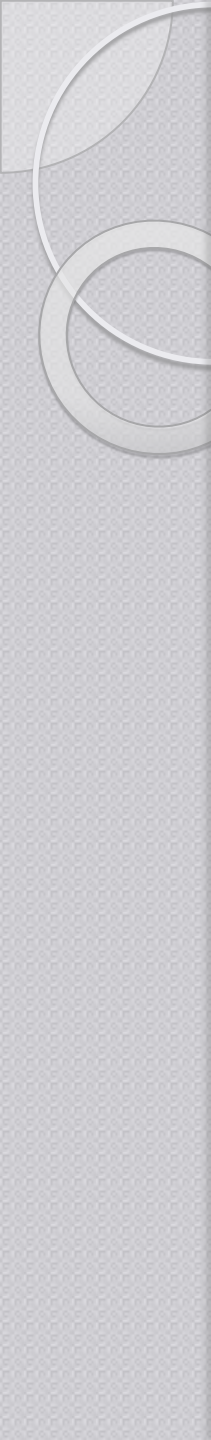




I/O Streams

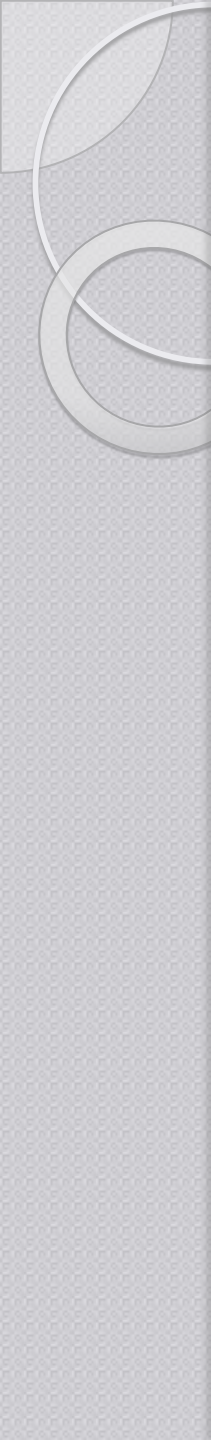
Major source of side effects found in traditional I/O

- read- return a different value each time
- display- multiple call must occur in proper order
- Functional languages rarely supports side effects
- One way to avoid these side effects is to model input and output as streams
- unbounded length lists whose elements are generated lazily

- 
- If we model input and output as streams, then a program takes the form
 - (define output (my_prog input))
 - When program needs i/p
 - Function 'my_prolog' forces evaluation of car of i/p
 - Passes cdr to rest of program
 - To drive execution, the language implementation repeatedly forces evaluation of the car of output, prints it and repeats

Example

write a purely functional program that prompts the user for a sequence of numbers (one at a time!) and prints their squares.



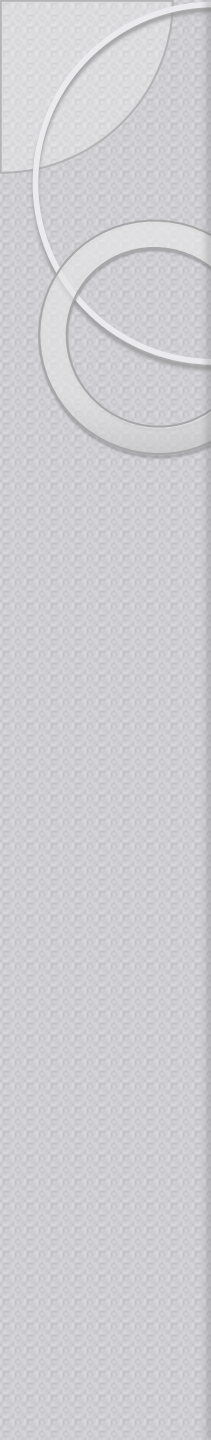
```
(define driver (lambda (s) //To repeat execution of a function
  (if (null? s) '() ; nothing left
      (display (car s))
      (driver (cdr s)))))
(driver output)
```

```
(define squares (lambda (s)
  (cons "please enter a number\n"
        (let ((n (car s)))
          (cons (* n n) (cons #\newline (squares (cdr s))))))))
(define output (squares input))
```

To execute: (driver(squares(input)))

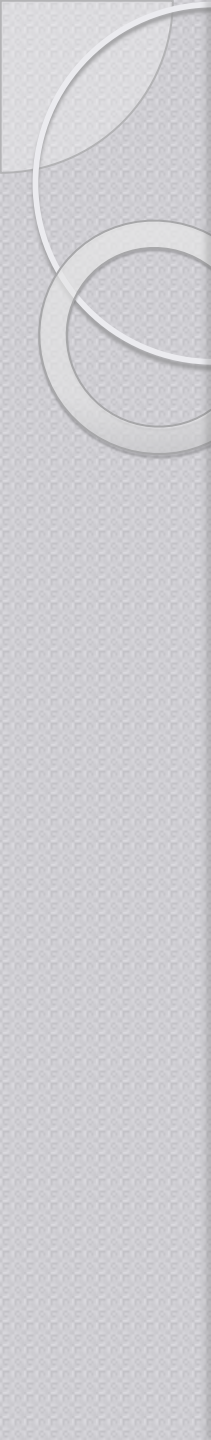
Monads

- Disadvantages of Streams
 - Streams donot work for graphics or random access files
 - They also make it difficult to accomodate i/o of different kinds
- Monads are used in such contexts
- Monads are higher order functions that allow the programmer to chain together a sequence of actions that must happen in order

- 
- Monad chains operations in some specific useful way
 - The IO monad performs operations sequentially
 - but passes a "hidden variable" which represents "the state of the world"

Example

- Creating a pseudorandom number generator(RNG)
 - Modifies hidden state as a side effect, allowing it to return a different value every time it is called
- This is not possible in a pure functional language

- 
- We can obtain a similar effect by passing the state of the function and having it return new state along with the random number
 - This is exactly how the built in function `random` works in Haskell




```
twoRandomInts :: StdGen -> ([Integer], StdGen)
```

```
-- type signature: twoRandomInts is a function that takes an  
-- StdGen (the state of the RNG) and returns a tuple containing  
-- a list of Integers and a new StdGen.
```

```
twoRandomInts gen = let  
    (rand1, gen2) = random gen  
    (rand2, gen3) = random gen2  
in ([rand1, rand2], gen3)
```

```
main = let  
    gen = mkStdGen 123 -- new RNG, seeded with 123  
    ints = fst (twoRandomInts gen) -- extract first element  
in print ints -- of returned tuple
```

- 
- gen2- return values from the first call to random passed as arg to 2nd call
 - gen3-a return values from the 2nd call is returned to main - where it would be passed to another function
 - The problem is : copies of RNG state must be "threaded through" every function that needs a random number



```
twoMoreRandomInts :: IO [Integer]
```

```
-- twoMoreRandomInts returns a list of Integers. It also implicitly accepts,  
and returns, all the state of the IO monad.
```

```
twoMoreRandomInts = do
```

```
  rand1 <- randomIO
```

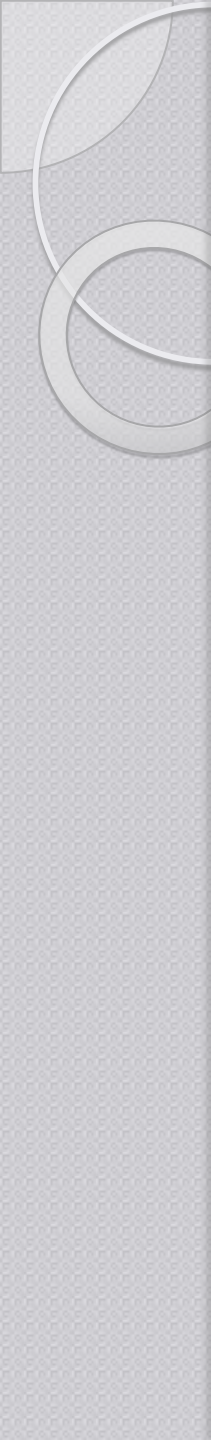
```
  rand2 <- randomIO
```

```
  return [rand1, rand2]
```

```
main = do
```

```
  moreInts <- twoMoreRandomInts
```

```
  print moreInts
```

- 
- There are several differences here
 - the type of the `twoMoreRandomInts` function has become `IO [Integer]`.
 - This identifies it as an IO action -function that invisibly accepts and returns the state of the IO monad
 - To thread the IO state from one action to the next, the bodies of `twoMoreRandomInts` and `main` use `do` notation rather than `let`
 - At each step it passes the state of the monad from one action to the next

Strings

- Strings in Haskell re simply list of characters
- `putStr :: String -> IO ()`
- `putStr s = sequence_ (map putChar s)`

map

- The map function takes a function f and a list l as argument, and returns a list that contains the results of applying f to the elements of l :
- $\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
- $\text{map } f [] = []$ -- base case
- $\text{map } f (h:t) = f h : \text{map } f t$
- -- ':' is like cons in Scheme

sequence_

- sequence_ converts this to a single action that prints a list
- It could be defined as follows.
- $\text{sequence_} :: [\text{IO } ()] \rightarrow \text{IO } ()$
- $\text{sequence_} [] = \text{return } ()$ -- base case
- $\text{sequence_} (a:\text{more}) = \text{do } a; \text{sequence_ more}$