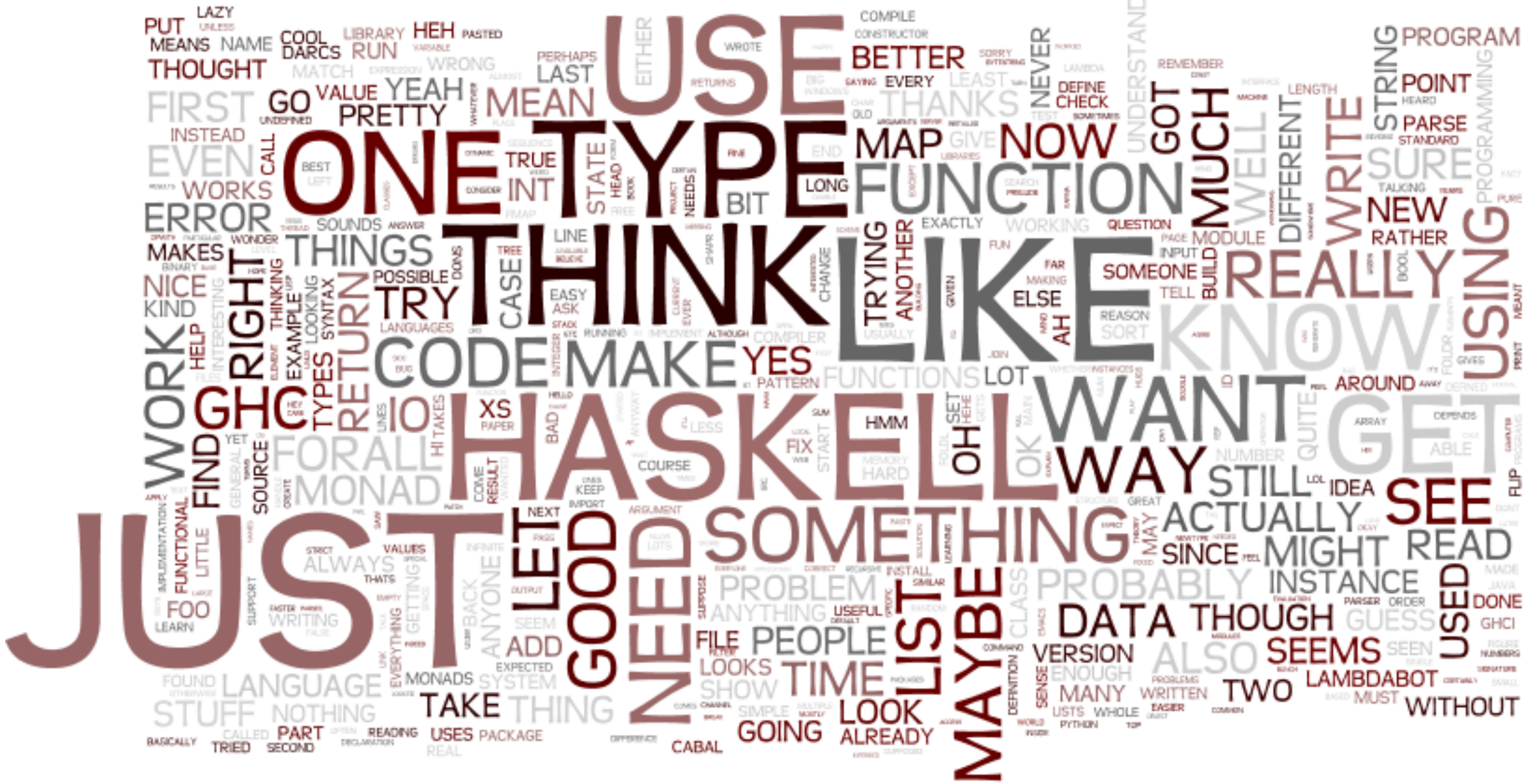
Freie Universität Berlin



Real World Haskell SoSe 2013
Julian Fleischer, Alexander Steen

Freie Universität Berlin

1: Introduction

Real World Haskell SoSe 2013
Julian Fleischer, Alexander Steen

```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network



main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = do
  (sock, host, _) <- accept socket


  let send = hPutStr sock
      text = "<h1>Hooray, Haskell</h1>"

  send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                (length text) text

  hFlush sock
  hClose sock

  printf "Anfrage von %s beantwortet\n" host
```

```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network



main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = do
  (sock, host, _) <- accept socket


  let send = hPutStr sock
      text = "<h1>Hooray, Haskell</h1>"

  send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                (length text) text

  hFlush sock
  hClose sock

  printf "Anfrage von %s beantwortet\n" host
```

```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network



main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = do
  (sock, host, _) <- accept socket


  let send = hPutStr sock
  text <- readFile "index.html"

  send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                (length text) text

  hFlush sock
  hClose sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception


main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket


  let send = hPutStr sock
  text <- readFile "index.html"

  send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                (length text) text

  hFlush sock
  hClose sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception


main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket


let send = hPutStr sock
text <- readFile "index.html"

send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
              (length text) text

hFlush sock
hClose sock

printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent

main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ do
    let send = hPutStr sock
    text <- readFile "index.html"

    send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                  (length text) text

    hFlush sock
    hClose sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent

main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ do
    let send = hPutStr sock
    text <- readFile "index.html"

    send $ printf "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                  (length text) text

    hFlush sock
    hClose sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent

main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text

    hFlush sock
    hClose sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent

main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text

    hFlush sock
    hClose sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent

main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text

    hFlush sock


  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent

main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

forkIO $ flip finally (hClose sock) $ do

  text <- readFile "index.html"

  hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
               (length text) text

  hFlush sock


printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.Environment
import System.IO
import Control.Monad
import Text.Printf
import Network
import Control.Exception
import Control.Concurrent


main = listenOn (PortNumber 8080) >>= forever . serve


serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.Environment (getArgs)
import System.IO (hFlush, hClose)
import Control.Monad (forever)
import Text.Printf (hPrintf, printf)
import Network (listenOn, accept, Socket, PortID (..))
import Control.Exception (handle, finally, SomeException)
import Control.Concurrent (forkIO)


main = listenOn (PortNumber 8080) >>= forever . serve


serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```


```haskell
import System.Environment (getArgs)
import System.IO (hFlush, hClose)
import Control.Monad (forever)
import Text.Printf (hPrintf, printf)
import Network (listenOn, accept, Socket, PortID (..))
import Control.Exception (handle, finally, SomeException)
import Control.Concurrent (forkIO)

main :: IO ()
main = listenOn (PortNumber 8080) >>= forever . serve

serve :: Socket -> IO ()
serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```


```
import System.Environment (getArgs)
import System.IO (hFlush, hClose)
import Control.Monad (forever)
import Text.Printf (hPrintf, printf)
import Network (listenOn, accept, Socket, PortID (..))
import Control.Exception (handle, finally, SomeException)
import Control.Concurrent (forkIO)

main :: IO ()
main = listenOn (PortNumber 8080) >>= forever . serve

serve :: Socket -> IO ()
serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```


```
import System.IO (hFlush, hClose)
import Control.Monad (forever)
import Text.Printf (hPrintf)
import Network (listenOn, accept, Socket, PortID (..))
import Control.Exception (handle, finally)
import Control.Concurrent (forkIO)
import System.Environment (getArgs)

main :: IO ()
main = getArgs >>= return . read . (!! 0)
               >>= listenOn . PortNumber . fromIntegral
               >>= forever . serve

serve :: Socket -> IO ()
serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```

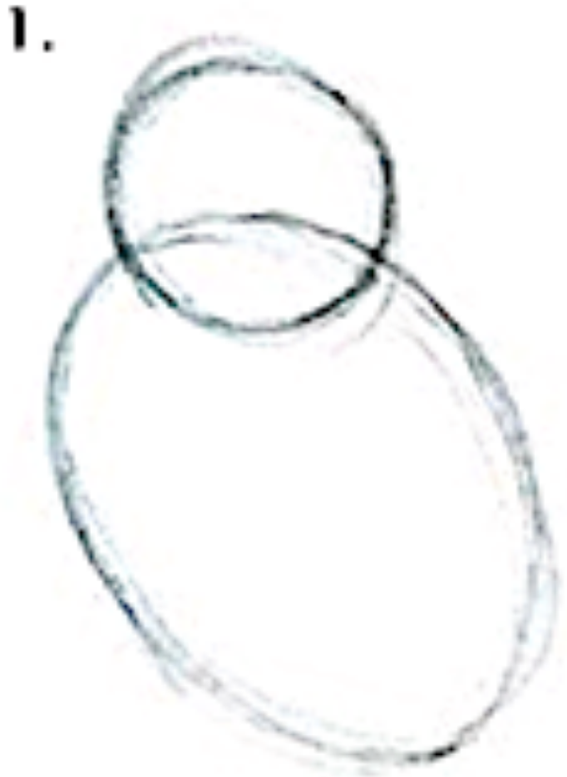

```haskell
import System.IO (hFlush, hClose)
import Control.Monad (forever)
import Text.Printf (hPrintf)
import Network (listenOn, accept, Socket, PortID (..))
import Control.Exception (handle, finally)
import Control.Concurrent (forkIO)
import System.Environment (getArgs)

main :: IO ()
main = getArgs >>= return . read . (!! 0)
               >>= listenOn . PortNumber . fromIntegral
               >>= forever . serve

serve :: Socket -> IO ()
serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```

Freie Universität Berlin

```haskell
main = do
  socket <- listenOn (PortNumber 8080)
  forever (serve socket)

serve socket = do
  (sock, host, _) <- accept socket


  let send = hPutStr sock
      text = "<h1>Hooray, Haskell</h1>"

  send $ printf  "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text

  hFlush sock
  hClose sock

  printf "Anfrage von %s beantwortet\n" host
```

```
main = getArgs >>= return . read . (!! 0)
               >>= listenOn . PortNumber . fromIntegral
               >>= forever . serve

serve socket = handle (\e -> print (e :: SomeException)) $ do
  (sock, host, _) <- accept socket

  forkIO $ flip finally (hClose sock) $ do

    text <- readFile "index.html"

    hPrintf sock "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s"
                 (length text) text >> hFlush sock

  printf "Anfrage von %s beantwortet\n" host
```

How to draw an owl



Draw some circles

Real World Haskell SoSe 2013
Julian Fleischer, Alexander Steen

main :: IO ()

Freie Universität Berlin

```haskell
{-# LANGUAGE Haskell2010 #-}

-- | Mein erstes echtes Haskell Programm
module Main where

import System.Environment
import System.IO

-- | Die main-Funktion ist die erste Funktion die aufgerufen wird
main :: IO ()
main = putStrLn "Hooray, Haskell!"

{- oder: main = getLine >>= putStrLn

   oder: main = do
           putStr "Your Name: "
           line <- getLine
           putStrLn $ "It is " ++ line -}
```

cabal

```
$ cabal init
> Package name? [default: ...] fu-haskell-webserver
> Package version? [default: 0.1.0.0] 1.0
> Please choose a license:
  ...
> Your choice [default: (none)]: <ENTER>
> Author name? Julian Fleischer
> Maintainer email? julian.fleischer@fu-berlin.de
> Project homepage URL? <ENTER>
> Project synopsis? My first cabal package
> Project category?
  ...
> Your choice [default: (none)]: <ENTER>
> What does the package build:
>   1) Library
>   2) Executable
> Your choice? 2
  ...
```

cabal (2)

Freie Universität Berlin

```
$ <EDIT> fu-haskell-webserver.cabal

executable fu-haskell-webserver
    -- main-is:
    -- other-modules:
    build-depends:       base ==4.5.*, network ==2.3.*
```



```
executable fu-haskell-webserver
    main-is: webserver.hs
    -- other-modules:
    build-depends:       base ==4.5.*, network ==2.3.*
```

cabal (3)

Freie Universität Berlin

```
$ cabal configure
Resolving dependencies...
Configuring fu-haskell-webserver-1.0...
...

$ cabal build
Building fu-haskell-webserver-1.0...
Preprocessing executable 'fu-haskell-webserver' for
fu-haskell-webserver-1.0...
[1 of 1] Compiling Main             ( ... )
Linking dist/build/fu-haskell-webserver/fu-haskell-webserver ...

$ ./dist/build/fu-haskell-webserver/fu-haskell-webserver
...

( $ cabal install )
```

runhaskell

Freie Universität Berlin

```
$ runhaskell webserver.hs
...
```

Freie Universität Berlin

1.

2.

Draw the rest of the fucking owl !

Real World Haskell SoSe 2013
Julian Fleischer, Alexander Steen

Appendix

- Search-Engine for function signatures
  http://www.haskell.org/hoogle
- The Haskell Platform ("Library Doc")
  http://www.haskell.org/platform/

Appendix (2)

```
main   :: IO ( )

(>>=)  :: IO a → (b → IO b)

(>>)   :: IO a → (IO b → IO b)
```

```
System.Environment (getArgs)

getArgs :: IO [String]
```

Appendix (3)

Freie Universität Berlin

```
do
   a <- f x1 x2 x3
   f2 a
```

```
f x1 x2 x3 >>= (\a -> f2 a)
```

Appendix (4)

Freie Universität Berlin

```
do
   doSomething
   doSomethingElse
```

```
doSomething >> doSomethingElse
```

Appendix (5)

```
id :: a → a
id x = x                ⇔    id = \x -> x

return :: a → M a
return x = M x          ⇔    return = M

(.) :: (b → c) → (a → b) → (a → c)
(.) f g x = f (g x)
                        ⇔  f . g = \x -> f (g x)
```