Intro to Haskell Notes: Part 1

Adrian Brasoveanu*

July 1, 2013

Contents

*Based primarily on *Learn You a Haskell for Great Good!* by Miran Lipovača, http://learnyouahaskell.com/.

1 Comments

```
ghci 1>    -- to comment out a line, use –
```

2 Simple arithmetic

```
ghci 2> 2 + 15
    17
```

```
ghci 3> 49 * 100
    4900
```

```
ghci 4> 5 / 2
    2.5
```

```
ghci 5> (50 * 100) - 4999
    1
```

```
ghci 6> 50 * 100 - 4999
    1
```

```
ghci 7> 50 * (100 - 4999)
    -244950
```

A little pitfall to watch out for here is negating numbers: if you want a negative number, it's always best to surround it with parentheses. Doing $5 * -3$ will make `ghci` yell at you, but doing $5 * (-3)$ will work just fine.

```
ghci 8> 5 * -3
```

```
ghci 9> 5 * (-3)
    -15
```

3 Boolean operators

ghci 10> $True \wedge False$
False

ghci 11> $True \wedge True$
True

ghci 12> $False \vee True$
True

ghci 13> $\neg\ False$
True

ghci 14> $\neg\ (True \wedge True)$
False

4 Testing for equality

ghci 15> $5 \equiv 5$
True

ghci 16> $1 \equiv 0$
False

ghci 17> $5 \not\equiv 5$
False

ghci 18> $5 \not\equiv 4$
True

ghci 19> `"hello"` $\equiv$ `"hello"`
True

5 No covert type conversion

```
ghci 20> 5 + "llama"
```

```
ghci 21> 5 == True
```

6 Functions

We've been using functions now all along. For instance, $*$ is a function that takes two numbers and multiplies them. As we've seen already, we call it by sandwiching it between them. This is what we call an infix function.

Most functions that aren't used with numbers are prefix functions. Let's take a look at them.

6.1 Calling functions

In most imperative languages, functions are called by writing the function name and then writing its parameters / arguments in parentheses, usually separated by commas.

In Haskell, functions are called by writing the function name, a space and then the parameters, separated by spaces. For starters, we'll try calling one of the most boring functions in Haskell.

```
ghci 22> succ 8
    9
```

The *succ* function takes anything that has a defined successor and returns that successor. As you can see, we just separate the function name from the parameter with a space.

Calling a function with several parameters is also simple. The functions *min* and *max* take two things that can be put in an order (like numbers); *min* returns the one that's less than the other and *max* returns the one that's greater:

```
ghci 23> min 9 10
    9
```

```
ghci 24> min 3.4 3.2
    3.2
```

```
ghci 25> max 100 101
    101
```

Function application (calling a function by putting a space after it and then typing out the parameters) has the highest precedence of them all. What that means for us is that these two statements are equivalent.

```
ghci 26> succ 9 + max 5 4 + 1
    16
```

```
ghci 27> (succ 9) + (max 5 4) + 1
    16
```

However, if we wanted to get the successor of the product of numbers 9 and 10, we couldn't write *succ* $9 * 10$ because that would get the successor of 9, which would then be multiplied by 10, so the result would be 100. We'd have to write *succ* $(9 * 10)$ to get 91.

```
ghci 28> succ 9 * 10
    100
```

```
ghci 29> succ (9 * 10)
    91
```

If a function takes two parameters, we can also call it as an infix function by surrounding it with backticks.

For instance, the *div* function takes two integers and does integral division between them. Doing *div* 92 10 results in a 9.

But when we call it like that, there may be some confusion as to which number is doing the division and which one is being divided. So we can call it as an infix function by doing 92 `` `div` `` 10 and it's much clearer.

```
ghci 30> div 92 10
    9
```

```
ghci 31> 92 `div` 10
    9
```

6.2 Making new functions

Now that we got a basic feel for calling functions, let's try making our own.

Functions are defined in a similar way that they are called. The function name is followed by parameters seperated by spaces. But when defining functions, there's a $=$ and after that we define what the function does. If we were to save this function in a separate file, e.g., `baby.hs`, we would just type the following code:

(1) *doubleMe* $x = x + x$

We would then navigate to where it's saved and run `ghci` from there. Once inside GHCI, we would do *:l baby* and the script would load.

But we will type (or copy-paste) the function directly into `ghci`, so we need to add the keyword **let** before the function definition. Doing **let** $a = 1$ inside `ghci` is the equivalent of writing $a = 1$ in a script and then loading it.

Remember that **let** is needed only when typing the function directly in the interpreter.

```
ghci 32> let doubleMe x = x + x
```

We can also add curly brackets around the function definition like so:

```
ghci 33> let {doubleMe x = x + x}
```

Or even add a semicolon at the end of the function definition:

```
ghci 34> let {doubleMe x = x + x; }
```

Let's play with the function now:

```
ghci 35> doubleMe 9
    18
```

Because + works on integers as well as on floating-point numbers (anything that can be considered a number, really), our function also works on any number.

```
ghci 36> doubleMe 8.3
    16.6
```

Let's make a function that takes two numbers and multiplies each by two and then adds them together.

```
ghci 37> let {doubleUs x y = x * 2 + y * 2}
```

We could have also defined it as *doubleUs x y = x + x + y + y*. Testing it out produces pretty predictable results.

```
ghci 38> doubleUs 4 9
    26
```

```
ghci 39> doubleUs 2.3 34.2
    73.0
```

```
ghci 40> doubleUs 28 88 + doubleMe 123
    478
```

As expected, you can call your own functions from other functions that you made. With that in mind, we could redefine *doubleUs* like this:

```
ghci 41> let {doubleUs x y = doubleMe x + doubleMe y}
```

It still works as expected:

```
ghci 42> doubleUs 4 9
    26
```

```
ghci 43> doubleUs 2.3 34.2
    73.0
```

```
ghci 44> doubleUs 28 88 + doubleMe 123
    478
```

This is a very simple example of a common pattern you will see throughout Haskell: making basic functions that are obviously correct and then combining them into more complex functions.

This way we avoid repetition, but more importantly, this is compositionality of meaning in action with Haskell programs, just as we see compositionality in action when we assign meaning to complex expressions in English or any other natural language.

Functions in Haskell don't have to be in any particular order, so it doesn't matter if you define *doubleMe* first and then *doubleUs* or if you do it the other way around. However, if you're working in `ghci` directly, you should define *doubleMe* first and then *doubleUs*.

6.3 A little bit on the if statement

Now we're going to make a function that multiplies a number by 2, but only if that number is smaller than or equal to 100 – because numbers bigger than 100 are already big enough as it is.

```
ghci 45> let {doubleSmallNumber x = if x > 100 then x else x * 2}
```

```
ghci 46> doubleSmallNumber 99
    198
```

```
ghci 47> doubleSmallNumber 100
    200
```

```
ghci 48> doubleSmallNumber 101
    101
```

We just introduced Haskell's **if** statement. The difference between Haskell's **if** statement and **if** statements in imperative languages is that the **else** part is mandatory in Haskell.

The **if** statement in Haskell is an expression. An expression is basically a piece of code that returns a value – just like an expression in a logical or natural language that gets assigned a semantic value. For example:

- 5 is an expression and it returns 5, i.e., it gets evaluated as 5
- $4 + 8$ is an expression, and it gets evaluated as 12

- more generally, $x + y$ is an expression and it returns the sum of x and y

Because the **else** is mandatory, an **if** statement will always return something and that's why it's an expression.

6.4 A bit more on operator precedence

If we wanted to add 1 to every number that's produced by our previous function, we could have written its body like this.

```
ghci 49> let {doubleSmallNumber2 x = (if x > 100 then x else x * 2) + 1}
```

```
ghci 50> doubleSmallNumber2 99
199
```

```
ghci 51> doubleSmallNumber2 100
201
```

```
ghci 52> doubleSmallNumber2 101
102
```

Had we omitted the parentheses, it would have added 1 only if x wasn't greater than 100.

```
ghci 53> let {doubleSmallNumber' x = if x > 100 then x else x * 2 + 1}
```

```
ghci 54> doubleSmallNumber' 99
199
```

```
ghci 55> doubleSmallNumber' 100
201
```

```
ghci 56> doubleSmallNumber' 101
101
```

6.5 The apostrophe

Note the ' at the end of the function name. That apostrophe doesn't have any special meaning in Haskell's syntax. It's a valid character to use in a function name. We usually use ' to either denote a strict version of a function (one that isn't lazy; more about that later) or a slightly modified version of a function or a variable. Because ' is a valid character in functions, we can make a function like this.

```
ghci 57> let {conanO'Brien = "It's a-me, Conan O'Brien!"}
```

```
ghci 58> conanO'Brien
    "It's a-me, Conan O'Brien!"
```

There are two noteworthy things here. The first is that we didn't capitalize Conan's name in the function name. That's because functions can't begin with uppercase letters.

The second thing is that this function doesn't take any parameters. When a function doesn't take any parameters, we usually say it's a definition (or a name). Because we can't change what names (and functions) mean once we've defined them, *conanO'Brien* and the string `"It's a-me, Conan O'Brien!"` can be used interchangeably.

7 Lists

Much like shopping lists in the real world, lists in Haskell are very useful. It's the most used data structure and it can be used in a multitude of different ways to model and solve a whole bunch of problems. In this section, we'll look at the basics of lists, strings (which are lists) and list comprehensions.

7.1 The basics

In Haskell, a list is a homogenous data structure: it stores elements of the same type. That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.

```
ghci 59> let lostNumbers = [4,8,15,16,23,42]
```

```
ghci 60> lostNumbers
    [4,8,15,16,23,42]
```

Lists are denoted by square brackets and the values in the lists are separated by commas. If we tried a list like `[1,2,'a',3,'b','c',4]`, Haskell would complain that characters (which are, by the way, denoted as a character between single quotes) are not numbers.

```
ghci 61> let badList = [1,2,'a',3,'b','c',4]
```

Speaking of characters, strings are just lists of characters. `"hello"` is just syntactic sugar for `['h','e','l','l','o']`. Because strings are lists, we can use list functions on them, which is really handy.

```
ghci 62> let greeting1 = "hello"
```

```
ghci 63> greeting1
    "hello"
```

```
ghci 64> let greeting2 = ['h','e','l','l','o']
```

```
ghci 65> greeting2
    "hello"
```

7.2 List concatenation

A common task is putting two lists together. This is done by using the ++ operator.

```
ghci 66> [1,2,3,4] ++ [9,10,11,12]
    [1,2,3,4,9,10,11,12]
```

```
ghci 67> "hello" ++ " " ++ "world"
    "hello world"
```

```
ghci 68> ['w','o'] ++ ['o','t']
    "woot"
```

Watch out when repeatedly using the ++ operator on long strings. When we put together two lists – even if we append a singleton list to a list, for instance, $[1,2,3] ++ [4]$, Haskell has to walk through the whole list on the left side of ++. That's not a problem when dealing with lists that aren't too big. But putting something at the end of a list that's 50 million entries long is going to take a while.

7.3 Prepending elements to lists

However, putting something at the beginning of a list using the : operator, also called the *cons* operator (for 'constructor', as in 'list constructor'), is instantaneous.

```
ghci 69> 'A':" SMALL CAT"
    "A SMALL CAT"
```

```
ghci 70> 5:[1,2,3,4,5]
    [5,1,2,3,4,5]
```

Notice how : takes a number and a list of numbers or a character and a list of characters, whereas ++ takes two lists. Even if we're adding an element to the end of a list with ++, we have to surround it with square brackets so it becomes a list, e.g.:

```
ghci 71> [1,2,3,4] ++ [5]
[1,2,3,4,5]
```

`[1,2,3]` is actually just syntactic sugar for `1:2:3:[]`. `[]` is an empty list. If we prepend 3 to it, it becomes `[3]`. If we prepend 2 to that, it becomes `[2,3]`, and so on.

```
ghci 72> 1:2:3:[]
[1,2,3]
```

`[]`, `[[]]` and `[[],[],[]]` are all different things. The first one is an empty list, the seconds one is a list that contains one empty list, the third one is a list that contains three empty lists.

7.4 List indexation

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

```
ghci 73> "Steve Buscemi" !! 6
'B'
```

```
ghci 74> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

But if we try to get the 6th element from a list that only has 4 elements, we'll get an error.

```
ghci 75> [9.4,33.2,96.2,11.2] !! 6
*** Exception: Prelude.(!!): index too large
```

Lists can also contain lists. They can also contain lists that contain lists that contain lists ...

```
ghci 76> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
```

```
ghci 77> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
```

```
ghci 78> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
```

```
ghci 79> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
```

```
ghci 80> b !! 2
    [1,2,2,3,4]
```

The lists within a list can be of different lengths but they can't be of different types. Just like we can't have a list that has some characters and some numbers, we can't have a list that contains lists of characters and also lists of numbers.

7.5 Comparing lists

Lists can be compared if the stuff they contain can be compared. When using <, <=, > and >= to compare lists, they are compared in lexicographical order. First the heads are compared. If they are equal, the second elements are compared etc.

```
ghci 81> [3,2,1] > [2,1,0]
    True
```

```
ghci 82> [3,2,1] > [2,10,100]
    True
```

```
ghci 83> [3,4,2] > [3,4]
    True
```

```
ghci 84> [3,4,2] > [2,4]
    True
```

```
ghci 85> [3,4,2] ≡ [3,4,2]
    True
```

7.6 Basic functions on lists

What else can we do with lists? Here are some basic functions that operate on lists.

head takes a list and returns its head. The head of a list is basically its first element.

```
ghci 86> head [5,4,3,2,1]
    5
```

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

```
ghci 87> tail [5,4,3,2,1]
    [4,3,2,1]
```

last takes a list and returns its last element.

```
ghci 88> last [5,4,3,2,1]
    1
```

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

```
ghci 89> init [5,4,3,2,1]
    [5,4,3,2]
```

What happens if we try to get the head of an empty list?

```
ghci 90> head []
    *** Exception: Prelude.head: empty list
```

When using *head*, *tail*, *last* and *init*, be careful not to use them on empty lists. This error cannot be caught at compile time so it's always good practice to take precautions against accidentally telling Haskell to give you some elements from an empty list.

length takes a list and returns its length, obviously.

```
ghci 91> length [5,4,3,2,1,5,4,3,2,1]
    10
```

null checks if a list is empty. If it is, it returns *True*, otherwise it returns *False*. Use this function instead of $xs \equiv []$ – if you have a list called *xs*, that is.

```
ghci 92> null [1,2,3]
    False
```

```
ghci 93> null []
    True
```

reverse reverses a list.

```
ghci 94> reverse [5,4,3,2,1]
    [1,2,3,4,5]
```

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

```
ghci 95> take 3 [5,4,3,2,1]
    [5,4,3]
```

```
ghci 96> take 1 [3,9,3]
[3]
```

```
ghci 97> take 5 [1,2]
[1,2]
```

```
ghci 98> take 0 [6,6,6]
[]
```

See how if we try to take more elements than there are in the list, it just returns the list. If we try to take 0 elements, we get an empty list.

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

```
ghci 99> drop 3 [8,4,2,1,5,6]
[1,5,6]
```

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

```
ghci 101> drop 100 [1,2,3,4]
[]
```

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

```
ghci 102> maximum [1,9,2,3,4]
9
```

minimum returns the smallest.

```
ghci 103> minimum [8,4,2,1,5,6]
1
```

sum takes a list of numbers and returns their sum.

```
ghci 104> sum [5,2,1,6,3,2,5,7]
31
```

product takes a list of numbers and returns their product.

```
ghci 105> product [6,2,1,2]
          24
```

```
ghci 106> product [1,2,5,6,7,9,2,0]
          0
```

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 107> 4 ∈ [3,4,5,6]
          True
```

```
ghci 108> 10 ∈ [3,4,5,6]
          False
```

Here it is in prefix notation:

```
ghci 109> elem 4 [3,4,5,6]
          True
```

```
ghci 110> elem 10 [3,4,5,6]
          False
```

```
ghci 111> elem [3,4,5,6] 4
```

Those were a few basic functions that operate on lists. We'll take a look at more list functions later.

7.7 List ranges

What if we want a list of all numbers between 1 and 20? We use ranges.

Ranges are a way of making lists that are arithmetic sequences of elements that can be enumerated. Numbers can be enumerated. Characters can also be enumerated. Names can't be enumerated.

To make a list containing all the natural numbers from 1 to 20, we just write $[1..20]$. That is the equivalent of writing $[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]$ and there's no difference between writing one or the other.

```
ghci 112> [1..20]
          [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

```
ghci 113> ['a'..'z']
    "abcdefghijklmnopqrstuvwxyz"
```

```
ghci 114> ['K'..'Z']
    "KLMNOPQRSTUVWXYZ"
```

You can also specify a step. What if we want all even numbers between 1 and 20? Or every third number between 1 and 20?

```
ghci 115> [2,4..20]
    [2,4,6,8,10,12,14,16,18,20]
```

```
ghci 116> [3,6..20]
    [3,6,9,12,15,18]
```

It's simply a matter of separating the first two elements with a comma and then specifying what the upper limit is.

While pretty smart, ranges with steps aren't as smart as some people expect them to be. You can't do $[1,2,4,8,16..100]$ and expect to get all the powers of 2. Why? Because we can only specify one step. And also because some sequences that aren't arithmetic are ambiguous if given only by a few of their first terms.

```
ghci 117> [1,2,4,8,16..100]
```

```
ghci 118> [1,2..100]
    [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
    32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,
    60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,
    88,89,90,91,92,93,94,95,96,97,98,99,100]
```

To make a list with all the numbers from 20 to 1, we can't just do $[20..1]$, we have to do $[20,19..1]$.

```
ghci 119> [20..1]
    []
```

```
ghci 120> [20,19..1]
    [20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

Watch out when using floating point numbers in ranges! Because they are not completely precise (by definition), their use in ranges can yield some pretty funky results.

ghci 121> $[0.1, 0.3 \,..\, 1]$
$[0.1, 0.3, 0.5, 0.7, 0.8999999999999999, 1.0999999999999999]$

ghci 122> $[0.1, 0.3 \,..\, 5.0]$
$[0.1, 0.3, 0.5, 0.7, 0.8999999999999999, 1.0999999999999999, 1.2999999999999998,$
$1.4999999999999998, 1.6999999999999997, 1.8999999999999997, 2.0999999999999996, 2.3,$
$2.5, 2.7, 2.9000000000000004, 3.1000000000000005, 3.3000000000000007, 3.500000000000001,$
$3.700000000000001, 3.9000000000000012, 4.100000000000001, 4.300000000000002,$
$4.500000000000002, 4.700000000000002, 4.900000000000002]$

Just don't use them in list ranges.

7.8 Infinite lists

You can also use ranges to make infinite lists by just not specifying an upper limit. Later we'll go into more detail on infinite lists. For now, let's examine how we would get the first 24 multiples of 13.

Sure, we could do $[13, 26 \,..\, 24 * 13]$. But there's a better way: *take* 24 $[13, 26 \,..]$. Because Haskell is lazy, it won't try to evaluate the infinite list immediately because it would never finish. It'll wait to see what we want to get out of that infinite lists. And here it sees we just want the first 24 elements and it gladly obliges.

ghci 123> $[13, 26 \,..\, 24 * 13]$
$[13, 26, 39, 52, 65, 78, 91, 104, 117, 130, 143, 156, 169, 182, 195, 208, 221, 234, 247, 260, 273, 286,$
$299, 312]$

ghci 124> *take* 24 $[13, 26 \,..]$
$[13, 26, 39, 52, 65, 78, 91, 104, 117, 130, 143, 156, 169, 182, 195, 208, 221, 234, 247, 260, 273, 286,$
$299, 312]$

7.9 Some functions that produce infinite lists

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

ghci 125> *take* 10 (*cycle* $[1, 2, 3]$)
$[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]$

ghci 126> *take* 12 (*cycle* `"LOL "`)
`"LOL LOL LOL "`

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

```
ghci 127> take 10 (repeat 5)
    [5,5,5,5,5,5,5,5,5,5]
```

But it's simpler to just use the *replicate* function if we want several tokens of the same element in a list.

```
ghci 128> replicate 3 10
    [10,10,10]
```

```
ghci 129> replicate 10 3
    [3,3,3,3,3,3,3,3,3,3]
```

7.10 List comprehension

List comprehension: build lists out of other lists. E.g., generate the first ten even natural numbers by doubling the first ten natural numbers:

```
ghci 130> [x * 2 | x ← [1..10]]
    [2,4,6,8,10,12,14,16,18,20]
```

The part before | is the output, x is the variable, $[1..10]$ is the input list.

We can add a condition, i.e., a predicate, to that comprehension. Predicates go after the binding parts (i.e., the variable and the input list) and are separated from them by a comma.

Let's say we want only the elements which, when doubled, are greater than or equal to 12.

```
ghci 131> [x * 2 | x ← [1..10], x * 2 ⩾ 12]
    [12,14,16,18,20]
```

What if we wanted all numbers from 50 to 100 whose remainder when divided by 7 is 3?

```
ghci 132> [x | x ← [50..100], x `mod` 7 ≡ 3]
    [52,59,66,73,80,87,94]
```

Weeding out lists by predicates is also called filtering. We took a list of numbers and we filtered it by the predicate.

Another example: we want a comprehension that replaces each odd number greater than 10 with `"BANG!"` and each odd number that's less than 10 with `"BOOM!"`. If a number isn't odd, we throw it out of our list.

For convenience, we'll put that comprehension inside a function so we can easily reuse it.

```
ghci 133> let {boomBangs xs = [if x < 10 then "BOOM!" else "BANG!" | x ← xs, odd x]}
```

```
ghci 134> [7..13]
    [7,8,9,10,11,12,13]
```

```
ghci 135> boomBangs [7..13]
    ["BOOM!","BOOM!","BANG!","BANG!"]
```

```
ghci 136> boomBangs [7..15]
    ["BOOM!","BOOM!","BANG!","BANG!","BANG!"]
```

We can include several predicates. If we wanted all numbers from 10 to 20 that are not 13, 15 or 19, we'd do:

```
ghci 137> [x | x ← [10..20], x ≠ 13, x ≠ 15, x ≠ 19]
    [10,11,12,14,16,17,18,20]
```

Not only can we have multiple predicates in list comprehensions (an element must satisfy all the predicates to be included in the resulting list), we can also draw from several lists.

When drawing from several lists, comprehensions produce all combinations of the given lists and then joins them by the output function we supply.

```
ghci 138> [x * y | x ← [2,5,10], y ← [8,10,11]]
    [16,20,22,40,50,55,80,100,110]
```

As expected, the length of the new list is 9. What if we wanted all possible products that are more than 50?

```
ghci 139> [x * y | x ← [2,5,10], y ← [8,10,11], x * y > 50]
    [55,80,100,110]
```

How about a list comprehension that combines a list of adjectives and a list of nouns?

```
ghci 140> let nouns = ["hobo","frog","pope"]
```

```
ghci 141> let adjectives = ["lazy","grouchy","scheming"]
```

```
ghci 142> [adjective ++ " " ++ noun | adjective ← adjectives, noun ← nouns]
    ["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
    "grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

8 Nameless variables

Let's write our own version of *length*. We'll call it *length'*.

```
ghci 143> let {length' xs = sum [1 | _ ← xs]}
```

_ means that we don't care what we draw from the list. So instead of writing a variable name that we'll never use, we just write _.

This function replaces every element of a list with 1 and then sums that up. This means that the resulting sum will be the length of our list.

```
ghci 144> [1 | _ ← "hello"]
    [1,1,1,1,1]
```

```
ghci 145> sum [1 | _ ← "hello"]
    5
```

Recall that since strings are lists, we can use list comprehensions to process and produce strings. For example, here's a function that takes a string and removes everything except uppercase letters from it.

```
ghci 146> let {removeNonUppercase st = [c | c ← st, c ∈ ['A'..'Z']]}
```

```
ghci 147> removeNonUppercase "Hahaha! Ahahaha!"
    "HA"
```

```
ghci 148> removeNonUppercase "IdontLIKEFROGS"
    "ILIKEFROGS"
```

The predicate here does all the work. It says that the character will be included in the new list only if it's an element of the list ['A'..'Z'].

Nested list comprehensions are also possible if we're operating on lists that contain lists. E.g., if a list contains several lists of numbers, we can remove all odd numbers without flattening the list.

```
ghci 149> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
```

```
ghci 150> [[x | x ← xs, even x] | xs ← xxs]
    [[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

You can write list comprehensions across several lines. So if we're not in `ghci`, it's better to split longer list comprehensions across multiple lines, especially if they're nested.

9 Tuples

In some ways, tuples are like lists: they are a way to store several values into a single value. However, there are a few fundamental differences.

A list of numbers is a list of numbers. That's its type and it doesn't matter if it has only one number in it or an infinite amount of numbers. Tuples, however, are used when we know exactly how many values we want to combine and its type depends on how many components it has and the types of the components. They are denoted with parentheses and their components are separated by commas.

Another key difference is that they don't have to be homogenous. Unlike a list, a tuple can contain a combination of several types.

Think about how we'd represent a two-dimensional vector in Haskell. One way would be to use a list. That would work, kind of.

So what if we wanted to put a couple of vectors in a list to represent a shape on a two-dimensional plane? We could do something like $[[1,2],[8,11],[4,5]]$. The problem with that method is that we could also do stuff like $[[1,2],[8,11,5],[4,5]]$, which Haskell has no problem with since it's still a list of lists of numbers. But it doesn't make sense for our intended application.

On the other hand, a tuple of size 2 (also called a pair) is its own type, which means that a list can't have a couple of pairs in it and then a triple (a tuple of size three), so it's better to use tuples instead.

Instead of surrounding the vectors with square brackets, we use parentheses: $[(1,2),(8,11),(4,5)]$. What if we tried to make a shape like $[(1,2),(8,11,5),(4,5)]$? Well, we'd get an error.

```
ghci 151> [(1,2),(8,11),(4,5)]
    [(1,2),(8,11),(4,5)]
```

```
ghci 152> [(1,2),(8,11,5),(4,5)]
```

You also couldn't make a list like $[(1,2),(\texttt{"One"},2)]$ because the first element of the list is a pair of numbers and the second element is a pair consisting of a string and a number.

```
ghci 153> [("Two",2),("One",2)]
    [("Two",2),("One",2)]
```

```
ghci 154> [(1,2),("One",2)]
```

Tuples can also be used to represent a wide variety of data. For instance, if we wanted to represent someone's name and age in Haskell, we could use a triple: `("James","Bond",85)`. As seen in this example, tuples can also contain lists.

Use tuples when you know in advance how many components some piece of data should have. Tuples are much more rigid because each different size is associated with its own type, so we can't write a general function to append an element to a tuple – we'd have to write a function for appending to a pair, one function for appending to a triple, one function for appending to a 4-tuple, etc.

Note: while there are singleton lists, there's no such thing as a singleton tuple.

Like lists, tuples can be compared with each other if their components can be compared. Only we can't compare two tuples of different sizes, whereas we can compare two lists of different sizes.

9.1 Two functions that operate on pairs

fst takes a pair and returns its first component.

```
ghci 155> fst (8,11)
8
```

```
ghci 156> fst ("Wow",False)
"Wow"
```

snd takes a pair and returns its second component.

```
ghci 157> snd (8,11)
11
```

```
ghci 158> snd ("Wow",False)
False
```

These functions operate only on pairs. They won't work on triples, 4-tuples, 5-tuples, etc. We'll go over extracting data from tuples in different ways a bit later.

10 Zip

A cool function that produces a list of pairs: *zip*. It takes two lists and then zips them together into one list by joining the matching elements into pairs.

It's a really simple function but it has lots of uses. It's especially useful when we want to combine two lists or traverse two lists simultaneously.

Here's a demonstration.

```
ghci 159> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
```

zip pairs up the elements and produces a new list. The first element in the first list goes with the first element in the second list, the second with the second etc. Because pairs can have different types in them, zip can take two lists that contain different types and zip them up.

```
ghci 160> zip [1..5] ["one","two","three","four","five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

What happens if the lengths of the lists don't match?

```
ghci 161> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

The longer list simply gets cut off to match the length of the shorter one.

Because Haskell is lazy, we can zip finite lists with infinite lists:

```
ghci 162> zip [1..] ["apple","orange","cherry","mango"]
          [(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

11 Bringing it all together: an example

Here's a problem that combines tuples and list comprehensions: which right triangles (if any) are such that they have sides of integer length, the lengths of the sides are equal to or smaller than 10, and the triangles have a perimeter of 24?

First, let's try generating all triangles with sides equal to or smaller than 10:

```
ghci 163> let triangles = [(a,b,c) | a ← [1..10], b ← [1..10], c ← [1..10]]
```

We're just drawing from three lists and our output function is combining them into a triple. If we evaluate that by typing out *triangles* in `ghci`, we'll get a list of all possible triangles with sides at most 10.

```
ghci 164> take 30 triangles
          [(1,1,1),(1,1,2),(1,1,3),(1,1,4),(1,1,5),(1,1,6),(1,1,7),(1,1,8),(1,1,9),(1,1,10),(1,2,
          1),(1,2,2),(1,2,3),(1,2,4),(1,2,5),(1,2,6),(1,2,7),(1,2,8),(1,2,9),(1,2,10),(1,3,1),(1,3,
          2),(1,3,3),(1,3,4),(1,3,5),(1,3,6),(1,3,7),(1,3,8),(1,3,9),(1,3,10)]
```

Next, we'll add a condition that they all have to be right triangles. We'll also modify this function by taking into consideration that side b isn't larger than the hypothenuse and that side a isn't larger than side b (this last constraint just eliminates unnecessary duplication).

```
ghci 165> let rightTriangles = [(a,b,c) | c ← [1..10], b ← [1..c], a ← [1..b],
                                a↑2 + b↑2 ≡ c↑2]
```

```
ghci 166> rightTriangles
          [(3,4,5),(6,8,10)]
```

We're almost done. Now, we just modify the function by saying that we want the ones where the perimeter is 24.

```
ghci 167> let rightTriangles' = [(a,b,c) | c ← [1..10], b ← [1..c], a ← [1..b],
                                 a↑2 + b↑2 ≡ c↑2, a + b + c ≡ 24]
```

```
ghci 168> rightTriangles'
          [(6,8,10)]
```

And there's our answer!

This is a common pattern in functional programming: we take a starting set that we know includes the solutions (if any) and then we apply transformations to that set and filter them until we get to the solutions.