CSc 372

Comparative Programming Languages

17 : Haskell — Input/Output

Department of Computer Science
University of Arizona

collberg@gmail.com

The World isn't Functional

- The real world isn't functional:
  1. The real world has state
  2. In the real world order matters
- For example, if I move my book from the table to the chair, I have changed the state of the world.
- Also, to hang a painting on the wall I should
  1. Hammer a nail in the wall
  2. Hang the picture on the nail.

  The reverse order doesn't work as well!

Actions in Haskell

- To be useful, Haskell programs need to interact with the real world.
- For example:
  - reading and writing to files
  - graphics
  - networking
- Haskell supports a notion of actions that happen in sequence.

IO in Haskell

- Consider this program:

```
main' :: IO ()
main' =
    do
        putStr "Please enter your name: "
        name <- getLine
        putStr ("Your name is '"++name++"'\n")
```

- and its execution:

```
> main'
Please entre your name: bob
Your name is 'bob'
```

What is ()?

- () is a type that contains exactly one element, namely ():

```
> ()
()
> show ()
"()"
> :type ()
() :: ()
```

- Note that () is both a type and a value!

What is `IO ()`?

- `IO ()` is the type of a function that performs some I/O actions, but which returns nothing.
- If we wanted our program to return a string after performing some actions, its type would be `IO String`.

What does do do?

- do sequences together multiple actions:

```
do
    action 1
    action 2
    ...
    action n
```

- You can use this alternate syntax:

```
do {action 1; action 2; ...; action n}
```

What is an action?

- There are essentially four kinds of actions:

```
do
    let n = expression
    v <- getLine
    putStr expression
    return expression
```

- `let` just binds a name to an expression, like we've seen before.
- `<-` reads some value, and binds it to a name.
- `putStr` (and other similar output functions) print some value.
- `return` "packages up a value" so that we can return it from an I/O function.

What I/O functions are there?

- `putStr :: String -> IO ()`
- `putChar :: Char -> IO ()`
- `getChar :: IO Char`
- `getLine :: IO String`
- `return :: Monad m => a -> m a`

How does a function return a read value?

- Reading a string:

```
getName :: IO String
getName =
    do
        putStr "Please enter your name: "
        name <- getLine
        return name
```

- Running the function:

```
> getName
Please entre your name: Bob
"Bob"
```

Can we return any type?

- Reading a string:

```
yes_no :: String -> IO Bool
yes_no prompt =
    do
        putStr prompt
        c <- getChar
        let ok = c == 'y'
        putStr "\n"
        return ok
```

Example: Password database

```
passwd :: [(String,String)]
passwd = [("john","monkey"),("alice","banana")]

update :: String -> String -> [(String,String)]
                           -> [(String,String)]
update name new_pw passwd =
   map (\ (n,p) -> if (n==name) then (n,new_pw)
                                else (n,p)) passwd

isUser :: String -> [(String,String)] -> Bool
isUser name passwd = foldr f False passwd
                     where f = (\(n,_) r ->
                               if n==name then True else r)

okPassword :: String -> String -> [(String,String)]
                                -> Bool
okPassword name pw passwd = elem (name,pw) passwd
```

Example: Password database

```
> passwd
[("john","monkey"),("alice","banana")]
> update "john" "chimp" passwd
[("john","chimp"),("alice","banana")]
> isUser "alice"
> isUser "alice" passwd
True
> okPassword "alice" "banana" passwd
True
```

Example: Password database

```
getUser :: IO String
getUser =
    do
        putStr "Enter your name: "
        name <- getLine
        if isUser name passwd then
            return name else
         do
            putStr "Not a user, try again!\n"
            getUser
```

Example: Password database

```
checkPW :: String -> IO ()
checkPW name =
    do
        putStr "Enter your password: "
        pw <- getLine
        if okPassword name pw passwd then
          return ()
        else
          do
              putStr "Wrong password, try again!\n"
              checkPW name
```

Example: Password database

```
changePW :: String -> IO ()
changePW name =
   do
      putStr "Enter new password: "
      new1 <- getLine
      putStr "Confirm new password: "
      new2 <- getLine
      if new1 == new2 then
         do
            let new_pw = update name new1 passwd
            putStr ("New passwd: "++show new_pw++"\n")
      else
         do
            putStr "Passwords don't match,try again!\n"
            changePW name
```

Sequencing actions

- The `sequence` function takes a list of actions as input and executes them in turn:

```
welcomeUser :: String -> IO [()]
welcomeUser name = sequence [
                        putStr "Hello ",
                        putStr name,
                        putChar '!',
                        putChar '\n']
```

- Here's the output:

```
Hello john!
[(),(),(),()]
```

- Here's another example:

```
> sequence (map putChar "hello\n")
hello
```

Sequencing actions...

- Here's another example:

```
> sequence (map putChar "hello\n")
hello
[(),(),(),(),(),()]
```

Example: Password database

```
main :: IO ()
main =
    do
        name <- getUser
        welcomeUser name
        checkPW name
        ok <- yes_no "Change password [y/n]?"
        if ok then
            changePW name
        else
            return ()
```

- Note the use of `return()` in the `else` clause!

Reading From Files

- Here is how we open a file:

```
type FilePath = String
openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()
data IOMode = ReadMode | WriteMode |
              AppendMode | ReadWriteMode
```

- We can (lazily!) read in the entire file:

```
hGetContents :: Handle -> IO String
```

Reading From Files

- Here's a password file:

```
john monkey
alice banana
```

- Reading in the entire file:

```
readF =
    do
        f <- openFile "passwd" ReadMode
        c <- hGetContents f
        return c
```

- And here's the file read as a string:

```
> readF
"john monkey\nalice banana\n\n"
```

Reading and Parsing the Password File

```
openPWFile = do
      handle <- Control.Exception.catch
                    (openFile "passwd" ReadMode)
                    (\ e -> error
                                (
      show (e::Control.Exception.IOException)
                                )
                    )
      return handle
```

Reading and Parsing the Password File...

```
readPWFile f = do
    file <- hGetContents f
    let d = map words (lines file)
    let d' = filter (/=[]) d
    let parse xs =  case xs of {
                        [a,b] -> (a,b);
                        _ -> error (
                          "error:"++show xs
                                   )
                    }
    let res = map parse d'
    return res
```

Reading and Parsing the Password File...

```
load =
    do
        handle <- openPWFile
        passwd <- readPWFile handle
        return passwd
```

- Example:

```
> load
[("john","monkey"),("alice","banana")]
```

Exercise I

- Write a program `copy.hs` that copies a file `input.txt` to `output.txt`, while converting all characters to upper case.
- Here's the main program:

```
main :: IO ()
main = do
       inh<-openFile "input.txt" ReadMode
       outh<-openFile "output.txt" WriteMode
       mainloop inh outh
       hClose inh
       hClose outh
```

- Write the `mainloop` function!

```
mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
```

Exercise I...

- Useful functions:
  1. `hPutStrLn` *`outhandle string`*
  2. *`string`* `<-` `hGetLine` *`inhandle`*
  3. *`bool`* `<-` `hIsEOF` *`inhandle`*
- Input file:

```
> cat input.txt
there was a young lady of niger
who smiled as she rode on a tiger;
they returned from the ride
with the lady inside,
and the smile on the face of the tiger.
```

- Output file:

```
> cat output.txt
THERE WAS A YOUNG LADY OF NIGER...
```

Acknowledgments

- Bryan O'Sullivan, Don Stewart, and John Goerzen, *Real World Haskell*, http://book.realworldhaskell.org/read/io.html