



# CS 115

## Functional Programming

*Lecture 12: April 29, 2016*

### The **IO** Monad (part 2)



*Functional Programming: Spring 2016*



# Today

- Basic **IO** functions
- Stand-alone Haskell programs
- **ghci** and the **IO** monad
- Mutable references (**IORef**)





# IO functions

- We've already seen the `getLine` and `putStrLn` functions for reading and printing lines of text
- Now we'll look at various related I/O functions
- For more detailed documentation, look up the `System.IO` documentation on Hoogle





# Hoogle

- Hoogle is "Google for the Haskell APIs"
- Web site: <http://haskell.org/hoogle/>
- This is the Haskell programmer's best friend!
  - Searchable, detailed documentation of all Haskell modules
  - Links to source code if the explanation isn't good enough
  - Don't program in Haskell without it!
- Get **System.IO** by typing **System.IO** in the search box and clicking on the link





# Functions for input

- **getChar :: IO Char**
  - reads and returns a single character from **stdin**
- **getLine :: IO String**
  - reads and returns a single line from **stdin**, without the EOL character
- **getContents :: IO String**
  - returns all input from **stdin** until EOF as a single string
  - read lazily as needed





# Functions for output

- **putChar :: Char -> IO ()**
  - prints a single character to **stdout**
- **putStr :: String -> IO ()**
  - prints a string to **stdout**
- **putStrLn :: String -> IO ()**
  - same as **putStr**, but also prints a newline
- **print :: Show a => a -> IO ()**
  - prints any value whose type is an instance of **Show** to **stdout**, and adds a newline





# File handles

- Haskell uses the **Handle** type to represent file handles
- Standard **Handles**:
  - **stdin**
  - **stdout**
  - **stderr**
- Many I/O functions have **h**-equivalents, which are their generalizations to arbitrary file handles





# File handles

- Examples:
- `hGetChar :: Handle -> IO Char`
  - reads a character from a file represented by `Handle`
- `hGetLine :: Handle -> IO String`
- `hGetContents :: Handle -> IO String`
- `hPutChar :: Handle -> Char -> IO ()`
- `hPutStr :: Handle -> String -> IO ()`
- `hPutStrLn :: Handle -> String -> IO ()`
- `hPrint :: Show a => Handle -> a -> IO ()`







# File handles

- Other **Handle**-related functions/types:

```
type FilePath = String
```

```
data IOMode =
```

```
    ReadMode
```

```
    | WriteMode
```

```
    | AppendMode
```

```
    | ReadWriteMode
```

- **openFile :: FilePath -> IOMode -> IO Handle**
  - opens a file with **IOMode**, returns a **Handle**
- **hClose :: Handle -> IO ()**
  - closes a file handle





# Other file functions

- `readFile :: FilePath -> IO String`
  - reads the contents of an entire file (lazily)
- `writeFile :: FilePath -> String -> IO ()`
  - writes a string to a file
- `appendFile :: FilePath -> String -> IO ()`
  - appends a string to a file
- For other file functions/types/etc., see Hoogle!





# Writing standalone programs

- Haskell is a compiled language
- The "normal" mode of operation of Haskell is to compile a program to a stand-alone executable
- `ghci` is mainly used for testing code interactively
- Basic principles for standalone executables:
  - The starting point of the program is the function `main`
  - `main` must be in module `Main`
  - `main` must have type `IO ()`





# Simple standalone program

- The classic "hello world" program:

```
module Main where
```

```
main :: IO ()
```

```
main = putStrLn "hello, world!"
```

- Compile and run as follows:

```
% ghc -o hello Hello.hs
```

```
% hello
```

```
hello, world!
```





# Simple standalone program

- If module declaration is left out, `Main` module is assumed
- If name of module is other than `Main`, it won't work
  - execution starts with `Main.main`
- I recommend you also use the `-W` warning option to `ghc` *i.e.*

```
% ghc -W -o hello Hello.hs
```

- This turns on lots of useful warnings (like in `ghci`)





# Chasing dependencies

- Typically, the file where `main` is defined is called `Main.hs` (though it doesn't have to be)
- This file imports other modules used in the program
- If you type

```
% ghc Main.hs -o program_name
```
- Then `ghc` will compile `Main.hs` and all modules referenced from `Main.hs` that need to be recompiled (very convenient!)





# main and IO

- The **main** function has type **IO ()**
- It does some I/O, and returns nothing of significance
- This illustrates how **IO** works in Haskell:
  - **IO** actions are chained together using **>>=**
  - Eventually, the entire program is represented as one giant composite **IO** action called **main**
  - The runtime system is in charge of "running" this action to produce the effect of executing the program
  - This "running" of **main** is inherently non-functional





# ghci

- **ghci** is an extremely useful tool for Haskell programmers
- **ghci** doesn't accept the exact same language that is used in files of Haskell source code
- Some differences are obvious
  - e.g. `:` commands like `:load`, `:type`, `:kind` are **ghci**-specific
- Even the Haskell language used in **ghci** has significant differences/limitations
  - though the difference is getting smaller with newer versions







# ghci

- **ghci** works line-by-line
- All Haskell expressions entered must fit on a single line
- Code that would require indentation in Haskell source code can't be written in **ghci** the same way
  - though we'll see later how to fake it
- Fortunately, all indentation-dependent syntax in Haskell has a non-indentation-dependent form, usually involving curly braces and semicolons





# ghci

- Example: in source code file:

```
let x = 1
    y = 2
in x + y
```

- In **ghci**:

```
Prelude> let { x = 1 ; y = 2 } in x + y
3
```

```
Prelude> let x = 1 ; y = 2 in x + y
3
```





# ghci limitations

- Some Haskell constructs cannot be written in a **ghci** session
- Surprisingly, function definitions *can* be entered, but as a **let**-expression:

```
Prelude> let double x = 2 * x
```

```
Prelude> double 10
```

```
20
```

- Such a **let** scopes over the rest of the session
  - Similar to a **let** inside a **do**-expression





# ghci and IO

- **ghci** reads an expression, evaluates it, and prints the result (read-eval-print loop or REPL)
- **ghci** "knows" about **IO** actions and handles them specially: an **IO** action that results from evaluating an expression is executed immediately:

```
Prelude> :t putStrLn "hello"
```

```
putStrLn "hello" :: IO ()
```

```
Prelude> putStrLn "hello"
```

```
hello
```





# ghci and IO

- If an expression entered into **ghci** has the type **IO a** for some value **a**, then
  - the **IO** action is executed
  - the return value of type **a** is printed if its type is an instance of **Show** and is not **()**
- For instance:

```
Prelude> do { putStrLn "hello"; return "yes" }
```

```
hello      [printed]
```

```
"yes"     [returned]
```





## Aside: `putStrLn` vs. `print`

- There are two ways to print strings in Haskell
  - one represented by `putStr` and `putStrLn`
  - one represented by `print`
- Main difference: `print` wraps quotes around the string

```
Prelude> putStrLn "hello"
```

```
hello
```

```
Prelude> print "hello"
```

```
"hello"
```





## Aside: `putStrLn` vs. `print`

- Values printed as part of `ghci`'s REPL use `print`, not `putStrLn`, to print the value

```
Prelude> "hello"
```

```
"hello"
```

```
Prelude> return "hello" :: IO String
```

```
"hello"
```





# ghci and IO

- **ghci** also allows you to write expressions as if it were executing inside a **do**-expression:

```
Prelude> a <- return 10
```

```
Prelude> a
```

```
10
```







# ghci and functions

- Functions with multiple cases are tedious to define in **ghci** because of the one-line restriction:

```
Prelude> let fact 0 = 1; fact n = n * fact (n - 1)
```

```
Prelude> fact 10
```

```
3628800
```

- Can wrap function definition in curly braces, but this doesn't get around the one-line restriction:

```
Prelude> let { fact 0 = 1 ; fact n = n * fact (n - 1) }
```





# ghci and functions

- **ghci** allows you to define pseudo-multiline functions using the `:{` and `:}` delimiters (each on its own line):

```
Prelude> :{  
| let fact 0 = 1  
|     fact n = n * fact (n - 1)  
:}
```

- All lines inside `:{` and `:}` are combined into one line (without the leading `|`s on each line), separated by semicolons which are added for you





# Mutable reference cells

- Haskell fully supports imperative programming in the **IO** monad
- We've already seen how Haskell uses the **IO** monad to do input and output
- Other aspects of imperative programming include:
  - mutable variables
  - mutable arrays
- The **IO** monad can handle these too





# Mutable reference cells

- Haskell has no notion of a "variable" as such
- What other languages call "variables" are represented as a "reference cell"
- This is like a box that stores one item
- You can read from it or write to it
- In Haskell, this is called an **IORef** and must be used inside the **IO** monad





# Mutable reference cells

- To use `IORefs` you have to import the `Data.IORef` module
- In source code:

```
import Data.IORef
```

- In `ghci`:

```
Prelude> :m +Data.IORef
```

- Alternatively, you can type:

```
Prelude> import Data.IORef
```





# Mutable reference cells

- Key functions from `Data.IORef`:
- `newIORef :: a -> IO (IORef a)`
  - creates a new `IORef` from a value
- `readIORef :: IORef a -> IO a`
  - reads the value stored in an `IORef`
- `writeIORef :: IORef a -> a -> IO ()`
  - writes a value into an `IORef`, overwriting the previous value





# Mutable reference cells

- **IORefs** are particularly easy to use in **ghci** because of **ghci**'s handling of **IO** values:

```
Prelude> x <- newIORef 10
```

```
Prelude> readIORef x
```

```
10
```

```
Prelude> writeIORef x 20
```

```
Prelude> readIORef x
```

```
20
```





# Example: the gcd function

- We'll use **IORefs** to write a Haskell equivalent to this C function to compute greatest common denominators:

```
int gcd(int x, int y) {  
    while (x != y) {  
        if (x > y) {  
            x = x - y;  
        } else {  
            y = y - x;  
        }  
    }  
    return x;  
}
```







# Note!

- No real Haskell programmer would write a `gcd` function this way!
- `gcd` can be written purely functionally e.g.

```
gcd :: Integer -> Integer -> Integer
```

```
gcd x y =
```

```
  case compare x y of
```

```
    EQ -> x
```

```
    LT -> gcd x (y - x)
```

```
    GT -> gcd (x - y) y
```





# Note!

- However, it's been said that "real programmers can write C in any language"
- This is certainly going to be true for Haskell
- Our function will use **IORefs**, so it will have an **IO** type signature:

```
gcd :: Integer -> Integer -> IO Integer
```





# while loops in Haskell

- The C code uses a **while** loop
- Haskell doesn't have a **while** loop, so we have to write one!
  - call it **whileIO**
- We will use **IORefs** to store the state variables for the function
- Type signature of **whileIO**:

**whileIO :: IO Bool -> IO () -> IO ()**





# while loops in Haskell

`whileIO :: IO Bool -> IO () -> IO ()`

- The first argument is the test
  - do some `IO` action (consulting some `IORefs`) then return a boolean, which is the result of the test
- The second argument is the "block" of code to execute in the `IO` monad
- The return type represents the result of running the while loop





# while loops in Haskell

```
whileIO :: IO Bool -> IO () -> IO ()  
whileIO test block =  
    do b <- test  
       if b  
         then block >> whileIO test block  
         else return ()
```

- If the test is true, execute the block and repeat the while loop, otherwise you're done





# while loops in Haskell

- Could write this without `>>` operator:

```
whileIO :: IO Bool -> IO () -> IO ()
```

```
whileIO test block =
```

```
    do b <- test
```

```
        if b
```

```
            then do block
```

```
                whileIO test block
```

```
            else return ()
```





# Imperative gcd in Haskell

- Let's sketch out our imperative `gcd` function:

```
gcd :: Integer -> Integer -> IO Integer
```

```
gcd m n =
```

```
    do x <- newIORef m
```

```
       y <- newIORef n
```

```
       whileIO <x value is not equal to y value>
```

```
           <modify value stored in x or y>
```

```
       readIORef x
```





# Imperative gcd in Haskell

```
gcd :: Integer -> Integer -> IO Integer
gcd m n =
    do x <- newIORef m
       y <- newIORef n
       whileIO
           (do x' <- readIORef x
              y' <- readIORef y
              return (x' /= y'))
           <modify value stored in x or y>
       readIORef x
```







# Imperative gcd in Haskell

```
gcd :: Integer -> Integer -> IO Integer
gcd m n =
  do x <- newIORef m
     y <- newIORef n
  whileIO
    (do x' <- readIORef x
        y' <- readIORef y
        return (x' /= y'))
    (do x' <- readIORef x
        y' <- readIORef y
        if x' < y'
          then writeIORef y (y' - x')
          else writeIORef x (x' - y'))
  readIORef x
```





# liftM2 (for experts)

```
gcd :: Integer -> Integer -> IO Integer
gcd m n =
  do x <- newIORef m
     y <- newIORef n
  whileIO
    (liftM2 (/=) (readIORef x) (readIORef y))
    (do x' <- readIORef x
       y' <- readIORef y
       if x' < y'
         then writeIORef y (y' - x')
         else writeIORef x (x' - y'))
  readIORef x
```





# liftM2 (for experts)

- The `liftM2` function (from the `Control.Monad` module) can "lift" a binary function/operator into an arbitrary monad
- This can make code shorter, more natural

```
ghci> :t liftM2
```

```
liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
```

```
ghci> :t (/=)
```

```
(/=) :: Eq a => a -> a -> Bool
```

```
ghci> :t liftM2 (/=)
```

```
liftM2 (/=) :: (Eq a2, Monad m) => m a2 -> m a2 -> m Bool
```

- Here, we lift `(/=)` into a function taking two monadic values as arguments and returning a monadic `Bool`





# Imperative gcd in Haskell

- Conclusion:
- Imperative programming is possible in Haskell, and fairly straightforward
- Imperative programming is quite tedious compared to C
  - Must be extremely explicit about everything
- In C: whether a variable represents a value or a reference is handled automatically based on whether it's on the LHS or the RHS of the = sign





# Imperative gcd in Haskell

- Imperative style is not recommended in general!
- However, situations exist where writing code in an imperative style is more efficient than functional alternatives (good to have choices!)
- Also, useful for foreign function interfaces to *e.g.* C code
- Generally, functional code is easier to write/debug and may even be faster





# Imperative gcd in Haskell

- We will see a different way to write code in imperative style later in the course, when we cover *state monads*
- State monads will give us a *purely functional* way to simulate imperative code (reading/writing from/to state variables, but not input/output)





# Next time

- Practical:
  - Arrays: **Array** and **IOArray**
- Theoretical:
  - The three monad laws

