Introduction to Programming: Lecture 18

K Narayan Kumar

Chennai Mathematical Institute

`http://www.cmi.ac.in/~kumar`

17 Oct 2013

Compiling into executables

- So far,
  - All inputs were supplied as arguments to functions from ghci
  - Outputs were printed out by ghci

Compiling into executables

- So far,
  - All inputs were supplied as arguments to functions from ghci
  - Outputs were printed out by ghci
- Works as long as programs are run from within an interpreter.

Compiling into executables

- So far,
  - All inputs were supplied as arguments to functions from ghci
  - Outputs were printed out by ghci
- Works as long as programs are run from within an interpreter.
- What if we want to compile programs into executables?

Compiling into executables

- So far,
  - All inputs were supplied as arguments to functions from ghci
  - Outputs were printed out by ghci
- Works as long as programs are run from within an interpreter.
- What if we want to compile programs into executables?
- The Haskell programs described so far cannot be compiled into executables.

Compiling into executables

- So far,
    - All inputs were supplied as arguments to functions from ghci
    - Outputs were printed out by ghci
- Works as long as programs are run from within an interpreter.
- What if we want to compile programs into executables?
- The Haskell programs described so far cannot be compiled into executables.
    - For eg. they don't specify what function has to be computed.

Compiling ...

- Consider the Haskell program

```
main = putStrLn ("My First Compilable Program")
```

Compiling ...

- Consider the Haskell program

```
main = putStrLn ("My First Compilable Program")
```

- Compile the program using ghc

```
ghc –make Out.hs
```

Compiling ...

- Consider the Haskell program

  ```
  main = putStrLn ("My First Compilable Program")
  ```

- Compile the program using ghc

  ```
  ghc –make Out.hs
  ```

- Computes all the module dependencies and compiles all the modules.

Compiling ...

- Consider the Haskell program

```
main = putStrLn ("My First Compilable Program")
```

- Compile the program using ghc

```
ghc –make Out.hs
```

- Computes all the module dependencies and compiles all the modules.
- Run by executing the program Out

Compiling ...

- Consider the Haskell program

  ```
  main = putStrLn ("My First Compilable Program")
  ```

- Compile the program using ghc

  ```
  ghc –make Out.hs
  ```

- Computes all the module dependencies and compiles all the modules.
- Run by executing the program Out
- How do we give inputs to a Haskell program that is compiled and executed?

Input/Output in Haskell

- Here is a simple program that does both input and output.

```
main = do
         putStrLn ("Please enter your name:")
         name <- getLine
         putStrLn ("Hello " ++ name)
```

Input/Output in Haskell

- Here is a simple program that does both input and output.

```
main = do
           putStrLn ("Please enter your name:")
           name <- getLine
           putStrLn ("Hello " ++ name)
```

- `main` is the name of the action that is executed when a compiled Haskell program is run.

Input/Output in Haskell

- Here is a simple program that does both input and output.

```
main = do
         putStrLn ("Please enter your name:")
         name <- getLine
         putStrLn ("Hello " ++ name)
```

- `main` is the name of the action that is executed when a compiled Haskell program is run.
- `main`, `putStrLn s`, `getLine` are all actions.

Input/Output in Haskell

- Here is a simple program that does both input and output.

```
main = do
           putStrLn ("Please enter your name:")
           name <- getLine
           putStrLn ("Hello " ++ name)
```

- `main` is the name of the action that is executed when a compiled Haskell program is run.
- `main`, `putStrLn s`, `getLine` are all actions.
- The `do` command puts together a sequence of actions into a larger action.

Input/Output in Haskell

- Here is a simple program that does both input and output.

```
main = do
           putStrLn ("Please enter your name:")
           name <- getLine
           putStrLn ("Hello " ++ name)
```

- `main` is the name of the action that is executed when a compiled Haskell program is run.
- `main`, `putStrLn s`, `getLine` are all actions.
- The `do` command puts together a sequence of actions into a larger action.
- These actions are executed sequentially, that is, one after the other.

The type of Actions

- What is the type of `main` in this example?

The type of Actions

- What is the type of `main` in this example?

  ```
  main :  IO ()
  ```

The type of Actions

- What is the type of `main` in this example?

  `main :  IO ()`
- The type `()` is the type with a single value, also denoted `()`.

The type of Actions

- What is the type of `main` in this example?

  `main :  IO ()`

- The type `()` is the type with a single value, also denoted `()`.
- What are the types of `putStrLn` and `getLine` ?

The type of Actions

- What is the type of `main` in this example?

  ```
  main :  IO ()
  ```

- The type `()` is the type with a single value, also denoted `()`.
- What are the types of `putStrLn` and `getLine` ?

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

The type of Actions

- What is the type of `main` in this example?

  ```
  main :  IO ()
  ```

- The type `()` is the type with a single value, also denoted `()`.
- What are the types of `putStrLn` and `getLine` ?

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

- The qualifier `IO` in all these types indicate that the function also performs some Input/Output.

The type of Actions

- What is the type of `main` in this example?

  `main :  IO ()`

- The type `()` is the type with a single value, also denoted `()`.
- What are the types of `putStrLn` and `getLine` ?

  `putStrLn ::  String -> IO ()`

  `getLine ::  IO String`

- The qualifier `IO` in all these types indicate that the function also performs some Input/Output.
- The type of a `do ..` statement is the type of the last action.

  The type of an action `v <- e` is the type of `e`.

Actions

- An action is an expression of type `IO a` for some `a`.
  `main`, `putStrLn s` and `getLine` are all actions.

Actions

- An action is an expression of type `IO a` for some `a`.

  `main`, `putStrLn s` and `getLine` are all actions.
- In Haskell, any Input/Output must occur within expressions of type `IO a` for some `a`.

Actions

- An action is an expression of type `IO a` for some `a`.

  `main`, `putStrLn s` and `getLine` are all actions.
- In Haskell, any Input/Output must occur within expressions of type `IO a` for some `a`.
- Actions are first class values and for most part can be used like other normal values.

  ```
  alist =  map putStrLn ["one","two","three"]
  c = head alist
  ```

Actions

- An action is an expression of type `IO a` for some `a`.

  `main`, `putStrLn s` and `getLine` are all actions.
- In Haskell, any Input/Output must occur within expressions of type `IO a` for some `a`.
- Actions are first class values and for most part can be used like other normal values.

  ```
  alist =  map putStrLn ["one","two","three"]
  c = head alist
  ```
- As expected, we have

  ```
  alist ::  [IO ()]
  ```

Actions

- An action is an expression of type `IO a` for some `a`.

  `main`, `putStrLn s` and `getLine` are all actions.
- In Haskell, any Input/Output must occur within expressions of type `IO a` for some `a`.
- Actions are first class values and for most part can be used like other normal values.

  ```
  alist =  map putStrLn ["one","two","three"]
  c = head alist
  ```
- As expected, we have

  ```
  alist ::  [IO ()]
  c ::  IO ()
  ```

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`
  What is the need for such a distinction?

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`
  What is the need for such a distinction?
- The kind of Haskell functions we have so far seen in this course are called pure functions.

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`
  What is the need for such a distinction?
- The kind of Haskell functions we have so far seen in this course are called pure functions.
  Their type gives all the information we need about them.

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`
  What is the need for such a distinction?
- The kind of Haskell functions we have so far seen in this course are called pure functions.
  Their type gives all the information we need about them.
- For functions that also do IO, the types of the arguments and the return value by themselves do not reveal everything.

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`
  What is the need for such a distinction?
- The kind of Haskell functions we have so far seen in this course are called pure functions.
  Their type gives all the information we need about them.
- For functions that also do IO, the types of the arguments and the return value by themselves do not reveal everything.
  - Input/Output involves changing the outside world.
    state change
  - I/O Actions have to be composed sequentially, that is, the order of execution is critical.

I/O ...

- A function with an integer for an argument and returning an integer has type
  `Int -> Int`
  while one that also does some IO has type
  `(Int -> IO Int)`
  What is the need for such a distinction?
- The kind of Haskell functions we have so far seen in this course are called pure functions.
  Their type gives all the information we need about them.
- For functions that also do IO, the types of the arguments and the return value by themselves do not reveal everything.
  - Input/Output involves changing the outside world.
    state change
  - I/O Actions have to be composed sequentially, that is, the order of execution is critical.
    (eg.) Reading in different orders will result in different behaviours.

Combining Pure and IO functions

- Haskell type system allows us use pure and `action` parts in a safe manner.

Combining Pure and IO functions

- Haskell type system allows us use pure and `action` parts in a safe manner.
- There is no mechanism to execute an action from within a pure function.

Combining Pure and IO functions

- Haskell type system allows us use pure and `action` parts in a safe manner.
- There is no mechanism to execute an action from within a pure function.
- I/O is performed by an action only if it that action is performed, i.e. executed from within another action.

Combining Pure and IO functions

- Haskell type system allows us use pure and `action` parts in a safe manner.
- There is no mechanism to execute an action from within a pure function.
- I/O is performed by an action only if it that action is performed, i.e. executed from within another action.

  The `main` action is where all the action begins!

I/O Examples ...

- Read a line and print it out twice

I/O Examples ...

- Read a line and print it out twice

```
main = do
         inp <- getLine
         putStrLn inp;
         putStrLn inp;
```

I/O Examples ...

- Read a line and print it out twice

```
main = do
         inp <- getLine
         putStrLn inp;
         putStrLn inp;
```

- Read a line and print it out as many times as its length

I/O Examples ...

- Read a line and print it out twice

```
main = do
        inp <- getLine
        putStrLn inp;
        putStrLn inp;
```

- Read a line and print it out as many times as its length

```
main = do
        inp <- getLine
        ltimes (length inp) inp

ltimes :: Int ->  String -> IO ()
ltimes 1 l = putStrLn l
ltimes n l = do
              putStrLn l
              ltimes (n-1) l
```

Example ...

- Read a line `w`. Read and output as many lines as length of `w`.

Example ...

- Read a line `w`. Read and output as many lines as length of `w`.

```
main = do
        linp <- getLine
        ltimesrw (length linp)

ltimesrw :: Int -> IO ()
ltimesrw 1 = do
              inp <- getLine
              putStrLn inp
ltimesrw n = do
              inp <- getLine
              putStrLn inp
              ltimesrw (n-1)
```

Example ...

- Read a line `w`. Read and output as many lines as length of `w`.

```
main = do
        linp <- getLine
        ltimesrw (length linp)

ltimesrw :: Int -> IO ()
ltimesrw 1 = do
              inp <- getLine
              putStrLn inp
ltimesrw n = do
              inp <- getLine
              putStrLn inp
              ltimesrw (n-1)
```

- Suggests that we should write a function to do an action n times.

`ntimes`

- Repeat an action *n* times.

```
ntimes :: Int -> IO () -> IO ()
ntimes 1 s = s
ntimes n s = do
              s
              ntimes (n-1) s
```

`ntimes`

- Repeat an action *n* times.

```
ntimes :: Int -> IO () -> IO ()
ntimes 1 s = s
ntimes n s = do
              s
              ntimes (n-1) s
```

- Then we can write

```
action1 = do
            inp <- getLine
            ntimes (length inp)
              (putStrLn inp)
```

ntimes

- Repeat an action *n* times.

```
ntimes :: Int -> IO () -> IO ()
ntimes 1 s = s
ntimes n s = do
              s
              ntimes (n-1) s
```

- Then we can write

```
action1 = do
            inp <- getLine
            ntimes (length inp)
              (putStrLn inp)
```

- and

```
action2 = do
            linp <- getLine
            ntimes (length linp)
                   (do
                     inp <- getLine
                     putStrLn inp)
```

Reading other types

- The function `readLn` reads a value of any type `a` that is an instance of `Read a`

  ```
  readLn ::  (Read a) => IO a
  ```

Reading other types

- The function `readLn` reads a value of any type `a` that is an instance of `Read a`

  ```
  readLn ::  (Read a) => IO a
  ```

- All basic types (`Int, Float, Bool, ...`) are instances of `Read`.

Reading other types

- The function `readLn` reads a value of any type `a` that is an instance of `Read a`

```
readLn ::  (Read a) => IO a
```

- All basic types (`Int, Float, Bool, ...`) are instances of `Read`.

```
main = do
        inp <- (readLn :: IO Bool)
        putStrLn (show inp)
```

Reading other types

- The function `readLn` reads a value of any type `a` that is an instance of `Read a`

```
readLn ::  (Read a) => IO a
```

- All basic types (`Int, Float, Bool, ...`) are instances of `Read`.

```
main = do
          inp <- (readLn :: IO Bool)
          putStrLn (show inp)
```

- `readLn` reads a value of the appropriate type appearing by itself in a line.

Reading other types

- The function `readLn` reads a value of any type `a` that is an instance of `Read a`

  ```
  readLn ::  (Read a) => IO a
  ```

- All basic types (`Int, Float, Bool, ...`) are instances of `Read`.

  ```
  main = do
          inp <- (readLn :: IO Bool)
          putStrLn (show inp)
  ```

- `readLn` reads a value of the appropriate type appearing by itself in a line.

  ```
  main = do
          inp <- (readLn :: IO Float)
          putStrLn (show (inp*inp))
  ```

IO Examples

- Read a list of positive integers, terminated by a `-1`, into a `list` and print the sum.

IO Examples

- Read a list of positive integers, terminated by a `-1`, into a `list` and print the sum.

```
main = do
        ls <- (readlist [])
        putStrLn (sum ls)
```

IO Examples

- Read a list of positive integers, terminated by a `-1`, into a `list` and print the sum.

```
main = do
        ls <- (readlist [])
        putStrLn (sum ls)
readlist :: [Int] -> IO [Int]
readlist l = do
              inp <- (readLn :: IO Int)
              if (inp == -1)
               then l
               else readlist (inp:l)
```

IO Examples

- Read a list of positive integers, terminated by a `-1`, into a `list` and print the sum.

```
main = do
        ls <- (readlist [])
        putStrLn (sum ls)
readlist :: [Int] -> IO [Int]
readlist l = do
              inp <- (readLn :: IO Int)
              if (inp == -1)
               then l
               else readlist (inp:l)
```

- This is not typed correctly. `l` has type `[Int]` and not `IO [Int]`.

Example ...

```
readlist :: [Int] -> IO [Int]
readlist l = do
              inp <- (readLn :: IO Int)
              if (inp == -1)
               then (return l)
               else readlist (inp:l)
```

Example ...

```
readlist :: [Int] -> IO [Int]
readlist l = do
              inp <- (readLn :: IO Int)
              if (inp == -1)
               then (return l)
               else readlist (inp:l)
```

- The function `return` sends value of type `a` to a value of type `IO a`

Example ...

```
readlist :: [Int] -> IO [Int]
readlist l = do
              inp <- (readLn :: IO Int)
              if (inp == -1)
               then (return l)
               else readlist (inp:l)
```

- The function `return` sends value of type `a` to a value of type `IO a`
- Note that there is no obvious way to construct a useful function of type `IO a -> b` where `b` is not an action.

Example ...

```
readlist :: [Int] -> IO [Int]
readlist l = do
              inp <- (readLn :: IO Int)
              if (inp == -1)
               then (return l)
               else readlist (inp:l)
```

- The function `return` sends value of type `a` to a value of type `IO a`
- Note that there is no obvious way to construct a useful function of type `IO a -> b` where `b` is not an action.

  This is towards a clean separation of the pure fragments of the program (that do no I/O) and the IO parts.