Programming in Haskell: Lecture 16

S P Suresh

October 9, 2019

Till now ...

- A **program** is a bunch of functions

Till now ...

- A **program** is a bunch of functions
- A function of type `a -> b` produces a result of type `b` on an argument of type `a`

Till now ...

- A **program** is a bunch of functions
- A function of type `a -> b` produces a result of type `b` on an argument of type `a`
- The programs are run in `ghci` – by invoking a function on some arguments

Till now ...

- A **program** is a bunch of functions
- A function of type `a -> b` produces a result of type `b` on an argument of type `a`
- The programs are run in `ghci` – by invoking a function on some arguments
- `ghci` automatically displays the result on the screen (provided it can be shown)

User interaction

- Can we execute programs outside `ghci`?

User interaction

- Can we execute programs outside `ghci`?
- How do we let the programs interact with users?

User interaction

- Can we execute programs outside `ghci`?
- How do we let the programs interact with users?
  - Accept user inputs midway through a program execution

User interaction

- Can we execute programs outside `ghci`?
- How do we let the programs interact with users?
  - Accept user inputs midway through a program execution
  - Print output and diagnostics on screen or to a file

User interaction

- Can we execute programs outside `ghci`?
- How do we let the programs interact with users?
  - Accept user inputs midway through a program execution
  - Print output and diagnostics on screen or to a file
- Can interaction with the outside world be achieved without violating the spirit of Haskell?

Standalone programs and `main`

- Execution of a Haskell program starts with the function `main`

Standalone programs and `main`

- Execution of a Haskell program starts with the function `main`
- Every standalone Haskell program should have a `main` function

First program

- First compilable program

```
main = putStr "Hello, world!\n"
```

First program

- First compilable program

```
main = putStr "Hello, world!\n"
```

- Save this into a file named `hw.hs`

First program

- First compilable program

```
main = putStr "Hello, world!\n"
```

- Save this into a file named `hw.hs`
- Compile it using the command `ghc hw.hs`

First program

- First compilable program

```
main = putStr "Hello, world!\n"
```

- Save this into a file named `hw.hs`
- Compile it using the command `ghc hw.hs`
- This generates the files `hw.hi`, `hw.o` and `hw` (with **execute permissions**)

First program

- First compilable program

```
main = putStr "Hello, world!\n"
```

- Save this into a file named `hw.hs`
- Compile it using the command `ghc hw.hs`
- This generates the files `hw.hi`, `hw.o` and `hw` (with **execute permissions**)
- Run the executable using the command `./hw`

ghc

- ghc is the Glasgow Haskell Compiler

ghc

- `ghc` is the Glasgow Haskell Compiler
- `ghci` is the **interactive** version of the compiler

ghc

- `ghc` is the Glasgow Haskell Compiler
- `ghci` is the **interactive** version of the compiler
- One can view `ghci` as an **interpreter** or a **playground** in which to test programs

ghc

- `ghc` is the Glasgow Haskell Compiler
- `ghci` is the **interactive** version of the compiler
- One can view `ghci` as an **interpreter** or a **playground** in which to test programs
- Software intended for use by others is written as a standalone program, compiled using `ghc` and shipped

ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in `ghci`

ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in `ghci`
- Check out commonly used compiler options using `ghc --help`

ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in `ghci`
- Check out commonly used compiler options using `ghc --help`
- Use `ghc --show-options` to know all options (a huge list!)

ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in `ghci`
- Check out commonly used compiler options using `ghc --help`
- Use `ghc --show-options` to know all options (a huge list!)
- The GHC Manual is a comprehensive document about both `ghc` and `ghci`

ghc

- Compiled versions of programs run much faster and use much less memory, compared to running them in `ghci`
- Check out commonly used compiler options using `ghc --help`
- Use `ghc --show-options` to know all options (a huge list!)
- The GHC Manual is a comprehensive document about both `ghc` and `ghci`
  - Available at `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/`

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some `b`
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some `b`
- The return value is not used at all, so perhaps it returns nothing of significance
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some `b`
- The return value is not used at all, so perhaps it returns nothing of significance
- The type `()`, pronounced **unit**
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some `b`
- The return value is not used at all, so perhaps it returns nothing of significance
- The type `()`, pronounced **unit**
  - Consists of a single value, also denoted by `()`
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some `b`
- The return value is not used at all, so perhaps it returns nothing of significance
- The type `()`, pronounced **unit**
  - Consists of a single value, also denoted by `()`
  - Can be used to model **nothing**
  ```

Hello, world!

- ```
  main = putStr "Hello, world!\n"
  ```
- `putStr str` prints the string `str` on screen
- Clearly `putStr` is of type `String -> b`, for some `b`
- The return value is not used at all, so perhaps it returns nothing of significance
- The type `()`, pronounced **unit**
  - Consists of a single value, also denoted by `()`
  - Can be used to model **nothing**
- So is `String -> ()` the type of `putStr`?
  ```

Hello, world!

- Is `putStr` of type `String -> ()`?

Hello, world!

- Is `putStr` of type `String -> ()`?
- No, because it does not return the value `()`!

Hello, world!

- Is `putStr` of type `String -> ()`?
- No, because it does not return the value `()`!
- Further, how do we account for the side effect of printing something on screen?

Hello, world!

- Is `putStr` of type `String -> ()`?
- No, because it does not return the value `()`!
- Further, how do we account for the side effect of printing something on screen?
- ```
  ghci> :t putStr
        putStr :: String -> IO ()

  ghci> :t putStr "Hello, world!"
        putStr "Hello, world!" :: IO ()
  ```
  ```

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`
  - If `a` is a type, then `Maybe a` and `[a]` are types

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`
  - If `a` is a type, then `Maybe a` and `[a]` are types
  - Likewise, `IO a` is a type whenever `a` is a type

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`
  - If `a` is a type, then `Maybe a` and `[a]` are types
  - Likewise, `IO a` is a type whenever `a` is a type
- Values of type `Maybe a`

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`
  - If `a` is a type, then `Maybe a` and `[a]` are types
  - Likewise, `IO a` is a type whenever `a` is a type
- Values of type `Maybe a`
  - Either `Nothing` or `Just x` for `x :: a`

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`
  - If `a` is a type, then `Maybe a` and `[a]` are types
  - Likewise, `IO a` is a type whenever `a` is a type
- Values of type `Maybe a`
  - Either `Nothing` or `Just x` for `x :: a`
  - `Nothing` and `Just` are value constructors

`IO a`

- `IO` is a type constructor, just like `Maybe` or `[]`
  - If `a` is a type, then `Maybe a` and `[a]` are types
  - Likewise, `IO a` is a type whenever `a` is a type
- Values of type `Maybe a`
  - Either `Nothing` or `Just x` for `x :: a`
  - `Nothing` and `Just` are value constructors
- Unlike other type constructors like `Maybe`, the internal structure and constructors of `IO` are not visible to the user

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

- So an object of type `IO a` is a function:

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

- So an object of type `IO a` is a function:
  - Takes the current state of the real world as input

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

- So an object of type `IO a` is a function:
  - Takes the current state of the real world as input
  - Produces a new state of the real world and a value of type `a`

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

- So an object of type `IO a` is a function:
  - Takes the current state of the real world as input
  - Produces a new state of the real world and a value of type `a`
- In other words, objects of `IO a` consist of two things:

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

- So an object of type `IO a` is a function:
  - Takes the current state of the real world as input
  - Produces a new state of the real world and a value of type `a`
- In other words, objects of `IO a` consist of two things:
  - A value of type `a` that can be extracted

`IO a`

- `IO a` can be understood to be the type

```
RealWorld -> (RealWorld,a)
```

- So an object of type `IO a` is a function:
  - Takes the current state of the real world as input
  - Produces a new state of the real world and a value of type `a`
- In other words, objects of `IO a` consist of two things:
  - A value of type `a` that can be extracted
  - A side effect (the change in state of the world)

`IO a` and actions

- Technically, an object of type `IO a` is not a function

`IO a` and actions

- Technically, an object of type `IO a` is not a function
- It is an **action** of return type `IO a`

`IO a` and actions

- Technically, an object of type `IO a` is not a function
- It is an **action** of return type `IO a`
- An IO action produces a side effect **when its value is extracted**

`IO a` and actions

- Technically, an object of type `IO a` is not a function
- It is an **action** of return type `IO a`
- An IO action produces a side effect **when its value is extracted**
- Any function that produces a side effect will have return type `IO a`

`putStr` *and* `main`

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

`putStr` *and* `main`

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`

`putStr` *and* `main`

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`
- Produces a side effect when the return value is extracted

`putStr` *and* `main`

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`
- Produces a side effect when the return value is extracted
- The side effect is that of printing the string on screen

`putStr` *and* `main`

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`
- Produces a side effect when the return value is extracted
- The side effect is that of printing the string on screen
- `main :: IO ()`

`putStr` *and* `main`

- `putStr :: String -> IO ()`
- `putStr` takes a string as argument and returns `()`
- Produces a side effect when the return value is extracted
- The side effect is that of printing the string on screen
- `main :: IO ()`
- `main` is always of type `IO a`

Side effects

- Kind of side effects

Side effects

- Kind of side effects
  - Printing on screen

Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal

Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening or closing a file

Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening or closing a file
  - Changing a directory

Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening or closing a file
  - Changing a directory
  - Writing into a file

Side effects

- Kind of side effects
  - Printing on screen
  - Reading a user input from the terminal
  - Opening or closing a file
  - Changing a directory
  - Writing into a file
  - Launching a missile

`putStr` *and* `putStrLn`

- `putStr "Hello world!"` prints the string on the screen

`putStr` *and* `putStrLn`

- `putStr "Hello world!"` prints the string on the screen
- `putStrLn "Hello world!"` prints the string and a newline (`'\n'`) on the screen

`putStr` *and* `putStrLn`

- `putStr "Hello world!"` prints the string on the screen
- `putStrLn "Hello world!"` prints the string and a newline (`'\n'`) on the screen
- `putStrLn str` is equivalent to `putStr (str ++ "\n")`

Chaining actions

- We use the command **do** to chain multiple actions

```
main = do
    putStrLn "Hello!"
    putStrLn "What's your name?"
```

Chaining actions

- We use the command `do` to chain multiple actions

```
main = do
    putStrLn "Hello!"
    putStrLn "What's your name?"
```

- `do` makes the actions take effect in sequential order, one after the other

Chaining actions

- We use the command `do` to chain multiple actions

```
main = do
    putStrLn "Hello!"
    putStrLn "What's your name?"
```

- `do` makes the actions take effect in sequential order, one after the other
- Indentation is important

Chaining actions

- We use the command `do` to chain multiple actions

```
main = do
    putStrLn "Hello!"
    putStrLn "What's your name?"
```

- `do` makes the actions take effect in sequential order, one after the other
- Indentation is important
- Alternative, friendlier syntax

```
main = do {
    putStrLn "Hello!";
    putStrLn "What's your name?";
}
```

Chaining actions

- Actions can occur inside `let`, `where`, `if-then-else` &c.

```
main = let fibs = 0:1:zipWith (+) fibs (tail fibs)
    in do {
        putStrLn $ show fibs!!5;
        putStrLn $ show fibs!!10;
    }
```

```
main = do {act1; act2;}
    where
        act1 = putStr "Hello, "
        act2 = putStrLn "world!"
```

More actions

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

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen
- `getLine :: IO String`

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen
- `getLine :: IO String`
  - Read a line from the standard input and return it as a string

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen
- `getLine :: IO String`
  - Read a line from the standard input and return it as a string
  - The side effect of `getLine` is the consumption of a line of input

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen
- `getLine :: IO String`
  - Read a line from the standard input and return it as a string
  - The side effect of `getLine` is the consumption of a line of input
  - The return value is a string

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen
- `getLine :: IO String`
  - Read a line from the standard input and return it as a string
  - The side effect of `getLine` is the consumption of a line of input
  - The return value is a string
- `getChar :: IO Char`

More actions

- `print :: Show a => a -> IO ()`
  - Output a value of any printable type to the standard output (screen), and add a newline
- `putChar :: Char -> IO ()`
  - Write the character argument to the screen
- `getLine :: IO String`
  - Read a line from the standard input and return it as a string
  - The side effect of `getLine` is the consumption of a line of input
  - The return value is a string
- `getChar :: IO Char`
  - Read the next character from the standard input

Binding

- `getLine` is of type `IO String`

Binding

- `getLine` is of type `IO String`
- Is there a way to use the return value?

Binding

- `getLine` is of type `IO String`
- Is there a way to use the return value?
- We need to bind the return value to an object of type `String` and use it elsewhere

Binding

- `getLine` is of type `IO String`
- Is there a way to use the return value?
- We need to bind the return value to an object of type `String` and use it elsewhere
- The syntax for binding is `<-`

```
main = do {
    putStrLn "Please type your name!";
    n <- getLine;
    putStrLn ("Hello, " ++ n);
}
```

Binding

- The syntax for binding is `<-`

```
main = do {
    putStrLn "Please type your name!";
    n <- getLine;
    putStrLn ("Hello, " ++ n);
}
```

Binding

- The syntax for binding is `<-`

```
main = do {
    putStrLn "Please type your name!";
    n <- getLine;
    putStrLn ("Hello, " ++ n);
}
```

- **Wrong usage** – `putStrLn ("Hello" ++ getLine)`

Binding

- The syntax for binding is `<-`

```
main = do {
    putStrLn "Please type your name!";
    n <- getLine;
    putStrLn ("Hello, " ++ n);
}
```

- Wrong usage – `putStrLn ("Hello" ++ getLine)`
- `getLine` is not a string

Binding

- The syntax for binding is `<-`

```
main = do {
    putStrLn "Please type your name!";
    n <- getLine;
    putStrLn ("Hello, " ++ n);
}
```

- Wrong usage – `putStrLn ("Hello" ++ getLine)`
- `getLine` is not a string
- It is an action that has a return value of type `String`

Binding

- The syntax for binding is `<-`

```
main = do {
    putStrLn "Please type your name!";
    n <- getLine;
    putStrLn ("Hello, " ++ n);
}
```

- Wrong usage – `putStrLn ("Hello" ++ getLine)`
- `getLine` is not a string
- It is an action that has a return value of type `String`
- The return value has to be extracted before use

Summary

- Haskell has a clean separation of pure functions and actions with side effects

Summary

- Haskell has a clean separation of pure functions and actions with side effects
- Actions are used to interact with the real world and perform input/output

Summary

- Haskell has a clean separation of pure functions and actions with side effects
- Actions are used to interact with the real world and perform input/output
- `main` is an action where computation begins

Summary

- Haskell has a clean separation of pure functions and actions with side effects
- Actions are used to interact with the real world and perform input/output
- `main` is an action where computation begins
- `ghc` can be used to compile and run programs