Input/Output

and

Monads

Questions

> What's the difference between this

```
*Main> "One " ++ "line."
"One line."
```

> and this?

```
*Main> putStrLn ("One " ++ "line.")
One line.
```

> How does Haskell's I/O notation work?

> How can input/output can be done at all using just pure functions with no side effects?

Interactive mode

> When you do this:

```
*Main> "One " ++ "line."
"One line."
```

> you are using an interactive 'debugging' mode, so Haskell prints the *value* of the expression (plus a newline), so there are quotes to show it is a string

> When you do this:

```
*Main> putStrLn ("One " ++ "line.")
One line.
```

> you are asking Haskell to do some I/O, so there are no quotes, just the characters of your string and a newline

The **putStrLn** Function

> The type of the **`putStrLn`** function is

```
putStrLn :: String -> IO ()
```

> An **`IO()`** object is an ***action***, i.e. an I/O operation where **`()`** means "returning nothing to the program"

> In fact **`()`** is a tuple, one that has no fields at all, and it is used as the equivalent of **`void`** in C/Java

Manipulating IO Objects

> Inside normal functions, you can manipulate IO objects, but you can't make them actually do anything

```
oneTwo :: IO ()
oneTwo = putStrLn "One." >> putStrLn "Two."
```

> This takes two actions and glues them together to form an action which prints two lines

> The result action only 'happens" when it is "offered" to the operating system:

```
*Main> oneTwo
One.
Two.
```

The `>>` Operator

> The **>>** operator takes two actions and produces a combined action which means "do the two things one after the other"

> Of course, it can be used repeatedly **a1>>a2>>a3** and you can write a loop to repeat something **n** times

```
run :: Int -> IO () -> IO ()
run 1 action = action
run n action = action >> run (n-1) action
```

```
*Main> run 3 (putStr "yes ")
yes yes yes *Main>
```

The do Notation

> There is an alternative notation using the **`do`** keyword instead of the **`>>`** operator

```
run :: Int -> IO () -> IO ()
run 1 action = action
run n action = do
   action
   run (n-1) action
```

> After **`do`**, each line is an action, and they are joined in sequence, so the code becomes ***sequential***

> It doesn't make much difference here, but the **`do`** notation gets more useful later

Input

> The opposite of **putStrLn** is **getLine** which reads a line of text in

```
*Main> getLine
Hi there
"Hi there"
```

> After typing **getLine**, the system stops and waits for you to type a line of text in, then it prints the result

> The type of **getLine** is an action returning a string:

```
getLine :: IO String
```

The >>= Operator

> The **>>=** operator joins two actions together, passing the result of the first to the second

> The second argument must be a function which takes one argument and returns an action

```
*Main> getLine >>= putStrLn
Hi there
Hi there
```

Extended do Notation

- The **do** notation is easier to use than the **>>=** operator, because it avoids the need to define one-argument functions to use as the second argument

```
square :: IO ()
square = do
   s <- getLine
   putStrLn (show (read s ^ 2))
```

```
*Main> square
42
1764
```

- The notation **x<-a** in a **do** sequence means "do action **a** and define a variable **x** to hold the result"

The return Action

> One more IO feature is needed to complete the picture
> The action **`return x`** means "do no actual input or output, but just return **`x`** as the result of the action"

```
*Main> return "Hi"
"Hi"
```

> The main use is in combining results

```
getLine :: IO String
getLine = do
   c <- getChar
   if c =='\n' then return "" else do
      s <- getLine
      return (c : s)
```

File IO

> To write and read files in the simplest way just try this

```
*Main> writeFile "junk.txt" "contents"
*Main> readFile "junk.txt"
"contents"
```

> The **writeFile** action writes to a given filename, filling it with the given new contents, and the **readFile** action reads a file, given its name, and returns the contents as a stream (i.e. string)

File Functions

> To build file actions, use the same IO notation

```
copyFile :: String -> String -> IO ()
copyFile fromName toName = do
   content <- readFile fromName
   writeFile toName content
```

> The content of the file is represented as a string, but it will ***not*** (necessarily) all be in memory at once, because of the streaming effect of laziness

> Further IO facilities, including lower level details, can be imported from the IO module

One Big Action

- Apart from combining little actions into bigger actions, the only other thing you can do is to give the entire program as a single big action to the system
- In ghci, you can type in and execute any IO action
- In proper Haskell (ghc, not the interactive tool ghci, if you want to ***compile*** a program, the program needs a thing called **`main`** which must have type **`IO ()`**

```
main :: IO ()
main = ...
```

What does it all Mean?

- Let's switch from programming with IO to asking questions about what it means
- If we were to implement the IO type for ourselves, what would it look like?
- Ignoring results of actions, we might start out something like this:

```
data IO = IO (Computer -> Computer)
```

- An action is some thing that changes the state of the computer, wrapped with a constructor to make sure that only official operations can be performed

The Traditional Picture

> It may seem a bit much passing the entire state of the computer to a program and getting a new state back

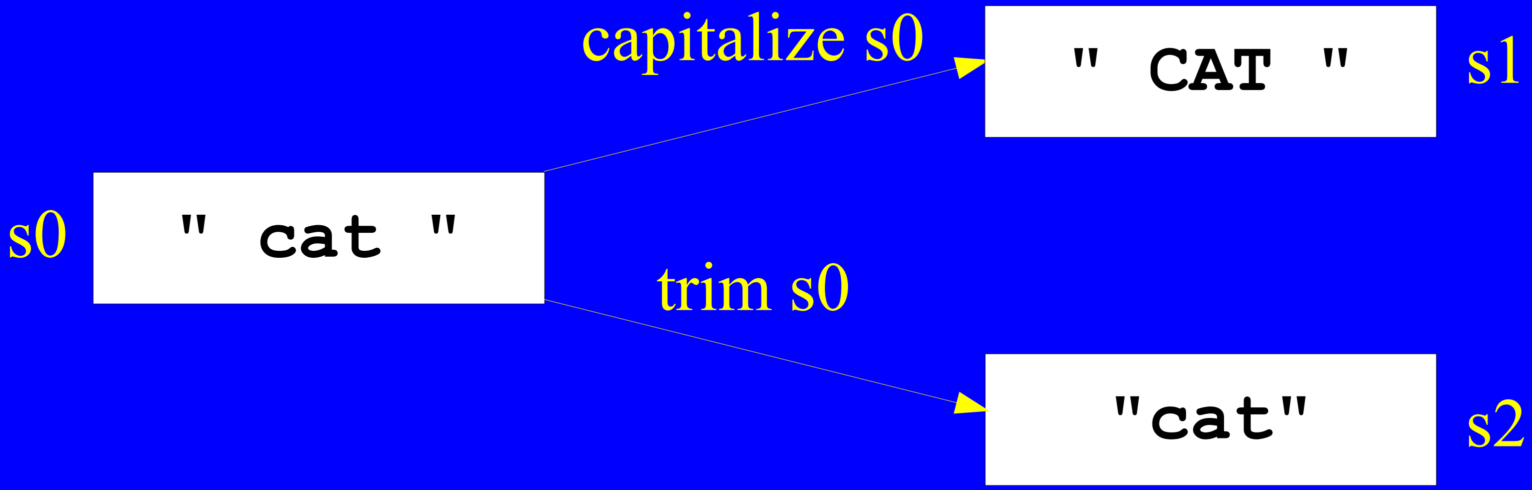
> But that's not Haskell's fault, it is the fault of the traditional view of programs

> If you download some binary program from the Web, say **`armageddon.exe`**, it can do anything it likes to your computer, e.g. give away all your most secret data and then wipe your hard disk clean

> People are only just waking up to the fact that this is a silly idea, because you can't trust programs that much nowadays, but we are stuck with the idea for now

Sequential Actions 1

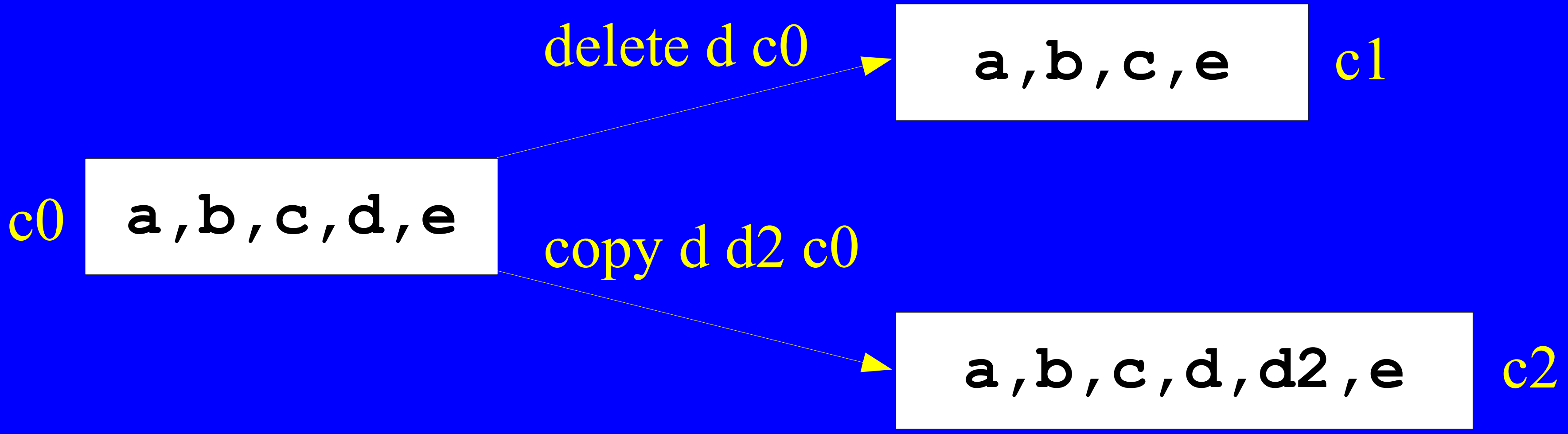
- Another aspect of the traditional view of programs is that the actions must form a linear sequence, unlike functions applied to normal data
- If you start with a string **`s0`**, you can capitalize it to form **`s1 = f s0`** and trim it to form **`s2 = g s0`**



Sequential Actions 2

> But you can't start with a computer state **`c0`**, delete a file to form **`c1 = f c0`**, then copy the same file to form **`c2 = g c0`**

> If **`c0`** has files **`a,b,c,d,e`** and **`d`** is deleted or copied:



Sequential Actions 3

- The standard convention is that the computer state is not copyable, there can only ever be one of it, actions are not regarded as reversible, and you can't undo a file deletion (except by manual intervention)
- There is a lot to be said for a system in which every action is undoable, and it isn't completely impossible (as in some editors and other applications) but we are stuck with the sequential model most of the time

A Safe Action Type

> We need to make sure that whenever we apply an action to a computer state, the old computer state is never used again

> What we can do is to make sure that the only thing you can do with actions is to combine them:

```
data IO = IO (Computer -> Computer)

(>>) :: IO -> IO -> IO
(IO f) >> (IO g) = IO h where h c = g (f c)
```

> An action is notionally a function of type **`Computer -> Computer`**, but protected inside an abstract type

Monads

> That brings us to something in Haskell called Monads

> The word monad is used in music, philosophy, maths, biology etc. to mean something 'singular'

> It is used with a specific meaning in Haskell (somewhat similar to its meaning in category theory)

> It means a type where the only operations provided are sequential, acting on a singular state

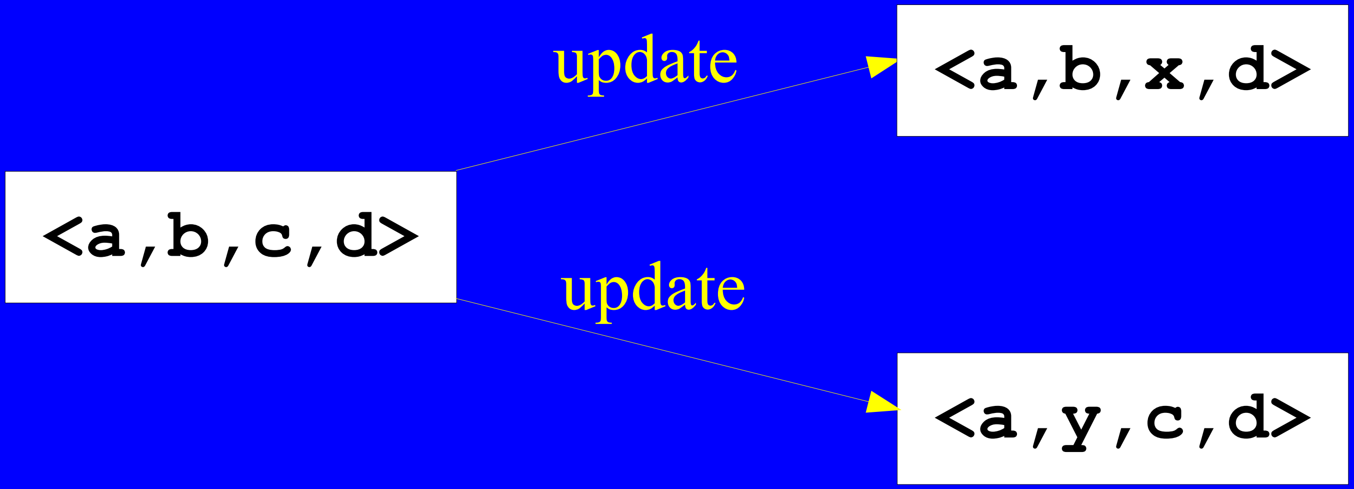
> The IO type is the most important monad type, but there are others

The Monad class

- Haskell has a Monad class
- It allows any type to be treated as a monad by defining the **`return`** function and the **`>>`** and **`>>=`** operators
- The **`do`** and **`<-`** notations then come for free
- The only types in the standard prelude which are in the Monad class are lists and Maybe
- These are probably not interesting or useful unless you have a special interest in monads or the monadic programming style

Arrays

> Arrays are potentially a problem in Haskell: suppose non-sequential operations are performed:



> Then each update operation appears to need a copy of the array to be made, in case the original is re-used

`Standard Arrays`

> The standard array type in **`Data.Array`** allows arbitrary operations, and makes copies

> Programmers are encouraged not to build arrays by updating one entry at a time, but by using functions provided to build an array holistically, e.g. by accumulation, all in one go

> In other words, they are intended for fast lookup, not for fast update in the procedural style

> If you use incremental updates to individual entries as in other languages, you will probably cause array copying which is very expensive

A Monadic Array type?

- You could imagine defining an array type which is monadic, so that updates on the array are forced to be sequential, and so the array space can be re-used in the same way as in procedural programs
- There is a problem though: what happens if you have an algorithm involving two arrays?
- Somehow, you need "the current state" to contain all arrays (and anything else) that the algorithm needs

The ST State Monad

- Haskell defines a general monad type called ST which consists of a a collection of named references to stateful values (and there are other state monads)
- Monadic arrays using the type **`STArray`** are defined to work within this general state monad as named items
- Beware: there are various different array libraries out there with different properties
- We will concentrate on the **`STArray`** type in the library **`Data.Array.ST`**

The Shuffling problem

> Let's use the shuffling problem as an example
> If you shuffle a list in Haskell using normal functions, you can't do better than O(n*log(n))
> One algorithm (assign a random number to each item in the list and then sort, taking care that duplicates don't spoil the randomness) illustrates why
> But shuffling can be done in O(n) time using arrays
> By the way, many computer scientists are beginning to think that O(1) array access is actually 'cheating' and that array access is 'really' O(log(n)))

The Fisher-Yates Shuffle

- An O(n) array algorithm is Fisher-Yates:
  - store the n items in an array a, indexed by 0..n-1
  - for i from n-1 down to 1 do
    - choose a random number r in the range 0..i
    - swap a[r] with a[i]
- Let's implement this in Haskell using STArray

Importing

> The imports we need are:

```
import Control.Monad.ST
import Data.Array.ST
import System.Random
```

> The first is the ST state monad type
> The second is the STArray type
> The third is for random numbers

The Inner Loop

> The inner loop is:

```
-- Shuffle items 0..i in a using gen
loop :: Int -> STArray s Int t ->
        StdGen -> ST s [t]
loop 0 a gen = getElems a
loop i a gen = do
    let (r, ngen) = randomR (0,i) gen
    x <- readArray a r
    y <- readArray a i
    writeArray a r y
    writeArray a i x
    loop (i-1) a ngen
```

Loop Types

> Actions `readArray, writeArray` etc. are overloaded to work on various different array types, so the type declaration for loop is necessary to tell Haskell which one we are using

> The type of a stateful action is `ST s a` where `s` is the type of the hidden state (which we aren't allowed to know, so it has to be left as a type variable) and `a` is the type of the result of the computation

> The type of the array is `STArray s Int t` where `s` is the state within which the array must live to enforce linear operations, `Int` is the type of the indexes, and `t` is the type of the elements

Loop Notes

- If you compare the loop with other Haskell shuffle implementations you may come across, one difference is in the handling of the random number generator
- Our implementation uses it in the purely functional style, but it can be regarded as having a state, and there is a monadic approach that can be used to avoid having to mention the state explicitly
- And a monadic random number generator can be mixed in with monadic arrays
- But it complicates the code, especially the types

Starting off

> To start the loop, we need another function which takes a list and a generator and creates the initial array

```
-- Create the shuffle action
doShuffle :: [t] -> StdGen -> ST s [t]
doShuffle xs gen = do
    let n = length xs
    arr <- newListArray (0, n-1) xs
    loop (n-1) arr gen
```

> The `let` keyword denotes a normal computation, whereas `<-` denotes an action to be performed

Wrapping

> We can wrap the action to form a pure list function

```
-- Shuffle a list
shuffle :: [t] -> [t]
shuffle xs =
  runST (doShuffle xs (mkStdGen 42))
```

> This uses a fixed generator, so will produce the same result every time we shuffle the same list (we would need an I/O based generator to avoid that)

> The **`runST`** function runs an action, starting with an empty state (which we can't do with I/O because it is too dangerous to have access to a real computer state)

Opinions

> Opinions differ on monadic programming in Haskell

> The theory it is based on is abstract, especially the types, so it is usually regarded as an advanced topic

> In some ways all it allows you to do is to write procedurally, as in other languages, but using an unfamiliar and somewhat restrictive syntax

> But, it allows I/O and fast arrays and other facilities to be expressed in a pure functional style, without breaking the fundamental principles

> And it is under very tight control, allowing analysis such as proofs