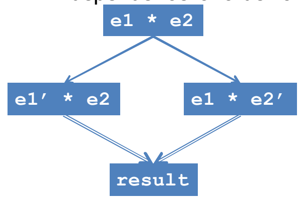
Haskell - part 8

Input/output

Beauty...

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 math function
  - Equational reasoning: if x = y, then fx = fy
  - Independence of order-of-evaluation (Church-Rosser)



The compiler can choose the best order in which to do evaluation, including skipping a term if it is not needed.

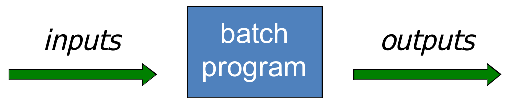
...and the Beast

- But to be *useful* as well as *beautiful*, a language must manage:
  - Input/Output
  - Imperative update
  - Error recovery (eg, timing out, catching divide by zero, etc.)
  - Foreign-language interfaces
  - Concurrency

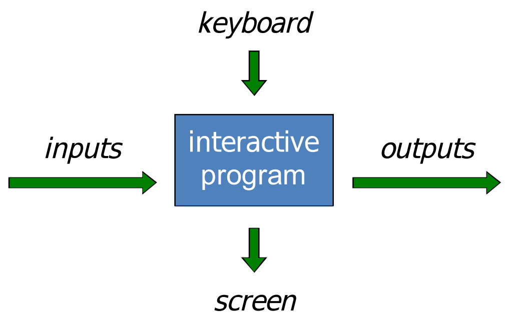
The whole point of a running a program is to affect the real world, an "update in place."

Introduction

To date, we have seen how Haskell can be used to write <u>batch</u> programs that take all their inputs at the start and give all their outputs at the end.



However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.



The Problem

Haskell programs are pure mathematical functions:

- Haskell programs have no side effects.

However, reading from the keyboard and writing to the screen are side effects:

- Interactive programs have side effects.

lazyness

In a lazy functional language, like Haskell, order of evaluation is deliberately undefined.

- `putchar ‘x’ + putchar ‘y’`
  Output depends on evaluation order of (+)
- `[putchar ‘x’, putchar ‘y]`
  Output (if any) depends on how the consumer evaluates the list
  e.g. **`length`** does not evaluate the elements!

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

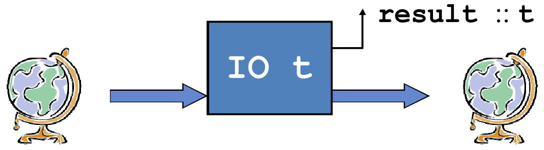
`IO t`

The type of actions that return a value of type t.

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)
```



Actions are first class

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

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

- "Actions" sometimes called "computations"
- An action is a first class value
- Evaluating an action has no effect; performing the action has an effect

For example:

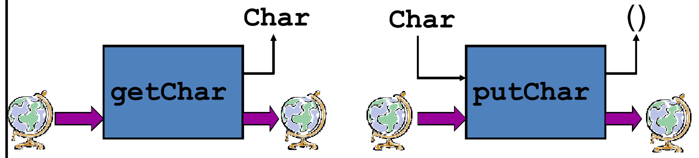
`IO Char` — The type of actions that return a character.

`IO ()` — The type of purely side effecting actions that return no result value.

Note:

- () is the type of tuples with no components.

Simple I/O



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

Basic Actions

- The action **getChar** reads a character from the keyboard, and returns the character as its result value, as an `IO char`:

```
getChar :: IO Char
```

- The action putChar c writes the character c to the screen, and returns no result value:

```
putChar :: Char → IO ()
```

13

How to execute an action?

- A special function `main` is an action of type `IO ()`

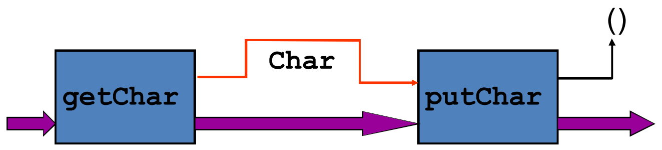
Main.hs
```
main :: IO ()
main = putChar 'x'
```

```
$ ghci
Prelude> :load Main.hs
*Main> main
x
```

```
$ runhaskell Main.hs
x
```

```
$ ghc --make Main.hs
$ ./Main
x
```

Connecting actions up

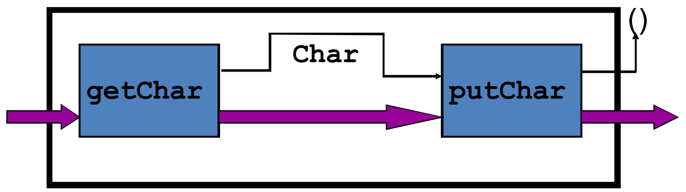


Goal: read a character and then write it back out

16

The (>>=) combinator

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



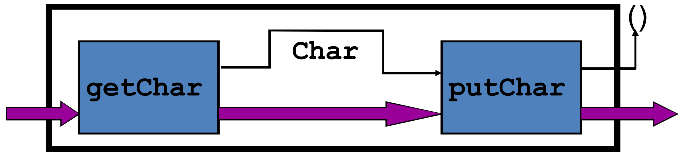
- We have connected two actions to make a new, bigger action.

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

17

The (>>=) combinator

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



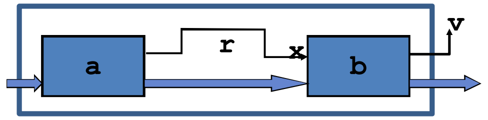
```
echo :: IO ()
echo = getChar >>= putChar
```

or

```
echo = getChar >>= (\c -> putChar c)
```

The (>>=) Combinator

- 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`



Printing a character twice

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

- The parentheses are optional

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

The (>>) combinator

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

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

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

m >> n  =  m >>= (\x -> n)
```

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

Getting two characters

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

- We want to return (c1,c2).
  - But, (c1,c2) :: (Char, Char)
  - And we need to return something of type IO(Char, Char)
- We need to have some way to convert values of "plain" type into the I/O Monad.

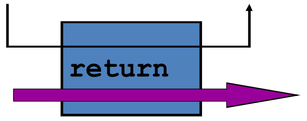
22

The return combinator

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

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

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



23

Notational convenience

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

- By design, the layout looks imperative

```
c1 = getchar();
c2 = getchar();
return (c1,c2);
```

24

Notational convenience

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

"do" notation adds only syntactic sugar

25

The "do" Notation

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

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

```haskell
-- 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:

```haskell
do { x<-e;es}    =   e>>= \x -> do {es}

do { e;es}       =   e>> do {es}

do { e }         =   e

do {let ds; es} =   let ds in do {es}
```

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

The last item in a "do" expression must be an expression.

Syntactic Variations

- The following are equivalent:

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

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

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

If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.

Getting a line

```haskell
getLine :: IO String
getLine  = do x <- getChar
              if x == '\n' then
                 return []
              else
                 do xs <- getLine
                    return (x:xs)
```

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

30

<-

- We have seen the <- symbol before:

```
[ x | x <- [1..10], even x ]
```

- The symbol is pronounced "drawn from"

```
main = do
  input <- getLine
  putStrLn ("you wrote: " ++ input)
```

- *“*`input` *is drawn from* `getLine`*”*
- *“type* `String` *is drawn from* `IO String`*”*

```
getLine :: IO String
```

31

- Writing a string to the screen:

```
putStr        :: String → IO ()
putStr []      = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

- Writing a string and moving to a new line:

```
putStrLn    :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

32

Example

We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen  = do putStr "Enter a string: "
             xs <- getLine
             putStr "The string has "
             putStr (show (length xs))
             putStrLn " characters"
```

33

For example:

```
> strlen

Enter a string: abcde
The string has 5 characters
```

Note:

Evaluating an action executes its side effects, with the final result value being discarded.

34

Control structures

Values of type `(IO t)` are first class
So we can define our own “control structures”

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

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

e.g. `repeatN 10 (putChar ‘x’)`

35

Loops

Values of type `(IO t)` are first class
So we can define our own “control structures”

```
for :: [a] -> (a -> IO b) -> IO ()
for []     fa = return ()
for (x:xs) fa = fa x >> for xs fa
```

e.g.

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

36

Loops

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)
                     }

for :: [a] -> (a -> IO b) -> IO ()
for xs fa = sequence (map fa xs) >>
            return ()
```

37

sequence

- **sequence** is pre-defined. Evaluates each IO action from a list of actions and returns a list of IO outputs

```
Prelude> sequence [getLine, getLine]
hello
world
["hello","world"]
```

- **print** is equivalent to **putStrLn.show**

```
Prelude> sequence (map print [1,2])
1
2
[(),()]  ???
```

38

GHCi

- prints the result of any expression

```
Prelude> 3+2
5
```

- … unless it is `IO ()`

```
Prelude> print (3+2)
5
```

```
Prelude> ()
()
```

```
Prelude> return ()
```

GHC

- compiled code prints only what is explicitly printed

Main.hs

```
main = sequence (map
print [1,2])
```

```
$ ghc —make Main
$ ./Main
1
2
```

39

Example

- Write a program that asks if the user wants to quit and, if the answer is not 'y', loops

```
main = do
  putStrLn "quit the program? y/n"
  ans <- getLine
  if ans /= "y" then do
    putStrLn "not quitting"
    main
  else ???
```

40

`return` and `<-`

- `return` is a function that "makes an I/O action out of a pure value"
- resembles the opposite of the `<-` syntax.

```
main = do
  input <- return "hello"
  putStrLn input
```

- `return` packs up a value into an IO box.
- `<-` extracts the value out of an IO box.
- but remember that `return` is a function, while `<-` is just a syntactic sugar: you can NEVER extract something from an IO action!
- real meaning:

```
do { x<-e;es}   =   e >>= \x -> do {es}
```

41

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, using the library System.IO

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

```
Prelude> :type hPutStr
Not in scope: `hPutStr'
Prelude> import System.IO
Prelude System.IO> :type hPutStr
hPutStr :: Handle -> String -> IO ()
```

openFile

```
openFile :: FilePath ->IOMode -> IO Handle
type FilePath = String
```

`IOMode`	Can read?	Can write?	Starting position	Notes
`ReadMode`	yes	no	Beginning of file	File must exist already
`WriteMode`	no	yes	Beginning of file	File is truncated (completely emptied) if it already existed.
`ReadWriteMode`	yes	yes	Beginning of file	File is created if it didn't exist; otherwise, existing data is left intact.
`AppendMode`	no	yes	End of file	File is created if it didn't exist; otherwise, existing data is left intact.

Lazy I/O

```
hGetContents :: Handle -> IO String
```

- logically, reads the whole file to RAM
- in practice, reads it lazily, only when needed

"At the moment you call hGetContents, nothing is actually read. Data is only read from the Handle as the elements (characters) of the list are processed. As elements of the String are no longer used, Haskell's garbage collector automatically frees that memory. All of this happens completely transparently to you. And since you have what looks like (and, really, is) a pure String, you can pass it to pure (non- IO) code."

from "Real World Haskell" by B. O'Sullivan, D. Stewart, and J. Goerzen

… but if the elements are used later on, the memory cannot be garbage collected!

Convert a file to uppercase

```
import System.IO
import Data.Char
main = do
   inh <- openFile "input.txt" ReadMode
   outh <- openFile "output.txt" WriteMode
   inpStr <- hGetContents inh
   let result = processData inpStr
   hPutStr outh result
   hClose inh
   hClose outh
processData = map toUpper
```

If we had used `inpStr` past the place where it was used (the call to `processData`), the program would have lost its memory efficiency. That's because the compiler would have been forced to keep `inpStr`'s value in memory for future use. Here it knows that `inpStr` will never be reused and frees the memory as soon as it is done with it.

hGetContents

- You are not required to consume all the data from the input file when using **`hGetContents`**.
- Whenever the Haskell system determines that the entire string **`hGetContents`** returned can be garbage collected, the out file is closed automatically.
- The same principle applies to data read from the file. Whenever a given piece of data will never again be needed, the Haskell environment releases the memory it was stored within.
- Strictly speaking, we wouldn't have to call **`hClose`** at all in this example program. However, it is still a good practice, as later changes to a program could make the call to **`hClose`** important.

readFile and writeFile

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

- `readFile` and `writeFile` are shortcuts for working with files as strings. They handle all the details of opening files, closing files, reading data, and writing data.
- `readFile` uses `hGetContents` internally.

```
import Data.Char(toUpper)
main = do
   inpStr <- readFile "input.txt"
   writeFile "output.txt" (map toUpper inpStr)
```

readFile and writeFile

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

- `readFile` and `writeFile` do not provide a Handle ⇒ nothing to `hClose`.
- `readFile` uses `hGetContents` internally. The underlying Handle is closed when the returned String is garbage-collected or all the input has been consumed.
- `writeFile` closes its underlying Handle when the entire String supplied to it has been written.

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 ...
```

- 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.
- Within the program, there is no way out!

Useful functions when dealing with text files

- These are not functions in the IO monad, but they are useful to read data from text files
- `lines :: String -> [String]`

```
>lines "This is a\nlong text\nindeed!"
["This is a","long text","indeed!"]
```

- `unlines :: [String] -> String`

```
>unlines ["various","words","together"]
"various\nwords\ntogether\n"
```

- `words :: String -> [String]`

```
> words "This is a\nlong text,indeed!"
["This","is","a","long","text,indeed!"]
```

Converting from string

- `Show` is the typeclass of the types that can be converted to String
- members implements function `show`
- `Read` is the typeclass of the types that can be read from String
- There is an indirect function `read` (cannot be reimplemented directly; one should reimplement readsPrec, which is a parser)

type can be inferred

```
>7+read "20"
27
```

```
>read "20"
ERROR
```

type cannot be inferred

```
>read "20" :: Int
20
```

converting numeric formats

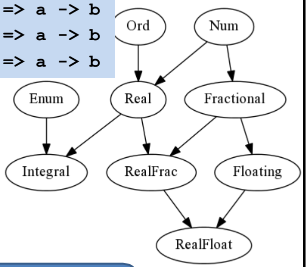
```
ceiling  :: (RealFrac a, Integral b) => a -> b
floor    :: (RealFrac a, Integral b) => a -> b
truncate :: (RealFrac a, Integral b) => a -> b
round    :: (RealFrac a, Integral b) => a -> b
```

Floating

Integral

```
>(sqrt 2) + floor (3/2)
ERROR
```

No automatic conversion! Either they can be interpreted as the same type, or it does not compile

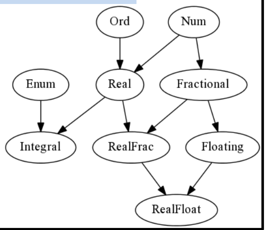


converting numeric formats

- `Int` and `Integer` are `Integral`

```
fromIntegral :: (Num b, Integral a) => a -> b
```

```
>(sqrt 2) + fromIntegral (floor (3/2))
2.414213562373095
```



Hangman

Consider the following version of hangman:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a sequence of guesses.
- For each guess, the computer indicates which letters in the secret word occur in the guess.

- The game ends when the guess is correct.

We adopt a top down approach to implementing hangman in Haskell, starting as follows:

```
hangman :: IO ()
hangman  =
   do putStrLn "Think of a word: "
      word ← sgetLine
      putStrLn "Try to guess it:"
      play word
```

55

The action sgetLine reads a line of text from the keyboard, echoing each character as a dash:

```
sgetLine :: IO String
sgetLine  = do x ← getCh
               if x == '\n' then
                  do putChar x
                     return []
               else
                  do putChar '-'
                     xs ← sgetLine
                     return (x:xs)
```

56

The action getCh reads a single character from the keyboard, without echoing it to the screen:

```
import System.IO

getCh :: IO Char
getCh  = do hSetEcho stdin False
            x <- getChar
            hSetEcho stdin True
            return x
```

57

The function play is the main loop, which requests and processes guesses until the game ends.

```
play     :: String → IO ()
play word =
   do putStr "? "
      guess <- getLine
      if guess == word then
         putStrLn "You got it!"
      else
         do putStrLn (match word guess)
            play word
```

58

The function match indicates which characters in one string occur in a second string:

```
match :: String → String → String
match xs ys =
   [if elem x ys then x else '-' | x ← xs]
```

For example:

```
> match "haskell" "pascal"

"-as--ll"
```

59

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.

A Monadic Skin

- 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.

Exercise

Implement the game of nim in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

62

- Two players take it turn about to remove one or more stars from the end of a single row.
- The winner is the player who removes the last star or stars from the board.

Hint:

Represent the board as a list of five integers that give the number of stars remaining on each row. For example, the initial board is [5,4,3,2,1].

IO programs that reverse text

- given a file containing some text, create an output file with **everything** in reverse

input.txt
```
hello world
```
output.txt
```
dlrow olleh
```

- given a file containing some text, create an output file with **every word** in reverse

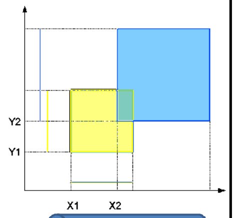
input.txt
```
hello world
```
output.txt
```
olleh dlrow
```

Esercizio

Un programma Haskell deve controllare le collisioni fra quadrati nel piano. Ogni quadrato è rappresentato da:

- identificatore: Int
- colore: stringa (senza spazi)
- X, Y, lato: Int

Il programma deve leggere un file di testo e visualizzare il numero di collisioni che ci sono fra i quadrati, ossia il numero di coppie di quadrati che hanno intersezione non nulla



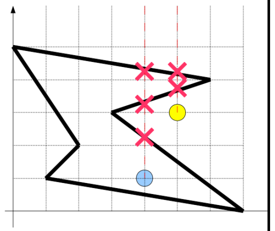
Si utilizzino funzioni di ordine superiore

http://www.unife.it/ing/informazione/fondamenti-info-1/materiale-didattico/testi-dei-compiti/quadrati.txt

Poligono

Un file di testo poligono.txt contiene le coordinate dei punti che rappresentano i vertici di un poligono; per ogni punto si hanno due coordinate intere: x e y. Nel file, l'ultimo punto coincide con il primo Scrivere un programma che legge da tastiera le coordinate di un ulteriore punto e comunica all'utente se è interno o esterno al poligono.

poligono.txt
```
1 1
2 2
0 5
6 4
3 3
7 0
1 1
```



Per verificare se un punto P è interno al poligono:

- si traccia una semiretta a partire dal punto P
- si calcola quanti lati del poligono intersecano tale semiretta
- se il numero delle intersezioni è pari,
  - allora il punto è esterno al poligono,
  - altrimenti (se è dispari) è interno

http://www.unife.it/ing/informazione/fondamenti-info-1/materiale-didattico/testi-dei-compiti/poligono.txt