**Input/output**

* To date, we have seen how Haskell can be used to write batch 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, When a function is applied, it computes and returns a result, but nothing else happens. The are no side effects.
* Good for modularity! Makes debugging and testing easier!
* However, reading from the keyboard and writing to the screen are side effects: Interactive programs have side effects.

Solution

* One way to avoid these side effects is to model input and output as streams
* Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects.

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

Example IO Char

Standalone Haskell programs

* The program should have a module called Main, containing a function called main:

module Main where

main :: IO ()

main = (...)

* The first line can be omitted, since the default module name is Main.

**Creating IO actions**

* The type  **IO** () represents "IO actions".
* Here are some functions (some may be familiar already) that return IO actions:

putStr :: String -> IO ()

print :: Show a => a -> IO ()

interact :: (String->String) -> IO ()

* The standard library provides a number of actions, including the following three primitives:
* The action getChar reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

getChar :: IO Char

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

putChar :: Char → IO ()

* The action return v simply returns the value v, without performing any interaction:

return :: a → IO a

* Sequencing:A sequence of actions can be combined as a single composite action using the keyword do. The do notation allows us to use the left arrow <- in the same way inside function definitions.

Example:

showTheDifference :: IO ()

showTheDifference = do putStrLn "Enter two numbers:"

x <- readLn

y <- readLn

putStr "The difference is: "

print (x-y)

We used these functions:

readLn :: Read a => IO a

print :: Show a => a -> IO ()

putStr :: String -> IO ()

putStrLn :: String -> IO ()

For example: act :: IO (Char,Char)

act = do x ← getChar

getChar

y ← getChar

return (x,y)

Doing IO and returning results

What if we want a function that returns the difference instead of showing it?

**getTheDifference :: IO Integer**

**getTheDifference = do x <- readLn**

**y <- readLn**

**return (x-y)**

**showTheDifference :: IO ()**

**showTheDifference = do d <- getTheDifference**

**putStr "The difference is: "**

**print d**

return :: a -> IO a

Why pure functions?

Purity is good for modularity and simplifies debugging and testing.

* Unexpected side effects is a major source of bugs.
* Automated testing with QuickCheck is an example where it is a good that functions are pure!
* Subexpressions can be computed in any order, even in parallel, without changing the result of the program.

Import module

import Data.List

Good Resources for Haskell

* <http://learnyouahaskell.com/chapters>
* [http://www.cse.chalmers.se/edu/year/2018/course/TDA452](http://www.cse.chalmers.se/edu/year/2018/course/TDA452/)

take 5([1..])

take 1 [1]

take 2 1,[1,2..]