Haskell
Starting Out

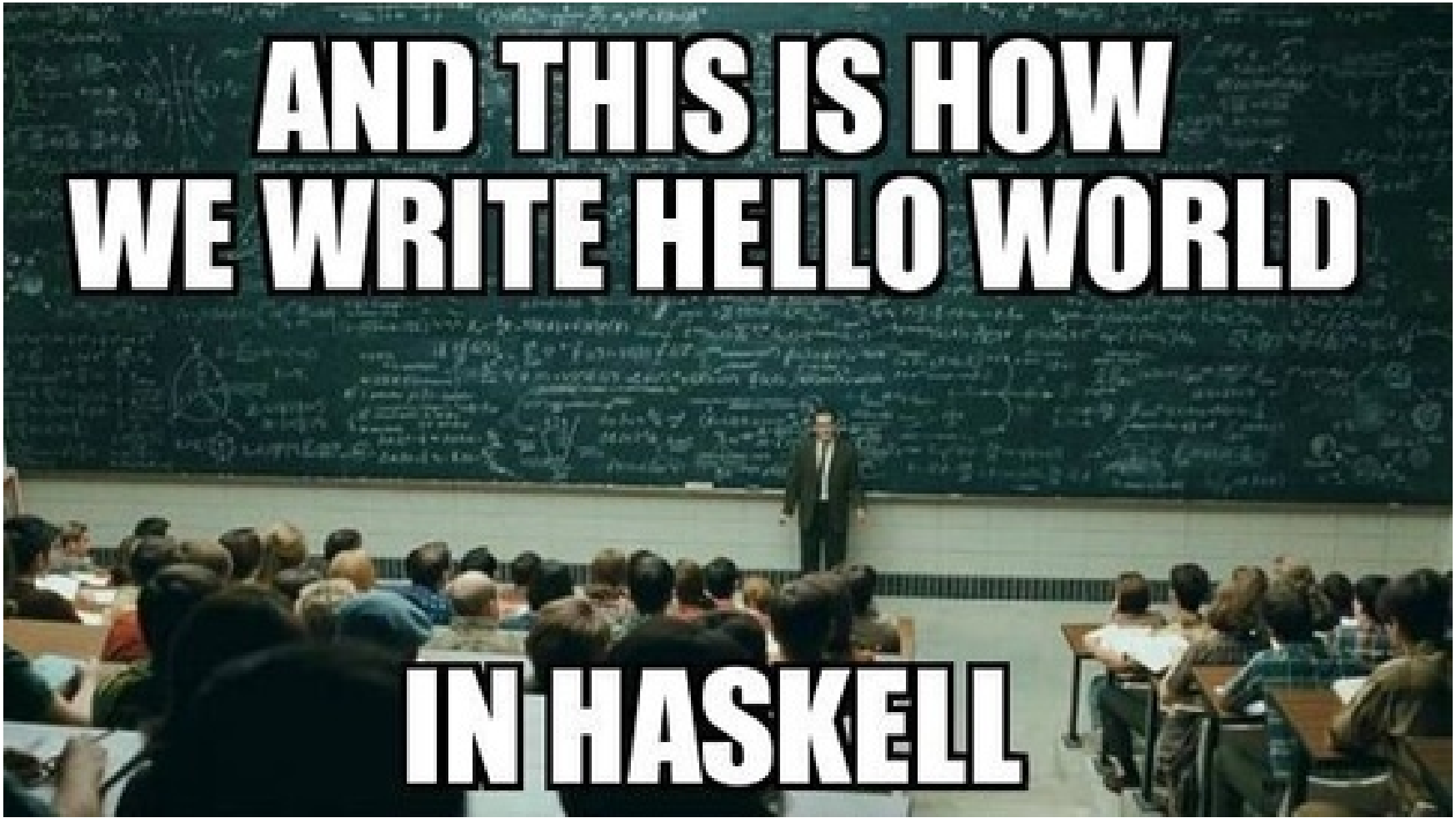
http://igm.univ-mlv.fr/~vialette/?section=teaching

Stéphane Vialette

LIGM, Université Gustave Eiffel

November 8, 2022

Ready, set, go!



Ready, set, go!

ghc's interactive mode

```
barbalala: ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/
:? for help
>>> 1 + 2
3
>>> 3 * 4
12
>>> 5 - 6
-1
>>> 7 / 8
0.875
```

Ready, set, go!

ghc's interactive mode

```
>>> True && True
True
>>> False || True
True
>>> not False
True
>>> not (True && True)
False
>>> :type True
True :: Bool
>>> :type False
False :: Bool
```

Ready, set, go!

ghc's interactive mode

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

Ready, set, go!

ghc's interactive mode

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

Functions

ghc's interactive mode

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

```
>>> :type (*)
(*) :: Num a => a -> a -> a
```

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.

Calling functions

Suppose that `f` is a function that takes two integers as arguments and returns some integer (in haskell `f :: Int -> Int -> Int`).

Call `f` **without parentheses**:

```
f 3 4
```

Do not write:

```
f(3, 4)
```

Functions

ghc's interactive mode

```
>>> succ 1
2
>>> :type succ
succ :: Enum a => a -> a
>>> :type succ 1
succ 1 :: (Enum a, Num a) => a
>>> succ 1.2
2.2
>>> :type succ 1.2
succ 1.2 :: (Enum a, Fractional a) => a
>>> succ False
True
>>> :type succ False
succ False :: Bool
>>> succ True
*** Exception: ghci.Enum.Bool.succ: bad argument
```

Functions

ghc's interactive mode

```
>>> min 2 1
1
>>> :type min
min :: Ord a => a -> a -> a
>>> :type min 2 1
min 2 1 :: (Num a, Ord a) => a
>>> max 2 1
2
>>> :type max
max :: Ord a => a -> a -> a
>>> :type max 2 1
max 2 1 :: (Num a, Ord a) => a
```

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.

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

First functions

doubleX

```
>>> doubleX x = x + x
>>> :type doubleX
doubleX :: Num a => a -> a
>>> doubleX 3
6
>>> doubleX 3.0
6.0
>>> doubleX 9/3
6.0
>>> :type (/)
(/) :: Fractional a => a -> a -> a
```

First functions

doubleXY

```
>>> doubleXY x y = 2*x + 2*y
>>> :type doubleXY
doubleXY :: Num a => a -> a -> a
>>> doubleXY 4 9
26
>>> doubleXY 2.3 34.2
73.0
>>> doubleXY 28 88 + doubleX 123
478
>>> doubleXY doubleX 2 doubleX 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
>>> doubleXY (doubleX 2) (doubleX 10)
48
```

First functions

```
doubleX x = x + x
doubleXY x y = doubleX x + doubleX 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

First functions

`doubleSmallNumber`

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.

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
```

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!"
```

First functions

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

doubleSmallNumber' x
  | x > 100   = x
  | otherwise = 2*x
```

Functional thinking

An introduction to lists

```
>>> numbers = [4,8,15,16,23,42]
>>> numbers
[4,8,15,16,23,42]
>>> :type numbers
numbers :: Num t => [t]


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

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:

```
>>> 'A' : " SMALL CAT"
"A SMALL CAT"
>>> 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.

An introduction to lists

```
(:) :: a -> [a] -> [a]

(++) :: [a] -> [a] -> [a]
```

An introduction to lists

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

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

You will rarely, if ever, use `!!`. If you happen to use `!!`, you should think again.

An introduction to lists

```
(:) :: a -> [a] -> [a]

(++) :: [a] -> [a] -> [a]

(!!) :: [a] -> Int -> a
```

An introduction to lists

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

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

An introduction to lists

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

An introduction to lists

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> l1 : l2
<interactive>:36:1:
    Non type-variable argument in the constraint: Num [t]
...
>>> l1 ++ l2
[1,2,3,4,5,6]
```

An introduction to lists

```
>>> l = 1 : 3 : l
>>> l !! 0
1
>>> l !! 1
3
>>> l !! 2
1
>>> l !! 3
3
>>> l -- don't do this
[1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,...
```

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.

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

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.

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

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.

```
>>> :type tail
tail :: [a] -> [a]
>>> tail "Hello"
"ello"
>>> tail [1,2,3,4,5]
[2,3,4,5]
>>> tail [1]
[]
>>> tail ['a']
""
>>> tail []
*** Exception: ghci.tail: empty list
```

An introduction to lists

Some basic functions that operate on lists

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

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

An introduction to lists

Some basic functions that operate on lists

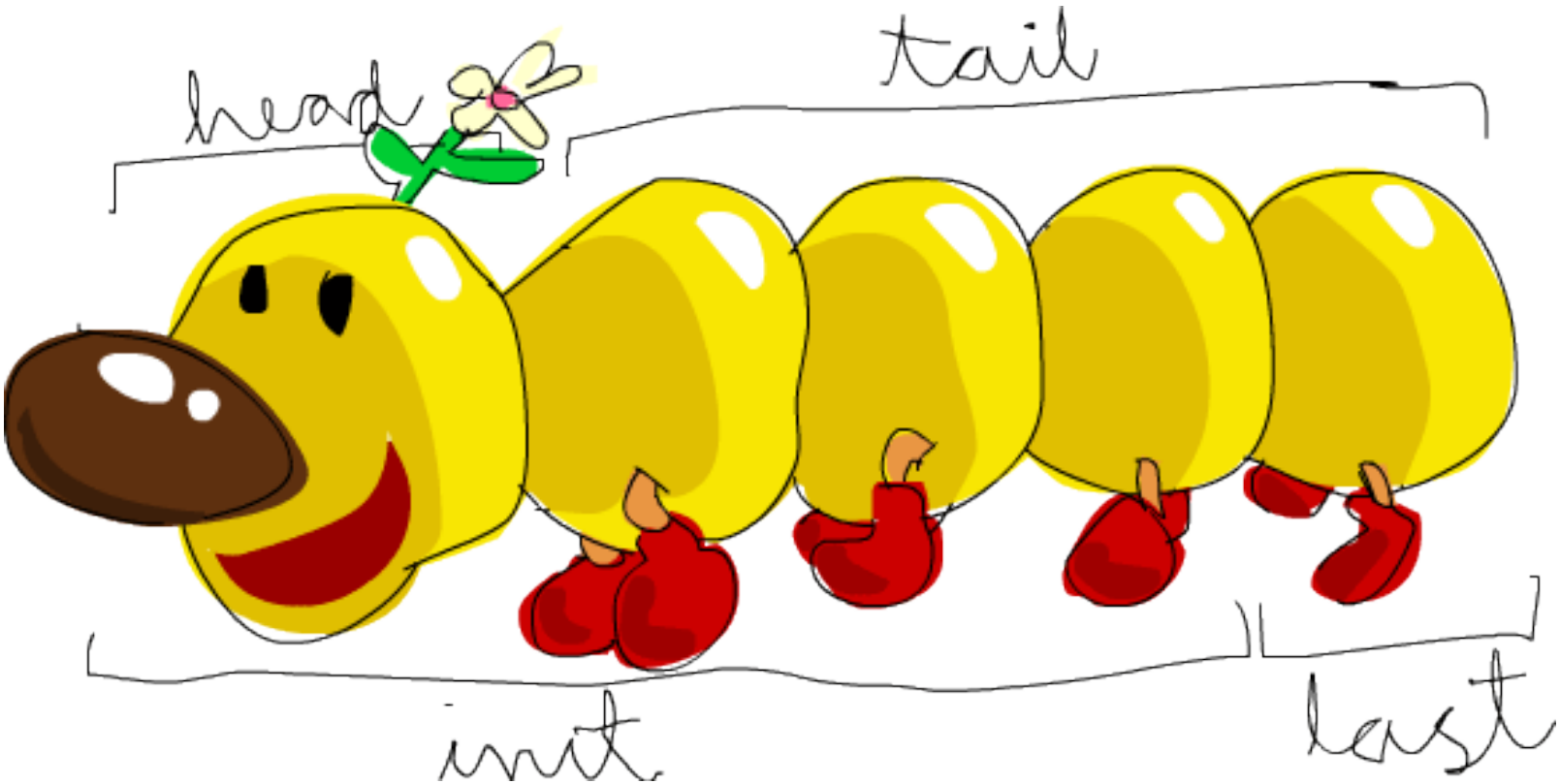
`init` takes a list and returns everything except its last element.

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

An introduction to lists

```
(:) :: a -> [a] -> [a]

(++) :: [a] -> [a] -> [a]

head :: [a] -> a

tail :: [a] -> [a]

last :: [a] -> a

init :: [a] -> [a]
```

An introduction to lists



An introduction to lists

Some basic functions that operate on lists

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

```
>>> :type length
length :: Foldable t => t a -> Int
>>> length "Hello"
5
>>> length [1,2,3,4,5]
5
>>> length []
0
```

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`)

```
>>> :type null
null :: Foldable t => t a -> Bool
>>> null "hello"
False
>>> null [1,2,3,4,5]
False
>>> null []
True
```

An introduction to lists

Some basic functions that operate on lists

`reverse` reverses a list.

```
>>> :type reverse
reverse :: [a] -> [a]
>>> reverse "hello"
"olleh"
>>> reverse [1,2,3,4,5]
[5,4,3,2,1]
>>> reverse []
[]
>>> "abcd" == reverse (reverse "abcd")
True
```

An introduction to lists

Some basic functions that operate on lists

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

```
>>> :type take
take :: Int -> [a] -> [a]
>>> take 0 [1,2]
[]
>>> take 1 [1,2]
[1]
>>> take 2 [1,2]
[1,2]
>>> take 3 [1,2]
[1,2]
>>> take 0 []
[]
>>> take 1 []
[]
```

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.

```
>>> :type drop
drop :: Int -> [a] -> [a]
>>> drop 0 [1,2,3]
[1,2,3]
>>> drop 1 [1,2,3]
[2,3]
>>> drop 2 [1,2,3]
[3]
>>> drop 3 [1,2,3]
[]
>>> drop 4 [1,2,3]
[]
```

An introduction to lists

Some basic functions that operate on lists

`maximum :: (Foldable t, Ord a) => t a -> a` takes a list of stuff that can be put in some kind of order and returns the biggest element. `minimum :: (Foldable t, Ord a) => t a -> a` returns the smallest.

```
>>> :type minimum
minimum :: (Ord a, Foldable t) => t a -> a
>>> minimum [3,4,2,5,1,6,9,8,7]
1
>>> :type maximum
maximum :: (Ord a, Foldable t) => t a -> a
>>> maximum [3,4,2,5,1,6,9,8,7]
9
>>> minimum []
*** Exception: ghci.minimum: empty list
>>> maximum []
*** Exception: ghci.maximum: empty list
```

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.

```
>>> :type sum
sum :: (Num a, Foldable t) => t a -> a
>>> sum []
0
>>> sum [1,2,3,4,5]
15
>>> :type product
product :: (Num a, Foldable t) => t a -> a
>>> product []
1
>>> product [1,2,3,4,5]
120
>>> fact n = product [1..n]
>>> fact 5
120
```

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.

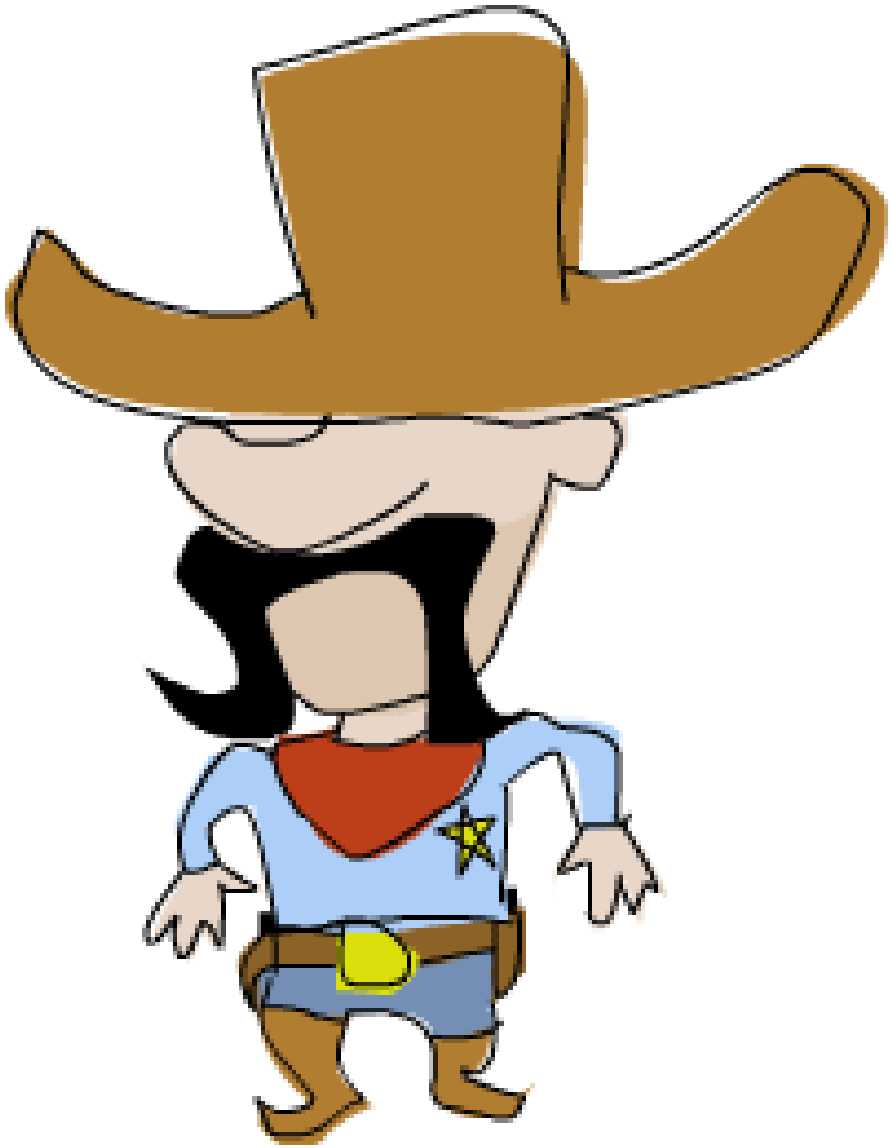
It's usually called as an infix function because it's easier to read that way.

```
>>> :type elem
elem :: (Eq a, Foldable t) => a -> t a -> Bool
>>> 3 `elem` [2,1,3,5,4]
True
>>> elem 3 [2,1,3,5,4]
True
>>> 6 `elem` [2,1,3,5,4]
False
>>> elem 6 [2,1,3,5,4]
False
```

An introduction to lists

```
length :: [a] -> Int

null :: [a] -> Bool

reverse :: [a] -> [a]

take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
sum :: Int -> [a] -> [a]
product :: Int -> [a] -> [a]

minimum :: [a] -> a
maximum :: [a] -> a

elem :: a -> [a] -> Bool
```

Texas ranges



Texas ranges

```
>>> [1,2,3,4,5,6,7,8,9,10]
[1,2,3,4,5,6,7,8,9,10]
>>> [1..10]
[1,2,3,4,5,6,7,8,9,10]
>>> [10..1]
[]
>>> [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]
>>> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
>>> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Texas ranges

Ranges are cool because you can also specify a step.

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

Texas ranges

Do not use floating point numbers in ranges!

```
>>> [0.1, 0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
>>> [1, 0.8..0]
[1.0,0.8,0.6000000000000001,0.4000000000000013,
 0.2000000000000018,2.220446049250313e-16]
```

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.

```
>>> l = [1..]
>>> :type l
l :: (Num t, Enum t) => [t]
>>> take 10 l
[1,2,3,4,5,6,7,8,9,10]
>>> 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

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

I'm a list comprehension

```
>>> [x | x <- [1..10]]
[1,2,3,4,5,6,7,8,9,10]
>>> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
>>> [x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]

>>> [(x, y) | x <- ['a'..'c'], y <- [1,2]]
[('a',1),('a',2),('b',1),('b',2),('c',1),('c',2)]
>>> [(y, x) | x <- ['a'..'c'], y <- [1,2]]
[(1,'a'),(2,'a'),(1,'b'),(2,'b'),(1,'c'),(2,'c')]

>>> [x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

I'm a list comprehension

```
>>> [x+y | x <- [1,2,3], y <- [10,20,30]]
[11,21,31,12,22,32,13,23,33]
>>> [x+y | x <- [1,2,3], y <- [10,20,30], x+y > 20]
[21,31,22,32,23,33]
>>> [x+y | x <- [1,2,3], y <- [10,20,30], x+y > 20, x+y < 32]
[21,31,22,23]

>>> [(x, y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
>>> [(y, x) | x <- [1..3], y <- [x..3]]
[(1,1),(2,1),(3,1),(2,2),(3,2),(3,3)]

>>> take 10 [x*x | x <- [1..]]
[1,4,9,16,25,36,49,64,81,100]
>>> take 10 (drop 10 [x*x | x <- [1..]])
[121,144,169,196,225,256,289,324,361,400]
```

I'm a list comprehension

```
>>> length' xs = sum [1 | _ <- xs]
>>> length' []
0
>>> length' [1..100]
100

>>> removeNonUppercase cs = [c | c <- cs, c `elem` ['A'..'Z']]
>>> removeNonUppercase "ABC"
"ABC"
>>> removeNonUppercase "def"
""
>>> removeNonUppercase "I don't LIKE FROGS."
"ILIKEFROGS"
```

I'm a list comprehension

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

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

>>> map (filter even) xss
[[2],[4],[6,8,10]]
>>> map (filter even) (filter (\xs -> length xs > 2) xss)
[[2],[6,8,10]]
```

I'm a list comprehension

```
>>> fibs = 0 : 1 : [a+b | (a, b) <- zip fibs (tail fibs)]
>>> :type fibs
fibs :: Num a => [a]
>>> take 0 fibs
[]
>>> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

I'm a list comprehension

```
>>> fibs' = 0 : 1 : zipWith (+) fibs' (tail fibs')
>>> :type fibs'
fibs :: Num a => [a]
>>> take 0 fibs'
[]
>>> take 10 fibs'
[0,1,1,2,3,5,8,13,21,34]
```

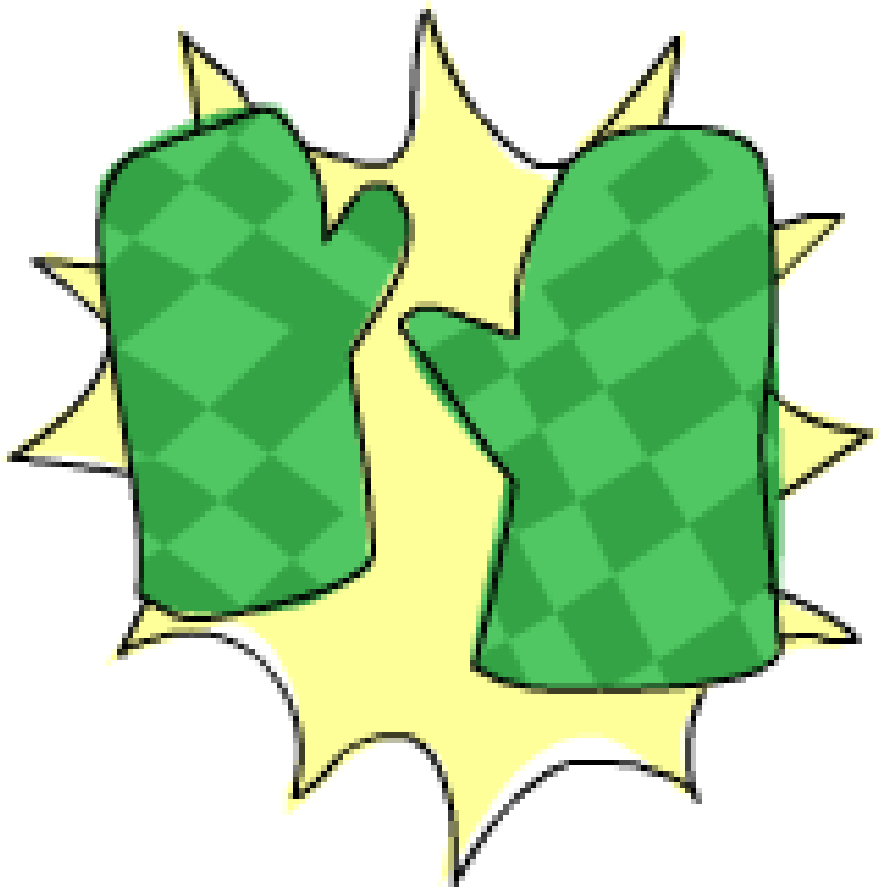
I'm a list comprehension

```
>>> binaries = [b : bs | bs <- "" : binaries, b <- ['0','1']]
>>> take 2 binaries
["0","1"]
>>> take 6 binaries
["0","1","00","10","01","11"]
>>> take 14 binaries
["0","1","00","10","01","11","000","100","010","110","001",
 "101","011","111"]
>>> take 6 (filter (\bs -> last bs == '0') binaries)
["0","00","10","000","100","010"]
```

I'm a list comprehension

```
>>> binaries = [b : bs | bs <- "" : binaries, b <- ['0','1']]
>>> binaries' = [b : bs | b <- ['0', '1'], bs <- "" : binaries']
>>> take 6 binaries'
["0","00","000","0000","00000","000000"]
>>> head (drop 10 binaries')
"00000000000"
```

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

```
>>> :type (1,2)
(1,2) :: (Num t1, Num t) => (t, t1)
>>> :type (1,2,3)
(1,2,3) :: (Num t2, Num t1, Num t) => (t, t1, t2)
>>> [(1,2),(8,11),(4,5)]
[(1,2),(8,11),(4,5)]
>>> [(1,2),(8,11,5),(4,5)]
<interactive>:68:8:
    Couldn't match expected type ...
```

Tuples

```
>>> :type ('a', 1, "hello")
('a', 1, "hello") :: Num t => (Char, t, [Char])
>>> (1, 2) < (3, 4)
True
>>> (1, 2) < (0, 1)
False
>>> (1, 2, 3) < (1, 2)
<interactive>:74:13:
    Couldn't match expected type ...
```

Tuples

```
>>>  :type ((1, 'a'), ("a", 1.2))
((1, 'a'), ("a", 1.2))
  :: (Fractional b, Num a) => ((a, Char), ([Char], b))
>>> :type (1, (2, (3, 4)))
(1, (2, (3, 4)))
  :: (Num a1, Num a2, Num a3, Num b) => (a1, (a2, (a3, b)))
>>> :type ([1,2], ['a', 'b'])
([1,2], ['a', 'b']) :: Num a => ([a], [Char])

>>> :type (,)
(,) :: a -> b -> (a, b)
>>> (,) 1 2
(1,2)
>>> (,,) 1 2 3
(1,2,3)
```

Tuples

Two useful functions that operate on pairs

```
>>> fst ('a', 2)
'a'
>>> snd ('a', 2)
2
>>> fst ('a', 2, "hello")
<interactive>:80:5:
    Couldn't match expected type ...

>>> third (_, _, z) = z
>>> third (1, 'a', "bc")
"bc"
```

Tuples

Two useful functions that operate on pairs

```
>>> :type fst
fst :: (a, b) -> a
>>> :type snd
snd :: (a, b) -> b
>>> fst (('a', 'b'),('c', 'd'))
('a','b')
>>> fst (('a', 'b', 'c'),('d', 'e', 'f'))
('a','b','c')
>>> fst (('a', 'b', 'c'),('d', 'e', 'f', 'g'))
('a','b','c')
```

Tuples

`zip`

Tuples

zip

```
>>> :type zip
zip :: [a] -> [b] -> [(a, b)]

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

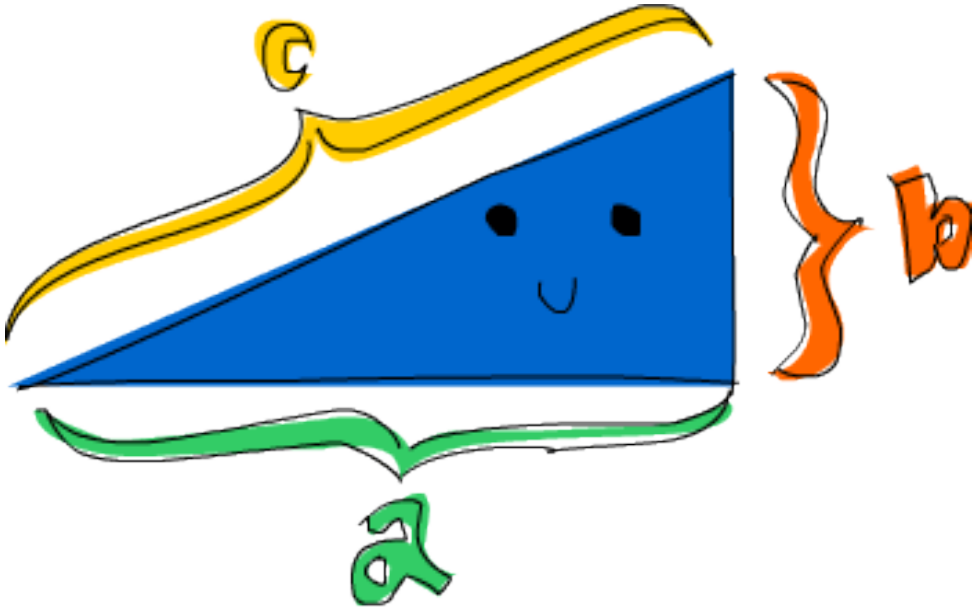
Tuples

zipWith

```
>>> :type zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
>>> zipWith (\ x y -> x + y) [0..5] [1,1..]
[1,2,3,4,5,6]
>>> zipWith (+) [0..5] [1,1..]
[1,2,3,4,5,6]

>>> rfibs = 0 : 1 : zipWith (+) rfibs (tail rfibs)
>>> take 10 rfibs
[0,1,1,2,3,5,8,13,21,34]
```

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?



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

Right triangle

```
>>> ts   = [(a,b,c) | a <- [1..10], b <- [1..10], c <- [1..10]]
>>> length ts
1000
>>> take 5 ts
[(1,1,1),(1,1,2),(1,1,3),(1,1,4),(1,1,5)]
>>> rts  = [(a,b,c) | (a,b,c) <- ts, a^2 + b^2 == c^2]
>>> rts
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
>>> rts' = [(a,b,c) | (a,b,c) <- ts, a^2 + b^2 == c^2
                    , a+b+c == 24]
>>> rts'
[(6,8,10),(8,6,10)]
```

Done!


ABSTRACTION
NO SIDE EFFECTS
LAZY
STRONG-TYPED
MINIMISES BUGS
ELEGANCE