Input and Output

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Input and Output

- So far, we still cannot receive input from keyboard or print output on the console screen
- We will cover the following basics:
  - What are I/O actions?
  - How do I/O actions enable us to do I/O?
  - When are I/O actions actually performed?
- Unfortunately, I/O actions create an issue that conflicts with how Haskell works?
- They are not **pure** because they have side effects

Separating the Pure from the Impure

- In Haskell, a function is not allowed to have **side effect**
- If a function is called two times with the same arguments, it must return the same result
- A function cannot change or update variables
  - When we map a list, we get a new list

```
ghci> original = [1,2,3]
ghci> new = map (*2) original
ghci> original
[1,2,3]
ghci> new
[2,4,6]
```

  - When we insert a new item into our binary search tree, it must return a new tree with new item inserted
  - If these expressions are in a file, it will cause an error:

```
myPi = 3.14
myPi = 3.1415
```

- Unfortunately, the output device such as the console screen is also considered to have a state
- To display something, simply change the state of the screen

Separating the Pure from the Impure

- This prevent a Haskell function to display output on the console screen
- Note that the output from `GHCi` is just a test platform
- We are talking about running the program from the console screen without running `GHCi`
- Here is an example of the `hello_world.hs`:

```
main = putStrLn "Hello World!!!"
```

- To compile, we use the command `ghc` with `--make` option:

```
$ ghc --make hello_world.hs
[1 of 1] Compiling Main             ( hello_world.hs, hello_world.o )
Linking hello_world ...
```

- Now, we can execute it just like a regular executable file:

```
$ ./hello_world
Hello World!!!
```

Separating the Pure from the Impure

- Let's check the `putStrLn` function in `GHCi`:

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "Hello World!!!"
putStrLn "Hello World!!!" :: IO ()
```

- The `putStrLn` takes an argument of type `String` and returns an **I/O action** that has a result of type `()`
- `()` is known as **unit**
- An I/O action is something that, when performed, will carry out an action with **side effect**, and will also present some result
- Since printing a string to the console does not really have any kind of return value, so a dummy value of `()` is used
- An I/O action will perform when we give it a name of `main` and then run the program

Gluing I/O Actions

- In Haskell, you can only bind a name to an expression
- A sequence of I/O actions must be glued into one to be able to bind with `main`
- Here is an example:

```
main = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn ("Hello " ++ name)
```

- The `do` keyword, takes multiple I/O actions and glues them into one I/O action
- Using the `do` keyword, the type of its I/O action will be the same as the type of the **last** I/O action
- In the above example, the type of `main` is `IO ()`

Separating the Pure from the Impure

- Let's take a look at the type of `getLine`

```
ghci> :t getLine
getLine :: IO String
```

- It indicates that `getLine` is an I/O action that returns a `String`
- Note that instead of

```
name = getLine
```

we use

```
name <- getLine
```

- This a how Haskell separates pure from impure
  - Since we perform `"Hello" ++ name`, `name` must have type `String`
  - But `getLine` has type `IO String`
  - To get the result out of an I/O action, we have to use `<-`
- If we want to deal with impure data, we must do it in an impure environment

Separating the Pure from the Impure

- Every I/O action yields a result

```
main = do
    foo <- putStrLn "What is your name?"
    name <- getLine
    putStrLn ("Hello " ++ name)
```

- Recall that `putStrLn` has type `String -> IO ()`
- `foo` will have type `()` (kind of useless)
- **Note** that in a `do` block, the last action **cannot** be bound to a name
- Again about I/O actions:
  - I/O action will be performed when they are given a name of `main` or they are inside a bigger I/O action that we composed with a `do` block
  - We can use a `do` block to glue a few I/O actions and then use it inside another `do` block
  - The type of a `do` block is the type of the last I/O action

Separating the Pure from the Impure

- To bind pure values to names, we use the `let` syntax
- We can use the `let` syntax inside I/O action

```
import Data.Char

main = do
    putStrLn "What is your name?"
    name <- getLine
    let bigName = map toUpper name
    putStrLn ("Hello " ++ bigName)
```

- Again only use `<-` to bind the result of an I/O action to a name
  - `getLine` has type `IO String`
  - With `name <- getLine`, `name` will have type `String`
- If we write `name = getLine`, `name` is another name of the `getLine` function

Caesar Cipher Program

- Here is a new version of our Caesar cipher:

```
import System.IO
import Data.Char

main = do
    putStrLn "Welcome to Caesar Cipher Program"
    putStrLn "================================="
    choice <- prompt "Enter 1 to Encode or 2 to Decode: "
    key <- prompt "Enter a key (integer): "
    msg <- prompt "Enter a message: "
    let result = cipher choice key msg
    putStrLn ("Result: " ++ result)

prompt :: String -> IO String
prompt str = do
    putStr str
    hFlush stdout
    getLine

cipher choice key msg = if (read choice :: Int) == 1
                        then encode (read key :: Int) msg
                        else decode (read key :: Int) msg

encode :: Int -> [Char] -> [Char]
encode n xs = map (chr . (+n) . ord) xs

decode :: Int -> [Char] -> [Char]
decode n xs = encode (negate n) xs
```

Caesar Cipher Program

- The following are `encode` and `decode` functions that we discused a while back:

```
encode :: Int -> [Char] -> [Char]
encode n xs = map (chr . (+n) . ord) xs

decode :: Int -> [Char] -> [Char]
decode n xs = encode (negate n) xs
```

- Consider this `cipher` function:

```
cipher choice key msg = if (read choice :: Int) == 1
                        then encode (read key :: Int) msg
                        else decode (read key :: Int) msg
```

  - Since the type of `choice` and `key` are `String`, we use the `read` function to turn them into `Int`
  - We can also use string comparison for `choice`

```
cipher choice key msg = if choice == "1"
                        then encode (read key :: Int) msg
                        else decode (read key :: Int) msg
```

Caesar Cipher Program

- Let's take a look at the `prompt` funciton:

```
prompt :: String -> IO String
prompt str = do
    putStr str
    hFlush stdout
    getLine
```

- This is an I/O action
- Inside are I/O actions under a `do` block
- The type of `prompt` function given a `String` is `IO String` which is the type of the `getLine` function
- With the `prompt` function, we can do this:

```
name <- prompt "Enter your name: "
```

- By default, `stdout` is bufferred until newline is found
- To print an output without newline, `hFlush stdout` is used to flush the `stdout`
- To use `hFlush`, we need to import `System.IO`

Caesar Cipher Program

- Here is the program again:

```
import System.IO
import Data.Char

main = do
    putStrLn "Welcome to Caesar Cipher Program"
    putStrLn "================================="
    choice <- prompt "Enter 1 to Encode or 2 to Decode: "
    key <- prompt "Enter a key (integer): "
    msg <- prompt "Enter a message: "
    let result = cipher choice key msg
    putStrLn ("Result: " ++ result)

prompt :: String -> IO String
prompt str = do
    putStr str
    hFlush stdout
    getLine

cipher choice key msg = if (read choice :: Int) == 1
                        then encode (read key :: Int) msg
                        else decode (read key :: Int) msg

encode :: Int -> [Char] -> [Char]
encode n xs = map (chr . (+n) . ord) xs

decode :: Int -> [Char] -> [Char]
decode n xs = encode (negate n) xs
```

Caesar Cipher Program

- To compile:

```
$ ghc --make caesar.hs
[1 of 1] Compiling Main             ( caesar.hs, caesar.o )
Linking caesar ...
```

- Here is a couple runs:

```
$ ./caesar
Welcome to Caesar Cipher Program
================================
Enter 1 to Encode or 2 to Decode: 1
Enter a key (integer): 5
Enter a message: Hello from Haskell
Result: Mjqqt%kwtr%Mfxpjqq

$ ./caesar
Welcome to Caesar Cipher Program
================================
Enter 1 to Encode or 2 to Decode: 2
Enter a key (integer): 5
Enter a message: Mjqqt%kwtr%Mfxpjqq
Result: Hello from Haskell
```

Example 1

- Go back to `main` is fine:

```
main = do
    putStrLn "What is your name? (0 - to exit)"
    name <- getLine
    if name == "0"
       then return ()
       else do
           putStrLn ("Hello " ++ name)
           main
```

- `main` is just another I/O action
- `then` and `else` must have the same type
  - The type of `else` is `IO ()`
  - We need `return ()` for `then`
- `return ()` does not cause the program to terminate
  - `return x` packs the value `x` into `IO a` where `a` is the type of `x`
  - `msg <- return "Hello"` will make the `msg` to have the value `"Hello"` of type `[Char]`

Example 2

- Enter a choice without checking:

```
main = do
    putStrLn "1 - This, 2 - That, 3 - Exit"
    choice <- getLine
    if choice /= "3"
       then doWork choice
       else return ()

doWork choice = if choice == "1"
                then putStrLn "This"
                else putStrLn "That"
```

- If a user enter something other than `1`, `2`, or `3`, it will print `That`

Example 3

- Ask until a valid choice is enter:

```
main = do
    choice <- getOneTwoOrThree
    if choice == "3"
        then return ()
        else doWork choice

getOneTwoOrThree = do
    putStrLn "1 - This, 2 - That, 3 - Exit"
    tempChoice <- getLine
    if tempChoice == "1" || tempChoice == "2" || tempChoice == "3"
       then return tempChoice
       else do
           putStrLn "Invalid input..."
           getOneTwoOrThree

doWork choice = if choice == "1"
                then putStrLn "This"
                else putStrLn "That"
```

The `putStr` Function

- `putStr` takes a string as an argument and returns an I/O action that will print that string to the terminal
- Example

```
main = do
    putStr "Hello "
    putStr "World"
    putStr "!!!"
```

- Output:

```
Hello World!!!
```

- Note that `putStr` does not jump into a new line after printing
- If you want to see the output without printing a newline, use `hFlush stdout`

The `putChar` Function

- `putChar` takes a character and returns an I/O action that will print it to the terminal
- Example:

```
main = do
    putChar 'D'
    putChar 'o'
    putChar 'g'
```

- Output:

```
Dog
```

- `putStr` can be defined using `putChar` as shown below:

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

The `print` Function

- `print` takes a value of any type that is an instance of the `Show` type class and output the value on the terminal followed by a newline
- Basically,

```
print x = putStrLn (show x)
```

- Examples:

```
main = do
    print True
    print [1,2,3]
    print "Hello"
    print 3.14159265
```

- Examples:

```
True
[1,2,3]
Hello
3.14159265
```

- Every time we type a value in GHCi, it actually call the `print` function on that value

The `when` Function

- `when` is a function that behaves like the `if` statement
- To use `when`, we need to import `Control.Monad`
- `when` takes a `Bool` and an I/O action
  - If the `Bool` is true, it returns the I/O action
  - Otherwise, it return `return ()` action
- Use `when`:

```
main = do
    input <- getLine
    when (input == "Hello") (putStrLn "What's Up???")
```

- Without `when`:

```
main = do
    input <- getLine
    if (input == "Hello")
        then putStrLn "What's Up???"
        else return ()
```

The `sequence` Function

- The `sequence` function takes a list of I/O actions and returns an I/O action that will perform those actions one after another
- The following program will wait for a user to enter three lines:

```
main = do
    lst <- sequence [getLine, getLine, getLine]
    print lst
```

- You may see something strange when the `sequence` function is used in GHCi:

```
ghci> sequence (map print [1,2])
1
2
[(),()]
```

- `sequence` returns the list of actions and GHCi prints it

The mapM Function

- Mapping a function that returns an I/O action over a list is quite common
- Instead of using `sequence` and `map`, the utility functions `mapM` and `mapM_` is provided
- `mapM` takes a function and a list, maps the function over the list, and then **sequence** it
- `mapM_` does the same thing, but it throws away the result later
- Example:

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> sequence [print 1, print 2, print 3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

The `forever` Function

- The `forever` function takes an I/O action and returns an I/O action, and repeats the I/O action forever
- To use the `forever` function, we need to import `Control.Monad`
- This program will as a user for his/her name forever:

```
import Control.Monad

main = forever (do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn ("Hello " ++ name))
```

The `forM` Function

- The `forM` function is similar to the `mapM` function, but its parameters are switched
  - The first argument is a list
  - The second argument is a function to map over the list
- Often time, if the function to map over the list is pretty long, put it at the end of an expression will make the code easier to read

I/O Action

- I/O actions are values in Haskell
- We can pass them as arguments
- Functions can return I/O actions as result
- I/O actions are performed if they fall into the `main` function (or in GHCi)
- The `do` block is used to glue multiple I/O actions into one and the type of the block is the type of the last I/O action