Intro to Haskell

Zachary Stigall

Colorado School of Mines

May 25, 2013

Install

- Linux:
  - Debian/Ubuntu:

    `apt-get install haskell-platform`
  - Fedora/CentOS/Redhat:

    `yum install haskell-platform`
  - or use justhub
  - Arch: Install from the AUR

Install

- Linux:
  - Debian/Ubuntu:

    `apt-get install haskell-platform`
  - Fedora/CentOS/Redhat:

    `yum install haskell-platform`
  - or use justhub
  - Arch: Install from the AUR
- Windows:
  - Go to: `http://www.haskell.org/platform/windows.html` download, install

Install

- Linux:
  - Debian/Ubuntu:

    `apt-get install haskell-platform`
  - Fedora/CentOS/Redhat:

    `yum install haskell-platform`
  - or use justhub
  - Arch: Install from the AUR
- Windows:
  - Go to: `http://www.haskell.org/platform/windows.html` download, install
- Mac:
  - Option 1: Go to `http://www.haskell.org/platform/mac.html` download, install. Requires command line devel tools from XCode
  - Option 2: MacPorts, its in there somewhere
  - Option 3: HomeBrew, again its in there somewhere

Why the Haskell Platform

- Why not just ghc?

Why the Haskell Platform

- Why not just ghc?
- What does the Haskell Platform provide?

Why the Haskell Platform

- Why not just ghc?
- What does the Haskell Platform provide?
- Can I get away with just ghc?

About Haskell

- Functional!. . . Why?

About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.

About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.
- Strong and Static Typing that can also be Inferred

About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.
- Strong and Static Typing that can also be Inferred
- First Class Functions

About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.
- Strong and Static Typing that can also be Inferred
- First Class Functions
- Lazy Evaluation

About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.
- Strong and Static Typing that can also be Inferred
- First Class Functions
- Lazy Evaluation
- Tons and Tons of syntactic sugar

About Haskell

- Functional!. . . Why?
- Pure! but has ways of handling impure stuff also.
- Strong and Static Typing that can also be Inferred
- First Class Functions
- Lazy Evaluation
- Tons and Tons of syntactic sugar
- Haskell is Compiled, but it has an interpreter also (GHCi)

GHC

- GHC is massive, so massive I gave it its own slide

GHC

- GHC is massive, so massive I gave it its own slide
- It is smarter than you

GHC

- GHC is massive, so massive I gave it its own slide
- It is smarter than you
- It knows haskell way better than you, listen to its suggestions

GHC

- GHC is massive, so massive I gave it its own slide
- It is smarter than you
- It knows haskell way better than you, listen to its suggestions
- If it actually compiles your code, your code 99% of the time will not crash, and will do exactly what you wanted

GHC

- GHC is massive, so massive I gave it its own slide
- It is smarter than you
- It knows haskell way better than you, listen to its suggestions
- If it actually compiles your code, your code 99% of the time will not crash, and will do exactly what you wanted
- It is really, really good at what it does, Haskell is a High Performance language, often programs compete with C++, Java, or C in terms of speed.

Getting Started in GHCI

- Run by typing ghci in a terminal

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`
  - `:r[eload]`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`
  - `:r[eload]`
  - `:e[dit]`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`
  - `:r[eload]`
  - `:e[dit]`
  - `:set editor <executable>`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`
  - `:r[eload]`
  - `:e[dit]`
  - `:set editor <executable>`
  - `:set prompt ''<prompt string>''`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`
  - `:r[eload]`
  - `:e[dit]`
  - `:set editor <executable>`
  - `:set prompt ''<prompt string>''`
  - `:main <args>`

Getting Started in GHCI

- Run by typing ghci in a terminal
- Try some basic math
- By default Prelude is imported
- 'it' can be used to reference the last returned value
- Useful builtins
  - `:t[ype] <something>`
  - `:l[oad] <hs file>`
  - `:r[eload]`
  - `:e[dit]`
  - `:set editor <executable>`
  - `:set prompt ''<prompt string>''`
  - `:main <args>`
  - `:h[elp]`

First File

In your editor of choice create a new file, call it example.hs
A lot of the examples are inspired by Learn You a Haskell for Great Good

First File

Functions basics

- Name has to start with lowercase letter
- Convention says to use camelCase, but underscores are fine too

Create a function

```
1  twice x = 2 * x
```

First File

Functions basics

- Name has to start with lowercase letter
- Convention says to use camelCase, but underscores are fine too

Add a type signature

```
1  twice :: Integer -> Integer
2  twice x = 2 * x
3
4  --Just to show a common idiom
5  twice' :: Integer -> Integer
6  twice' x = x + x
```

First File

Using functions

- Function application looks a lot like the definition

```
1 twiceTwo :: Integer -> Integer -> Integer
2 twiceTwo x y = twice x + twice y
```

Control Structures

if..then..else

- All if statements must have both a then clause *AND* an else clause

```haskell
-- The 'even' function in Prelude does just this
-- Lets through a Type Class in here as well
isEven :: Integral a => a -> Bool
isEven x = if x `mod` 2 == 0
           then True
           else False
```

Control Structures

case *expr* of . . .

```
1  isEven :: Integral a => a -> Bool
2  isEven x = case x `mod` 2 of
3      0 -> True
4      1 -> False
```

Control Structures

Loops

- You don't need them

Control Structures

Loops

- You don't need them
- You don't have them

Control Structures

Loops

- You don't need them
- You don't have them
- You have to use some form of a map

Control Structures

Loops

- You don't need them
- You don't have them
- You have to use some form of a map
- Or a List Comprehension

Control Structures

Loops

- You don't need them
- You don't have them
- You have to use some form of a map
- Or a List Comprehension
- . . . or use recursion

Lists

Lists are the bread and butter of functional programming
You will use lists everywhere

Lists

Lists are the bread and butter of functional programming
You will use lists everywhere
Lists are homogeneous, meaning you can't mix types like you would in Python, Ruby, or other high level languages

Lists

Lists are the bread and butter of functional programming
You will use lists everywhere
Lists are homogeneous, meaning you can't mix types like you would in Python, Ruby, or other high level languages

```
1 aList :: [Integer]
2 aList = [1,2,3,4]
```

Ranges and Construction

The range operator

```
1  -- .. is inclusive on both ends
2  >>= [1..4]
3  [1,2,3,4]
4  >>= [1..]
5  [1,2,3,...]
6  >>= [0,2..]
7  [0,2,4,...]
8  -- If you want to go backwards
9  >>= [9,8..0]
10 [9,8,7,6,5,4,3,2,1,0]
```

Ranges and Construction

List construction

```
1  -- : pronounced cons => prepends an item
2  >>= 'a' : ['b', 'c']
3  "abc"
4
5  -- ++ concatenate
6  >>= "hello" ++ " " ++ "world"
7  "hello world"
```

List Groups

```
1  -- head : tail
2  >>= head [10..20]
3  10
4  >>= tail [1..5]
5  [2,3,4,5]
6
7  -- init ++ [last]
8  >>= init [1..5]
9  [1,2,3,4]
10 >>= last [1..5]
11 5
```

List Info

```
1  -- null :: [a] -> Bool
2  >>= null []
3  True
4  >>= null ['a'..'f']
5  False
6
7  -- length :: [a] -> Int
8  >>= length []
9  0
10 >>= length ['0..'z']
11 75
```

More Functions

```
1  >>= reverse [1..4]
2  [4,3,2,1]
3
4  >>= take 5 [10,20..]
5  [10,20,30,40,50]
6
7  >>= drop 3 [2..9]
8  [5,6,7,8,9]
```

More Functions

```
>>= maximum [7,2,3,10,5,9]
10

>>= minimum [7,2,3,10,5,9]
2

>>= sum [7,2,3,10,5,9]
36

>>= product [7,2,3,10,5,9]
18900
```

Check and Access

```
1  >>= 'a' `elem` "hello world"
2  False
3
4  >>= elem 'w' "hello world"
5  True
6
7  >>= "hello world" !! 4
8  'o'
9
10 >>= head $ tail $ tail $ tail $ tail "hello world"
11 'o'
```

Map

Applies a function to each element in a list and returns the new list

```
1 map :: (a -> b) -> [a] -> [b]
```

Map

Applies a function to each element in a list and returns the new list

```haskell
map :: (a -> b) -> [a] -> [b]
```

```python
#A little python code
for i in lst:
    newval = func(i)
    newlst.append(newval)

#Or use python's map
newlst = map(func, lst)

#Or a list comp
newlst = [func(i) for i in lst]
```

Map

Applies a function to each element in a list and returns the new list

```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
-- Multiplies each item in a list by 2
timesTwo lst = map (\x -> 2 * x) lst
```

Map

Applies a function to each element in a list and returns the new list

```
1  map :: (a -> b) -> [a] -> [b]
```

```
1  -- Lets Clean up our function def a little
2  -- First lets get rid of the lambda
3
4  timesTwo lst = map (* 2) lst
```

Map

Applies a function to each element in a list and returns the new list

```
1  map :: (a -> b) -> [a] -> [b]
```

```
1  -- Lets Clean up our function def a little
2  -- First lets get rid of the lambda
3  -- Second lets get rid of excess 'points'
4  timesTwo = map (* 2)
```

Map

Applies a function to each element in a list and returns the new list

```haskell
map :: (a -> b) -> [a] -> [b]
```

```haskell
-- Lets Clean up our function def a little
-- First lets get rid of the lambda
-- Second lets get rid of excess 'points'
timesTwo = map (* 2)
```

```haskell
-- How about a list comp
timesTwo xs = [ 2 * x | x <- xs ]
```

Filter

Checks each item against some condition. True ⇒ keep, False ⇒ Discard

```
1  filter :: (a -> Bool) -> [a] -> [a]
```

Filter

Checks each item against some condition. True ⇒ keep, False ⇒ Discard

```haskell
filter :: (a -> Bool) -> [a] -> [a]
```

```python
#More python
for i in lst:
    if func(i):
        newlst.append(i)

#Or python's filter
newlst = filter(func, lst)

#Or a list comp
newlst = [i for i in lst if func(i)]
```

Filter

Checks each item against some condition. True ⇒ keep, False ⇒ Discard

```
1 filter :: (a -> Bool) -> [a] -> [a]
```

```
1 -- Looks a lot like map
2 evens xs = filter even xs
```

Filter

Checks each item against some condition. True ⇒ keep, False ⇒ Discard

```haskell
filter :: (a -> Bool) -> [a] -> [a]
```

```haskell
-- Looks a lot like map
evens xs = filter even xs
```

```haskell
-- Cleaned up a bit
evens = filter even
```

Filter

Checks each item against some condition. True ⇒ keep, False ⇒ Discard

```
1  filter :: (a -> Bool) -> [a] -> [a]
```

```
1  -- Looks a lot like map
2  evens xs = filter even xs
```

```
1  -- Cleaned up a bit
2  evens = filter even
```

```
1  -- As a list comp
2  evens xs = [ x | x <- xs, even x ]
```

Folds

Applies a function to each item of a list and an accumulator, returns the accumulator

```
1 foldl :: (a -> b -> a) -> a -> [b] -> a
2 foldr :: (a -> b -> b) -> b -> [a] -> b
```

Folds

Applies a function to each item of a list and an accumulator, returns the accumulator

```haskell
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```python
#Python once more
for i in lst:
    accum = func(accum, i)

#Clear as mud right?
#Python also has a fold it is called reduce
from functools import reduce
reduce(func, lst)
```

Folds

Applies a function to each item of a list and an accumulator, returns the accumulator

```
1  foldl :: (a -> b -> a) -> a -> [b] -> a
2  foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
1  -- Lets make a difference function
2  -- Continually subtracts each number
3  diffl x:xs = foldl (-) x xs
4  -- Or foldr
5  diffr xs = foldr (-) (last xs) (init xs)
```

Folds

Applies a function to each item of a list and an accumulator, returns the accumulator

```
1  foldl :: (a -> b -> a) -> a -> [b] -> a
2  foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
1  -- These don't behave the same
2  >>= diffl [1..10]
3  -53
4
5  >>= diffr [1..10]
6  -5
```

Types

- Haskell is strongly typed

Types

- Haskell is strongly typed
- . . . And it doesn't really like type casting (you can still do it though)

Types

- Haskell is strongly typed
- . . . And it doesn't really like type casting (you can still do it though)
- So far. . .

Type Signatures

Your understanding so far...

Type Signatures

Your understanding so far...

```
1  funcName :: ArgT1 -> ArgT2 -> ArgT3 -> ReturnT
2  -- More appropriately
3  funcName :: a -> b -> c -> d
```

Type Signatures

Your understanding so far...

```
1 funcName :: ArgT1 -> ArgT2 -> ArgT3 -> ReturnT
2 -- More appropriately
3 funcName :: a -> b -> c -> d
```

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying There is a reason we use arrows for all args and return

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying There is a reason we use arrows for all args and return The return is not necessarily the last item

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying There is a reason we use arrows for all args and return The return is not necessarily the last item

```
1 func1 :: Int a => a -> a -> a -> a
2 func2 :: Int a => a -> a -> a
3 func3 :: Int a => a -> a
4 func4 :: Int a => a
```

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying There is a reason we use arrows for all args and return The return is not necessarily the last item

```
1  func1 :: Int a => a -> a -> a -> a
2  func2 :: Int a => a -> a -> a
3  func3 :: Int a => a -> a
4  func4 :: Int a => a
5
6  --Lets define these a bit
7  func1 a b c = a + b + c
8  func2 a b = func1 10 a b
9  func3 a = func2 20 a
10 func4 = func3 30
```

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying

```
1  func1 :: Int a => a -> a -> a -> a
2  func2 :: Int a => a -> a -> a
3  func3 :: Int a => a -> a
4  func4 :: Int a => a
5
6  --Lets define these a bit
7  func1 a b c = a + b + c
8  func2 a b = func1 10 a b
9  func3 a = func2 20 a
10 func4 = func3 30
```

Another way to look at it

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying

```
1  func1 :: Int a => a -> a -> a -> a
2  func2 :: Int a => a -> a -> a
3  func3 :: Int a => a -> a
4  func4 :: Int a => a
5
6  --Lets define these a bit
7  func1 a b c = a + b + c
8  func2 a b = func1 10 a b
9  func3 a = func2 20 a
10 func4 = func3 30
```

Another way to look at it Given something, returns what is left

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying

```
1  func1 :: Int a => a -> a -> a -> a
2  func2 :: Int a => a -> a -> a
3  func3 :: Int a => a -> a
4  func4 :: Int a => a
5
6  --Lets define these a bit
7  func1 a b c = a + b + c
8  func2 a b = func1 10 a b
9  func3 a = func2 20 a
10 func4 = func3 30
```

Another way to look at it Given something, returns what is left So given a function that takes at most 3 values

```
1  func1 :: a -> b -> c -> d
```

Type Signatures

You are limiting yourself, and preventing Haskell from doing what it's good at, Currying

```
1  func1 :: Int a => a -> a -> a -> a
2  func2 :: Int a => a -> a -> a
3  func3 :: Int a => a -> a
4  func4 :: Int a => a
5
6  --Lets define these a bit
7  func1 a b c = a + b + c
8  func2 a b = func1 10 a b
9  func3 a = func2 20 a
10 func4 = func3 30
```

Another way to look at it Given something, returns what is left So given a function that takes at most 3 values

```
1  func1 :: a -> b -> c -> d
```

Creating your first executable

```haskell
main = do
  putStr "What is your name? "
  user <- getLine
  putStr "Hi "
  putStrLn user
```

Creating your first executable

```haskell
main = do
  putStr "What is your name? "
  user <- getLine
  putStr "Hi "
  putStrLn user
```

To Run:

```
runhaskell <yourfile>
```

Or:

```
ghc <yourfile> -o <exename>; ./<exename>
```

Creating your first executable

```
1  -- Alternate way if you prefer
2  --   braces and semicolons
3  main = do {
4    putStr "What is your name? ";
5      user <- getLine;
6        putStr "Hi ";
7          putStrLn user;
8  }
9  -- The excess indetation is to show
10 --   that with this notation haskell
11 --   ignores whitespace
```

Creating your first executable

```
1  -- Alternate way using Monad operators
2  -- Note: This is technically one line
3  --   you are allowed to play with whitespace some
4  main = putStr "What is your name? " >>
5         getLine >>=
6         putStrLn . (++) "Hi "
```

Your first Monad

You just did it, congrats

Your first Monad

You just did it, congrats
We are not creating monads today, that is beyond the scope of this presentation

Your first Monad

We are not creating monads today, that is beyond the scope of this presentation
Monads you will use as a haskell programmer, and not realize

Your first Monad

We are not creating monads today, that is beyond the scope of this presentation
Monads you will use as a haskell programmer, and not realize

- IO

Your first Monad

We are not creating monads today, that is beyond the scope of this presentation
Monads you will use as a haskell programmer, and not realize

- IO
- Maybe

Your first Monad

We are not creating monads today, that is beyond the scope of this presentation
Monads you will use as a haskell programmer, and not realize

- IO
- Maybe
- Either

Your first Monad

We are not creating monads today, that is beyond the scope of this presentation
Monads you will use as a haskell programmer, and not realize

- IO
- Maybe
- Either
- List

Your first Monad

We are not creating monads today, that is beyond the scope of this presentation
Monads you will use as a haskell programmer, and not realize

- IO
- Maybe
- Either
- List
- Many More

Questions and Resources

- Book/Web: Learn You a Haskell For Great Good
- Book/Web: Real World Haskell
- Web: School of Haskell

Questions and Resources

- Book/Web: Learn You a Haskell For Great Good
- Book/Web: Real World Haskell
- Web: School of Haskell

Questions?