CS4450/7450
Chapter 2 LYAHGG: Starting Out
Principles of Programming Languages

Dr. William Harrison

University of Missouri

September 17, 2018

GHCi is basically a fancy calculator

```
$ ghci
GHCi, version 7.10.3: http://www.haskell.org/
    ghc/  :? for help
Prelude> 4 + 2
6
Prelude> not (True && True)
False
Prelude> max 5 4
5
```

Type errors are your friends

```
Prelude> 99 + "Hey"
<interactive>:5:4:
    No instance for (Num [Char]) arising from
         a use of `+`
    In the expression: 99 + "Hey"
    In an equation for `it`: it = 99 + "Hey"
Prelude>
```

GHCi Commands

Some Pragmatics

- `:l` or `:load` — load a file or module
- `:t:` or `:type` — give the type of an expression
- `:i` or `:info` — produce information about a definition
- `:q` or `:quit` — quit, derp.

GHCi Commands

Some Pragmatics

- `:l` or `:load` — load a file or module
- `:t:` or `:type` — give the type of an expression
- `:i` or `:info` — produce information about a definition
- `:q` or `:quit` — quit, derp.

```
Prelude> :t not
not :: Bool -> Bool
Prelude> :i not
not :: Bool -> Bool
        -- Defined in `GHC.Classes`
Prelude>
```

Review

Entered in a file `Chap2.hs`:

```
module Chap2 where

doubleMe x = x + x
```

Review, cont'd

```
$ ghci
GHCi, version 7.10.3: http://www.haskell.org/
    ghc/  :? for help
Prelude> :l Chap2.hs
[1 of 1] Compiling Chap2                ( Chap2.
    hs, interpreted )
Ok, modules loaded: Chap2.
*Chap2> doubleMe 9
18
*Chap2> doubleMe 3.14
6.28
*Chap2> :t doubleMe
```

Review, cont'd

```
$ ghci
GHCi, version 7.10.3: http://www.haskell.org/
    ghc/  :? for help
Prelude> :l Chap2.hs
[1 of 1] Compiling Chap2                ( Chap2.
    hs, interpreted )
Ok, modules loaded: Chap2.
*Chap2> doubleMe 9
18
*Chap2> doubleMe 3.14
6.28
*Chap2> :t doubleMe
```

```
doubleMe :: Num a => a -> a
*Chap2>
```

Lists, an Introduction to

```
Prelude> let lostNumbers = [4,8,15,16,23,42]
Prelude> lostNumbers
[4,8,15,16,23,42]

Prelude> 99 : lostNumbers
[99,4,8,15,16,23,42]

Prelude> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]

Prelude> "hello" ++ " " ++ "world"
"hello world"

Prelude> ['w','0'] ++ ['0','t']
"w00t"
```

Some Facts about Lists

- `[]`, `[[]]` and `[[],[],[]]` are all different things. What are their types? Can check that with GHCi.

Some Facts about Lists

- `[]`, `[[]]` and `[[],[],[]]` are all different things. What are their types? Can check that with GHCi.
- Lists are *uniform* in Haskell. E.g., `[1,2,3]` is legal and `[1,2,'c']` is not.

Some Facts about Lists

- `[]`, `[[]]` and `[[],[],[]]` are all different things. What are their types? Can check that with GHCi.
- Lists are *uniform* in Haskell. E.g., `[1,2,3]` is legal and `[1,2,'c']` is not.
- The `data` declaration for lists in Haskell is:

```
data [a] = [] | a : [a]
```

Basic Function on Lists

`head` takes a list and returns its head. The head of a list is its first element (if it exists).

```
ghci> head [5,4,3,2,1]
5
```

- What is the type of `head`?
- How do we write `head` in Haskell?

Basic Function on Lists

`tail` takes a list and returns its tail. In other words, it chops off a list's head.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

- What is the type of `tail`?
- How do we write `tail` in Haskell?

Basic Function on Lists

If you want to get an element out of a list by index, use `!!`. The indices start at 0.

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

- What is the type of `!!`?
- How do we write `!!` in Haskell?

More Basic Functions on Lists

`last` takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

- What is the type of `last`?
- How do we write `last` in Haskell?

More Basic Functions on Lists

`init` takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

- What is the type of `init`?
- How do we write `init` in Haskell?

More Basic Functions on Lists

`length` takes a list and returns its length, obviously.

```
ghci> length [5,4,3,2,1]
5
```

- What is the type of `length`?
- How do we write `length` in Haskell?

More Basic Functions on Lists

`null` checks if a list is empty. If it is, it returns `True`, otherwise it returns `False`. Use this function instead of `xs == []` (if you have a list called `xs`).

```
ghci> null [1,2,3]
False
ghci> null []
True
```

- What is the type of `null`?
- How do we write `null` in Haskell?

More Basic Functions on Lists

`reverse` reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

- What is the type of `reverse`?
- How do we write `reverse` in Haskell?

An Aside on Efficiency

Here's a simple way to write `reverse` and append (++).

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Why is this inefficient?

An Aside on Efficiency

Here's a simple way to write `reverse` and append (++).

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Why is this inefficient? Here's why:

```
reverse [x0,...,xn]
  = reverse [x0,...,xn−1] ++ [xn−1]  -- (n × reverse)
      ⋮
  = [x0] ++ ... ++ [xn−1]            -- (n−1 × ++)
```

Accumulator Passing Style

This is more efficient. Why?

```
rev :: [a] -> [a]
rev xs = rev' [] xs
   where rev' :: [a] -> [a] -> [a]
         rev' acc []     = acc
         rev' acc (x:xs) = rev' (x:acc) xs
```

Accumulator Passing Style

This is more efficient. Why?

```
rev :: [a] -> [a]
rev xs = rev' [] xs
   where rev' :: [a] -> [a] -> [a]
         rev' acc []     = acc
         rev' acc (x:xs) = rev' (x:acc) xs
```

```
rev [x0,...,xn]
  = rev' [] [x0,...,xn]
  = rev' [x0] [x1,...,xn]
    ⋮
  = rev' [xn,...,x0] []
  = [xn,...,x0]              -- (n+1 × rev')
```

More Basic Functions on Lists

`take` takes number and a list. It extracts that many elements from the beginning of the list.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

- What is the type of `take`?
- How do we write `take` in Haskell?

More Basic Functions on Lists

`drop` works in a similar way, only it drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

- What is the type of `drop`?
- How do we write `drop` in Haskell?

More Basic Functions on Lists

`maximum` takes a list of stuff that can be put in some kind of order and returns the biggest element.

```
ghci> maximum [1,9,2,3,4]
9
```

- What is the type of `maximum`?
- How do we write `maximum` in Haskell?

More Basic Functions on Lists

`sum` takes a list of numbers and returns their sum.

```
ghci> sum [5,2,1,6,3,2,5,7]
31
```

- What is the type of `sum`?
- How do we write `sum` in Haskell?

More Basic Functions on Lists

`elem` takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

- What is the type of `elem`?
- How do we write `elem` in Haskell?

Texas (?!) Ranges

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]


ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"

ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

Removing Duplicates

The function `nub` removes duplicates from a list. It is defined in `Data.List`, so you have to import that module to use it.

```
ghci> nub "steve buscemi"
   "stev bucmi"
```

Infinite Lists

`repeat` takes an element and produces an infinite list of just that element. It's like cycling a list with only one element.

```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

- What is the type of `repeat`?
- How do we write `repeat` in Haskell?

Infinite Lists

`cycle` takes a list and cycles it into an infinite list. If you just try to display the result, it will go on forever so you have to slice it off somewhere.

```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```

- What is the type of `cycle`?
- How do we write `cycle` in Haskell?

Set Comprehensions

In the lingo of Mathematics, the following definition is a *set comprehension*:

$$S = \{2 * x \mid x \in Nat,\ x \leq 10\}$$

- $S = \{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$.

Set Comprehensions

In the lingo of Mathematics, the following definition is a *set comprehension*:

$$S = \{2 * x \mid x \in Nat,\ x \leq 10\}$$

- $S = \{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$.
- There is a similar notion in Haskell known as a *list comprehension*.

```
ghci> [2*x | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

List Comprehension Examples

- Multiple Generators:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

List Comprehension Examples

- Multiple Generators:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

- Adding constraints:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

List Comprehension Examples

- Multiple Generators:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

- Adding constraints:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

- As part of functions:

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Testing it out:

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

Tuples

- These are built-in type constructors for ordered pairs, ordered triples, etc. E.g., `("Wow",'a')` is an ordered pair.

Tuples

- These are built-in type constructors for ordered pairs, ordered triples, etc. E.g., `("Wow",'a')` is an ordered pair.
- Tuple types are written in the same style as tuple expressions:

```
ghci> :t ("Wow",'a')
("Wow",'a') :: ([Char], Char)
```

Tuples

- These are built-in type constructors for ordered pairs, ordered triples, etc. E.g., `("Wow",'a')` is an ordered pair.
- Tuple types are written in the same style as tuple expressions:

```
ghci> :t ("Wow",'a')
("Wow",'a') :: ([Char], Char)
```

- There are Prelude-defined functions for *pairs*:

```
:t fst
fst :: (a, b) -> a
ghci> :t snd
snd :: (a, b) -> b
ghci> fst ("Wow",'a')
"Wow"
ghci> snd ("Wow",'a')
'a'
```

Zippers

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]

ghci> zip [1 .. 5] ["one", "two", "three", "
    four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four")
    ,(5,"five")]

ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","
    turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

- Notice the input lists need not be of the same length.
- What is the type of `zip`?
- How do we write `zip` in Haskell?