Side Effects and Haskell

- Haskell is **pure:**
  - Evaluating expressions has **no side-effects**
  - Expressions are evaluated only for obtaining their **values**
- But sometimes we want our programs to affect the real world (printing, controlling a robot, drawing a picture, etc).

How do we reconcile these two aspects?

In Haskell, certain "pure values" are "worldly actions" that can be ***performed***

– **Types:** An expression with type *IO a* has as its value a **computation** (in the *IO*-***monad***) that can be understood as returning a value of type *a*.

Alternative explanation: An expression with type *IO a* has possible ***actions*** associated with its execution, while returning a value of type *a*

– **Syntax:** The **do** syntax sequences several actions (using layout)

The do Syntax

```
main = do                                          –– Users.hs
  s ← readFile "/etc/passwd"
  putStrLn $ "/etc/passwd has " ⧺ show (length s) ⧺ " characters"
  let logins = map (takeWhile (':' ≠)) $ lines s
  putStrLn $ "There are " ⧺ show (length logins) ⧺ " logins"
  let funny = filter (all (‘notElem‘ "AEIOUaeiou")) logins
  putStrLn $ unwords $ "Funny logins:" : funny
```

- *readFile* "/etc/passwd" :: *IO String* is an action.
- We use the **do** syntax to bind the result of that action to the variable *s*, and sequence this action with other actions that depend on *s*.
- Inside **do**, one may write **let** without **in**.

When IO Actions are Performed

> An expression with type *IO a* has as its value a **computation** that, when performed, may return a value of type *a*.

- A value of type *IO a* is an **action**, but it is still a *value:* it will **only** have an **effect** when it is **performed**.
- In Haskell, a program's value is the value of the variable *main* in the module *Main*.

  That value has to have type *IO a*.

  It will be **performed** upon execution of the program.
- In Hugs and GHCi, you can type any expression to the prompt.

  If the expression has type *IO a* it will be performed; otherwise its value will be printed on the display.

Predefined IO Actions

```
–– write a string to terminal (without/with adding a newline)
putStr, putStrLn :: String → IO ()

putChar :: Char → IO ()                    –– write one character to terminal
getChar :: IO Char                         –– get one character from keyboard
getLine :: IO String                       –– get a whole line from keyboard
readFile :: FilePath → IO String           –– read a file as a String
writeFile :: FilePath → String → IO ()     –– write a String to a file
```

With "**import** *System*":

```
getArgs     :: IO [String]                 –– obtain command-line arguments
getProgName :: IO String                   –– obtain program name
getEnv      :: String → IO String          –– get environment variable value
system      :: String → IO ExitCode        –– run command
```

IO Example

```
import qualified System                         −− Cat.hs

main = do
  args ← System.getArgs
  putStrLn (shows (length args) " arguments")
  let (flags, files) = span (("-"≡) ∘ take 1) args
  print flags
  mapM (λ file → readFile file >>= putStrLn) files
```

Compile and run:

```
ghc --make -o Cat Cat.hs
./Cat -flag1 -q -v -flag4 file1 qwerty -what file4
```

Another *IO* Example

```
module Main ( main ) where −− this is the default module header --- WC.hs

main = do
 line ← getLine
 let ws = words line
 case ws of
   [ ] → return ()
   _ → do
     putStrLn ("You entered " ++ show (length ws) ++ " words")
     main
```

Compile and run:

```
ghc --make -o WC WC.hs
./WC
```

Adding Line Numbers

```
module Main ( main ) where                 −− WC2.hs

main = count 1

count :: Integer → IO ()
count n = do
 line ← getLine
 let ws = words line
 case ws of
   [ ] → return ()
   _ → do
     putStrLn ("Line " ++ show n ++ " has " ++ show (length ws) ++ " words")
     count (n + 1)
```

The “state” is managed as argument of a **parameterised action.**

Catching I/O Exceptions

catch is not a keyword, but a prelude function:

```
catch :: IO a → (IOError → IO a) → IO a
```

Example:

```
main = do                                            −− Catch.hs
   s1 ← catch (readFile "infile1")
              (λ e → do
                 putStrLn $ "Error reading infile1: " ++ show e
                 return "")
   s2 ← readFile "infile2"
        ‘catch‘ λ e → do
                 putStrLn $ "Error reading infile2: " ++ show e
                 return ""
   writeFile "outfile" (s1 ++ s2)
     ‘catch‘ λ e → putStrLn $ "Error writing outfile: " ++ show e
   putStrLn "Finished"
```

Recursive Actions with Results — *getLine*

getLine can be defined recursively in terms of simpler actions:

```
getLine :: IO String
getLine =
 do c <- getChar                 –– get a character
    if c == ’\n’                 –– if it’s a newline
       then return ""            –– then return empty string
       else do l <- getLine      –– otherwise get rest of
                                 ––    line recursively,
               return (c:l)      –– and return entire line
```

The function *return* $:: a \to IO\ a$ takes a value of type a, and turns it into an action of type $IO\ a$, which does nothing but return the value.