Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

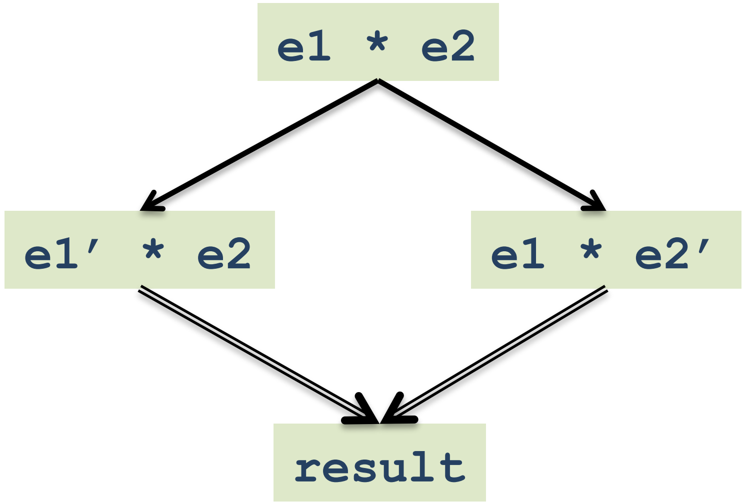
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 29

- Monads in Haskell
- The IO Monad

Pros of Functional Programming

- Functional programming is beautiful:
  - Concise and powerful abstractions
    - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
  - Close correspondence with mathematics
    - Semantics of a code function is the mathematical function
    - Equational reasoning: if x = y, then f x = f y
    - Independence of order-of-evaluation (Confluence, aka Church-Rosser)



The compiler can choose the best sequential or parallel evaluation order

Problems…

- But to be *useful* as well as *beautiful*, a language must be able to manage “impure features”:
  - Input/Output
  - Imperative update
  - Error recovery (eg, timeout, divide by zero, etc.)
  - Foreign-language interfaces
  - Concurrency control

The whole point of a running a program is to interact with the external environment and affect it

The Direct Approach

- Just add imperative constructs "the usual way"
  - I/O via "functions" with side effects:

```
putchar 'x' + putchar 'y'
```

  - Imperative operations via assignable reference cells:

```
z = ref 0; z := !z + 1;
f(z);
w = !z     (* What is the value of w? *)
```

  - Error recovery via exceptions
  - Foreign language procedures mapped to "functions"
  - Concurrency via operating system threads
- Can work if language determines evaluation order
  - Ocaml, Standard ML are good examples of this approach

But what if we are "lazy"?

In a lazy functional language, like Haskell, the order of evaluation is deliberately undefined, so the "direct approach" will not work.

- Example: `res = putchar 'x' + putchar 'y'`
  - Output depends upon the evaluation order of (+).
- Example: `ls = [putchar 'x', putchar 'y']`
  - Output depends on how list is used
  - If only used in `length ls`, nothing will be printed because `length` does not evaluate elements of list

Fundamental question

- Is it possible to regard pure Haskell as the basic programming paradigm, and add imperative features without changing the meaning of pure Haskell expressions?
- **Yes!** Using the concept of **monad**
  - Formally defined as a **type constructor class**
  - Each **monadic type constructor** defines certain **monadic values** (sometimes called **actions**) and how to compose them sequentially

The Maybe type constructor

```
data Maybe a = Nothing | Just a

-- example
sqrt :: Int -> Maybe Real
```

- A value of type **Maybe a** is a possibly undefined value of type **a**
- A function **f :: a -> Maybe b** is a partial function from **a** to **b**

Composing partial function

```haskell
father :: Person -> Maybe Person  -- partial function
mother :: Person -> Maybe Person  -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just mom -> father mom
```

```haskell
bothGrandfathers :: Person -> Maybe (Person, Person)
    bothGrandfathers p =
        case father p of
            Nothing -> Nothing
            Just dad ->
                case father dad of
                    Nothing -> Nothing
                    Just gf1 ->                          -- found first grandfather
                        case mother p of
                            Nothing -> Nothing
                            Just mom ->
                                case father mom of
                                    Nothing -> Nothing
                                    Just gf2 ->          -- found second grandfather
                                        Just (gf1, gf2)
```

The Monad type class and the Maybe monad

```
class Monad m where
        return :: a -> m a
        (>>=)  :: m a -> (a -> m b) -> m b -- "bind"
         ... -- + something more
```

- **m** is a type constructor
- **m a** is the type of **monadic values**

```
instance Monad Maybe where
        return :: a -> Maybe a
        return x = Just x
        (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
        y >>= g  = case  y of
                    Nothing -> Nothing
                    Just x -> g x
```

- **bind (>>=)** shows how to “propagate” undefinedness

Use of **bind** of the **Maybe** monad to compose partial functions

```
father :: Person -> Maybe Person   -- partial function
mother :: Person -> Maybe Person   -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
        Nothing -> Nothing
        Just mom -> father mom
```

```
maternalGrandfather p = mother p >>= father

bothGrandfathers p =
       father p >>=
           (\dad -> father dad >>=
               (\gf1 -> mother p >>=
                   (\mom -> father mom >>=
                       (\gf2 -> return (gf1,gf2) ))))
```

Alternative, imperative-style syntax: **do**

```
bothGrandfathers p =
      father p >>=
          (\dad -> father dad >>=
              (\gf1 -> mother p >>=
                  (\mom -> father mom >>=
                      (\gf2 -> return (gf1,gf2) ))))
```

```
bothGrandfathers p = do {
       dad <- father p;
       gf1 <- father dad;
       mom <- mother p;
       gf2 <- father mom;
       return (gf1, gf2);
     }
```

```
bothGrandfathers p = do
       dad <- father p
       gf1 <- father dad
       mom <- mother p
       gf2 <- father mom
       return (gf1, gf2)
```

- **do** syntax is just syntactic sugar for **>>=**

Some Haskell Monads

Monad	Imperative semantics
Maybe	Exception (Anonymous)
Error	Exception (with error description)
State	Global state
IO	Input/output
[] (lists)	Non-determinism
Reader	Environment
Writer	Logger

Understanding Monads as containers

```
class Monad m where   -- definition of Monad type class
        return :: a -> m a
        (>>=)   :: m a -> (a -> m b) -> m b -- "bind"
         ... -- + something more + a few axioms
```

- The monadic constructor can be seen as a ***container***: let's see this for *lists*
- Getting **bind** from more basic operations

```
map :: (a -> b) -> [a] -> [b] -- seen. "fmap" for Functors

return  :: a -> [a] -- container with single element
return x = [x]

concat :: [[a]] -> [a] -- flattens two-level containers
      Example: concat [[1,2],[],[4]] = [1,2,4]

(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f  =  concat(map f xs)

Exercise: define map and concat using bind and return
```

Understanding Monads as computations

```
class Monad m where  -- definition of Monad type class
       return :: a -> m a
       (>>=)  :: m a -> (a -> m b) -> m b -- "bind"
       (>>)   :: m a -> m b -> m b        -- "then"
        ... -- + something more + a few axioms
```

- A value of type **`m a`** is a computation returning a value of type **`a`**
- For any value, there is a computation which "does nothing" and produces that result. This is given by function **`return`**
- Given two computations **`x`** and **`y`**, one can form the computation **`x >> y`** which intuitively "runs" **`x`**, throws away its result, then runs **`y`** returning its result
- Given computation **`x`**, we can use its result to decide what to do next. Given **`f: a -> m b`**, computation **`x >>= f`** runs **`x`**, then applies **`f`** to its result, and runs the resulting computation.

```
Note that we can define then using bind:
x >> y = x >>= (\_ -> y)
```

Understanding Monads as computations (2)

```
class Monad m where   -- definition of Monad type class
        return :: a -> m a
        (>>=)  :: m a -> (a -> m b) -> m b -- "bind"
        (>>)   :: m a -> m b -> m b         -- "then"
         ... -- + something more + a few axioms
```

- **return**, **bind** and **then** define basic ways to compose computations
- They are used in Haskell libraries to define more complex composition operators and control structures (sequence, for-each loops, …)
- If a type constructor defining a library of computations is monadic, one gets automatically benefit of such libraries

Example: MAYBE

- `f:a -> Maybe b` is a ***partial*** function
- **bind** applies a partial function to a possibly undefined value, propagating undefinedness

Example: LISTS

- `f:a -> [b]` is a ***non-deterministic*** function
- **bind** applies a non-deterministic function to a list of values, collecting all possible results

Example: Parsing, handling errors, IO, backtracking….

Monadic Input and Output
The IO Monad

Problem

A functional program defines a pure function, with no side effects

The whole point of running a program is to have some side effect

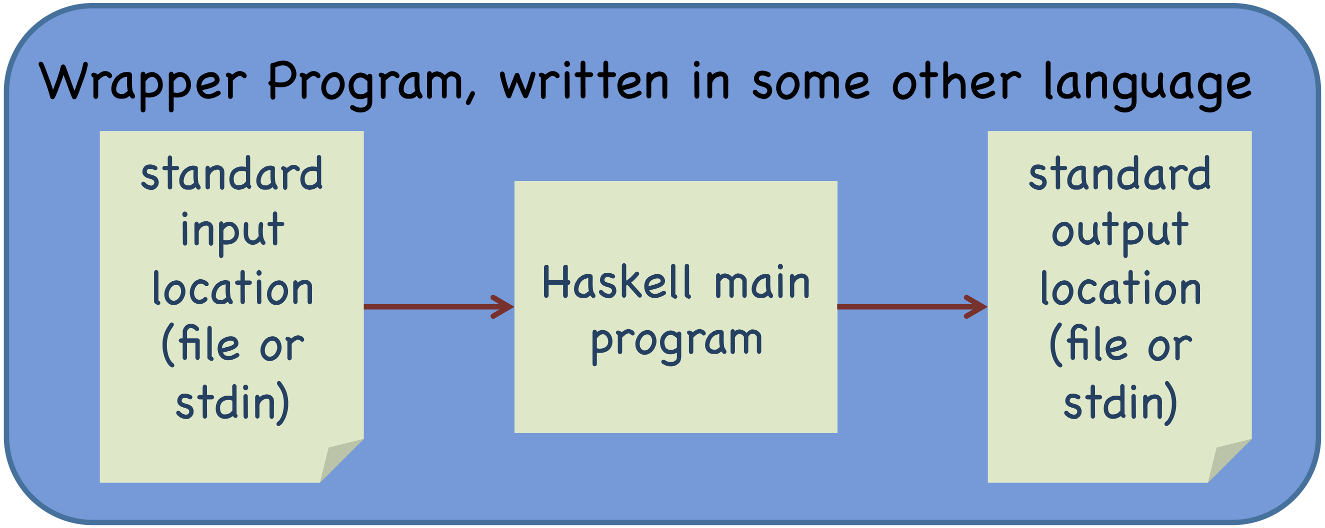
The term “side effect” itself is misleading

Before Monads

- Streams
  - Program sends stream of requests to OS, receives stream of responses
- Continuations
  - User supplies continuations to I/O routines to specify how to process results
- World-Passing
  - The "State of the World" is passed around and updated, like other data structures
  - Problem: how to guarantee single-threaded access to the world
- Haskell 1.0 Report adopted Stream model
  - Stream and Continuation models were discovered to be inter-definable

Stream Model: Basic Idea

- Move "side effects" outside of functional program
- Haskell `main :: String -> String`



- Gets more complicated …
  - But what if you need to read more than one file? Or delete files? Or communicate over a socket? ...

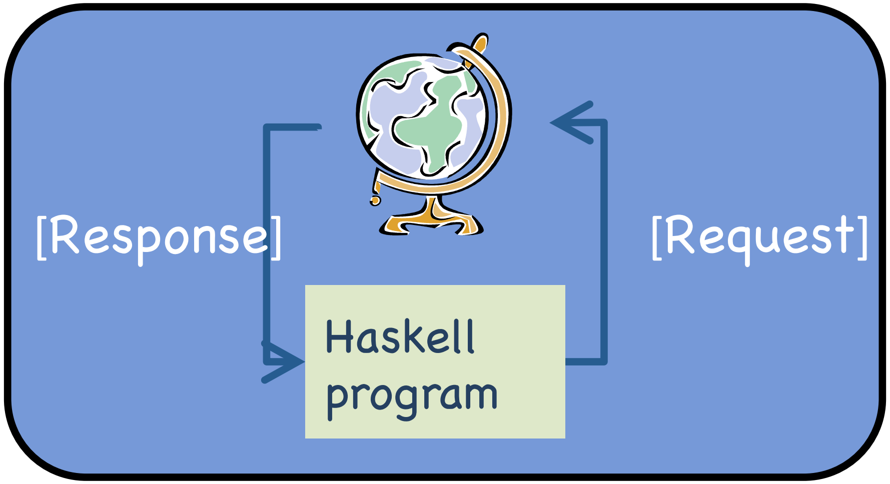
Stream Model

- Enrich argument and return type of `main` to include all input and output events.

```
main :: [Response] -> [Request]
data Request =  ReadFile Filename
          |  WriteFile FileName String
          |  …
data Response=  RequestFailed
          |  ReadOK String
          |  WriteOk
          |  Success   |  …
```

- Wrapper program interprets requests and adds responses to input.
- Move side effects outside of functional program

Stream Model: `main::[Response] -> [Request]`



- Problem: Laziness allows program to generate requests prior to processing any responses.
- Hard to extend
  - New I/O operations require adding new constructors to Request and Response types, modifying wrapper
- Does not associate Request with Response
  - easy to get "out-of-step," which can lead to deadlock
- Not composable
  - no easy way to combine two "main" programs
- ... and other problems!!!

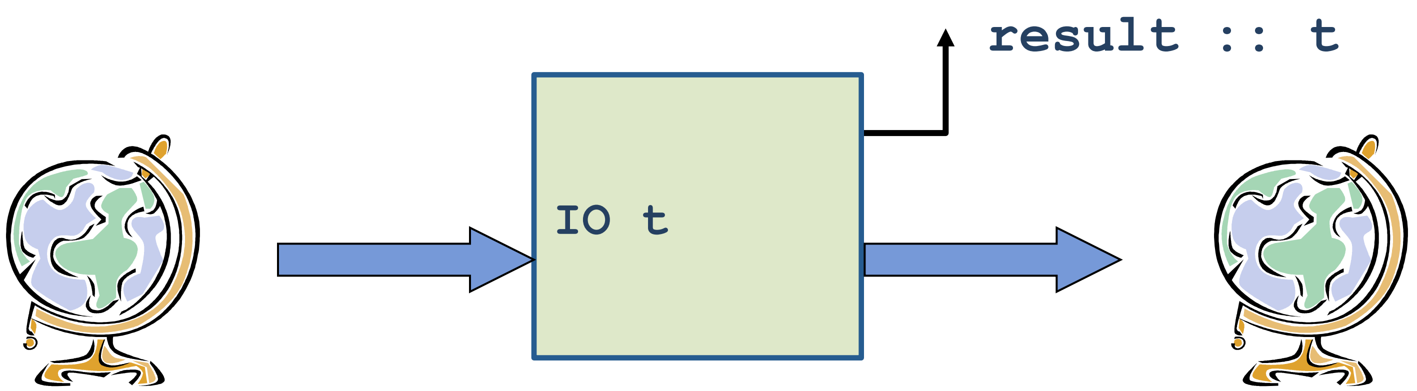
Monadic I/O: The Key Ideas

- **IO** is a type constructor, instance of **Monad**
- A value of type **`(IO t)`** is a computation or "action" that, when performed, may do some input/output before delivering a result of type **`t`**
- **return** returns the value without making I/O
- **then** (>>) [and also **bind** (>>=)] composes two actions sequentially into a larger action
- The only way to perform an action is to call it at some point, directly or indirectly, from **`Main.main`**.

A Helpful Picture

A value of type (IO t) is an "action." When performed, it may do some input/output before delivering a result of type t.

```
type IO t = World -> (t, World)
```



- An action is a first-class value
- **Evaluating** an action has no effect; **performing** the action has the effect

Simple I/O actions

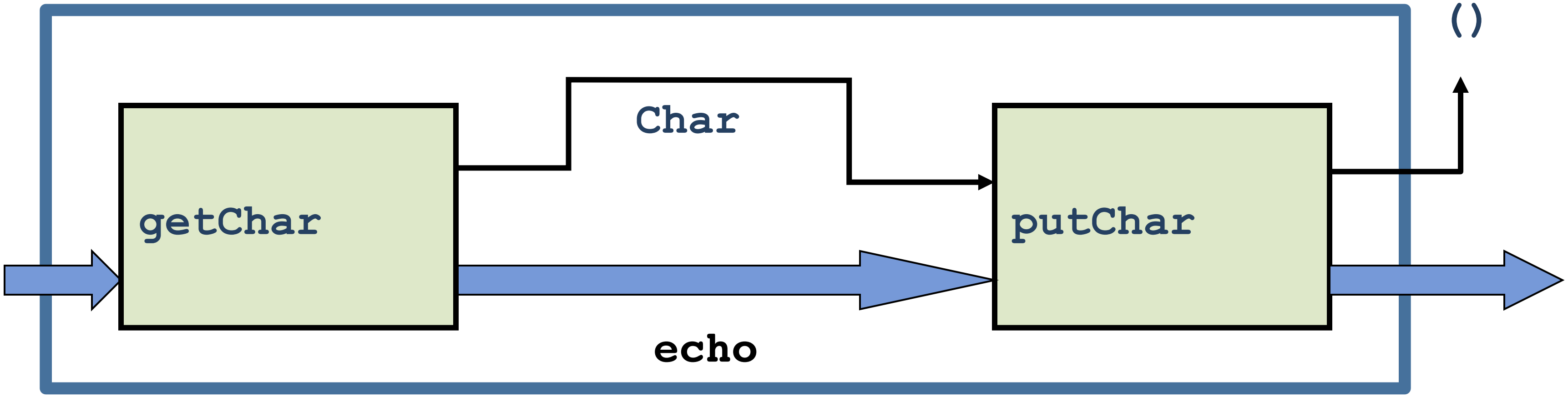

```
getChar :: IO Char
putChar :: Char -> IO ()

main :: IO ()
main = putChar 'x'
```

Main program is an action of type IO ()

The Bind Combinator (>>=)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```



- We have connected two actions to make a new, bigger action.

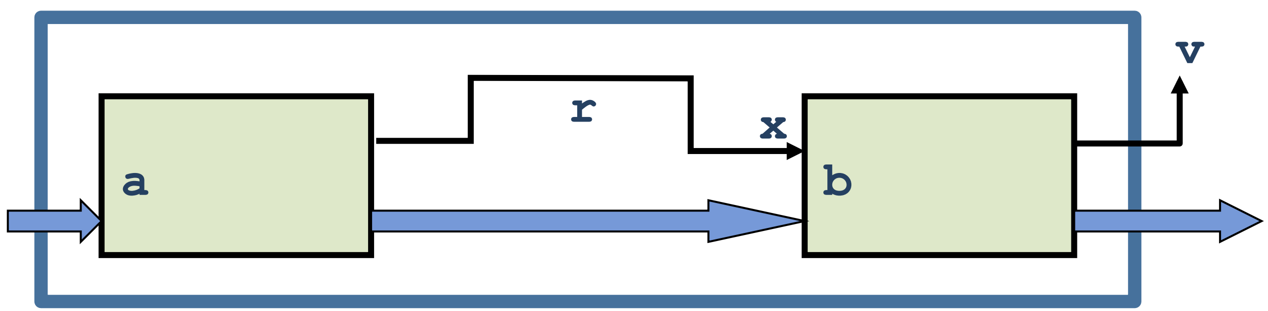
```
getChar :: IO Char
putChar :: Char -> IO ()
```

```
echo :: IO ()
echo = getChar >>= putChar
```

The (>>=) Combinator

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- Operator is called **bind** because it binds the result of the left-hand action in the action on the right
- Performing compound action **`a >>= \x->b`** :
  - performs action **`a`**, to yield value **`r`**
  - applies function **`\x->b`** to **`r`**
  - performs the resulting action **`b{x <- r}`**
  - returns the resulting value **`v`**



The (>>) Combinator

- The “then” combinator (>>) does sequencing when there is no value to pass:

```
(>>) :: IO a -> IO b -> IO b
-- defined from bind
(>>=) :: IO a -> (a -> IO b) -> IO b
m >> n  =  m >>= (\_ -> n)
```

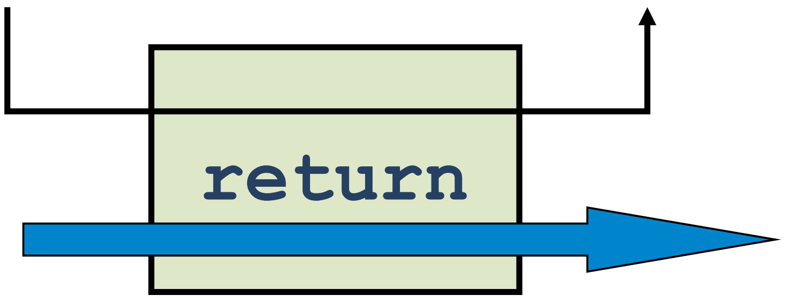
```
echoDup :: IO ()
echoDup = getChar  >>= \c  ->
          putChar c  >>
          putChar c
```

```
echoTwice :: IO ()
echoTwice = echo >> echo
```

The return Combinator

- The action (return v) does no IO and immediately returns v:

```
return :: a -> IO a
```




```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar   >>= \c1 ->
              getChar   >>= \c2 ->
              return (c1,c2)
```

The “do” Notation

- The “do” notation adds syntactic sugar to make monadic code easier to read.

```
-- Plain Syntax
getTwoChars :: IO (Char,Char)
getTwoChars = getChar   >>= \c1 ->
              getChar   >>= \c2 ->
              return (c1,c2)
```

```
-- Do Notation
getTwoCharsDo :: IO(Char,Char)
getTwoCharsDo = do { c1 <- getChar ;
                     c2 <- getChar ;
                     return (c1,c2) }
```

- Do syntax designed to look imperative.

Desugaring “do” Notation

- The “do” notation *only* adds syntactic sugar:

```
do { x }           =  x
do { x; stmts }    =  x >> do { stmts }
do { v<-x; stmts }    =  x >>=  \v -> do { stmts }
do {let ds; stmts }    =     let ds in do { stmts }
```

The scope of variables bound in a generator is the rest of the “do” expression.

- The following are equivalent:

```
do { x1 <- p1; ...; xn <- pn; q }
```

```
do   x1 <- p1; ...; xn <- pn; q
```

```
do x1 <- p1
   ...
   xn <- pn
   q
```

Bigger Example

- The `getLine` function reads a line of input:

```
getLine :: IO [Char]
getLine = do { c <- getChar ;
               if c == '\n' then
                   return []
               else
                   do { cs <- getLine;
                        return (c:cs) }}
```

Note the "regular" code mixed with the monadic operations and the nested "do" expression.

Control Structures on Monads

- Exploiting the monadic combinators, we can define control structures that work for any monad

```
repeatN 0 x = return ()
repeatN n x = x >> repeatN (n-1) x
repeatN :: (Num a, Monad m, Eq a) => a -> m a1 -> m ()
```

```
Main> repeatN 5 (putChar 'h')
```

```
for []     fa = return ()
for (x:xs) fa = fa x  >>  for xs fa
for :: Monad m => [t] -> (t -> m a) -> m ()
```

```
Main> for [1..10] (\x -> putStr (show x))
```

Sequencing

A list of IO actions.

An IO action returning a list.

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a:as) = do { r  <- a;
                       rs <- sequence as;
                       return (r:rs) }
sequence :: Monad m => [m a] -> m [a]
```

- Example use:

```
Main> sequence [getChar, getChar, getChar]
```

Slogan: First-class actions let programmers write application-specific control structures.

IO Provides Access to Files

- The IO Monad provides a large collection of operations for interacting with the "World."
- For example, it provides a direct analogy to the Standard C library functions for files:

```
openFile :: FilePath -> IOMode -> IO Handle
hPutStr  :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
hClose   :: Handle -> IO ()
```

References

- The IO operations let us write programs that do I/O in a strictly sequential, imperative fashion.
- Idea: We can leverage the sequential nature of the IO monad to do other imperative things!

```
data IORef a    -- Abstract type
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

- A value of type IORef a is a reference to a mutable cell holding a value of type a.

Example Using References

```
import Data.IORef  -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
   { r <- newIORef 0;
     addToN r 1 }
  where
    addToN :: IORef Int -> Int -> IO Int
    addToN r i | i > n      = readIORef r
               | otherwise = do
                 { v <- readIORef r
                 ; writeIORef r (v + i)
                 ; addToN r (i+1)}
```

But this is terrible! Contrast with: sum [1..n]. Claims to need side effects, but doesn't really.

Example Using References

```
import Data.IORef  -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
   { r <- newIORef 0;
     addToN r 1 }
  where
    addToN :: IORef Int -> Int -> IO Int
    addToN r i | i > n      = readIORef r
               | otherwise = do
                 { v <- readIORef r
                 ; writeIORef r (v + i)
                 ; addToN r (i+1)}
```

Just because you can write C code in Haskell, doesn't mean you should!

The IO Monad as ADT

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b

getChar :: IO Char
putChar :: Char -> IO ()
... more operations on characters ...
openFile :: [Char] -> IOMode -> IO Handle
... more operations on files ...
newIORef :: a -> IO (IORef a)
... more operations on references …
```

- All operations return an IO action, but only **bind** (>>=) takes one as an argument.
- **Bind** is the only operation that combines IO actions, which forces sequentiality.
- In pure Haskell, there is no way to transform a value of type `IO a` into a value of type `a`
- There are unsafe primitives for this

Implementation

- GHC uses "world-passing semantics" for the IO monad

```
type IO t = World -> (t, World)
```

- It represents the "world" by an un-forgeable token of type `World`, and implements `bind` and `return` as:

```
return :: a -> IO a
return a = \w -> (a,w)
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) m k = \w -> case m w of (r,w') -> k r w'
```

- Using this form, the compiler can do its normal optimizations. The dependence on the world ensures the resulting code will still be single-threaded.
- The code generator then converts the code to modify the world "in-place."

Summary

- A complete Haskell program is a single IO action called **`main`**. Inside IO, code is single-threaded.
- Big IO actions are built by gluing together smaller ones with bind (>>=) and by converting pure code into actions with return.
- IO actions are first-class.
  - They can be passed to functions, returned from functions, and stored in data structures.
  - So it is easy to define new "glue" combinators.
- The IO Monad allows Haskell to be pure while efficiently supporting side effects.
- The type system separates the pure from the effectful code.

Comparison

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because it is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- So it is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.

Monad Laws

```
1) return x  >>=  f  =  f x
2) m  >>=  return    =  m
3) (x >>= f) >>= g = x >>= (\v -> f v >>= g)
```

- In do-notation:

```
1) do { w <- return v; f w }
   = do { f v }

2) do { v <- x; return v }
   = do { x }
```

```
3) do { x <- m1;      =    do { y <- do { x <- m1;
        y <- m2;                           m2 }
        m3 }                      m3}
```

```
x not in free vars of m3
```

Derived Laws for (>>) and done

```
(>>) :: IO a -> IO b -> IO b
m >> n  =  m >>= (\_ -> n)

done :: IO ()
done = return ()
```

```
done >>  m            = m
m  >>  done           = m
m1 >> (m2 >> m3)  = (m1 >> m2) >> m3
```