Week 6: Input/Output in Haskell

ECS713
Functional Programming

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary
University of London

Core Haskell / Types / IO Actions / Haskell "stack" Tool / Parsing

Core Haskell: Working with modules

- import modules using the "import" keyword
- organise your code into modules using the "module" keyword

Types: The "unit" type ()

- understand the role the unit type plays when working with IO actionsWeek 6/Task 1

IO Actions: Pure vs impure programming

- define the notion of computational "side effect"
- understand the difference between "pure" (no side effects) and "impure" (with side effects) programs

IO Actions: The IO type function

- familar with the basic Haskell functions that have IO type: print, putStrLn, readLnWeek 6/Task 1
- understand how impure programming in Haskell is controlled via the IO type functionWeek 6/Task 1

IO Actions: The "do" notation

- understand how IO action can be combined using the "do" notationWeek 6/Task 1

IO Actions: Working with files

- read contents from a text file (readFile), and and write to a text file (writeFile)Week 6/Task 1

Haskell "stack" Tool: Basic stack usage

- create a new project using "stack new"
- build an executable using "stack build"

Parsing: Recursive descent parsing

- understand the concept of "recursive descent parsing" and how it can be used to parse data defined via context-free grammars

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 - basic usage

Create 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
```

stack - basic usage

Create new project (omit template for standard project):

```
$ stack new helloworld
```

Stack setup installs the appropriate ghc for the project.


```
$ stack setup
```

Build executable:

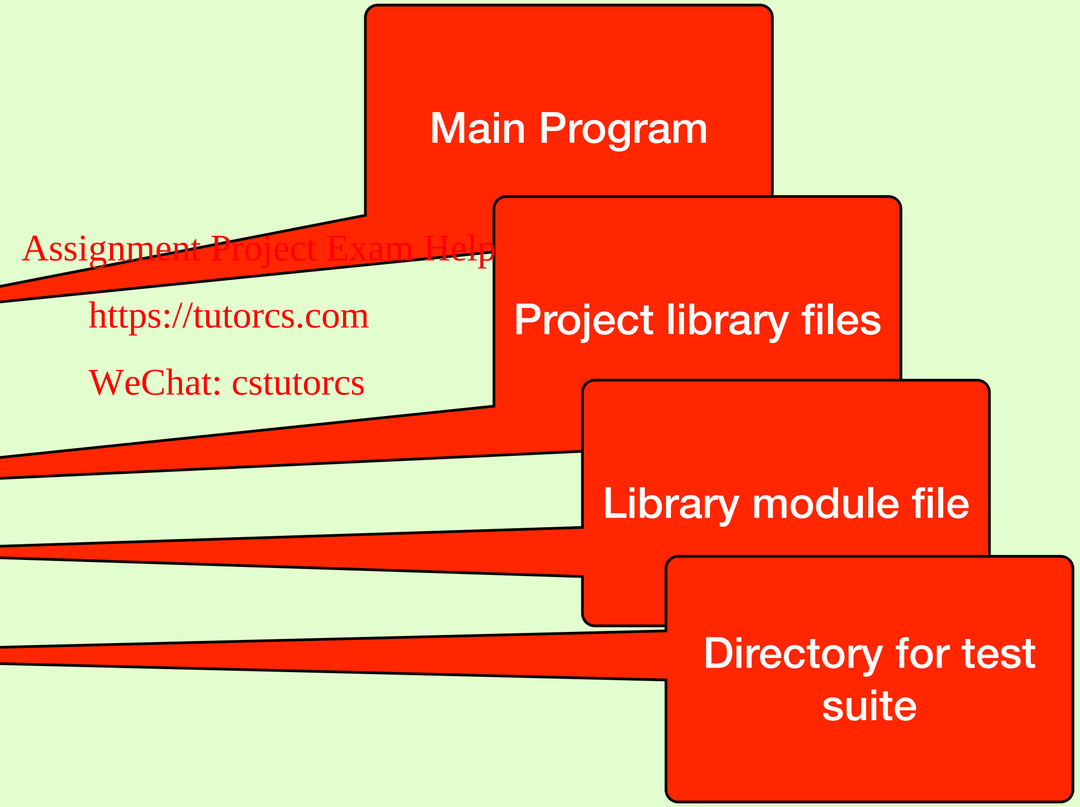
```
$ stack build
```

Run ghci

```
$ stack ghci
```

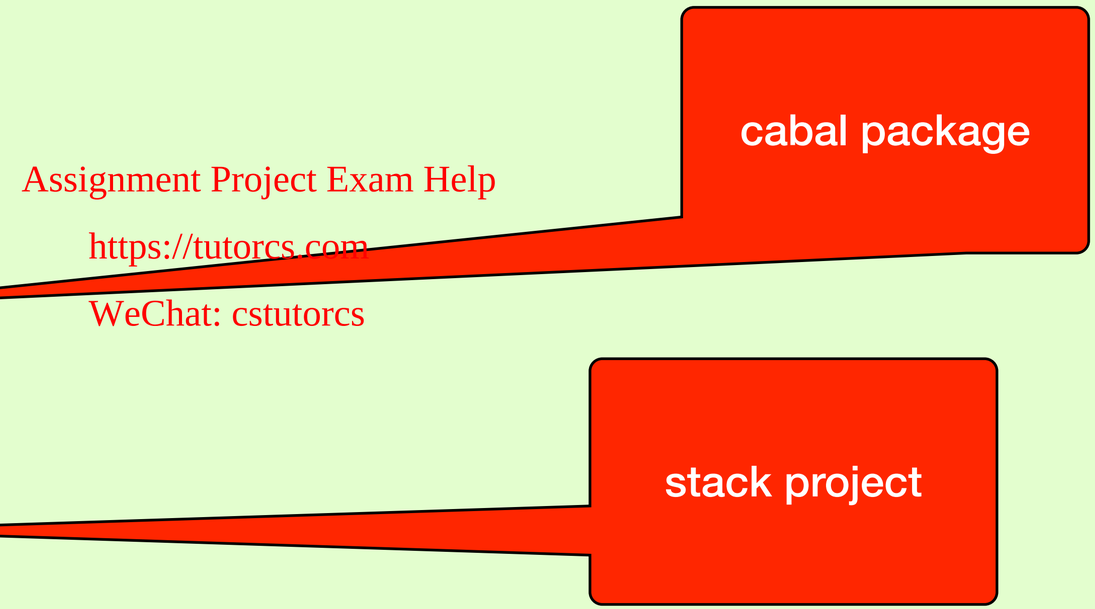
stack - project layout

```
Edmunds-MacBook-Pro:tmp edmundr$ tree demo
demo
├── ChangeLog.md
├── LICENSE
├── README.md
├── Setup.hs
├── app
│   └── Main.hs
├── demo.cabal
├── package.yaml
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 10 files
```



stack - build files

```
Edmunds-MacBook-Pro:tmp edmundr$ tree demo
demo
├── ChangeLog.md
├── LICENSE
├── README.md
├── Setup.hs
├── app
│   └── Main.hs
├── demo.cabal
├── package.yaml
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 10 files
```



.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

.Modules, packages, projects

- Project: Stack
- Package: Cabal
- Module: Haskell

- A **module** is a single source file
- It contains declarations, and an export list
- The name of the module is basically the name of the file
- Modules are the units Haskell uses to import code into programs
- The Haskell language knows about modules, but not packages or projects

Creating Modules

```
-- 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

Using Modules

```
-- file: CrawlerDB.hs

module CrawlerDB ( printURLs ) where

import CrawlerHTTP
import Database.HDBC as Db
import Database.HDBC.Sqlite3 (connectSQlite3)

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)
```

modules are imported through import directives

rename module

import list

Importing Modules

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

Dealing with name clashes


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

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

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

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

.Modules, packages, projects

- Project: Stack

- Package: Cabal

- Module: Haskell

- A **package** is a library of Haskell modules known to the compiler
- Cabal is a system for building and packaging Haskell libraries and programs.
- The build of each package is controlled by a “cabal” file.
- Packages have to be installed and registered in a database in order to be usable by the compiler.
- Packages are identified by a base name and version number:
- The Haskell language knows about modules, but not packages or projects

.Modules, packages, projects

- Project: Stack
- Package: Cabal
- Module: Haskell

HTTP

A library for client-side HTTP

HTTP-4000.3.16 A library for client-side HTTP

Network
- Network.Browser
- Network.BufferType
- Network.HTTP
  - Network.HTTP.Auth
  - Network.HTTP.Base
  - Network.HTTP.Cookie
  - Network.HTTP.HandleStream
  - Network.HTTP.Headers
  - Network.HTTP.Proxy
  - Network.HTTP.Stream
- Network.Stream
- Network.StreamDebugger
- Network.StreamSocket
- Network.TCP

.Modules, packages, projects

- Project: Stack
- Package: Cabal
- Module: Haskell

HTTP

A library for client-side HTTP

HTTP-4000.3.16 — A library for client-side HTTP

```
Library
  Exposed-modules:
                Network.BufferType,
                Network.Stream,
                Network.StreamDebugger,
                Network.StreamSocket,
                Network.TCP,
                Network.HTTP,
                Network.HTTP.Headers,
                Network.HTTP.Base,
                Network.HTTP.Stream,
                Network.HTTP.Auth,
                Network.HTTP.Cookie,
                Network.HTTP.Proxy,
                Network.HTTP.HandleStream,
                Network.Browser
  Other-modules:
                Network.HTTP.Base64,
                Network.HTTP.MD5Aux,
                Network.HTTP.Utils
                Paths_HTTP
  GHC-options: -fwarn-missing-signatures -Wall

  -- note the test harness constraints should be kept in sync with these
  -- where dependencies are shared
  Build-depends: base >= 4.3.0.0 && < 4.17, parsec >= 2.0 && < 3.2
  Build-depends: array >= 0.3.0.2 && < 0.6, bytestring >= 0.9.1.5 && < 0.12
  Build-depends: time >= 1.1.2.3 && < 1.13
```

.Modules, packages, projects

- Project: Stack
- Package: Cabal
- Module: Haskell

- Stack deals with **projects**
- A project can contain multiple packages, but may just contain one.
- The build of a project is controlled by a “stack.yaml” file.
- The stack.yaml file specifies which packages to build, and where to get library files from if they are needed.

stack.yaml

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

```
# 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
extra-deps:
- HTTP-4000.3.3
```


resolver containing curated package set

package and version

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
```


package names, sometimes with version constraints

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”

Input/Output in Haskell

Pure versus Impure

Pure	Impure
Definitions	*Commands*
Stateless	*State (e.g. global variables)*
No side effects	*Side effects*
Easy to reason about program	*Easy to interact with outside world*

Pure versus Impure

Pure	Impure
Definitions	*Commands*
Stateless	*State (e.g. global variables)*
No side effects	*Side effects*
Easy to reason about program	*Easy to interact with outside world*

Pure versus Impure

Pure	Impure
Definitions	*Commands*
Stateless	*State (e.g. global variables)*
No side effects	*Side effects*
Easy to reason about program	*Easy to interact with outside world*

Contained in IO types

RECAP

Algebraic Types

Just like functions, our algebraic type can be **parametrised** by another type


```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

You call these "**type functions**"

```
-- Maybe is often used when a value might not be available
first :: [a] -> Maybe a
first [] = Nothing
first (x:xs) = Just x
```

The IO Type Function

`IO` is a (special) type function

`IO a`

We call the type "IO a" an "IO action of type a".
When performed it might carry out an action with side-effect and yield 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 ()
```

The IO Type Function

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

The IO Type Removal Question

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


```
Prelude> getLine
hello
"hello"
Prelude> length getLine

<interactive>:17:1: error:
```

length only applies to pure Strings

getLine is an IO String

What function can I apply to my IO String to extract the String???

The IO Type Removal Answer

What function can I apply to my IO String to extract the String???


```
Prelude> getLine
hell
"hel
Prelude> length getLine

<int
```

length only applies to pure Strings

Answer: there isn't one

getLine

And that is deliberate!

The IO Type Removal Answer

What function can I apply to my IO String to extract the String???


```
Prelude> getLine
hell
"hel
Prelude> length getLine

<int
```

length only applies to pure Strings

Anything acquired through an IO action stays in IO type

getLine

It is forever tainted!

But you can unpack it locally inside a do block.

Greeting Example

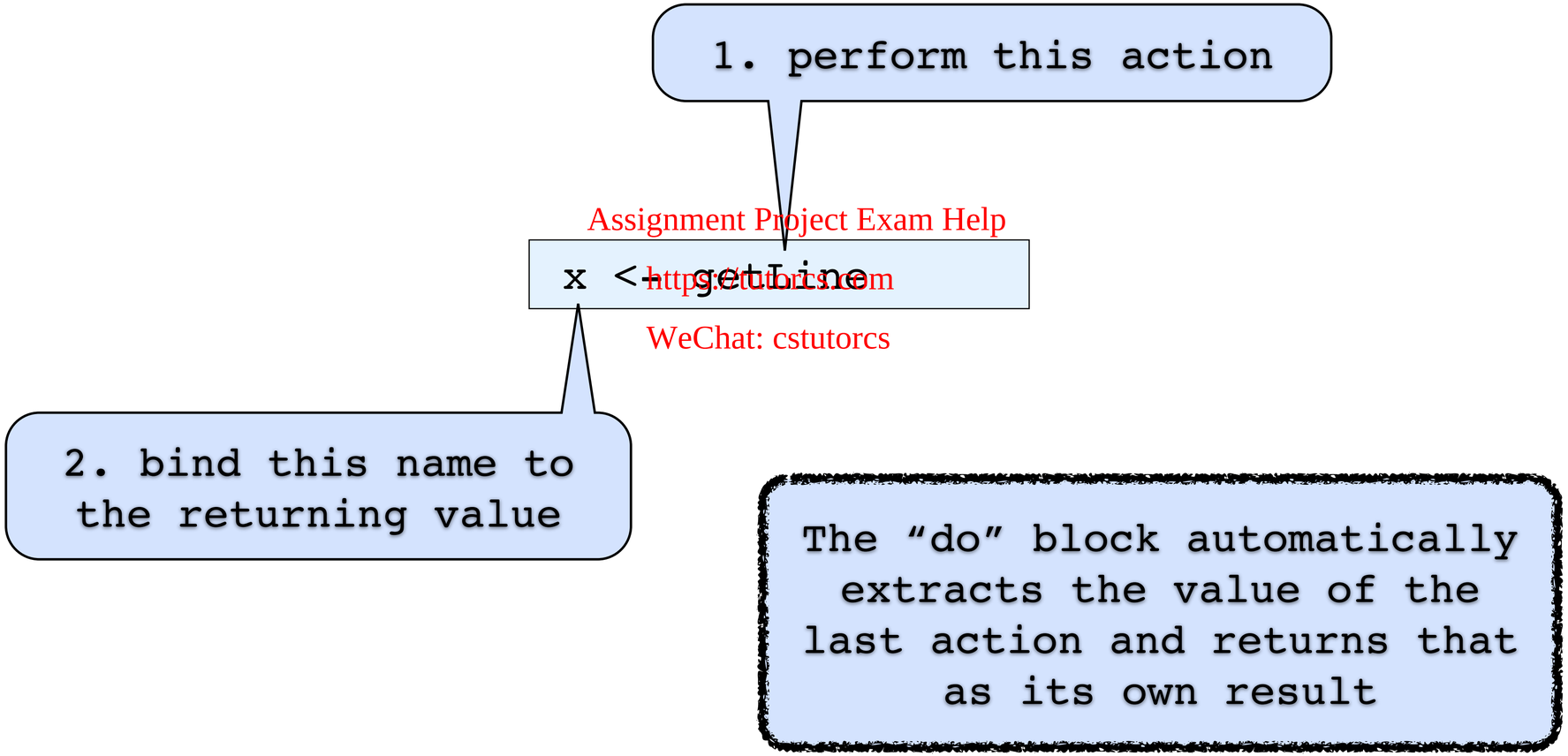
```
-- file: week06.hs
main = do
    putStrLn "Greetings!  What is your name?"
    x <- getLine
    putStrLn $ "Welcome to Haskell, " ++ x ++ "!"
```

"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



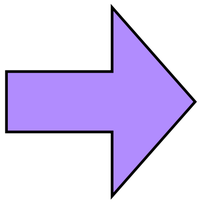
readFile / writeFile

```
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
```