

Think more effectively with Haskell

Evan Misshula

2017-02-14

Outline

- 1 Acknowledgements
- 2 If you are stuck
- 3 Why another language?
- 4 Tooling
- 5 The fun begins
- 6 Fun with numbers
- 7 More fun with parenthesis
- 8 syntax gotcha
- 9 Boolean variables
- 10 Tests
- 11 Asking the wrong question
- 12 Creating a bad test
- 13 Types
- 14 prefix demo
- 15 Precedence
- 16 Making our own functions
- 17 Functions with 2 arguments

These slides are based on

- Learn You Haskell for Great Good
- Haskell Tutorials from
 - Programming Languages by Peter Drake
- Videos from the New York and Boston Haskell Meetups
 - Special thanks to NYC's:
 - Gershon Bazerman
 - Evie Borthwick
 - Ryan Trinkle
 - Brian Hurt
 - Cat Chuang
 - Boston:
 - Edward Kmet

These slides are based on

- Learn You Haskell for Great Good
- Haskell Tutorials from
 - Programming Languages by Peter Drake
- Videos from the New York and Boston Haskell Meetups
 - Special thanks to NYC's:
 - Gershon Bazerman
 - Evie Borthwick
 - Ryan Trinkle
 - Brian Hurt
 - Cat Chuang
 - Boston:
 - Edward Kmet
 - I have watched his videos so many times I think we have met
- Bartoz Milewski's
 - Category Theory blog
 - Video Tutorials
- The brilliant and funny Eugenia Cheng

- Internet Relay Chat
 - Freenode
 - #Haskell
 - Or our gmail group

What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
 - In Haskell computations is *lazy*.
 - In Haskell a variables is *immutable*.
 - In Haskell a variable is never implicitly coerced into another type.

What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
 - In Haskell computations is *lazy*.
 - In Haskell a variables is *immutable*.
 - In Haskell a variable is never implicitly coerced into another type.
 - immutability means you can build Single Page [Web] Apps (SPA)

What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
 - In Haskell computations is *lazy*.
 - In Haskell a variables is *immutable*.
 - In Haskell a variable is never implicitly coerced into another type.
 - immutability means you can build Single Page [Web] Apps (SPA)
 - immutability means you can transpile into JavaScript and build native mobile apps without coding in Java, Objective-C or Swift

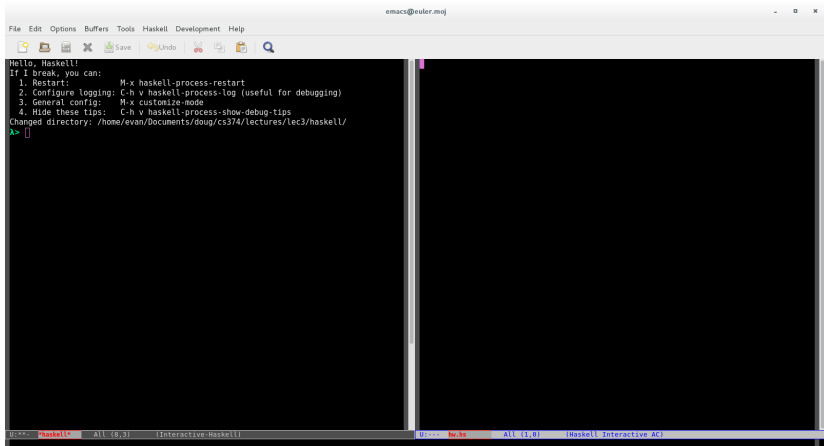
What is Haskell?

- A purely functional statically typed language.
- Aspires to declaration over instruction
 - In Haskell computations is *lazy*.
 - In Haskell a variables is *immutable*.
 - In Haskell a variable is never implicitly coerced into another type.
- immutability means you can build Single Page [Web] Apps (SPA)
- immutability means you can transpile into JavaScript and build native mobile apps without coding in Java, Objective-C or Swift
- Immutability, functional and lazy gives you code you can run in parallel by default

What do you need?

- You need an editor and the Haskell compiler. We will install Haskell stack, Emacs and code completion.

let's open Emacs



The screenshot shows the Emacs editor window titled "emacs@euler.moj". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", "Haskell", "Development", and "Help". The toolbar contains icons for opening files, saving, undo, redo, and searching. The main editing area is split into two panes. The left pane displays the following text:

```
Hello, Haskell!  
If I break, you can:  
  1. Restart:      M-x haskell-process-restart  
  2. Configure logging: C-h v haskell-process-log (useful for debugging)  
  3. General Config: M-x customize-mode  
  4. Hide these tips: C-h v haskell-process-show-debug-tips  
Changed directory: /home/evan/Documents/doug/cs374/lectures/lec3/haskell/  
λ> 
```

The right pane is currently empty. The status bar at the bottom shows the current buffer is "haskell" (All (8,3)) and the active window is "Haskell Interactive AC" (All (1,0)).

2 + 15

49 * 100

1892 - 1472

5 / 2

1=1

17

4900

420

2.5

Compound arithmetic computations

`(50 * 100) - 4999`

`50 * 100 - 4999`

`50 * (100 - 4999)`

`1==1`

`1`

`1`

`-244950`

Watch out for negative numbers

- $5 * -3$ doesn't work

Watch out for negative numbers

- $5 * -3$ doesn't work
- $5 * (-3)$ does

usual rules with Boolean variables

- `&&` == Boolean *and*
- `||` == Boolean *or*
- `not` == negation

testing for equality

True && False

True && True

False || True

not False

not (True && True)

1=1

False

True

True

True

False

Bad addition

```
5+"llama"
```

```
1==1
```

```
<interactive>:252:2:
```

```
No instance for (Num [Char]) arising from a use of ‘+’
```

```
Possible fix: add an instance declaration for (Num [Char])
```

```
In the expression: 5 + "llama"
```

```
In an equation for ‘it’: it = 5 + "llama"
```

Bad addition

```
5+"llama"
```

```
1==1
```

```
<interactive>:252:2:
```

```
No instance for (Num [Char]) arising from a use of ‘+’
```

```
Possible fix: add an instance declaration for (Num [Char])
```

```
In the expression: 5 + "llama"
```

```
In an equation for ‘it’: it = 5 + "llama"
```

result

Bad test

```
5 == True
1==1
```

```
<interactive>:255:1:
```

```
No instance for (Num Bool) arising from the literal '5'
Possible fix: add an instance declaration for (Num Bool)
In the first argument of '(==)', namely '5'
In the expression: 5 == True
In an equation for 'it': it = 5 == True
```

Bad test

```
5 == True
1==1
```

```
<interactive>:255:1:
```

```
No instance for (Num Bool) arising from the literal '5'
Possible fix: add an instance declaration for (Num Bool)
In the first argument of '(==)', namely '5'
In the expression: 5 == True
In an equation for 'it': it = 5 == True
```

```
<interactive>:340:1:
```

```
No instance for (Num Bool) arising from the literal '5'
```

```
Possible fix: add an instance declaration for (Num Bool)
```

```
In the first argument of '(==)', namely '5'
```

```
In the expression: 5 == True
```

```
In an equation for 'it': it = 5 == True
```

GHCI can tell that the types don't match

- '+' expects left and right to be number
- '==' expects two things that are of the same type

infix and *prefix* functions

infix and *prefix* functions

- '+' is *infix* because it goes between its arguments

infix and *prefix* functions

- '+' is *infix* because it goes between its arguments
- 'succ' is a *prefix* function because it goes before its argument

```
succ 8  
min 9 10  
min 3.4 3.2  
max 100 101  
1=1  
  
9  
9  
3.2  
101
```

Haskell relies on precedence

- Many Lisp/Scheme/Clojure programmers put in more parenthesis than is idiomatic in Haskell
- These two statements are the same

```
succ 9 + max 5 4 + 1  
(succ 9) + (max 5 4) + 1  
1==1
```

16

16

imperative steps

imperative steps

- ① open Emacs
- ② create the following director in Documents
`~/Documents/cs374/haskell/`
- ③ create the file `baby.hs`
- ④ start the GHCi

imperative steps

- 1 open Emacs
 - 2 create the following director in Documents
`~/Documents/cs374/haskell/`
 - 3 create the file `baby.hs`
 - 4 start the GHCi
-
- 1 type `doubleMe x = 2*x`
 - 2 load the file into `ghci`

take integers or floats

take integers or floats

❶ `type doubleUS x y = 2*x + 2*y in baby.hs`

take integers or floats

- ❶ `type doubleUS x y = 2*x + 2*y in baby.hs`
- ❶ `type doubleUS x y = x + x + y + y in baby.hs`

take integers or floats

- ❶ `type doubleUS x y = 2*x + 2*y in baby.hs`
- ❶ `type doubleUS x y = x + x + y + y in baby.hs`
- ❶ `type doubleUS x y = doubleMe x + doubleMe y in baby.hs`

piece-wise functions

piece-wise functions

❶ type the following

```
let doubleSmallNumber x = if (x > 100) then x else 2*x  
doubleSmallNumber 54  
doubleSmallNumber 103  
1==1
```

piece-wise functions on one line

piece-wise functions on one line

① type the following

```
doubleSmallNumber' x = (if x > 100 then x else 2*x) + 1
```


character functions

- 1 type the following

character functions

- 1 type the following
- 1 can't capitalize the name

character functions

- 1 type the following
- 1 can't capitalize the name

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

- 1 type the following `let lostNumbers = [4,8,15,16,23,42]`

- 1 type the following `let lostNumbers = [4,8,15,16,23,42]`
- 1 '++' is the concatenate operator

❶ type the following `let lostNumbers = [4,8,15,16,23,42]`

❶ `'++'` is the concatenate operator

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

```
"hello" ++ " " ++ "world"
```

```
['w','o'] ++ ['o','t']
```

```
1=1
```

```
[1,2,3,4,9,10,11,12]
```

```
hello world
```

```
woot
```

- 1 `'::'` adds an element to the front of a list $O(1)$ time (`cons`)

naming lists

- 1 `:` adds an element to the front of a list $O(1)$ time (`cons`)
- 1 `++` works in $O(n)$

naming lists

- 1 `' : '` adds an element to the front of a list $O(1)$ time (`cons`)
- 1 `' ++ '` works in $O(n)$
- 1 `[1,2,3]` is really `1:2:3:[]`

naming lists

- 1 `' : '` adds an element to the front of a list $O(1)$ time (`cons`)
- 1 `' ++ '` works in $O(n)$
- 1 `[1,2,3]` is really `1:2:3:[]`
- 1 `' !! '` takes element out of a list

naming lists

- 1 `' : '` adds an element to the front of a list $O(1)$ time (`cons`)
- 1 `' ++ '` works in $O(n)$
- 1 `[1,2,3]` is really `1:2:3:[]`
- 1 `' !! '` takes element out of a list

```
"Steve Buscemi" !! 6
```

```
[9.4,33.2,96.2,11.2,23.25] !! 1
```

```
1==1
```

```
'B'
```

```
33.2
```

Four $O(1)$ list functions

- 1 'head' takes a list and returns only its first element

Four $O(1)$ list functions

- ① 'head' takes a list and returns only its first element
- ① 'tail' takes a list and returns everything except its first element

Four $O(1)$ list functions

- ① 'head' takes a list and returns only its first element
- ① 'tail' takes a list and returns everything except its first element
- ① 'last' takes a list and returns only its last element

Four $O(1)$ list functions

- ① 'head' takes a list and returns only its first element
- ① 'tail' takes a list and returns everything except its first element
- ① 'last' takes a list and returns only its last element
- ① 'init' takes a list and returns everything except its last element

Four $O(1)$ list functions

- ① 'head' takes a list and returns only its first element
- ① 'tail' takes a list and returns everything except its first element
- ① 'last' takes a list and returns only its last element
- ① 'init' takes a list and returns everything except its last element

```
head "Steve Buscemi"
```

```
head [9.4,33.2,96.2,11.2,23.25]
```

```
tail "Steve Buscemi"
```

```
tail [9.4,33.2,96.2,11.2,23.25]
```

```
last "Steve Buscemi"
```

```
last [9.4,33.2,96.2,11.2,23.25]
```

```
init "Steve Buscemi"
```

```
init [9.4,33.2,96.2,11.2,23.25]
```

```
1==1
```

```
'S'
```

```
9.4
```


Matrices can be represented as nested lists

- We use nested lists
- We can also apply the list functions multiple times to access parts of the list

```
let firstMat = [[9,8,7],[6,5,4],[3,2,1]]  
firstMat !! 1  
(firstMat !! 1) !! 2  
1==1
```

Prelude|

```
<interactive>:298:1: parse error on input 'firstMat'  
<interactive>:298:2: Not in scope: 'firstMat'
```

Length examples

- 'length' returns the length of a list

```
length [3,9,3]
```

```
length [1,2]
```

```
length [6,6,6]
```

```
1==1
```

```
3
```

```
2
```

```
3
```

Null examples

- 'null' let's us know if a list is empty

```
null [1,2,3]
```

```
null []
```

```
1==1
```

```
False
```

```
True
```

Functions to reduce a list

- 'drop' returns the list without the first n elements

```
drop 1 [3,9,3]
```

```
drop 5 [1,2]
```

```
drop 0 [6,6,6]
```

```
1==1
```

```
[9,3]
```

```
[]
```

```
[6,6,6]
```

Take examples

- 'take' returns the first n elements of a list

```
take 1 [3,9,3]
```

```
take 5 [1,2]
```

```
take 0 [6,6,6]
```

```
1==1
```

```
[3]
```

```
[1,2]
```

```
[]
```

There is a built in function to reverse a list

- 'reverse' doesn't reduce
 - returns a list in the reverse order
 - can't be bound to the same variable

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

```
1==1
```

```
[1,2,3,4,5]
```

There are a built in function to return extremes of a list

- 'minimum' returns the minimum of a list
- 'maximum' returns the maximum of a list

```
minimum [5,4,3,2,1]
```

```
maximum [5,4,3,2,1]
```

```
1==1
```

```
1
```

```
5
```

There are a built in functions to sum and multiply elements

- 'sum' returns the sum of the elements of a list
- 'product' returns the product of the elements of a list

```
sum [5,4,3,2,1]
```

```
product [5,4,3,2,1]
```

```
1==1
```

```
15
```

```
120
```


There is a built in function to check membership

- 'elem' returns the sum of the elements of a list
- the backtick key `` allows you to use a *prefix* function as *infix*

```
4 'elem' [3,4,5,6]
10 'elem' [3,4,5,6]
1==1
```

True

False

Enumeration and range

- If we can enumerate all the elements of a set we can use ranges:
 - numbers
 - letters
 - capital letters

```
[1..20]
```

```
['a'..'z']
```

```
['X'..'Z']
```

```
1==1
```

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

```
abcdefghijklmnopqrstuvwxyz
```

```
XYZ
```

ranges can have steps

- We can get even the even numbers
- We can get the multiples of three

```
[2,4..20]
```

```
[3,6..20]
```

```
1==1
```

```
[2,4,6,8,10,12,14,16,18,20]
```

```
[3,6,9,12,15,18]
```

Here are functions that produce infinite data structures

- First 24 multiples of 13

Here are functions that produce infinite data structures

- First 24 multiples of 13
- `take 24 [13,26..]`

Here are functions that produce infinite data structures

- First 24 multiples of 13
- `take 24 [13,26..]`
- Haskell is lazy so it won't compute until you ask it for something

Here are functions that produce infinite data structures

- First 24 multiples of 13
- take 24 [13,26..]
- Haskell is lazy so it won't compute until you ask it for something
- 'cycle' takes a finite list and makes it infinite

```
take 24 [13,26..]
take 10 (cycle [1,2,3])
take 12 (cycle "LOL ")
1==1
```

[13,26,39,52,65,78,91,104,117,130,143,156,169,182,195,208,221,234,247,260,273,286,299,312,325,338,351,364,377,390,403,416,429,442,455,468,481,494,507,520,533,546,559,572,585,598,611,624,637,650,663,676,689,702,715,728,741,754,767,780,793,806,819,832,845,858,871,884,897,910,923,936,949,962,975,988,1001,1014,1027,1040,1053,1066,1079,1092,1105,1118,1131,1144,1157,1170,1183,1196,1209,1222,1235,1248,1261,1274,1287,1300,1313,1326,1339,1352,1365,1378,1391,1404,1417,1430,1443,1456,1469,1482,1495,1508,1521,1534,1547,1560,1573,1586,1599,1612,1625,1638,1651,1664,1677,1690,1703,1716,1729,1742,1755,1768,1781,1794,1807,1820,1833,1846,1859,1872,1885,1898,1911,1924,1937,1950,1963,1976,1989,2002,2015,2028,2041,2054,2067,2080,2093,2106,2119,2132,2145,2158,2171,2184,2197,2210,2223,2236,2249,2262,2275,2288,2301,2314,2327,2340,2353,2366,2379,2392,2405,2418,2431,2444,2457,2470,2483,2496,2509,2522,2535,2548,2561,2574,2587,2600,2613,2626,2639,2652,2665,2678,2691,2704,2717,2730,2743,2756,2769,2782,2795,2808,2821,2834,2847,2860,2873,2886,2899,2912,2925,2938,2951,2964,2977,2990,3003,3016,3029,3042,3055,3068,3081,3094,3107,3120,3133,3146,3159,3172,3185,3198,3211,3224,3237,3250,3263,3276,3289,3302,3315,3328,3341,3354,3367,3380,3393,3406,3419,3432,3445,3458,3471,3484,3497,3510,3523,3536,3549,3562,3575,3588,3601,3614,3627,3640,3653,3666,3679,3692,3705,3718,3731,3744,3757,3770,3783,3796,3809,3822,3835,3848,3861,3874,3887,3900,3913,3926,3939,3952,3965,3978,3991,4004,4017,4030,4043,4056,4069,4082,4095,4108,4121,4134,4147,4160,4173,4186,4199,4212,4225,4238,4251,4264,4277,4290,4303,4316,4329,4342,4355,4368,4381,4394,4407,4420,4433,4446,4459,4472,4485,4498,4511,4524,4537,4550,4563,4576,4589,4602,4615,4628,4641,4654,4667,4680,4693,4706,4719,4732,4745,4758,4771,4784,4797,4810,4823,4836,4849,4862,4875,4888,4901,4914,4927,4940,4953,4966,4979,4992,5005,5018,5031,5044,5057,5070,5083,5096,5109,5122,5135,5148,5161,5174,5187,5200,5213,5226,5239,5252,5265,5278,5291,5304,5317,5330,5343,5356,5369,5382,5395,5408,5421,5434,5447,5460,5473,5486,5499,5512,5525,5538,5551,5564,5577,5590,5603,5616,5629,5642,5655,5668,5681,5694,5707,5720,5733,5746,5759,5772,5785,5798,5811,5824,5837,5850,5863,5876,5889,5902,5915,5928,5941,5954,5967,5980,5993,6006,6019,6032,6045,6058,6071,6084,6097,6110,6123,6136,6149,6162,6175,6188,6201,6214,6227,6240,6253,6266,6279,6292,6305,6318,6331,6344,6357,6370,6383,6396,6409,6422,6435,6448,6461,6474,6487,6500,6513,6526,6539,6552,6565,6578,6591,6604,6617,6630,6643,6656,6669,6682,6695,6708,6721,6734,6747,6760,6773,6786,6799,6812,6825,6838,6851,6864,6877,6890,6903,6916,6929,6942,6955,6968,6981,6994,7007,7020,7033,7046,7059,7072,7085,7098,7111,7124,7137,7150,7163,7176,7189,7202,7215,7228,7241,7254,7267,7280,7293,7306,7319,7332,7345,7358,7371,7384,7397,7410,7423,7436,7449,7462,7475,7488,7501,7514,7527,7540,7553,7566,7579,7592,7605,7618,7631,7644,7657,7670,7683,7696,7709,7722,7735,7748,7761,7774,7787,7800,7813,7826,7839,7852,7865,7878,7891,7904,7917,7930,7943,7956,7969,7982,7995,8008,8021,8034,8047,8060,8073,8086,8099,8112,8125,8138,8151,8164,8177,8190,8203,8216,8229,8242,8255,8268,8281,8294,8307,8320,8333,8346,8359,8372,8385,8398,8411,8424,8437,8450,8463,8476,8489,8502,8515,8528,8541,8554,8567,8580,8593,8606,8619,8632,8645,8658,8671,8684,8697,8710,8723,8736,8749,8762,8775,8788,8801,8814,8827,8840,8853,8866,8879,8892,8905,8918,8931,8944,8957,8970,8983,8996,9009,9022,9035,9048,9061,9074,9087,9100,9113,9126,9139,9152,9165,9178,9191,9204,9217,9230,9243,9256,9269,9282,9295,9308,9321,9334,9347,9360,9373,9386,9399,9412,9425,9438,9451,9464,9477,9490,9503,9516,9529,9542,9555,9568,9581,9594,9607,9620,9633,9646,9659,9672,9685,9698,9711,9724,9737,9750,9763,9776,9789,9802,9815,9828,9841,9854,9867,9880,9893,9906,9919,9932,9945,9958,9971,9984,9997,10010,10023,10036,10049,10062,10075,10088,10101,10114,10127,10140,10153,10166,10179,10192,10205,10218,10231,10244,10257,10270,10283,10296,10309,10322,10335,10348,10361,10374,10387,10400,10413,10426,10439,10452,10465,10478,10491,10504,10517,10530,10543,10556,10569,10582,10595,10608,10621,10634,10647,10660,10673,10686,10699,10712,10

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's
- take 10 (repeat 5)

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's
- take 10 (repeat 5)
- 'replicate' produces a list of a given size of all a given number

Here are functions that produce constant vectors

- First 10 of an infinite list of 5's
- `take 10 (repeat 5)`
- `'replicate'` produces a list of a given size of all a given number
- `'cycle'` takes a finite list and makes it infinite

```
take 10 (repeat 5)
```

```
replicate 3 10
```

```
1==1
```

```
[5,5,5,5,5,5,5,5,5,5]
```

```
[10,10,10]
```

Just like Python, Haskell has its own twist on list comprehensions

Definition (In set notation)

- $S = \{2 \cdot x \mid x \in \mathbb{N}, x \leq 10\}$
 - $2 \cdot x$ the output function
 - ' x ' is the variable
 - \mathbb{N} is the input set
 - $x \leq 10$ is the predicate

```
[x*2 | x<- [1..100] , x <=10, x>=6]  
[ x | x <- [50..100], mod x 7==3]  
1==1
```

```
[12,14,16,18,20]  
[52,59,66,73,80,87,94]
```

We can combine functions and list comprehensions

- We can even get product lists

```
let boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- boomBangs [7..13]
[ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[ x*y | x <- [2,5,10], y <- [8,10,11]]
[ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
1==1
```

Prelude|

```
<interactive>:359:1: parse error on input 'boomBangs'
[10,11,12,14,16,17,18,20]
[16,20,22,40,50,55,80,100,110]
[55,80,100,110]
```

We can string functions and list comprehensions

- We can even get combine lists of adjectives and nouns for some laughs

```
let nouns = ["hobo","frog","pope"]  
let adjectives = ["lazy","grouchy","scheming"]  
[adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
```

We can even use projections

- If we keep track of how many elements we throw away we can redefine the 'length' function
- ' _ ' means we don't care what the element is

```
let length' xs = sum [1 | _ <- xs]  
i==i
```

We can even use membership to keep only a certain items

- Here we keep only capital letters
- Let's write a function that keeps only lower case

```
let removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
removeNonUppercase "Hahaha! Ahahaha!"
removeNonUppercase "IdontLIKEFROGS"
let removeNonLowercase st = [ c | c <- st, c `elem` ['a'..'z']]
removeNonLowercase "i love AND HATE haskell"
i==i
```

Prelude|

```
<interactive>:373:1: parse error on input 'removeNonUppercase'
```

```
<interactive>:373:1: Not in scope: 'removeNonUppercase'
```

Prelude|

```
<interactive>:376:1: parse error on input 'removeNonLowercase'
```


Nested lists can be used to remove odd numbers in a list of lists

- We could do this across several lines

```
let xxs = [[1,3,5,2,3,1,2,4,5], [1,2,3,4,5,6,7,8,9], [1,2,4,2,1,6]]
          [[ x | x <- xs, even x ] | xs <- xxs]
i==i
```

Ordered pairs in Haskell

- Tuples in Haskell have stronger type checking

```
[[1,2],[8,11,5],[4,5]] --doesn't throw an error  
[(1, 2), (8, 11, 5), (4, 5)]  
i==i
```

```
[[1,2],[8,11,5],[4,5]]
```

```
<interactive>:383:10:
```

```
Couldn't match expected type '(t0, t1)'  
      with actual type '(t2, t3, t4)'
```

```
In the expression: (8, 11, 5)
```

```
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
```

```
In an equation for 'it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

A 2-tuple is also called an ordered pair

- There are two special functions which return the components

```
fst ("One", 1)
```

```
snd ("Two", 2)
```

```
i==i
```

```
One
```

```
2
```

Like Python, Haskell has a zip operation

- We can zip two lists into a list of tuples
- The two lists do not need to be the same type
- One list can be shorter than the other
- We can zip finite lists with infinite ones (lazy)

```
zip [1,2,3,4,5] [5,5,5,5,5]
zip [1 .. 5] ["one", "two", "three", "four", "five"]
zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
zip [1..] ["apple", "orange", "cherry", "mango"]
i==i

[(1,5),(2,5),(3,5),(4,5),(5,5)]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
[(5,"im"),(3,"a"),(2,"turtle")]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

Let's find a problem that puts constraints on tuples

- which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?

Let's find a problem that puts constraints on tuples

- which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?
- crack the problem like an egg

Let's find a problem that puts constraints on tuples

- which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?
- crack the problem like an egg
- generate all tuples of sides less than 10

```
length [(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10]]  
i==i
```

```
1000
```

Let's add constraints

Let's add constraints

- make $b < c$

Let's add constraints

- make $b < c$
- make only right triangles
- Here is the first

```
length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z])  
i==i
```

450

Let's add constraints

Let's add constraints

- make $a^2 + b^2 = c^2$

Let's add constraints

- make $a^2 + b^2 = c^2$
- the perimeter equal 24
- $a + b + c = 24$

```
length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z,(x^2 + y^2 == z^2)])
```

4

Let's add constraints

- the perimeter equal 24
- $a + b + c = 24$

```
:set +m
```

```
length([(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],  
y<z,  
x+y+z==24,  
(x^2 + y^2 == z^2)])  
[(x,y,z) | x<-[1..10],y<-[1..10],z<-[1..10],y<z,  
x+y+z==24,  
(x^2 + y^2 == z^2)]  
i==i
```

```
Prelude> Prelude| Prelude| Prelude| 2  
Prelude| Prelude| [(6,8,10),(8,6,10)]
```