Programming in Haskell: Lecture 18

S P Suresh

October 16, 2019

Reading a list of integers

- Read a list of non-negative integers (one on each line and terminated by a negative integer)

```
main = do {ls <- readIntList; print ls;}
readIntList :: IO [Int]
readIntList = do {
    inp <- readLn :: IO Int;
    if (inp < 0) then return [];
    else do {l <- readIntList; return (inp:l);}
}
```

Reading a list of integers

- Read a list of non-negative integers (one on each line and terminated by a negative integer)

```
main = do {ls <- readIntList; print ls;}
readIntList :: IO [Int]
readIntList = do {
    inp <- readLn :: IO Int;
    if (inp < 0) then return [];
    else do {l <- readIntList; return (inp:l);}
}
```

- What if we want to signal the end of input by some other means?

Reading a list of integers

- Read a list of non-negative integers (one on each line and terminated by a negative integer)

```
main = do {ls <- readIntList; print ls;}
readIntList :: IO [Int]
readIntList = do {
    inp <- readLn :: IO Int;
    if (inp < 0) then return [];
    else do {l <- readIntList; return (inp:l);}
}
```

- What if we want to signal the end of input by some other means?
- Say, input is from a file and we process each line till the file ends

Reading a list of integers

- Use `isEOF` (requires `import System.IO`)

```
import System.IO
main = do {ls <- readIntList; print ls;}
readIntList = do
    exitCond <- isEOF
    if exitCond then return [] else do {
        inp <- readLn :: IO Int;
        l <- readIntList; return (inp:l);
    }
```

Reading a list of integers

- Use `isEOF` (requires `import System.IO`)

```
import System.IO
main = do {ls <- readIntList; print ls;}
readIntList = do
    exitCond <- isEOF
    if exitCond then return [] else do {
        inp <- readLn :: IO Int;
        l <- readIntList; return (inp:l);
    }
```

- `isEOF` returns `True` when end of file is reached

Reading a list of integers

- Use `isEOF` (requires `import System.IO`)

```
import System.IO
main = do {ls <- readIntList; print ls;}
readIntList = do
    exitCond <- isEOF
    if exitCond then return [] else do {
        inp <- readLn :: IO Int;
        l <- readIntList; return (inp:l);
    }
```

- `isEOF` returns `True` when end of file is reached
- If input is provided from keyboard, indicate end of input by **Ctrl-D**

Repetition using `forever`

- Repeatedly read a list of integers on each line and print its reverse

Repetition using `forever`

- Repeatedly read a list of integers on each line and print its reverse
- Use `forever` to repeatedly perform an action (requires `import Control.Monad`)

Repetition using `forever`

- Repeatedly read a list of integers on each line and print its reverse
- Use `forever` to repeatedly perform an action (requires `import Control.Monad`)
- Use `exitSuccess` to exit the loop (requires `import System.Exit`)

Repetition using `forever`

- Repeatedly read a list of integers on each line and print its reverse
- Use `forever` to repeatedly perform an action (requires `import Control.Monad`)
- Use `exitSuccess` to exit the loop (requires `import System.Exit`)
- Check when to exit using `isEOF`

Repetition using **forever**

- Repeatedly read a list of integers on each line and print its reverse

```
import System.IO
import System.Exit
import Control.Monad
main = forever $ do {
    exitCond <- isEOF;
    if exitCond then exitSuccess else do {
        inList <- readLn :: IO [Int];
        print (reverse inList);
    }
}
```

Repetition using `forever`

- Convenient to use `when` along with `forever` to handle the exit case (requires `import Control.Monad`)

```
import System.IO
import System.Exit
import Control.Monad
main = forever $ do {
    exitCond <- isEOF;
    when exitCond exitSuccess;
    inList <- readLn :: IO [Int];
    print (reverse inList);
}
```

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen
- But Haskell is lazy!

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen
- But Haskell is lazy!
  - So only the portion of the input that is needed to produce a line of output is consumed

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen
- But Haskell is lazy!
  - So only the portion of the input that is needed to produce a line of output is consumed
  - No waiting for user to provide the whole input

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen
- But Haskell is lazy!
  - So only the portion of the input that is needed to produce a line of output is consumed
  - No waiting for user to provide the whole input
  - The line of output is printed to `stdout`

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen
- But Haskell is lazy!
  - So only the portion of the input that is needed to produce a line of output is consumed
  - No waiting for user to provide the whole input
  - The line of output is printed to `stdout`
  - Rest of the input is processed (including waiting for user to provide input)

The magic of `interact`

- The cleanest way of processing input is `interact` (requires `System.IO`)

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

- `interact f` applies `f` (a string function) to the entire input, and produces the entire output to the screen
- But Haskell is lazy!
  - So only the portion of the input that is needed to produce a line of output is consumed
  - No waiting for user to provide the whole input
  - The line of output is printed to `stdout`
  - Rest of the input is processed (including waiting for user to provide input)
  - Truly interactive!

The magic of `interact`

- Typically `f` is a function that processes one line of input

The magic of `interact`

- Typically `f` is a function that processes one line of input
- Produces output corresponding to that line of input

The magic of `interact`

- Typically `f` is a function that processes one line of input
- Produces output corresponding to that line of input
- The library functions `lines` and `unlines` come to the rescue

```
lines "One\nTwo\nThree" = ["One", "Two", "Three"]
unlines ["One", "Two", "Three"] = "One\nTwo\nThree\n"
```

The magic of `interact`

- Typically `f` is a function that processes one line of input
- Produces output corresponding to that line of input
- The library functions `lines` and `unlines` come to the rescue

```
lines "One\nTwo\nThree" = ["One", "Two", "Three"]
unlines ["One", "Two", "Three"] = "One\nTwo\nThree\n"
```

- Typical use of `interact`

```
    main = interact (unlines . map f . lines)
```

The magic of `interact`

- Typically `f` is a function that processes one line of input
- Produces output corresponding to that line of input
- The library functions `lines` and `unlines` come to the rescue

```
lines "One\nTwo\nThree" = ["One", "Two", "Three"]
unlines ["One", "Two", "Three"] = "One\nTwo\nThree\n"
```

- Typical use of `interact`

```
    main = interact (unlines . map f . lines)
```

- Localises input-output to one line of code

The magic of `interact`

- Typically `f` is a function that processes one line of input
- Produces output corresponding to that line of input
- The library functions `lines` and `unlines` come to the rescue

```
lines "One\nTwo\nThree" = ["One", "Two", "Three"]
unlines ["One", "Two", "Three"] = "One\nTwo\nThree\n"
```

- Typical use of `interact`

```
    main = interact (unlines . map f . lines)
```

- Localises input-output to one line of code
- `f` is a pure function

The magic of `interact`

- Typical use of `interact`

```
main = interact (unlines . map f . lines)
```

The magic of `interact`

- Typical use of `interact`

```
main = interact (unlines . map f . lines)
```

- Equivalent to the following:

```
main = forever $ do {
    exitCond <- isEOF;
    when exitCond exitSuccess;
    inp <- getLine;
    putStrLn $ f inp;
}
```

The magic of `interact`

- Repeatedly read a list of integers on each line and print its reverse

The magic of `interact`

- Repeatedly read a list of integers on each line and print its reverse
- Using `interact`

```
import System.IO
main = interact (unlines . map f . lines)

f :: String -> String
f inp = show (reverse (read inp :: [Int]))
```

The magic of `interact`

- Repeatedly read a list of integers on each line and print its reverse
- Using `interact`

```
import System.IO
main = interact (unlines . map f . lines)

f :: String -> String
f inp = show (reverse (read inp :: [Int]))
```

- `f` is required to be of type `String -> String`

The magic of `interact`

- Repeatedly read a list of integers on each line and print its reverse
- Using `interact`

```
import System.IO
main = interact (unlines . map f . lines)

f :: String -> String
f inp = show (reverse (read inp :: [Int]))
```

- `f` is required to be of type `String -> String`
- Hence we apply `read` to the input first, process it, and then apply `show` at the end

The bind operator

- Two fundamental functions used to construct and combine actions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

The bind operator

- Two fundamental functions used to construct and combine actions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

- Execution of `act1 >>= act2`

The bind operator

- Two fundamental functions used to construct and combine actions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

- Execution of `act1 >>= act2`
  - executes `act1`

The bind operator

- Two fundamental functions used to construct and combine actions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

- Execution of `act1 >>= act2`
  - executes `act1`
  - unboxes and extracts the return value (of type `a`)

The bind operator

- Two fundamental functions used to construct and combine actions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

- Execution of `act1 >>= act2`
  - executes `act1`
  - unboxes and extracts the return value (of type `a`)
  - executes `act2`, perhaps using the previously extracted value

The bind operator

- Two fundamental functions used to construct and combine actions

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

- Execution of `act1 >>= act2`
  - executes `act1`
  - unboxes and extracts the return value (of type `a`)
  - executes `act2`, perhaps using the previously extracted value
- The return value of `act2` is returned by the combined action

The bind operator

- Actually, `return` and `(>>=)` are functions common to all **monads**

The bind operator

- Actually, `return` and `(>>=)` are functions common to all **monads**
- `IO` is an example of a monad

The bind operator

- Actually, `return` and `(>>=)` are functions common to all **monads**
- `IO` is an example of a monad
- Many other type constructors we have already seen produce monads – `[]`, `Maybe` &c.

The bind operator

- Actually, `return` and `(>>=)` are functions common to all **monads**
- `IO` is an example of a monad
- Many other type constructors we have already seen produce monads – `[]`, `Maybe` &c.
- We will (perhaps!) see other examples of monads later

The bind operator

- Actually, `return` and `(>>=)` are functions common to all **monads**
- `IO` is an example of a monad
- Many other type constructors we have already seen produce monads – `[]`, `Maybe` &c.
- We will (perhaps!) see other examples of monads later
- Functions like `readLn`, `putStrLn`, `print` &c. are specific to the `IO` monad

Using bind

- Read a line and print it

```
getLine >>= putStrLn
```

Using bind

- Read a line and print it

```
getLine >>= putStrLn
```

- Read a line and print its length

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

getLine >>= (\str ->
    print (length str)
    )
```

Using bind

- Read a line and print its length twice

```
getLine >>= (\str ->
    print (length str) >>=
    print (length str)
    )
```

Using bind

- Read a line and print its length twice

```
getLine >>= (\str ->
    print (length str) >>=
    print (length str)
    )
```

- This produces a type error

Using bind

- Read a line and print its length twice

```
getLine >>= (\str ->
    print (length str) >>=
    print (length str)
    )
```

- This produces a type error
  - The second `(>>=)` expects a second argument of type `() -> IO c`

Using bind

- Read a line and print its length twice

```
getLine >>= (\str ->
    print (length str) >>=
    print (length str)
    )
```

- This produces a type error
  - The second `(>>=)` expects a second argument of type `() -> IO c`
  - But `print x` is of type `IO ()`

Using bind

- Read a line and print its length twice

```
getLine >>= (\str ->
    print (length str) >>=
    print (length str)
    )
```

- This produces a type error
  - The second `(>>=)` expects a second argument of type `() -> IO c`
  - But `print x` is of type `IO ()`
- Correct code!

```
getLine >>= (\str -> print (length str) >>=
                (\str' -> print (length str)))
```

Bind without arguments

- A simpler version of the previous action:

```
getLine >>= (\str ->
    print (length str) >>
    print (length str)
)
```

Bind without arguments

- A simpler version of the previous action:

```
getLine >>= (\str ->
    print (length str) >>
    print (length str)
)
```

- If we do not want to unbox and use the result of the preceding action, we use `(>>)`

Bind without arguments

- A simpler version of the previous action:

```
getLine >>= (\str ->
    print (length str) >>
    print (length str)
)
```

- If we do not want to unbox and use the result of the preceding action, we use `(>>)`
- `act1 >> act2` is equivalent to the following (where the name `n` is not used in `act2`):

```
act1 >>= (\n -> act2)
```

Bind without arguments

Consider the definitions (where `y` does not occur in `exp2`)

```
f x = exp1
g y = exp2
h = g (f 10)
```

- `f 10` is not evaluated when computing `h`

Bind without arguments

Consider the definitions (where `y` does not occur in `exp2`)

```
f x = exp1
g y = exp2
h = g (f 10)
```

- `f 10` is not evaluated when computing `h`
- Given actions `act1` and `act2`, executing `act1 >> act2` always executes `act1`, even though its return value is not used in `act2`

Bind without arguments

Consider the definitions (where `y` does not occur in `exp2`)

```
f x = exp1
g y = exp2
h = g (f 10)
```

- `f 10` is not evaluated when computing `h`
- Given actions `act1` and `act2`, executing `act1 >> act2` always executes `act1`, even though its return value is not used in `act2`
- The operators `(>>=)` and `(>>)` force execution of both the arguments, the left one first and then the right one

`do` *is syntactic sugar*

- The `do` blocks introduced earlier can be translated in terms of `(>>=)` and `(>>)`

`do` *is syntactic sugar*

- The `do` blocks introduced earlier can be translated in terms of `(>>=)` and `(>>)`
- A single action needs no `do`

```
do {putStrLn "Hello world!";}
```

translates to

```
putStrLn "Hello world!"
```

`do` *is syntactic sugar*

- If there is no `<-` in the first action, we use `>>`

```
do {act1; S}
```

translates to

```
act1 >> do {S}
```

`do` *is syntactic sugar*

- If there is no `<-` in the first action, we use `>>`

```
do {act1; S}
```

translates to

```
act1 >> do {S}
```

- If there is `<-` in the first action, we use `>>=`

```
do {n <- act1; S}
```

translates to

```
act1 >>= \n -> do {S}
```