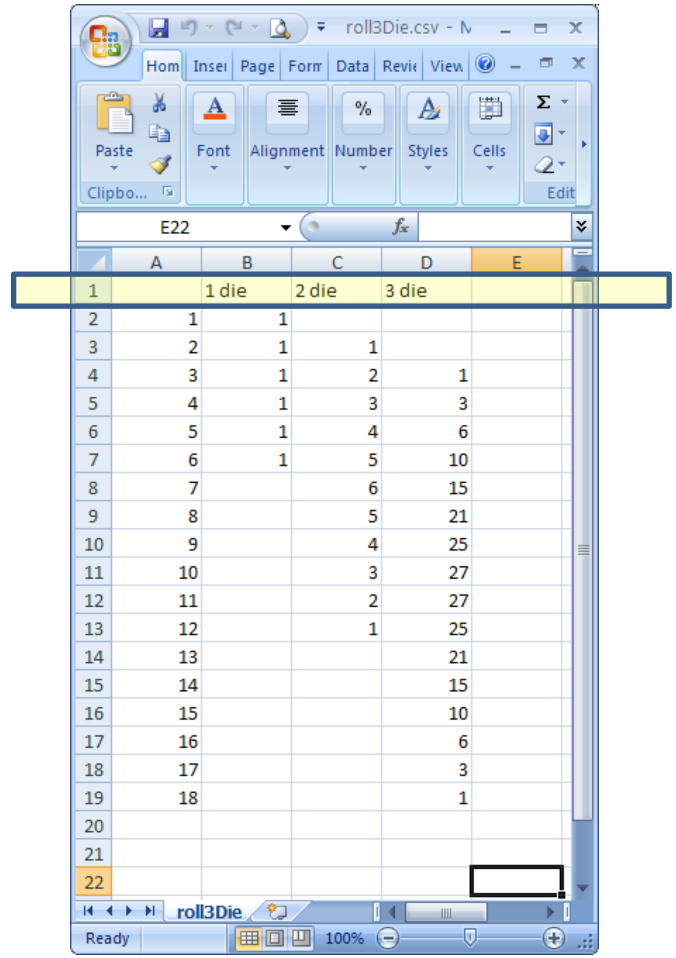
Part of the code

```
import Control.Monad.State
type Line a = State [String] a
type Reader a = State [[String]] a

getLine6b :: Line (String,[Maybe Int])
getLine6b =
   do { count <- string
      ; rolls <- list 3 (blank int)
      ; return(count,rolls)
      }
```

The first line is different

```
getLine1 :: Line [String]
getLine1 =
   do { skip
      ; list 3 string
      }
```



	A	B	C	D	E
1		1 die	2 die	3 die	
2	1	1			
3	2	1	1		
4	3	1	2	1	
5	4	1	3	3	
6	5	1	4	6	
7	6	1	5	10	
8	7		6	15	
9	8		5	21	
10	9		4	25	
11	10		3	27	
12	11		2	27	
13	12		1	25	
14	13			21	
15	14			15	
16	15			10	
17	16			6	
18	17			3	
19	18			1	
20					
21					
22					

What do int and string etc look like?

```
int:: Line Int
int = mapState f (return ())
  where f ((),s:ss) = (read s,ss)
        f ((),[]) =
          error "No more strings to read an 'Int' from"

list:: Int -> Line a -> Line [a]
list 0 r = return []
list n r = do { x <- r; xs <- list (n-1) r; return(x:xs)}

blank:: Line a -> Line(Maybe a)
blank (State g) = State f
  where f ("":xs) = (Nothing, xs)
        f  xs = case g xs of
                  (y,ys) -> (Just y,ys)
```

From lines to files

```
type Reader a = State [[String]] a

lineToReader:: Line a -> Reader a
lineToReader l1 = State g
  where g (line:lines) = (evalState l1 line, lines)

getN ::  Int -> Line a -> Reader [a]
getN 0 line = return []
getN n line =
  do { x <- lineToReader line
     ; xs <- getN (n-1) line
     ; return(x:xs)
     }
```

Reading a whole file

```
getFile :: Reader ([String],[(String,[Maybe Int])])
getFile =
  do { labels <- lineToReader getLine1
     ; pairs <- getN 18 getLine6b
     ; return(labels,pairs)
     }

importCSV :: Reader a -> String -> IO a
importCSV reader file =
      do { r <- parseCSVFromFile file; (f r)}
  where f (Left err) =
            error ("Error reading from file: "++
                   file++"\n"++show err)
        f (Right xs) = return(evalState reader xs)

test1 = importCSV getFile "roll3Die.csv"
```

Thoughts

- We use state monad at two different states
  - Lines where the state is [String]
  - Files where the state is [[String]]
- The use of the do notation makes the ordering explicit and is much cleaner than using nested case and threading (although this still happens)
- We have defined higher-order programs like list, blank, and lineToReader

Can we do better?

- Recall that monadic computations are first class.
- Can we capture patterns of use in our example to make things even simpler and more declarative.
- What patterns do we see again and again?

Patterns

```
getFile =
  do { labels <- lineToReader getLine1
     ; pairs <- getN 18 getLine6b
     ; return(labels,pairs)
     }

getN n line =
  do { x <- lineToReader line
     ; xs <- getN (n-1) line
     ; return(x:xs)
     }

getLine6b =
   do { count <- string
      ; rolls <- list 3 (blank int)
      ; return(count,rolls)
      }
```

Run two computations in order and then combine the two results

Generic Monad Operations

```
infixr 3 `x`
x :: Monad m => m b -> m c -> m(b,c)
r1 `x` r2 = do { a <- r1; b <- r2; return(a,b) }

many :: Monad m => Int -> m c -> m [c]
many n r = sequence (replicate n r)

sequence [] = return []
sequence (c:cs) =
   do { x <- c; xs <- sequence cs; return(x:xs)}
```

Declarative description

```
row:: (a -> b) -> Line a -> Reader b
row f line1 = lineToReader line2
  where line2 = do { x <- line1; return(f x) }


get3DieEx2::Reader ([[Char]],[([Char],[Maybe Int])])

get3DieEx2 =      (row snd (skip `x` list 3 string))
             `x`  (many 18 (row id cols2_18))
 where cols2_18 = (string `x` list 3 (blank int))
```

What if we get it wrong?

```
get3DieEx2 =       (row snd (skip `x` list 3 string))
             `x`   (many 18 (row id cols2_18))
 where cols2_18 = (string `x` list 4 (blank int))
```

```
D:\Programs\ghc-6.8.3\bin\ghci.exe
GHCi, version 6.8.3: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
[1 of 2] Compiling CSV              ( CSV.hs, interpreted )
[2 of 2] Compiling CSVLine          ( D:/work/sheard/Courses/CyberMil/web/Code/C
SVReader.hs, interpreted )
Ok, modules loaded: CSVLine, CSV.
*CSVLine>
*CSVLine> test2
Loading package parsec-2.1.0.1 ... linking ... done.
Loading package mtl-1.1.0.1 ... linking ... done.
(["1 die","2 die","3 die"],[("1",[Just 1,Nothing,Nothing,*** Exception: No more
strings to read an 'Int' from
*CSVLine>
```

Not very informative about where the error occurred

Thread more information in the state

```
type Line a = State (Int,Int,[String]) a
```

Line and column information

```
type Reader a = State (Int,[[String]]) a
```

Line information

- Where do we need to make changes?
- Remarkably, very few places

Only at the interface to the monad

```
report l c message =
  error ("\n at line: "++show l++
         ", column: "++show c++
         "\n   "++message)

bool:: Line Bool
bool = (State f) where
   f (l,c,"True" : ss) = (True,(l,c+1,ss))
   f (l,c,"False" : ss) = (False,(l,c+1,ss))
   f (l,c,x:xs) =
      report l c ("Non Bool in reader bool: "++x)
   f (l,c,[]) =
      report l c "No more strings to read a 'Bool' from"
```

```haskell
string:: Line String
string = State f
  where f (l,c,s:ss) = (s,(l,c+1,ss))
        f (l,c,[]) = report l c
                        "No more strings to read a 'String' from"

int:: Line Int
int = mapState f (return ())
  where f ((),(l,c,s:ss)) = (read s,(l,c+1,ss))
        f ((),(l,c,[])) = report l c
                            "No more strings to read an 'Int' from"

skip:: Line ()
skip = State f
  where f (l,c,s:ss) = ( (), (l,c+1,ss))
        f (l,c,[]) = report l c "No more strings to 'skip' over"

blank:: Line a -> Line(Maybe a)
blank (State g) = State f
  where f (l,c,"":xs) = (Nothing, (l,c+1,xs))
        f  xs = case g xs of
                  (y,ys) -> (Just y,ys)
```

Some thoughts

```
lineToReader:: Line a -> Reader a
lineToReader l1 = mapState f (return ())
  where f ((),(l,line:lines)) =
            (evalState l1 (l,1,line),(l+1,lines))
```

- Changes occur only where they matter
- Other functions use the same monadic interface
- The “plumbing” is handled automatically, even in the generic monad functions like `x` and `many`

We can see where the error occurred