Roll your own IRC bot

This tutorial is designed as a practical guide to writing real world code in Haskell and hopes to intuitively motivate and introduce some of the advanced features of Haskell to the novice programmer. Our goal is to write a concise, robust and elegant IRC bot in Haskell.

1 Getting started

You'll need a reasonably recent version of GHC or Hugs. Our first step is to get on the network. So let's start by importing the Network package, and the standard IO library and defining a server to connect to.

```
import Network
import System.IO

server = "irc.freenode.org"
port   = 6667

main = do
    h <- connectTo server (PortNumber (fromIntegral
port))
    hSetBuffering h NoBuffering
    t <- hGetContents h
    print t
```

The key here is the `main` function. This is the entry point to a Haskell program. We first connect to the server, then set the buffering on the socket off. Once we've got a socket, we can then just read and print any data we receive.

Put this code in the module `1.hs` and we can then run it. Use whichever system you like:

Using runhaskell:

```
   $ runhaskell 1.hs
   "NOTICE AUTH :*** Looking up your hostname...\r\nNOTICE
AUTH :***
   Checking ident\r\nNOTICE AUTH :*** Found your hostname\r\n ...
```

Or we can just compile it to an executable with GHC:

```
   $ ghc --make 1.hs -o tutbot
   Chasing modules from: 1.hs
   Compiling Main             ( 1.hs, 1.o )
   Linking ...
   $ ./tutbot
   "NOTICE AUTH :*** Looking up your hostname...\r\nNOTICE
AUTH :***
   Checking ident\r\nNOTICE AUTH :*** Found your hostname\r\n ...
```

Or using GHCi:

```
   $ ghci 1.hs
   *Main> main
   "NOTICE AUTH :*** Looking up your hostname...\r\nNOTICE
AUTH :***
   Checking ident\r\nNOTICE AUTH :*** Found your hostname\r\n ...
```

Or in Hugs:

```
   $ runhugs 1.hs
   "NOTICE AUTH :*** Looking up your hostname...\r\nNOTICE
AUTH :***
   Checking ident\r\nNOTICE AUTH :*** Found your hostname\r\n ...
```

Great! We're on the network.

2 Talking IRC

Now we're listening to the server, we better start sending some information back. Three details are important: the nick, the user name, and a channel to join. So let's send those.

```
import Network
import System.IO
import Text.Printf

server = "irc.freenode.org"
port   = 6667
chan   = "#tutbot-testing"
nick   = "tutbot"

main = do
    h <- connectTo server (PortNumber (fromIntegral
port))
    hSetBuffering h NoBuffering
    write h "NICK" nick
    write h "USER" (nick++" 0 * :tutorial bot")
    write h "JOIN" chan
    listen h

write :: Handle -> String -> String -> IO ()
write h s t = do
    hPrintf h "%s %s\r\n" s t
    printf    "> %s %s\n" s t

listen :: Handle -> IO ()
listen h = forever $ do
    s <- hGetLine h
    putStrLn s
  where
    forever a = do a; forever a
```

Now, we've done quite a few things here. Firstly, we import `Text.Printf`, which will be useful. We also set up a channel name and bot nickname. The `main` function has been extended to send messages back to the IRC server using a `write` function. Let's look at that a bit more closely:

```
write :: Handle -> String -> String -> IO
()
write h s t = do
    hPrintf h "%s %s\r\n" s t
    printf    "> %s %s\n" s t
```

We've given `write` an explicit type to help document it, and we'll use explicit types signatures from now on, as they're just good practice (though of course not required, as Haskell uses type inference to work out the types anyway).

The `write` function takes 3 arguments; a handle (our socket), and then two strings representing an IRC protocol action, and any arguments it takes. `write` then uses `hPrintf` to build an IRC message and write it over the wire to the server. For debugging purposes we also print to standard output the message we send.

Our second function, `listen`, is as follows:

```
listen :: Handle -> IO ()
listen h = forever $ do
    s <- hGetLine h
    putStrLn s
  where
    forever a = do a; forever
a
```

This function takes a Handle argument, and sits in an infinite loop reading lines of text from the network and printing them. We take advantage of two powerful features; lazy evaluation and higher order functions to roll our own loop control structure, `forever`, as a normal function! `forever` takes a chunk of code as an argument, evaluates it and recurses - an infinite loop function. It is very common to roll our own control structures in Haskell this way, using higher order functions. No need to add new syntax to the language, lisp-like macros or meta programming - you just write a normal function to implement whatever control flow you wish. We can also avoid `do`-notation, and directly write: `forever a = a >> forever a`.

Let's run this thing:

```
$ runhaskell 2.hs
> NICK tutbot
> USER tutbot 0 * :tutorial bot
> JOIN #tutbot-testing
NOTICE AUTH :*** Looking up your hostname...
NOTICE AUTH :*** Found your hostname, welcome back
NOTICE AUTH :*** Checking ident
NOTICE AUTH :*** No identd (auth) response
:orwell.freenode.net 001 tutbot :Welcome to the freenode IRC Network
tutbot
:orwell.freenode.net 002 tutbot :Your host is orwell.freenode.net
...
:tutbot!n=tutbot@aa.bb.cc.dd JOIN :#tutbot-testing
:orwell.freenode.net MODE #tutbot-testing +ns
:orwell.freenode.net 353 tutbot @ #tutbot-testing :@tutbot
:orwell.freenode.net 366 tutbot #tutbot-testing :End of /NAMES list.
```

And we're in business! From an IRC client, we can watch the bot connect:

```
   15:02 -- tutbot [n=tutbot@aa.bb.cc.dd] has joined #tutbot-
testing
   15:02  dons> hello
```

And the bot logs to standard output:

```
   :dons!i=dons@my.net PRIVMSG #tutbot-
testing :hello
```

We can now implement some commands.

3 A simple interpreter

Add these additional imports before changing the `listen` function.

```
import Data.List
import
System.Exit
```

```
listen :: Handle -> IO ()
listen h = forever $ do
    t <- hGetLine h
    let s = init t
    if ping s then pong s else eval h (clean s)
    putStrLn s
  where
    forever a = a >> forever a

    clean     = drop 1 . dropWhile (/= ':') . drop
1

    ping x    = "PING :" `isPrefixOf` x
    pong x    = write h "PONG" (':' : drop 6 x)
```

We add 3 features to the bot here by modifying `listen`. Firstly, it responds to `PING` messages: `if ping s then pong s ...` . This is useful for servers that require pings to keep clients connected. Before we can process a command, remember the IRC protocol generates input lines of the form:

```
:dons!i=dons@my.net PRIVMSG #tutbot-testing :!id
foo
```

so we need a `clean` function to simply drop the leading ':' character, and then everything up to the next ':', leaving just the actual command content. We then pass this cleaned up string to `eval`, which then dispatches bot commands.

```
eval :: Handle -> String -> IO ()
eval h    "!quit"               = write h "QUIT" ":Exiting" >> exitWith
ExitSuccess
eval h x | "!id " `isPrefixOf` x = privmsg h (drop 4 x)
eval _   _                      = return () -- ignore everything else
```

So, if the single string "!quit" is received, we inform the server and exit the program. If a string beginning with "!id" appears, we echo any argument string back to the server (`id` is the Haskell identity function, which just returns its argument). Finally, if no other matches occur, we do nothing.

We add the `privmsg` function - a useful wrapper over `write` for sending `PRIVMSG` lines to the server.

```
privmsg :: Handle -> String -> IO ()
privmsg h s = write h "PRIVMSG" (chan ++ " :" ++
s)
```

Here's a transcript from our minimal bot running in channel:

```
   15:12 -- tutbot [n=tutbot@aa.bb.cc.dd] has joined #tutbot-
testing
   15:13  dons> !id hello, world!
   15:13  tutbot> hello, world!
   15:13  dons> !id very pleased to meet you.
   15:13  tutbot> very pleased to meet you.
   15:13  dons> !quit
   15:13 -- tutbot [n=tutbot@aa.bb.cc.dd] has quit [Client Quit]
```

Now, before we go further, let's refactor the code a bit.

4 Roll your own monad

A small annoyance so far has been that we've had to thread around our socket to every function that needs to talk to the network. The socket is essentially *immutable state*, that could be treated as a global read only value in other languages. In Haskell, we can implement such a structure using a state *monad*. Monads are a very powerful abstraction, and we'll only touch on them here. The interested reader is referred to All About Monads. We'll be using a custom monad specifically to implement a read-only global state for our bot.

The key requirement is that we wish to be able to perform IO actions, as well as thread a small state value transparently through the program. As this is Haskell, we can take the extra step of partitioning our stateful code from all other program code, using a new type.

So let's define a small state monad:

```
data Bot = Bot { socket :: Handle
}

type Net = ReaderT Bot IO
```

Firstly, we define a data type for the global state. In this case, it is the `Bot` type, a simple struct storing our network socket. We then layer this data type over our existing IO code, with a *monad transformer*. This isn't as scary as it sounds and the effect is that we can just treat the socket as a global read-only value anywhere we need it. We'll call this new io + state structure the `Net` monad. `ReaderT` is a *type constructor*, essentially a type function, that takes 2 types as arguments, building a result type: the `Net` monad type.

We can now throw out all that socket threading and just grab the socket when we need it. The key steps are connecting to the server, followed by the initialisation of our new state monad and then to run the main bot loop with that state. We add a small function, which takes the intial bot state and evaluates the bot's `run` loop "in" the Net monad, using the Reader monad's `runReaderT` function:

```
loop st = runReaderT run
st
```

where `run` is a small function to register the bot's nick, join a channel, and start listening for commands.

While we're here, we can tidy up the main function a little by using `Control.Exception.bracket` to explicitly delimit the connection, shutdown and main loop phases of the program - a useful technique.

```
main :: IO ()
main = bracket connect disconnect
loop
  where
    disconnect = hClose . socket
    loop st    = runReaderT run st
```

That is, the higher order function `bracket` takes 3 arguments: a function to connect to the server, a function to disconnect and a main loop to run in between. We can use `bracket` whenever we wish to run some code before and after a particular action - like `forever`, this is another control structure implemented as a normal Haskell function.

Rather than threading the socket around, we can now simply ask for it when needed. Note that the type of `write` changes - it is in the Net monad, which tells us that the bot must already by connected to a server (and thus it is ok to use the socket, as it is initialised).

```
--
-- Send a message out to the server we're currently connected
to
--
write :: String -> String -> Net ()
write s t = do
    h <- asks socket
    io $ hPrintf h "%s %s\r\n" s t
    io $ printf    "> %s %s\n" s t
```

In order to use both state and IO, we use the small `io` function to *lift* an IO expression into the Net monad making that IO function available to code in the `Net` monad.

```
io :: IO a -> Net
a
io = liftIO
```

Similarly, we can combine IO actions with pure functions by lifting them into the IO monad. We can therefore simplify our `hGetLine` call:

```
do t <- io (hGetLine
h)
   let s = init t
```

by lifting `init` over IO:

```
do s <- init `fmap` io (hGetLine
h)
```

The monadic, stateful, exception-handling bot in all its glory:

```
import Data.List
```

```
import Network
import System.IO
import System.Exit
import Control.Arrow
import Control.Monad.Reader
import Control.Exception
import Text.Printf

server = "irc.freenode.org"
port   = 6667
chan   = "#tutbot-testing"
nick   = "tutbot"

-- The 'Net' monad, a wrapper over IO, carrying the bot's immutable state.
type Net = ReaderT Bot IO
data Bot = Bot { socket :: Handle }

-- Set up actions to run on start and end, and run the main loop
main :: IO ()
main = bracket connect disconnect loop
  where
    disconnect = hClose . socket
    loop st    = runReaderT run st

-- Connect to the server and return the initial bot state
connect :: IO Bot
connect = notify $ do
    h <- connectTo server (PortNumber (fromIntegral port))
    hSetBuffering h NoBuffering
    return (Bot h)
  where
    notify a = bracket_
        (printf "Connecting to %s ... " server >> hFlush stdout)
        (putStrLn "done.")
        a

-- We're in the Net monad now, so we've connected successfully
-- Join a channel, and start processing commands
run :: Net ()
run = do
    write "NICK" nick
    write "USER" (nick++" 0 * :tutorial bot")
    write "JOIN" chan
    asks socket >>= listen

-- Process each line from the server
listen :: Handle -> Net ()
listen h = forever $ do
    s <- init `fmap` io (hGetLine h)
    io (putStrLn s)
    if ping s then pong s else eval (clean s)
  where
    forever a = a >> forever a
    clean     = drop 1 . dropWhile (/= ':') . drop 1
    ping x    = "PING :" `isPrefixOf` x
    pong x    = write "PONG" (':' : drop 6 x)
```

```
-- Dispatch a command
eval :: String -> Net ()
eval     "!quit"               = write "QUIT" ":Exiting" >> io (exitWith
ExitSuccess)
eval x | "!id " `isPrefixOf` x = privmsg (drop 4 x)
eval     _                     = return () -- ignore everything else

-- Send a privmsg to the current chan + server
privmsg :: String -> Net ()
privmsg s = write "PRIVMSG" (chan ++ " :" ++ s)

-- Send a message out to the server we're currently connected to
write :: String -> String -> Net ()
write s t = do
    h <- asks socket
    io $ hPrintf h "%s %s\r\n" s t
    io $ printf    "> %s %s\n" s t

-- Convenience.
io :: IO a -> Net a
io = liftIO
```

Note that we threw in a new control structure, `notify`, for fun. Now we're almost done! Let's run this bot. Using runhaskell:

```
   $ runhaskell
4.hs
```

or using GHC:

```
   $ ghc --make 4.hs -o tutbot
   Chasing modules from: 4.hs
   Compiling Main             ( 4.hs,
4.o )
   Linking ...
   $ ./tutbot
```

If you're using Hugs, you'll have to use the `-98` flag:

```
   $ runhugs -98
4.hs
```

And from an IRC client we can watch it connect:

```
   15:26 -- tutbot [n=tutbot@aa.bb.cc.dd] has joined #tutbot-
testing
   15:28  dons> !id all good?
   15:28  tutbot> all good?
   15:28  dons> !quit
   15:28 -- tutbot [n=tutbot@aa.bb.cc.dd] has quit [Client Quit]
```

So we now have a bot with explicit read-only monadic state, error handling, and some basic IRC operations. If we wished to add read-write state, we need only change the `ReaderT` transformer to `StateT`.

5 Extending the bot

Let's implement a basic new command: uptime tracking. Conceptually, we need to remember the time the bot starts. Then, if a user requests, we work out the total running time and print it as a string. A nice way to do this is to extend the bot's state with a start time field:

```
import
System.Time

data Bot = Bot { socket :: Handle, starttime :: ClockTime
}
```

We can then modify the initial `connect` function to also set the start time.

```
connect :: IO Bot
connect = notify $ do
    t <- getClockTime
    h <- connectTo server (PortNumber (fromIntegral
port))
    hSetBuffering h NoBuffering
    return (Bot h t)
```

We then add a new case to the `eval` function, to handle uptime requests:

```
eval "!uptime" = uptime >>=
privmsg
```

This will just run the `uptime` function and send it back to the server. `uptime` itself is:

```
uptime :: Net String
uptime = do
    now  <- io getClockTime
    zero <- asks starttime
    return . pretty $ diffClockTimes now
zero
```

That is, in the Net monad, find the current time and the start time, and then calculate the difference, returning that

number as a string. Rather than use the normal representation for dates, we'll write our own custom formatter for dates:

```
--
-- Pretty print the date in '1d 9h 9m 17s' format
--
pretty :: TimeDiff -> String
pretty td =
  unwords $ map (uncurry (++) . first show) $
  if null diffs then [(0,"s")] else diffs
  where merge (tot,acc) (sec,typ) = let (sec',tot') = divMod tot
sec
                                    in (tot',(sec',typ):acc)
        metrics = [(86400,"d"),(3600,"h"),(60,"m"),(1,"s")]
        diffs = filter ((/= 0) . fst) $ reverse $ snd $
                foldl' merge (tdSec td,[]) metrics
```

And that's it. Running the bot with this new command:

```
   16:03 -- tutbot [n=tutbot@aa.bb.cc.dd] has joined #tutbot-
testing
   16:03  dons> !uptime
   16:03  tutbot> 51s
   16:03  dons> !uptime
   16:03  tutbot> 1m 1s
   16:12  dons> !uptime
   16:12  tutbot> 9m 46s
```

6 Where to now?

This is just a flavour of application programming in Haskell, and only hints at the power of Haskell's lazy evaluation, static typing, monadic effects and higher order functions. There is much, much more to be said on these topics. Some places to start:

Or take the bot home and hack! Some suggestions:

- Use `forkIO` to add a command line interface, and you've got yourself an irc client with 4 more lines of code.
- Port some commands from Lambdabot.

Author: Don Stewart