"The most important thing in a programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language." --- Professor Donald Knuth

Side-Effecting Functions

Assume that the function `getChar` were defined to read a single char, `getchar :: Char` and we defined the following function:

```
get2chars = [ getchar, getchar]
```

There are two not-so-obvious issues here:

- Haskell treats functions as pure and assumes no side effects, so `get2chars` could evaluate `getChar` only once and return `[a,a]`
- If in fact two function calls are made that read characters from the input, you could get `[a,b]` or `[b,a]` because order of evaluation is not guaranteed
Forcing Order of Evaluation

- We can fix the partial evaluation problem by forcing the evaluation of getchar by providing a different argument to each invocation:

  ```
  getchar :: Int -> Char
  get2chars :: String
  get2chars = [getchar 1, getchar 2]
  ```

- We can fix the order of evaluation problem by creating a data dependency that forces sequencing:

  ```
  getchar :: Int -> (Char, Int)
  get2chars = [a,b] where
  (a,i) = getchar 1
  (b,j) = getchar i
  ```

But Things Get Messier

- A compiler could figure out that we don’t use the constant arguments to getchar and optimize the two calls into one. So we would need to write use a variable:

  ```
  get2chars x = [a,b] = where (a,y) = getchar x
  (b,z) = getchar y
  ```

- What if we want 4 chars? We get the same problem and have to hack up get2chars to compensate again:

  ```
  get4chars = [get2chars 1, get2chars 2]
  ```

- Maybe pure functional programming isn’t that great after all?! Do we take the Red Pill or the Blue Pill at this point?
How to Declare and Imperative

Input/Output is fundamentally a side-effecting imperative operation requiring explicit sequencing. So how does one deal with I/O in a pure functional language like Haskell?

Answer: hide it within a functional data abstraction. In the Haskell Prelude, the IO type is built via an internal mechanism, but we can think of it defined as as a function type:

```haskell
type IO a = World -> (a, World)
```

Recall that I mentioned before that the function main is defined as:

```haskell
main :: World -> (a, World), i.e., main :: IO a
```

The State of the World

- `getChar :: IO Char`
- `putChar :: Char -> IO ()`
The World

In a pure functional language the state of a program is implicit, but we can use functional abstraction and data abstraction to make it explicit.

To make state explicit, we have to keep track of the state of the world explicitly by “threading” it through the program.

```haskell
main :: World -> (a, World)
main world0 = let (c1, world1) = getChar world0
  (c2, world2) = getChar world1
  in
  ([c1,c2], world2)
```

Welcome to the Haskell Matrix

Like in the Matrix, you need to have a special functional mechanism for passing information back-and-forth between the World and the Haskell Matrix.

Monads are Haskell’s solution to the problem of interacting with the imperative real world of explicit state and imperative action, and the purely functional matrix world of implicit state and function evaluation.

We can think of an IO action as a function of type World -> (a, World), but without requiring the programmer to explicitly keep track of the state of the World. You can remain blissfully ignorant of the real world from within the Haskell Matrix.
The IO Monad

- The IO Monad is an instance of a Monad that hides the explicit World state inside the IO Monad and provides a pure functional interface using >>=, >> and return.
- The IO Monad is primitive and the implementation is internal to the Haskell runtime

```haskell
data IO a = -- builtin datatype of IO actions

instance Monad IO where
  (>>=) = primbindIO
  return = primretIO
  fail s = ioError (userError s)
```

Examples of IO a

- Some common IO functions operating on the standard input (stdin) and the standard output (stdout)

```haskell
main :: IO ()
getChar :: IO Char
putChar :: Char \rightarrow IO ()
putStr :: String \rightarrow IO ()
putStrLn :: String \rightarrow IO ()
putStrLn s = do { putStr s; putChar ‘\n’ }
print :: Show a \rightarrow a \rightarrow IO ()
getLine, getContents :: IO String
readIO :: Read a \rightarrow String \rightarrow IO a
readLn :: Read a \rightarrow IO a
```
IO Actions

Recall that the >> operator in the Monad type provides sequencing of monad actions

\[(>>): IO \ a \to IO \ b \to IO \ b\]

\texttt{putChar >> putChar}

Or more conveniently, we use the do notation syntactic sugar:

\texttt{main = do \{ putChar \ ‘a’; putChar \ ‘b’ \}}

\texttt{-- or equivalently}

\texttt{do \ putChar \ ‘a’}
\texttt{\ putChar \ ‘b’}

Sequencing IO Actions

\texttt{getChar :: IO Char result :: Char}
\texttt{putChar :: Char \to IO ()}

\texttt{getChar \gg \ putChar}
IO Actions with the World

You can think of IO actions as causing implicit state change in the World as if the World were passed around as an extra parameter.

--recall that \( \text{IO} \ a :: \text{World} \to (a, \text{World}) \)

\( (\gg) :: \text{IO} \ a \to \text{IO} \ b \to \text{IO} \ b \)

\[
(\text{action1} \gg \text{action2}) = \text{action}
\]

\[
\text{where action world0} = \text{let} \ (a, \text{world1}) = \text{action1 world0} \\
\ (b, \text{world2}) = \text{action2 world1} \\
\ (b, \text{world2})
\]

\( (\gg\gg) :: \text{IO} \ a \to (a \to \text{IO} \ b) \to \text{IO} \ b \)

\[
(\text{action1} \gg\gg \text{funaction}) = \text{action}
\]

\[
\text{where action} = \text{let} \ (a, \text{world1}) = \text{action1 world0} \\
\ (b, \text{world2}) = \text{funaction} \ a \text{world1} \\
\ (b, \text{world2})
\]

\( \text{return} :: a \to \text{IO} \ a \)

\[
\text{return} \ a \text{ world0} = (a, \text{world0})
\]

Example IO Actions

To use IO actions, we need to start in the main function since that is how a Haskell program gets its first World state that “threads” all subsequent IO actions

\[
\text{main} = \text{do}
\]

\[
\text{putStr} \ “\text{Enter some input:”} \\
\ a \leftarrow \text{readLn} -- \text{a contains the input} \\
\text{putStr} \ “\text{Enter some more:”} \\
\ b \leftarrow \text{readLn} \\
\text{print} \ (a, b)
\]

-- or equivalently

\[
\text{putStr} \ “\text{Enter some input:”} \gg \text{readLn} \\
\gg (a \to \text{putStr} \ “\text{Enter some more:”} \gg \text{readLn} \\
\gg (\ b \to \text{print} \ (a, b)))
\]
Monad Sequencing Functions

sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (c:cs) = do { x<-c; xs<-sequence cs; return (x:xs) }

sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ()

mapM :: Monad m => (a->m b) -> [a] => [m b]
mapM f = sequence . map f

Note the function composition operator
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

Lists of IO Actions

Since the type IO a is a just a function type, we can use IO actions just like other values. For example, we can put IO actions into a list and later evaluate them. Note that they are not evaluated in the definition of the list because until we “thread” them with an initial implicit World state from main.

ioactions = [print "Hello, world", putChar 'x']

main = sequence_ ioactions
The System.IO module contains many useful IO definitions that operate on file "handles"

import System.IO

stdin, stdout, stderr :: Handle

Useful Handle based functions

hIsEOF :: Handle -> IO Bool
isEOF :: IO Bool -- where isEOF = hIsEOF stdin
hFlush :: Handle -> IO ()
hReady, hisOpen, hIsClosed :: Handle -> IO Bool
hIsReadable, hIsWriteable :: Handle -> IO Bool
hIsTerminalDevice :: Handle -> IO Bool
hSetBinaryMode :: Handle -> Bool -> IO ()

Read/Writing files using handles

hLookahead, hGetChar :: Handle -> IO Char
hGetLine, hGetContents :: Handle -> IO String

hPutChar :: Handle -> Char -> IO ()
hPutStr, hPutStrLn :: Handle -> String -> IO ()
hPrint :: Show a => Handle -> a -> IO ()
hShow :: Handle -> IO String

Setting the buffering mode

data BufferMode = NoBuffering | LineBuffering | BlockBuffering
    deriving (Eq, Ord, Read, Show)

hSetBuffering :: Handle -> BufferMode -> IO ()

hgetBuffering :: Handle -> BufferMode -> IO ()

hFlush :: Handle -> IO ()
System.IO Module

Useful data types for use with files and related functions:

type FilePath = String

data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
    deriving (Enum, Eq, Ord, Read, Show, Ix)

openFile :: FilePath -> IOMode -> IO Handle
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
openTempFile :: FilePath -> String -> IO (FilePath, Handle)

A Read-Eval-Print-Loop

A REPL loop prompts for input, reads an input string, evaluates the input string, and prints the result of the evaluation. A REPL for a monadic parser might be:

module Main (main) where
repl :: IO ()
repl = do putStr "repl> "
    hFlush stdout -- stdout is line buffered by default, so flush prompt
eof <- isEOF -- check for ctrl-d (EOF) on stdin
if eof -- when using if in a do block, then/else need new do blocks
    then do { putStrLn ""; return () }
    else do input <- getLine
            case (parse startRule input) of
                Left _ -> putStrLn "syntax error!"
                Right (s, _) -> putStrLn s
            repl
main = do { putStrLn "Welcome to REPL! Use Ctrl-d to exit"; repl }
A Haskell While Loop Function

 Loopy is sequencing of actions combined with tail recursion and a mutable variable for the loop counter

```
-- from the Haskell Prelude
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

foreach = flip mapM_

while test action = do val <- test
    if val
    then do { action; while test action }
    else return ()
```

Mutable Variables in the IO Monad

 Now that the State of the World is safely encapsulated within the IO Monad, Haskell uses the IO Monad to implement mutable data

```
data IORef a -- builtin
newIORef :: a -> IO (IORef a)
readIORef : IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
```
Looping in Haskell using an IORef

In Python, we imperatively write:

```python
for x in range(1,11):
    print x
i = 0
while i < 5:
    print "looping.."
    i += 1
```

In Haskell, we monadically write using the do notation:

```haskell
incr ref = modifyIORef ref (+1)
test ref p = do { v <- readIORef ref; return (p v) }
main = do foreach [1..10] print
      i <- newIORef 0  -- IORef Int
      while (test i (< 5))
        (do  print "looping...
             incr i )
```

More Monadic Control Functions

- Infinite loop
  ```haskell
  forever :: IO () -> IO ()
  forever a = a >> forever a
  ```

- Repeat n times
  ```haskell
  repeatN :: Int -> IO a -> IO ()
  repeatN 0 a = return ()
  repeatN n a = a >> repeat (n-1) a
  ```

- For loop
  ```haskell
  for :: [a] -> (a -> IO ()) -> IO ()
  for [] fa = return ()
  for (n:ns) fa = fa n >> for ns fa
  -- e.g., printNums = for [1..10] print
  ```