Haskell - introduction
Functional Programming

Jens Egholm Pedersen and Anders Kalhauge

cphbusiness

Spring 2018

Outline

Running Haskell

- Installation
- Creating and Compiling Projects
- Read Evaluate Print Loop

Haskell language

- Some basic examples
- Haskell Data Types
- Type Synonyms
- Haskell Type Classes

Running Haskell

- Installation
- Creating and Compiling Projects
- Read Evaluate Print Loop

Haskell language

- Some basic examples
- Haskell Data Types
- Type Synonyms
- Haskell Type Classes

Installation

The Haskell Tool **Stack**

- Installing GHC in an isolated location
- Installing packages
- Building the project
- Running tests

Installation Windows

Download:

`https://www.stackage.org/stack/windows-x86_64-installer`

Run the installer

Installation Linux

In a console:

```
$ curl -sSL https://get.haskellstack.org/ | sh
```

or:

```
$ wget -qO- https://get.haskellstack.org/ | sh
```

Installation Mac

In a console:

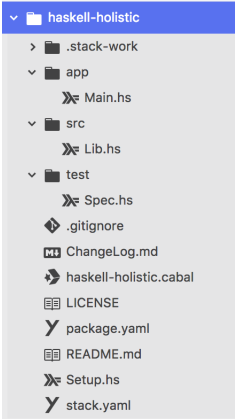
```
$ brew update
$ brew install haskell-stack
```

Creating a Haskell-project with Stack

In the root directory of your new project:

```
$ stack new haskell-holistic
$ cd haskell-holistic
```

This will create the `haskell-holistic` project folder and project files:



```
haskell-holistic
  .stack-work
  app
    Main.hs
  src
    Lib.hs
  test
    Spec.hs
  .gitignore
  ChangeLog.md
  haskell-holistic.cabal
  LICENSE
  package.yaml
  README.md
  Setup.hs
  stack.yaml
```

The GHCi Compiler

Installing the compiler if necessary:

```
$ stack setup
```

Building the Project:

```
$ stack build
```

And running the application

```
$ stack exec haskell-holistic
```

GHCi REPL

```
$ stack ghci
Configuring GHCi with the following packages:
GHCi, version 8.0.2: http://www.haskell.org/ghc/
      :? for help
Loaded GHCi configuration from ...
Prelude> 2 + 2
4
Prelude> :quit
Leaving GHCi.
$
```

Running Haskell

- Installation
- Creating and Compiling Projects
- Read Evaluate Print Loop

Haskell language

- Some basic examples
- Haskell Data Types
- Type Synonyms
- Haskell Type Classes

Read and Write to console

```
main :: IO ()
main = do
  putStrLn "Hi,␣what's␣your␣name?"
  name <- getLine
  putStrLn ("Hello␣"++name)
```

In GHCi

```
ghci> :t getLine
getLine :: IO String
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci>
```

Show and Read

```haskell
main :: IO ()
main = do
  putStrLn "Hi,␣what's␣your␣age?"
  line <- getLine
  let age = (read line :: Int)
  putStrLn ((show age)++"␣years")
```

Reading lines from file

```haskell
import System.IO

main :: IO ()
main = do
  withFile "numbers.txt" ReadMode (\handle -> do
    contents <- hGetContents handle
    let numberLines = lines contents
    putStr (contents++(numberLines!!2)++"\n")
    )
```

Define function

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)

main :: IO ()
main = do
  putStrLn ("50!␣=␣"++(show (fact 50)))
```

```
50! = 30414093201713378043612608166064768844377641568
      960512000000000000
```

Exercise 1 - Greatest Common Divisor

Create the Greatest Common Divisor function

$$gcd(a, 0) = a$$

$$gcd(a, b) = gcd(b, a \mod b)$$

```
gcd :: Integer -> Integer -> Integer
```

Hint: Haskell modulus function is `mod`

Exercise 2 - Prime number?

Create a function that determins if a number is a prime

```haskell
isPrime :: Integer -> Bool
```

Hint: Smallest divisor of n must be less than $\sqrt{n}$

data types

Elm: Union type

```
data Bool = False | True
```

Custom data types

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

```
data Person = { name :: String
              , age :: Int
              , email :: String
              } deriving (Show)
```

```
data Maybe a = Nothing | Just a
```

type Synonyms

Elm: **type alias**

```
phoneBook :: [(String, String)]
phoneBook =
  [ ("Kurt", "12345678")
  , ("Sonja", "98765432")
  , ("Ib", "47117913")
  ]
```

```
type PhoneBook = [(String, String)]

phoneBook :: PhoneBook
phoneBook = [ ... ]
```

Build in Type Classes

- `Ord` ordered type, supporting `<`, `>`, . . .
- `Eq` equatable type, supporting `==` and `/=`
- `Enum` enumeratable type, works with `[1..10]`
- `Bounded` types with `minBound` and `maxBound`
- `Num` numeral type, supporting `+`, `-`, . . .
  - `Integral` integer numbers
  - `Floating` real numbers
- `Show` showable type, supporting `show` (*toString*) function
- `Read` showable type, supporting `read` (*fromString*) function