Haskell Basics

Stephen A. Edwards

Columbia University

Fall 2021

Haskell
Useful Websites

- https://www.haskell.org/
  Downloads, documentation
  E.g., the Haskell Wiki, the GHC User’s Guide, The Haskell 2010 language report, Hackage (package library), Hoogle (Haskell API search)

- http://docs.haskellstack.org
  The Haskell Tool Stack: a powerful system for downloading and installing packages, etc.
  We will be using the Haskell Stack to make sure everybody’s environment is consistent.
GHCI

GHCI is the Glasgow Haskell Compiler (the major Haskell compiler release)
GHCI is the REPL (Read-Eval-Print Loop, a.k.a., command-line interface)
Run ghci with stack:

$ stack ghci
Configuring GHCI with the following packages:
GHCI, version 8.10.6: http://www.haskell.org/ghc/ :? for help
Loaded GHCI configuration from /tmp/haskell-stack-ghci/2a3bbd58/..
Prelude> :?
Commands available from the prompt:

<statement> evaluate/run <statement>
:quit exit GHCI
The material on the following slides is adapted from

Miran Lipovača.
Learn You a Haskell for Great Good!

http://learnyouahaskell.com/
Comments

Single-line comments start with two dashes: --

```
Prelude> -- Single-line comment
```

Multi-line comments start with {-, end with -}, and may nest.

```
In GHCi only, multi-line definitions, etc.
may be written with :{ and :}; these are
unnecessary in source (.hs) files.
```

```
Prelude> :{
Prelude| {- This is a
Prelude|   multi-line comment -}
Prelude| :}
```

Alternately enable multi-line input mode in GHCi:

```
Prelude> :set +m
Prelude> {-
Prelude|   A multi-line
Prelude|       Comment
Prelude|   -}
Prelude> {- Another
Prelude|     one -}
```
Basic Arithmetic

17
32
7
2.5
1
Booleans and Equality

Haskell is case-sensitive

Prelude> True && False
False
Prelude> False || True
True
Prelude> not True || True
True
Prelude> not (True || True)
False

Prelude> 5 == 5
True
Prelude> 5 == 0
False
Prelude> 5 /= 5
False
Prelude> 5 /= 0
True
Prelude> "hello" == "hello"
True

Prelude> "llama" == 5
<interactive>:25:12: error:
  * No instance for (Num [Char]) arising from the literal '5'
  * In the second argument of '(%==)', namely '5'
    In the expression: "llama" == 5
    In an equation for 'it': it = "llama" == 5
Function Application

Juxtaposition indicates function application. Don’t use parentheses or commas for arguments.

Prelude> succ 41
42
Prelude> min 42 17
17
Prelude> max 42 17
42

Juxtaposition binds tightly; use parentheses to group arguments

Prelude> succ 3 * 2
8
Prelude> succ (3 * 2)
7
**Backticks and parentheses**

Backticks make a function an infix operator. This is sometimes a more natural way to write expressions.

```
Prelude> 5 `max` 3
5
Prelude> 5 `max` 8
8
```

Parentheses around a binary operator turns it into a two-argument function. This is most useful when you want to pass it as an argument (later).

```
Prelude> (+) 17 25
42
```
In recent versions of GHCi, just use = to bind things to names

```haskell
Prelude> x = 7
Prelude> x * x
49
```

Just add one or more arguments to define a function

```haskell
Prelude> sqr x = x * x
Prelude> sqr 7
49
Prelude> y = 8
Prelude> sqr y
64
```
You can similarly define a function in a source file:

```
sqr.hs:   sqr x = x * x
```

In GHCi, :l means "load"

```
Prelude> :l sqr
[1 of 1] Compiling Main                 ( sqr.hs, interpreted )
Ok, one module loaded.
*Main> sqr 7
49
```
Lists: Homogeneous Sequences

Square brackets and commas denote list literals

Prelude> fiveprimes = [2,3,5,7,11]
Prelude> fiveprimes
[2,3,5,7,11]

Strings are just lists of characters

Prelude> ['h','e','l','l','o']
"hello"

++ performs list concatenation

Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
Prelude> ['h','e','l','l','o'] ++ " world"
"hello world"
The Cons Operator : Prepends a List Element

The bracket notation is just syntactic sugar for Cons.

Prelude> 1 : [2,3,4]
[1,2,3,4]
Prelude> 1 : 2 : [3,4]
[1,2,3,4]
Prelude> 1 : 2 : 3 : 4 : []
[1,2,3,4]

List elements must all be the same type

Prelude> 1 : ['h','e']
<interactive>:10:1: error:
  * No instance for (Num Char) arising from the literal '1'
  * In the first argument of '(:)', namely '1'
    In the expression: 1 : ['h', 'e']
    In an equation for 'it': it = 1 : ['h', 'e']
From Learn You a Haskell for Great Good!
Prelude> x = [0,1,2,3,4]
Prelude> head x
0
Prelude> tail x
[1,2,3,4]
Prelude> last x
4
Prelude> length x
5
Prelude> init x
[0,1,2,3]
Prelude> reverse x
[4,3,2,1,0]
Prelude> null x
False
Prelude> null []
True

Prelude> [5,6,7] !! 2
7
Prelude> "Monty Python" !! 6
'P'
Prelude> take 3 x
[0,1,2]
Prelude> drop 2 x
[2,3,4]
Prelude> maximum x
4
Prelude> minimum x
0
Prelude> sum x
10
Prelude> product x
0
List Ranges

Prelude> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"

Linear sequences only
Floating point numbers problematic
Infinite Lists

Haskell supports infinite lists (and other infinite data structures). Hint: *don’t print out the whole thing.* E.g., use `take` to see the first elements.

```haskell
Prelude> take 5 [1..]
[1,2,3,4,5]
Prelude> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 10 [1,2,3]
[1,2,3]
Prelude> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
Prelude> take 16 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1]
Prelude> take 17 (repeat 5)
[5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]
Prelude> replicate 15 6
[6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6]
```
Prelude> [ x^2 | x <- [1..19] ]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361]

Prelude> [ x^2 | x <- [1..20], (x^2) `mod` 2 == 0 ]
[4,16,36,64,100,144,196,256,324,400]

Prelude> [ x^2 | x <- [1..20], even (x^2) ]
[4,16,36,64,100,144,196,256,324,400]

Prelude> [ y | x <- [1..20], let y = x^2, even y ]
[4,16,36,64,100,144,196,256,324,400]
List Comprehensions

Multiple guards must all be true

```
Prelude> [ x | x <- [1..100], x `mod` 7 == 0 ]
[7,14,21,28,35,42,49,56,63,70,77,84,91,98]
```

```
Prelude> [ x | x <- [1..100], x `mod` 7 == 0, x `mod` 5 == 0 ]
[35,70]
```

Multiple generators apply right-to-left:

```
Prelude> [ x + y | x <- [100,200..400], y <- [0..3] ]
[100,101,102,103,200,201,202,203,300,301,302,303,400,401,402,403]
```
Application: CS Research Jargon Generator

```
Prelude> :set +m
Prelude> [ adjective ++ " " ++ noun |
Prelude|   adjective <- ["An integrated","A type-safe"],
Prelude|   noun <- ["network","architecture","hypervisor"] ]
["An integrated network","An integrated architecture",
 "An integrated hypervisor","A type-safe network",
 "A type-safe architecture","A type-safe hypervisor"]

https://www.cs.purdue.edu/homes/dec/essay.topic.generator.html
```
List Comprehensions

Here’s an awkward way to code the standard Prelude’s length function:

```
Prelude> length' xs = sum [ 1 | _ <- xs ]
Prelude> length' [5,6,2,1,0]
Prelude> length' (replicate 11 []) -- List of eleven empty lists
```

Names (variable identifiers) start with a lowercase letter followed by zero or more letters, digits, underscores, and single quotes.

_ alone means “don’t give this a name”

```
Prelude> onlyLetters s = [ c | c <- s, c `elem` ['A'..'Z'] ++ ['a'..'z'] ]
Prelude> onlyLetters "Does this do what I think it should?"
"Does this do what I think it should?"
```
Tuples: Pairs and More of Heterogeneous Objects

Lists are zero or more things of the same type; a tuple is two or more of (potentially) different types.

Prelude> (5,10)
(5,10)
Prelude> ("a",15)
("a",15)
Prelude> ("Douglas","Adams",42)
("Douglas","Adams",42)
Prelude> sae = ("Stephen", "Edwards")
Prelude> fst sae
"Stephen"
Prelude> snd sae
"Edwards"
Zip and Pythagorean Triples

Form a list of pairs from two lists. Shorter of the two lists dominates; convenient with infinite lists

Prelude> zip [1,2,3] [100,200,300]
[(1,100),(2,200),(3,300)]

Prelude> zip "Stephen" [1..]
[('S',1),('t',2),('e',3),('p',4),('h',5),('e',6),('n',7)]

Prelude> [ (a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b],
a^2 + b^2 == c^2 ]
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20)]
The Handshake Problem

Number of handshakes among a group of $n$ friends?

Prelude> handshakes n = [ (a,b) | a <- [1..n-1], b <- [a+1..n] ]
Prelude> handshakes 3
[(1,2),(1,3),(2,3)]
Prelude> handshakes 5
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
Prelude> length (handshakes 5)
10
Prelude> [ length (handshakes n) | n <- [1..10] ]
[0,1,3,6,10,15,21,28,36,45]
Prelude> [ n * (n-1) `div` 2 | n <- [1..10] ]
[0,1,3,6,10,15,21,28,36,45]