Input/Output

ECS713: Functional Programming
Week 06

Prof. Edmund Robinson
Dr Paulo Oliva

Week 6: Contents

- The IO type constructor
- Pure vs impure functional programming
- Interactive programs
- Streams
- Reading/writing files
- Recursive descent parsing

Learning Outcomes

- Distinguish between purely functional programs and programs that have side-effects
- Understand I/O Actions, be able to use the main actions putStr, getLine, etc.
- Understand concept of a stream, and be able to read from files using streams

The IO Type Constructor

The IO Type Constructor

IO type is used for operations that interact with the “outside world”

```
Prelude> :type print
print :: Show a => a -> IO ()
Prelude> :type readFile
readFile :: FilePath -> IO String
Prelude> :type getLine
getLine :: IO String
```

given something that can be “shown”, shows it on screen and returns ()

given a file path, reads contents of the file and returns this content

reads one line of user input and returns that list of characters

like void in C ==> no return value

return string

everithing like io string, connection, has to....

The “main” function

```
Prelude> :type print
String -> IO ()
```

```
-- File “hello.hs”

main :: IO ()
main = print “Hello World”
```

```
$ ghc hello.hs
[1 of 1] Compiling Main          ( hello.hs, hello.o )
Linking hello ...
$ ./hello
“Hello World!”
```

The “main” function

```
-- File “friend.hs”

main :: IO ()
main = do
  print “Hi, what’s your name?”
  name <- getLine
  print $ “Dear “ ++ name ++ “, do you wanna be my friend?”
```

CIN in C

get charaacther

```
$ ghc friend.hs
[1 of 1] Compiling Main             ( friend.hs, friend.o )
Linking friend ...
$ ./friend
“Hi, what’s your name?”
Paulo
“Dear Paulo, do you wanna be my friend?”
```

I/O Actions

Pure versus Impure

Pure	**Impure**
Definitions	*Commands* enscapsulated
Stateless != memory, no alocating memory	*State (e.g. global variables)* marking position of memory - pointer of memory
No side effects	*Side effects*
Easy to reason about program	*Easy to interact with outside world*

Hello, world!

```haskell
-- file: week06.hs

main = putStrLn "Hello, world!"
```

```
$ runghc week06.hs
Hello, world!
```

```
$ ghc week06.hs
[1 of 1] Compiling Main      (week06.hs, week06.o)
Linking week06...
$ ./week06
Hello, world!
```

putStrLn

```
Prelude> :type putStrLn
putStrLn :: String -> IO ()
Prelude> let x = putStrLn "coffee"
Prelude> :type x
IO ()
Prelude> x
coffee
```

‘x’ above is an IO action
let’s see what these are...

I/O Actions

`IO is a (special) type constructor`

`IO a`

An IO action of type "a". When performed will carry out an action with side-effect and present a result of type a

For instance, we could have:

the "unit" type

`IO Int` `IO Bool` `IO [Char]` `IO ()`

The unit type ( )

() is the type containing only one element, namely ()

```
Prelude> :type ()
() :: ()
```

It plays the role of "void" in languages such as C and Java

Hence, a function that is only meant to do an IO action but not return any value will have return type "IO ()"

```
Prelude> :type print
print :: Show a => a -> IO ()
```

expect output
sometihing

just do something

getLine

```
Prelude> :type getLine
getLine :: IO String
Prelude> let x = getLine
Prelude> :type x
x :: IO String
Prelude> x
Coffee
“Coffee”
```

‘x’ above is an IO action of type String

When performed it asks for a user input and then returns the input string

I/O Actions

```
Prelude> let x = putStrLn "Hello"
Prelude> :type x
x :: IO ()
Prelude> x
Hello
Prelude> x
Hello
Prelu
<inte
    Co
    ...
```

IO actions need to be "performed" before returning a value

IO actions are performed when:
- called inside a function of type "IO a" as
  x <- action
- called in the prompt of ghci

Greeting Example

```
-- file: week06.hs
main = do
       putStrLn "Greetings!  What is your name?"
       inpStr <- getLine
       putStrLn $ "Welcome to Haskell, " ++ inpStr ++ "!"
```

"do" glues IO actions together

type of the whole action is the type of the last action

```
$ runghc week06.hs
Greetings!  What is your name?
Paulo
Welcome to Haskell, Paulo!
```

I/O Actions

1. perform this action

```
inpStr <- getLine
```

2. bind this variable to returning value

The "do" block automatically extracts the value of the last action and returns that as its own result

In-class Exercise

ghci is big do
so it allows you using let
directly

```
-- Q1: Would this work?

main = do
       foo <- putStr "What's your name?"
       name <- getLine
       putStr $ "Nice to meet you, " ++ name

-- Q2: What is the type of foo?
```

A1: Yes it would work.
A2: foo has type (), which is the unit type

Using "let" in do blocks

```haskell
import Data.Char

main = do
       putStrLn "What is your name?"
       name <- getLine
       let bigName = map toUpper name
       putStrLn $ "Hey " ++ bigName ++ ", how are you?!"
```

use "<-" to perform an action and bind the result

use "let" to bind pure values

```
$ runghc week06.hs
What is your name?
Paulo
Hey PAULO, how are you?!
```

Let and Return

```
main = do
     a <- return "nice"
     b <- return "work"
     putStrLn $ a ++ " " ++ b
```

does sometihing, not return

Use "return" to turn something pure into an IO action

```
main = do
     let a = "nice"
         b = "work"
     putStrLn $ a ++ " " ++ b
```

So, these two programs have the same behaviour

```
Prelude> let x = return "Coffee" :: IO String
Prelude> :type x
IO String
Prelude> x
"Coffee"
```

Other I/O Functions

```
-- same as putStrLn but without a “new line”
putStr :: String -> IO ()
```

```
-- print a character
putChar :: Char -> IO ()
```

```
-- print a “showable” value (print = putStrLn . show)
print :: Show a => a -> IO ()
```

Interactive Programs

Interactive Programs

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords = unwords . map reverse . words
```

use “return” to turn a pure value into an IO action

why do we need this “do”?

like bracket in java

make as 1 statement

```
$ runghc week06.hs
Let us do a test
teL su od a tset
Nice it works!
eciN ti !skrow
```

When ...

```
when :: Bool -> IO () -> IO ()
```

```
"when" takes a boolean and an IO action
if the boolean is true, run the IO action
if it is false run "return ()"
```

```haskell
import Control.Monad

main = do
   line <- getLine
   when (not $ null line) $ do
        putStrLn $ reverseWords line
        main

reverseWords = unwords . map reverse . words
```

Forever ...

```
forever :: IO a -> IO b
```

“forever” takes an IO action and performs that action forever

```
import Control.Monad

main = forever $ do
    line <- getLine
    when (not $ null line) (putStrLn $ reverseWords line)

reverseWords = unwords . map reverse . words
```

this program does NOT terminate!

An Interactive Program

```
import Control.Monad

main = do
   print $ “Choose action: [1] ... [2] ..."
   line <- getLine
   when (line==“1”) $ do
      action1
      main
   when (line==“2”) $ do
      action2
      main
```

Command Line Input

```
-- file: week06.hs

import System.Environment

main = do
     args <- getArgs
     progName <- getProgName
     putStrLn $ “First argument is:” ++ (args !! 0)
     putStrLn $ “The program name is:” ++ progName
```

```
getArgs :: IO [String]
getProgName :: IO String
```

```
$ ghc week06.hs
[1 of 1] Compiling Main
Linking week06 ...
$ ./week06 test
First argument is: test
The program name is: week06
```

Streams

Input Redirection

poem.txt

```
To be or not to be
that's the question
```

capslocker.hs

```
import Control.Monad
import Data.Char

main = forever $ do
       l <- getLine
       putStrLn $ map toUpper l
```

run forever

```
$ ghc capslocker.hs
[1 of 1] Compiling Main
Linking capslocker ...
$ ./capslocker < poem.txt
TO BE OR NOT TO BE
THAT'S THE QUESTION
capslocker <stdin>: hGetLine: end of file
```

getContents

only contents

poem.txt

```
To be or not to be
that's the question
```

capslocker.hs

```haskell
import Data.Char

main = do
      ct <- getContents
      putStrLn $ map toUpper ct
```

same type IO String as getLine

BUT LAZY!

```
$ ghc capslocker.hs
[1 of 1] Compiling Main
Linking capslocker ...
$ ./capslocker
This is a test
THIS IS A TEST
Very nice
VERY NICE
```

The “interact” function

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

poem.txt

```
To be or not to be
that's the question
```

week06.hs

```haskell
main = interact toUpperStr

toUpperStr = map toUpper
```

```haskell
toUpperStr = map toUpper
```

```
$ ghc week06.hs
[1 of 1] Compiling Main
Linking week06 ...
$ ./week06 < poem.txt
TO BE OR NOT TO BE
THAT'S THE QUESTION
```

readFile / writeFile

readFile / writeFile

```
import System.IO
import Data.Char

main = do
     contents <- readFile “poem.txt”
     writeFile “poem-cap.txt” (map toUpper contents)
```

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

poem.txt

```
To be or not to be
that’s the question
```

poem-cap.txt

```
TO BE OR NOT TO BE
THAT’S THE QUESTION
```

Files and Handles

openFile

```
Prelude> :module System.IO
Prelude System.IO> :type openFile
openFile :: FilePath -> IOMode -> IO Handle
```

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

```
type FilePath = String
```

			Beginning	File must exist
`WriteMode`	*No*	*Yes*	*Beginning*	File emptied if already exists
`ReadWriteMode`	*Yes*	*Yes*	*Beginning*	File created if didn't exist; otherwise existing data left intact
`AppendMode`	*No*	*Yes*	*End*	

Predefined Handles

```
Prelude> :module System.IO
                                  keyboard
Prelude System.IO> :type stdin
stdin :: Handle
                                  string
Prelude System.IO> :type stdout
stdout :: Handle
                                  error
Prelude System.IO> :type stderr
stderr :: Handle
```

Handles

```
-- file: week06.hs

import System.IO

main = do
      handle <- openFile “poem.txt” ReadMode
      contents <- hGetContents handle
      putStr contents
      hClose handle
```

```
getContents = hGetContents stdin
```

```
$ runghc week06.hs
To be or not to be
that’s the question
```

Working with Modules

Importing Modules

```haskell
import Data.List                      -- import all
```

Dealing with name clashes

```haskell
import Data.List (words, sort)        -- selective import
```

```haskell
import Data.List hiding (sort)        -- import all except sort
```

```haskell
import qualified Data.Map             -- call as Data.Map.filter
```

```haskell
import qualified Data.Map as M        -- call as M.filter
```

Creating Modules

```haskell
-- file: CrawlerDB.hs

module CrawlerDB ( printURLs ) where

import CrawlerHTTP
import Database.HDBC
import Database.HDBC.Sqlite3

printURLs :: IO ()
printURLs = do urls <- getURLs
               mapM_ print urls

getURLs :: IO [URL]
getURLs = do conn <- connectSqlite3 "urls.db"
             res <- quickQuery' conn "SELECT url FROM urls" []
             return $ map fromSql (map head res)
```

module name should be the same as file name

exported functions

function only used locally

stack

stack

- Cross-platform program for Haskell projects
- Tackle common build issues in Haskell
- Around since June of 2015
- A .cabal file for each package defines package-level metadata (stack uses cabal)
- A stack.yaml file provides information on where dependencies come from

stack

Creat new project:

```
$ stack new helloworld new-template
```

Configure new project: (cd into new folder)

```
$ stack setup
```

Build executable:

```
$ stack build
```

Run your program

```
$ stack exec helloworld-exe
```

.yaml vs .cabal

- A project can have multiple packages
- Each project has a stack.yaml
- Each package has a .cabal file
- The .cabal file specifies which packages are dependencies
- stack.yaml specifies which packages are available
- .cabal specifies the components, modules, and build flags provided by a package
- stack.yaml specifies which packages to include

stack.yaml

http://docs.haskellstack.org/en/stable/yaml_configuration/

```yaml
# file stack.yaml
...
# Resolver to choose a 'specific' stackage snapshot or a compiler version.
# A snapshot resolver dictates the compiler version and the set of packages
# to be used for project dependencies. For example:
#
# resolver: lts-3.5
# resolver: ghc-7.10.2
resolver: lts-3.5
...
# Packages to be pulled from upstream that are not in the resolver
extra-deps:
- HTTP-4000.3.3
```

helloworld.cabal

https://docs.haskellstack.org/en/stable/GUIDE/#stackyaml-vs-cabal-files

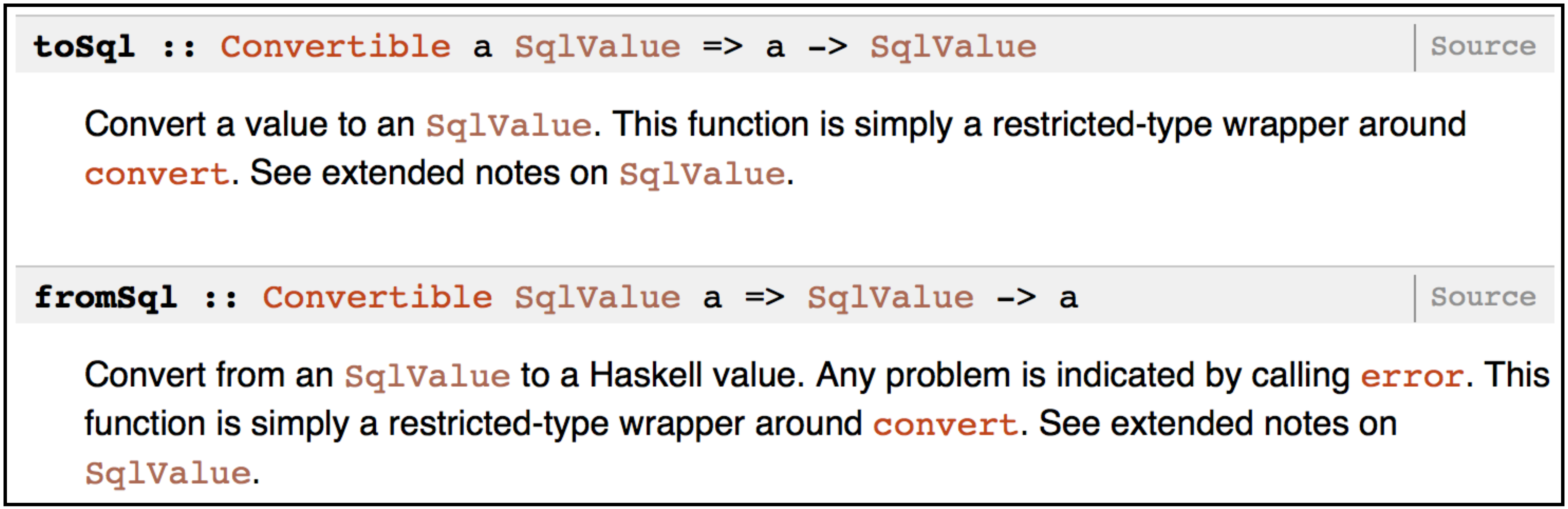
```
name:                   helloworld
version:                0.1.0.0
...
library
  hs-source-dirs:         src
  exposed-modules:        Lib
  build-depends:          base >= 4.7 && < 5
                        , text
                        , HTTP
  default-language:       Haskell2010
...
source-repository head
  type:     git
  location: https://github.com/githubuser/helloworld
```

stack on ITL

- Use Linux
- First time run
  $ stack-check
- Will create symbolic link
  /home/USER/.stack
  -> /import/scratch/ECS713P_stack
- From there on use stack as “normal”

Generate HTML documentation

hackage.haskell.org



Generated using the **haddock** tool
available in the Haskell Platform

http://hackage.haskell.org/package/HDBC-2.4.0.1/docs/src/Database-HDBC-SqlValue.html#toSql

```haskell
{- | Convert a value to an 'SqlValue'.  This function is simply
     a restricted-type wrapper around 'convert'.
     See extended notes on 'SqlValue'. -}
toSql :: Convertible a SqlValue => a -> SqlValue
toSql = convert

{- | Conversions to and from 'SqlValue's and standard Haskell types.
     This function converts from an 'SqlValue' to a Haskell value.
     Many people will use the simpler 'fromSql' instead. This function
     is simply a restricted-type wrapper around 'safeConvert'. -}
safeFromSql :: Convertible SqlValue a => SqlValue -> ConvertResult a
safeFromSql = safeConvert
```

http://hackage.haskell.org/package/HDBC-2.4.0.1/docs/Database-HDBC.html

toSql :: Convertible a SqlValue => a -> SqlValue   Source

Convert a value to an `SqlValue`. This function is simply a restricted-type wrapper around `convert`. See extended notes on `SqlValue`.

fromSql :: Convertible SqlValue a => SqlValue -> a   Source

Convert from an `SqlValue` to a Haskell value. Any problem is indicated by calling `error`. This function is simply a restricted-type wrapper around `convert`. See extended notes on `SqlValue`.

Using haddock

```haskell
import Package1
import Package2

module Package3 where

-- | This comment will be considered by haddock
myFunction :: Int -> IO String
myFunction n = ...
```

```
$ haddock package3.hs -h -o docs
Haddock coverage:
  50% ( 2 / 4) in ...
```

-h to generate html

-o gives destination

http://lambda.haskell.org/platform/doc/current/ghc-doc/haddock/

Recursive Descent Parsing

```
"Paul 12 John 55 Sarah 23"
```

```
Grammar in BNF
Name ::= [A-z]+
Age ::= [0-9]+
Item ::= Name " " Age
Input ::= Item (" " Item)*
```

```haskell
type Name = String
type Age = String
type Item = (Name, Age)
type Input = [Item]
```

```haskell
parseName :: String -> (Name, String)
parseName = span isAlpha
```

```haskell
parseAge :: String -> (Age, String)
parseAge = span isNumber
```

```haskell
parseItem :: String -> (Item, String)
parseItem xs = ((name, age), rest2)
  where (name, rest1) = parseName xs
        (age, rest2) = parseAge (tail rest1)
```

```haskell
parseInput :: String -> (Input, String)
parseInput xs = if r1==[] then ([y],r1) else (y:ys,r2)
   where (y, r1) = parseItem xs
         (ys, r2) = parseInput (tail r1)
```

References

- Learn you a Haskell for Great Good
  Miran Lipovača, Chapters 8 and 9
- Programming in Haskell
  Graham Hutton, Chapter 9
- Real World Haskell
  B. O'Sullivan et al, Chapter 7