Haskell
Starting Out

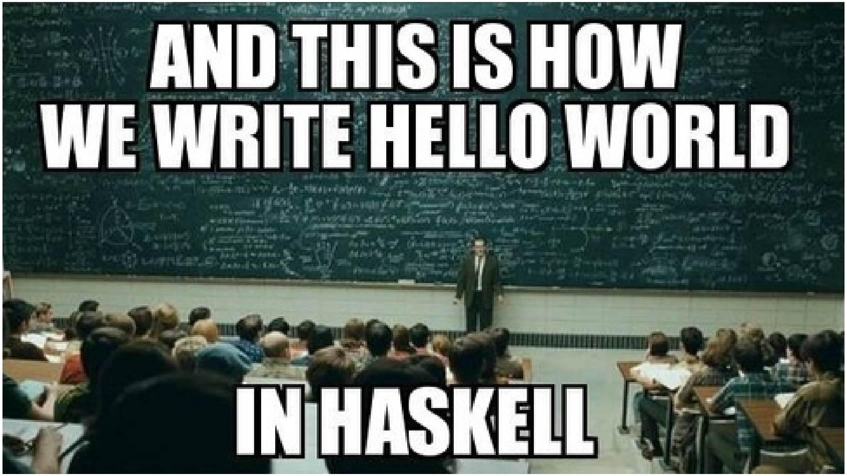
http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

November 13, 2016

Ready, set, go!



Ready, set, go!

ghc's interactive mode

```
barbalala-vialette: ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/
:? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :set prompt "ghci> "
ghci> 1 + 2
3
ghci> 3 * 4
12
ghci> 5 - 6
-1
ghci> 7 / 8
0.875
ghci>
```

Ready, set, go!

ghc's interactive mode

```
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Ready, set, go!

ghc's interactive mode

```
ghci> 1 == 1
True
ghci> 1 == 2
False
ghci> 1 /= 2
True
ghci> 1 /= 1
False
ghci> False && True
False
```

Ready, set, go!

ghc's interactive mode

```
ghci> 1 == True
<interactive>:17:1:
    No instance for (Num Bool) arising from the literal '1'
    In the first argument of '$\boxed{(==}$)', namely '1'
    In the expression: 1 == True
    In an equation for 'it': it = 1 == True
ghci>
```

Functions

ghc's interactive mode

$*$ is a function that takes two numbers and multiplies them.

As you've seen, 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.

Functions

ghc's interactive mode

```
ghci> succ 1
2
ghci> :type succ
succ :: Enum a => a -> a
ghci>
ghci> min 2 1
1
ghci> :type min
min :: Ord a => a -> a -> a
ghci>
ghci> max 2 1
2
ghci> :type max
max :: Ord a => a -> a -> a
ghci>
```

Functions

ghc's interactive mode

Function application has the highest precedence of them all. What that means for us is that these two statements are equivalent.

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

Baby's first functions

```
barbalala-vialette: cat Baby.hs
doubleMe x = x + x
barbalala-vialette: ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/
:? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :load "Baby"
[1 of 1] Compiling Main           ( baby.hs, interpreted )
Ok, modules loaded: Main.
*Main> doubleMe 5
10
*Main> doubleMe 100.0
200.0
*Main>
```

Baby's first functions

```haskell
doubleUs x y = 2*x + 2*y
```

```
*Main> doubleUs 4 9
26
*Main> doubleUs 2.3 34.2
73.0
*Main> doubleUs 28 88 + doubleMe 123
478
*Main> doubleUs doubleMe 2 doubleMe 10
<interactive>:11:1:
    No instance for (Num ((a0 -> a0) -> a0 -> a0))
      arising from a use of 'it'
    In a stmt of an interactive GHCi command: print it
*Main> doubleUs (doubleMe 2) (doubleMe 10)
48
*Main>
```

Baby's first functions

```haskell
doubleMe x = x + x
doubleUs x y = doubleMe x + doubleMe y
```

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

Baby's first functions

Write 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 big enough as it is!

```
doubleSmallNumber x = if x > 100
                      then x
                      else 2*x
```

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

Baby's first functions

Another thing about the if statement in Haskell is that it is an expression.

An expression is basically a piece of code that returns a value.

`5` is an expression because it returns 5, `1+2` is an expression, `x+y` is an expression because 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.

If we wanted to add one to every number that's produced in our previous function, we could have written its body like this:

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

Baby's first functions

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) 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:

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

An introduction to lists

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

```
Prelude> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
Prelude> "hello" ++ " " ++ "world"
"hello world"
Prelude> ['w','o'] ++ ['o','t']
"woot"
Prelude>
```

An introduction to lists

When you put together two lists (even if you append a singleton list to a list, for instance: `[1,2,3] ++ [4]`), internally, 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.

However, putting something at the beginning of a list using the `:` operator (also called the cons operator) is instantaneous:

```
Prelude> 'A':" SMALL CAT"
"A SMALL CAT"
Prelude> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
Prelude>
```

Notice how `:` takes a number and a list of numbers or a character and a list of characters, whereas `++` takes two lists.

An introduction to lists

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

```
Prelude> [1,2,3,4,5] !! 0
1
Prelude> "Hello" !! 1
'e'
Prelude> [1,2,3,4,5] !! (-1)
*** Exception: Prelude.(!!): negative index
Prelude> [1,2,3,4,5] !! 5
*** Exception: Prelude.(!!): index too large
Prelude>
```

An introduction to lists

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

```
Prelude> let l = [[1,2,3],[4,5,6],[7,8,9]]
Prelude> l
[[1,2,3],[4,5,6],[7,8,9]]
Prelude> :t l
l :: Num t => [[t]]
Prelude> l ++ [[6,6,6]]
[[1,2,3],[4,5,6],[7,8,9],[6,6,6]]
Prelude> [0,0,0]:l
[[0,0,0],[1,2,3],[4,5,6],[7,8,9]]
Prelude> l !! 1
[4,5,6]
Prelude>
```

An introduction to 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 then the second elements are compared, etc.

```
Prelude> [3,2,1] > [2,1,0]
True
Prelude> [3,2,1] > [2,10,100]
True
Prelude> [3,4,2] > [3,4]
True
Prelude> [3,4,2] > [2,4]
True
Prelude> [3,4,2] == [3,4,2]
True
Prelude>
```

An introduction to lists

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.

```
Prelude> head "Hello"
'H'
Prelude> head [1,2,3,4,5]
1
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

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

```
Prelude> tail "Hello"
"ello"
Prelude> tail [1,2,3,4,5]
[2,3,4,5]
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

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

```
Prelude> last "Hello"
'o'
Prelude> last [1,2,3,4,5]
5
Prelude> last []
*** Exception: Prelude.last: empty list
Prelude>
```

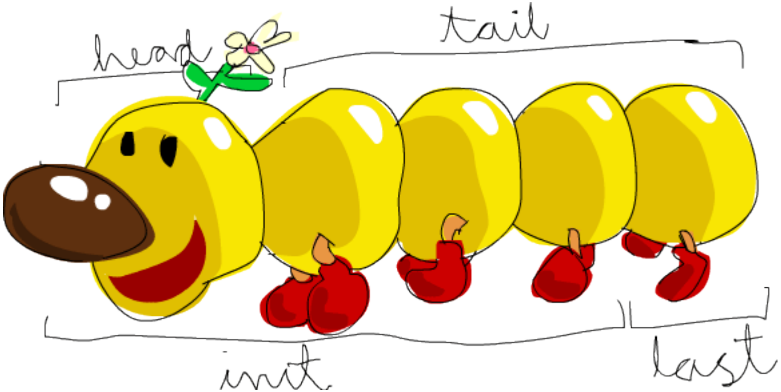
An introduction to lists

Some basic functions that operate on lists

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

```
Prelude> init "Hello"
"Hell"
Prelude> init [1,2,3,4,5]
[1,2,3,4]
Prelude> init []
*** Exception: Prelude.init: empty list
Prelude>
```

An introduction to lists



An introduction to lists

Some basic functions that operate on lists

length takes a list and returns its length, obviously.

```
Prelude> length "Hello"
5
Prelude> length [1,2,3,4,5]
5
Prelude> length []
0
Prelude>
```

An introduction to lists

Some basic functions that operate 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`)

```
Prelude> null "hello"
False
Prelude> null [1,2,3,4,5]
False
Prelude> null []
True
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

`reverse` reverses a list.

```
Prelude> reverse "hello"
"olleh"
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
Prelude> reverse []
[]
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

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

```
Prelude> take 0 [1,2]
[]
Prelude> take 1 [1,2]
[1]
Prelude> take 2 [1,2]
[1,2]
Prelude> take 3 [1,2]
[1,2]
Prelude> take 0 []
[]
Prelude> take 1 []
[]
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

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

```
Prelude> drop 0 [1,2,3]
[1,2,3]
Prelude> drop 1 [1,2,3]
[2,3]
Prelude> drop 2 [1,2,3]
[3]
Prelude> drop 3 [1,2,3]
[]
Prelude> drop 4 [1,2,3]
[]
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

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

`minimum` returns the smallest.

```
Prelude> minimum [3,4,2,5,1,6,9,8,7]
1
Prelude> maximum [3,4,2,5,1,6,9,8,7]
9
Prelude> minimum []
*** Exception: Prelude.minimum: empty list
Prelude> maximum []
*** Exception: Prelude.maximum: empty list
Prelude>
```

An introduction to lists

Some basic functions that operate on lists

sum takes a list of numbers and returns their sum.

product takes a list of numbers and returns their product.

```
Prelude> sum []
0
Prelude> sum [1,2,3,4,5]
15
Prelude> product []
1
Prelude> product [1,2,3,4,5]
120
Prelude> let fact n = product [1..n]
Prelude> fact 5
120
Prelude>
```

An introduction to lists

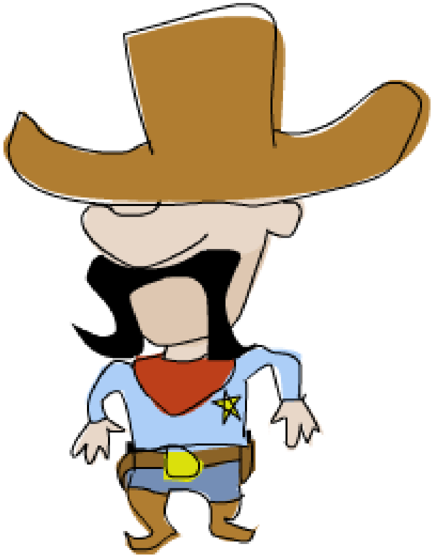
Some basic functions that operate on lists

`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.

```
Prelude> 3 `elem` [2,1,3,5,4]
True
Prelude> elem 3 [2,1,3,5,4]
True
Prelude> 6 `elem` [2,1,3,5,4]
False
Prelude> elem 6 [2,1,3,5,4]
False
Prelude>
```

Texas ranges



Texas ranges

```
Prelude> [1,2,3,4,5,6,7,8,9,10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> [10..1]
[]
Prelude> [1.0..10.0] -- don't do this!
[1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
Prelude> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Prelude>
```

Texas ranges

Ranges are cool because you can also specify a step.

```
Prelude> [10,13..20]
[10,13,16,19]
Prelude> ['a','e'..'z']
"aeimquy"
Prelude> [1,2,4,8,16..100] -- expecting the powers of 2 !
<interactive>:181:12: parse error on input '..''
Prelude> [20,18..5]
[20,18,16,14,12,10,8,6]
Prelude>
```

Texas ranges

Ranges are cool because you can also specify a step.

```
Prelude> [10,13..20]
[10,13,16,19]
Prelude> ['a','e'..'z']
"aeimquy"
Prelude> [1,2,4,8,16..100] -- expecting the powers of 2 !
<interactive>:181:12: parse error on input '..''
Prelude> [20,18..5]
[20,18,16,14,12,10,8,6]
Prelude>
```

Texas ranges

Do not use floating point numbers in ranges!

```
Prelude> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
Prelude> [1, 0.8 .. 0]
[1.0,0.8,0.6000000000000001,0.4000000000000001З,
 0.20000000000000018,2.220446049250313e-16]
Prelude>
```

Texas ranges

You can also use ranges to make infinite lists by just not specifying an upper limit.

Because Haskell is lazy, it won't try to evaluate the infinite list immediately.

```
Prelude> let l = [1..]
Prelude> :t l
l :: (Num t, Enum t) => [t]
Prelude> take 10 l
[1,2,3,4,5,6,7,8,9,10]
Prelude> l -- don't do this
[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,...
```

I’m a list comprehension

I'm a list comprehension

A basic comprehension for a set that contains the first ten even natural numbers is

$$\{2x \mid x \in \mathbb{N}, x \leq 10\}$$

In Hashell

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

I'm a list comprehension

```
Prelude>  [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
Prelude> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
Prelude> :{  -- starts multiline-input mode
Prelude| let ab xs = [if x<10 then "a" else "b" |
Prelude|                x <- xs, odd x]
Prelude| :}  -- terminates multiline-input mode
Prelude> ab [7..13]
["a","a","b","b"]
Prelude> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
Prelude>
```

I'm a list comprehension

```
Prelude> [x+y | x <- [1,2,3], y <- [100,200,300]]
[101,201,301,102,202,302,103,203,303]
Prelude> :{
Prelude| [x+y | x <- [1,2,3], y <- [100,200,300],
Prelude|           x+y > 200]
Prelude| :}
[201,301,202,302,203,303]
Prelude> :{
Prelude| [x+y | x <- [1,2,3], y <- [100,200,300],
Prelude|           x+y > 200, x+y < 300]
Prelude| :}
[201,202,203]
Prelude>
```

I'm a list comprehension

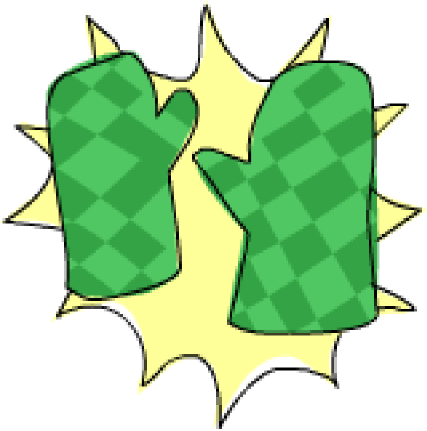
```
Prelude> let length' xs = sum [1 | _ <- xs]
Prelude> length' []
0
Prelude> length' [1..100]
100
Prelude> :{
Prelude| let removeNonUppercase cs = [c |
Prelude|                               c <- cs,
Prelude|                               c `elem` ['A'..'Z']]
Prelude| :}
Prelude> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
Prelude> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
Prelude>
```

I'm a list comprehension

Nested list comprehensions are also possible if you're operating on lists that contain lists.

```
Prelude> let xxs = [[1,2,3],[4,5],[6,7,8,9,10]]
Prelude> [[x | x <- xs, even x] | xs <- xxs]
[[2],[4],[6,8,10]]
Prelude> [[x | x <- xs, even x] | xs <- xxs, length' xs > 2]
[[2],[6,8,10]]
Prelude>
```

Tuples



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 you know exactly how many values you 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.

Tuples

```
Prelude> :t (1,2)
(1,2) :: (Num t1, Num t) => (t, t1)
Prelude> :t (1,2,3)
(1,2,3) :: (Num t2, Num t1, Num t) => (t, t1, t2)
Prelude> [(1,2),(8,11),(4,5)]
[(1,2),(8,11),(4,5)]
Prelude> [(1,2),(8,11,5),(4,5)]
<interactive>:68:8:
    Couldn't match expected type '(t, t3)'
                with actual type '(t0, t1, t2)'
    Relevant bindings include
      it :: [(t, t3)] (bound at <interactive>:68:1)
    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)]
Prelude>
```

Tuples

```
Prelude> :t ('a', 1, "hello")
('a', 1, "hello") :: Num t => (Char, t, [Char])
Prelude> (1, 2) < (3, 4)
True
Prelude> (1, 2) < (0, 1)
False
Prelude> (1, 2, 3) < (1, 2)
<interactive>:74:13:
    Couldn't match expected type '(t0, t1, t2)''
                with actual type '(t3, t4)''
    In the second argument of '(<)', namely '(1, 2)'
    In the expression: (1, 2, 3) < (1, 2)
    In an equation for 'it': it = (1, 2, 3) < (1, 2)
Prelude>
```

Tuples

Two useful functions that operate on pairs

```
Prelude> fst ('a', 2)
'a'
Prelude> snd ('a', 2)
2
Prelude> fst ('a', 2, "hello")
<interactive>:80:5:
    Couldn't match expected type '(a, b0)'
                with actual type '(Char, t0, [Char])'
    Relevant bindings include it :: a (bound at <interactive>:8
    In the first argument of 'fst', namely '('a', 2, "hello")'
    In the expression: fst ('a', 2, "hello")
    In an equation for 'it': it = fst ('a', 2, "hello")
Prelude>
```

Tuples

Two useful functions that operate on pairs

```
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
Prelude> :type fst
fst :: (a, b) -> a
Prelude> :type snd
snd :: (a, b) -> b
Prelude> fst (('a', 'b'),('c', 'd'))
('a','b')
Prelude> fst (('a', 'b', 'c'),('d', 'e', 'f'))
('a','b','c')
Prelude> fst (('a', 'b', 'c'),('d', 'e', 'f', 'g'))
('a','b','c')
Prelude>
```

Tuples

`zip`

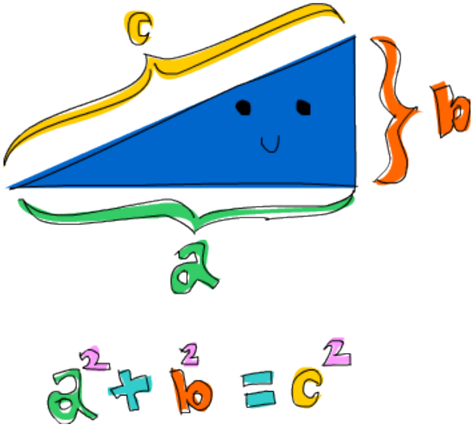
Tuples

`zip`

```
Prelude> zip [1,2,3,4] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
Prelude> zip [1,2,3,4,5] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
Prelude> zip [1,2,3,4] ['a','b','c','d','e']
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
Prelude> zip [1,2,3,4] ['a'..]
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
Prelude> zip [1..] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c'),(4,'d')]
Prelude> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
Prelude>
```

Right triangle

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



Right triangle

```
Prelude> :{
Prelude| let triangles = [(a,b,c) |
Prelude|                  a <- [1..10], b <- [1..10],
Prelude|                  c <- [1..10]]
Prelude| :}
Prelude> :{
Prelude| let rightTriangles = [(a,b,c) |
Prelude|                       (a,b,c) <- triangles,
Prelude|                       a^2 + b^2 == c^2]
Prelude| :}
Prelude> :{
Prelude| let rightTriangles' = [(a,b,c) |
Prelude|                        (a,b,c) <- triangles,
Prelude|                        a^2 + b^2 == c^2,
Prelude|                        a+b+c == 24]
Prelude| :}
Prelude> rightTriangles'
[(6,8,10),(8,6,10)]
Prelude>
```

Done!

ABSTRACTION
NO SIDE EFFECTS
LAZY
STRONG-TYPED
MINIMISES BUGS
ELEGANCE