Recollecting Haskell, Part I

(Based on Chapters 1 and 2 of LYAH*)

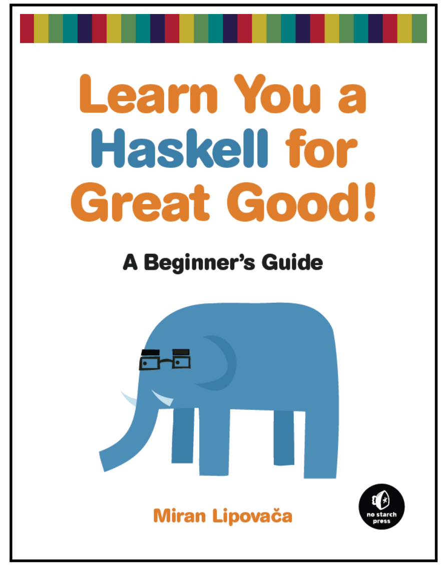
CIS 351/Spring 2016

Programming Languages

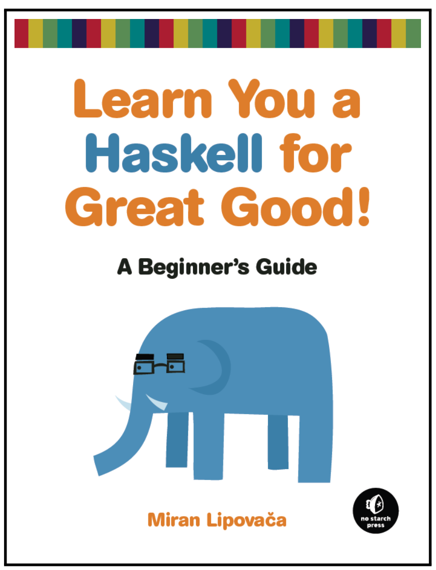
January 19, 2016

*LYAH = *Learn You a Haskell for Great Good*

Two (too?) big assumptions

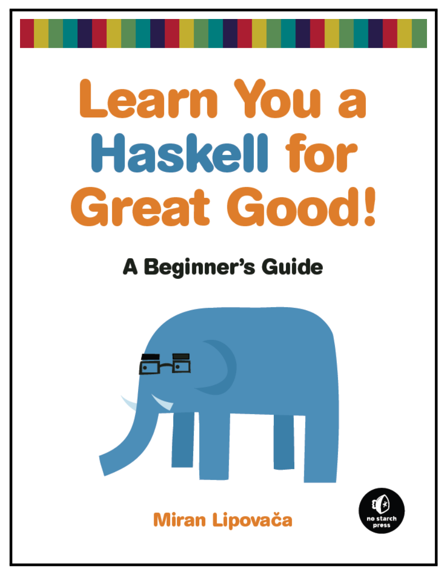


Two (too?) big assumptions



1. You can read LYAH

Two (too?) big assumptions



1. You can read LYAH
2. You *will* read LYAH.

The Blurb from `wiki.haskell.org`

Haskell is an advanced purely-functional programming language.

...it allows rapid development of robust, concise, correct software.

With strong support for

- *integration with other languages,*
- *built-in concurrency and parallelism,*
- *debuggers,*
- *profilers,*
- *rich libraries and*
- *an active community,*

Haskell makes it easier to produce flexible, maintainable, high-quality software.

So *why* do we care about Haskell in this course?

- Haskell is great for prototyping.
- Forces you to think compositionally.
- Semi-automated testing: QuickCheck
- Haskell can give you executable specifications.
- Good for "model building"
  e.g., direct implementations of operational semantics

So *why* do we care about Haskell in this course?

- Haskell is great for prototyping.
- Forces you to think compositionally.
- Semi-automated testing: QuickCheck
- Haskell can give you executable specifications.
- Good for "model building"
  e.g., direct implementations of operational semantics

...and beyond this course

- Many of the new systems/applications languages
  (e.g., Swift and Rust) steal lots of ideas from Haskell and ML.
- These ideas are a lot clearer in Haskell and ML
  than the munged versions in Swift, Rust, etc.,

Set up

- Go visit `https://www.haskell.org/downloads#platform`.
- Download the appropriate version of the current Haskell Platform and install it.

!!! Do the above even if you have an old copy of the Haskell Platform from a previous year.
You want the latest version of the GHC compiler and libraries.

!!! Use a reasonable editor.
Using Notepad or Word is a waste of your time.
See `http://www.haskell.org/haskellwiki/Editors`.
Emacs is my weapon of choice.

A sample session: ghci as a calculator

```
[Post:~] jimroyer% ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> 2+3
5
Prelude> 2*3
6
Prelude> 2-3
-1
Prelude> 2/3
0.6666666666666666
Prelude> :q
Leaving GHCi.
[Post:~] jimroyer%
```


Fussy bits

- ✘ `5 * -3`
- ✔ `5 * (-3)`
- ✘ `5 * 10 - 49` $\not\equiv$ `5 * (10 - 49)`
- ✔ `5 * 10 - 49` $\equiv$ `(5 * 10) - 49`
- ✘ `5 * True`

```
Prelude> 5 + True

<interactive>:2:3:
    No instance for (Num Bool) arising from a use of ‘+’
    In the expression: 5 + True
    In an equation for ‘it’: it = 5 + True
```

(What does all this mean?)

Using functions

```
Prelude> succ 4
5
Prelude> succ 4 * 10
50
Prelude> succ (4 * 10)
41
Prelude> max 5 3
5
Prelude> 1 + max 5 3
6
Prelude> max 5 3 + 1
6
Prelude> max 5 (3 + 1)
5
Prelude> 10 `max` 23
23
Prelude> (+) 3 5
8
```


Using functions

```
Prelude> succ 4
5
Prelude> succ 4 * 10
50
Prelude> succ (4 * 10)
41
Prelude> max 5 3
5
Prelude> 1 + max 5 3
6
Prelude> max 5 3 + 1
6
Prelude> max 5 (3 + 1)
5
Prelude> 10 `max` 23
23
Prelude> (+) 3 5
8
```


baby.hs

```
doubleMe x = x + x
```

Using functions

```
Prelude> succ 4
5
Prelude> succ 4 * 10
50
Prelude> succ (4 * 10)
41
Prelude> max 5 3
5
Prelude> 1 + max 5 3
6
Prelude> max 5 3 + 1
6
Prelude> max 5 (3 + 1)
5
Prelude> 10 `max` 23
23
Prelude> (+) 3 5
8
```

baby.hs

```
doubleMe x = x + x
```

```
Prelude> :load baby
[1 of 1] Compiling Main
                 ( baby.hs, interpreted )
Ok, modules loaded: Main.
*Main> doubleMe 5
10
*Main> doubleMe 7.7
15.4
```


Defining and using functions, continued

baby.hs

```
doubleMe x            = x + x

doubleUs x y          = 2*x+2*y
doubleUs' x y         = doubleMe x + doubleMe y

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

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


Lists

- A *list*: a sequence of things *of the same type*.
  - ✔ `[2,3,5,7,11,13,17,19]` :: `[Int]`
  - ✔ `[True,False,False,True]` :: `[Bool]`
  - ✔ `['b','o','b','c','a','t']` ≡ `"bobcat"` :: `[Char]`
  - ✔ `[]` :: `[a]`
  - ✔ `[[],[1],[2,3],[4,5,6]]` :: `[[Int]]`
  - ✘ `[[1],[[2],[3]]]` *fuss*
  - ✘ `[2,True,"foo"]` *fuss*
- If you want to bundle together things of different types, use tuples *(e.g.,* `(2,True,"foo")` *... explained later)*.

Lists

- A *list*: a sequence of things *of the same type*.
  - ✔ `[2,3,5,7,11,13,17,19]` :: `[Int]`
  - ✔ `[True,False,False,True]` :: `[Bool]`
  - ✔ `['b','o','b','c','a','t']` ≡ `"bobcat"` :: `[Char]`
  - ✔ `[]` :: `[a]`
  - ✔ `[[],[1],[2,3],[4,5,6]]` :: `[[Int]]`
  - ✘ `[[1],[[2],[3]]]` *fuss*
  - ✘ `[2,True,"foo"]` *fuss*
- If you want to bundle together things of different types, use tuples *(e.g.,* `(2,True,"foo")` *... explained later)*.

Notation

$expression_1 \leadsto expression_2$ means $expression_1$ evaluates to $expression_2$

E.g.: 2+3 $\leadsto$ 5

Lists: Building them up

Write them down: [$item_1$, $item_2$, ..., $item_n$]

- [2,3,5,7,11,13,17,19], [], *etc.*

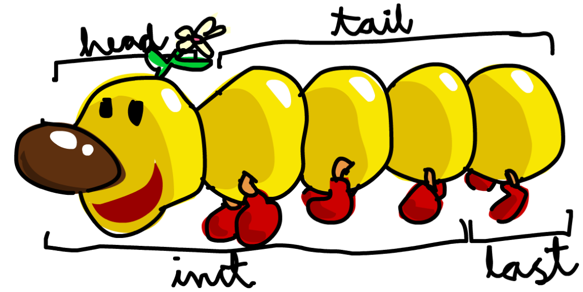
Concatenation: $list_1$++ $list_2$

- [1,2,3,4]++[10,20,30] ⇝ [1,2,3,4,10,20,30]
- "salt"++"potato" ⇝ "saltpotato"
- []++[1,2,3] ⇝ [1,2,3]   [1,2,3]++[] ⇝ [1,2,3]
- 1++[2,3] ⇝ ERROR   [1,2]++3 ⇝ ERROR

Cons: *item* : *list*

- 1:[2,3] ⇝ [1,2,3]   [1,2]:3 ⇝ ERROR
- [1,2,3] is *syntactic sugar* for 1:2:3:[] ≡ 1:(2:(3:[]))
- You can tell (:) is important because they gave it such a short name.

Lists: Tearing them down



- head [1,2,3] ⇝ 1
- tail [1,2,3] ⇝ [2,3]
- head [] ⇝ ERROR
- tail [] ⇝ ERROR
- last [1,2,3] ⇝ 3
- init [1,2,3] ⇝ [1,2]
- last [] ⇝ ERROR
- init [] ⇝ ERROR

Lists: More operations

- length :: [a] -> Int
- (!!) :: [a] -> Int -> a
- null :: [a] -> Bool
- reverse :: [a] -> [a]
- drop, take :: Int -> [a] -> [a]
- sum, product :: (Num a) => [a] -> a
- minumum, maximum :: (Ord a) => [a] -> a
- elem, notElem :: (Eq a) => a -> [a] -> Bool

Lists: More operations

- length :: [a] -> Int
- (!!) :: [a] -> Int -> a
- null :: [a] -> Bool
- reverse :: [a] -> [a]
- drop, take :: Int -> [a] -> [a]
- sum, product :: (Num a) => [a] -> a
- minumum, maximum :: (Ord a) => [a] -> a
- elem, notElem :: (Eq a) => a -> [a] -> Bool

You can look up what these do on:

- Hoogle: https://www.haskell.org/hoogle/
- Hayoo: http://hayoo.fh-wedel.de

Ranges

[m..n] $\rightsquigarrow$ [m,m+1,m+2,...,n]

- `[1..5]` $\rightsquigarrow$ `[1,2,3,4,5]`
- `[5..1]` $\rightsquigarrow$ `[]`
- `['a'..'k']` $\rightsquigarrow$ `"abcdefghijk"`
- `[1..]` $\rightsquigarrow$ `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,`

[m,p..n] $\rightsquigarrow$ [m, m+(p-m), m+2(m-p), ... ,n*]

- `[3,5..10]` $\rightsquigarrow$ `[3,5,7,9]`
- `[5,4..1]` $\rightsquigarrow$ `[5,4,3,2,1]`
- `[9,7..2]` $\rightsquigarrow$ `[9,7,5,3]`

*Or the "closest" number before n that is in the sequence.

List comprehensions

Set comprehensions in math (CIS 275 review)

- $\{ x \mid x \in N, x = x^2 \} = \{ 0, 1 \}$
- $\{ x \mid x \in N, x > 0 \}$ = the positive integers
- $\{ x^2 \mid x \in N \}$ = squares
- $\{ (x,y) \mid x \in N, y \in N, x \le y \}$

Suppose we have `lst = [5,10,13,4,10]`

- `[2*n+1 | n <- lst]` $\rightsquigarrow$ `[11,21,27,9,21]`
- `[even n | n <- lst]` $\rightsquigarrow$ `[False,True,False,True,True]`
- $[\ \underbrace{\texttt{2*n+1}}_{\textit{transform}}\ |\ \underbrace{\texttt{n <- lst}}_{\textit{generator}},\ \underbrace{\texttt{even n}}_{\textit{filter}},\ \underbrace{\texttt{n>5}}_{\textit{filter}}\ ]$ $\rightsquigarrow$ `[21,21]`

Example: Squaring every element of a list

squares.hs

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```

```
squares [1,2,3]
```


Example: Squaring every element of a list

squares.hs

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```


```
   squares [1,2,3]
=        { xs = [1,2,3] }
   [ x*x | x <- [1,2,3] ]
```

Example: Squaring every element of a list

squares.hs

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```

```
    squares [1,2,3]
=       { xs = [1,2,3] }
    [ x*x | x <- [1,2,3] ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1*1 ] ++ [ 2*2 ] ++ [ 3*3 ]
```


Example: Squaring every element of a list

squares.hs

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```



```
    squares [1,2,3]
=       { xs = [1,2,3] }
    [ x*x | x <- [1,2,3] ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1*1 ] ++ [ 2*2 ] ++ [ 3*3 ]
=
    [1]++[4]++[9]
```


Example: Squaring every element of a list

squares.hs

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]
```



```
    squares [1,2,3]
=       { xs = [1,2,3] }
    [ x*x | x <- [1,2,3] ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1*1 ] ++ [ 2*2 ] ++ [ 3*3 ]
=
    [1]++[4]++[9]
=
    [1,4,9]
```

Example lifted from Phil Wadler.

Example: Odd elements of a list

odds.hs

```
odds :: [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```

```
    odds [1,2,3]
```

Example: Odd elements of a list

odds.hs

```
odds :: [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```

```
    odds [1,2,3]
=       { xs = [1,2,3] }
    [ x | x <- [1,2,3], odd x ]
```


Example: Odd elements of a list

odds.hs

```
odds :: [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```


```
    odds [1,2,3]
=       { xs = [1,2,3] }
    [ x | x <- [1,2,3], odd x ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1 | odd 1 ] ++ [ 2 | odd 2 ] ++ [ 3 | odd 3 ]
```


Example: Odd elements of a list

odds.hs

```
odds :: [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```


```
    odds [1,2,3]
=       { xs = [1,2,3] }
    [ x | x <- [1,2,3], odd x ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1 | odd 1 ] ++ [ 2 | odd 2 ] ++ [ 3 | odd 3 ]
=
    [ 1 | True ] ++ [ 2 | False ] ++ [ 3 | True ]
```


Example: Odd elements of a list

odds.hs

```
odds :: [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```


```
    odds [1,2,3]
=       { xs = [1,2,3] }
    [ x | x <- [1,2,3], odd x ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1 | odd 1 ] ++ [ 2 | odd 2 ] ++ [ 3 | odd 3 ]
=
    [ 1 | True ] ++ [ 2 | False ] ++ [ 3 | True ]
=
    [1]++[]++[3]
```

Example: Odd elements of a list

odds.hs

```
odds :: [Integer] -> [Integer]
odds xs = [ x | x <- xs, odd x ]
```

```
    odds [1,2,3]
=       { xs = [1,2,3] }
    [ x | x <- [1,2,3], odd x ]
=       { x=1 }, { x=2 }, { x=3 }
    [ 1 | odd 1 ] ++ [ 2 | odd 2 ] ++ [ 3 | odd 3 ]
=
    [ 1 | True ] ++ [ 2 | False ] ++ [ 3 | True ]
=
    [1]++[]++[3]
=
    [1,3]
```

Example lifted from Phil Wadler.

Example: Sum of the squares of the odd elements, 1

sumSqOdds.hs

```
squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]

odds :: [Integer] -> [Integer]
odds xs    = [ x | x <- xs, odd x]

f :: [Integer] -> Integer
f xs  = sum (squares (odds xs))

f' :: [Integer] -> Integer
f' xs = sum [ x*x | x <- xs, odd x ]
```

Another example lifted from Phil Wadler.

Example: Sum of the squares of the odd elements, 2

```
f xs  = sum (squares (odds xs))
```

```
    f [1,2,3]
```


Example: Sum of the squares of the odd elements, 2


```
f xs  = sum (squares (odds xs))
```


```
    f [1,2,3]
=       { xs = [1,2,3] }
    sum (squares (odds [1,2,3]))
```

Example: Sum of the squares of the odd elements, 2

```
f xs  = sum (squares (odds xs))
```

```
    f [1,2,3]
=       { xs = [1,2,3] }
    sum (squares (odds [1,2,3]))
=
    sum (squares [1,3])
```


Example: Sum of the squares of the odd elements, 2

```
f xs  = sum (squares (odds xs))
```

```
    f [1,2,3]
=       { xs = [1,2,3] }
    sum (squares (odds [1,2,3]))
=
    sum (squares [1,3])
=
    sum [1,9]
```


Example: Sum of the squares of the odd elements, 2

```
f xs  = sum (squares (odds xs))
```

```
    f [1,2,3]
=       { xs = [1,2,3] }
    sum (squares (odds [1,2,3]))
=
    sum (squares [1,3])
=
    sum [1,9]
=
    10
```


Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```

```
    f' [1,2,3]
```

Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```

```
    f' [1,2,3]
=       { xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x]
```


Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```


```
    f' [1,2,3]
=       { xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x]
=       { x=1 }, { x=2 }, { x=3 }
    sum ([ 1*1 | odd 1 ] ++ [ 2*2 | odd 2 ] ++ [ 3*3 | odd 3 ])
```


Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```


```
    f' [1,2,3]
=       { xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x]
=       { x=1 }, { x=2 }, { x=3 }
    sum ([ 1*1 | odd 1 ] ++ [ 2*2 | odd 2 ] ++ [ 3*3 | odd 3 ])
=
    sum ([ 1 | True ] ++ [ 4 | False ] ++ [ 9 | True])
```


Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```


```
    f' [1,2,3]
=       { xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x]
=       { x=1 }, { x=2 }, { x=3 }
    sum ([ 1*1 | odd 1 ] ++ [ 2*2 | odd 2 ] ++ [ 3*3 | odd 3 ])
=
    sum ([ 1 | True ] ++ [ 4 | False ] ++ [ 9 | True])
=
    sum ([ 1 ] ++ [] ++ [ 9 ])
```

Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```

```
    f' [1,2,3]
=       { xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x]
=       { x=1 }, { x=2 }, { x=3 }
    sum ([ 1*1 | odd 1 ] ++ [ 2*2 | odd 2 ] ++ [ 3*3 | odd 3 ])
=
    sum ([ 1 | True ] ++ [ 4 | False ] ++ [ 9 | True])
=
    sum ([ 1 ] ++ [] ++ [ 9 ])
=
    sum [1,9]
```


Example: Sum of the squares of the odd elements, 3

```
f' xs  = sum [ x*x | x <- xs, odd x]
```


```
    f' [1,2,3]
=       { xs = [1,2,3] }
    sum [ x*x | x <- [1,2,3], odd x]
=       { x=1 }, { x=2 }, { x=3 }
    sum ([ 1*1 | odd 1 ] ++ [ 2*2 | odd 2 ] ++ [ 3*3 | odd 3 ])
=
    sum ([ 1 | True ] ++ [ 4 | False ] ++ [ 9 | True])
=
    sum ([ 1 ] ++ [] ++ [ 9 ])
=
    sum [1,9]
=
    10
```


Example: Sum of the squares of the odd elements, 4

sumSqOdds.hs

```
import Test.QuickCheck

squares :: [Integer] -> [Integer]
squares xs = [ x*x | x <- xs ]

odds :: [Integer] -> [Integer]
odds xs    = [ x | x <- xs, odd x]

f :: [Integer] -> Integer
f xs  = sum (squares (odds xs))

f' :: [Integer] -> Integer
f' xs = sum [ x*x | x <- xs, odd x]

f_prop :: [Integer] -> Bool
f_prop xs = f xs == f' xs
```

```
*Main> :load sumSqOdds
[1 of 1] Compiling Main
  ( sumSqOdds.hs, interpreted )
Ok, modules loaded: Main.

*Main> quickCheck f_prop
+++ OK, passed 100 tests.
*Main>
```


Tuples

Cartesian products in math (More CIS 275 Review)

Suppose A, B, C, ... are sets.

$$A \times B = \{ (a,b) \mid a \in A, \, b \in B \}.$$
$$A \times B \times C = \{ (a,b,c) \mid a \in A, \, b \in B, \, c \in C \}.$$
$$\textit{etc.}$$

Tuple types are Haskell's version of Cartesian products

```
✔ (1,2)                 :: (Int,Int)
✔ (True,'a')            :: (Bool,Char)
✔ (3,'q',"foo")         :: (Int,Char,[Char])
✘ [(1,2),(3,4,5),(6,7)]    (Why?)
✘ [(1,2),('d',False)]      (Why?)
```

Pairs

- ✔ `fst :: (a,b) -> a`
  `fst ("muffin",99)` ⇝ `"muffin"`
- ✔ `snd :: (a,b) -> b`
  `snd ("muffin",99)` ⇝ `99`
- ✘ `fst (4.1,True,'a')` ⇝ ERROR
- ✘ `snd (4.1,True,'a')` ⇝ ERROR
- ✔ `zip :: [a] -> [b] -> [(a,b)]`
  `zip [1..5] ['a','b','c','d','e']`
  ⇝ `[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]`
  `zip [1..]  "abcde"`
  ⇝ `[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]`

Finding Pythagorean Triples

Problem

Find all possible values of (a, b, c) such that a, b, and c are integers ≤ 10 that are the edge-lengths of a right triangle with perimeter 24.

Finding Pythagorean Triples

Problem

Find all possible values of (a, b, c) such that a, b, and c are integers ≤ 10 that are the edge-lengths of a right triangle with perimeter 24.

- triples1
  ```
  = [(a,b,c) | a <- [1..10], b <- [1..10], c <- [1..10]]
  ```


Finding Pythagorean Triples

Problem

Find all possible values of (a, b, c) such that a, b, and c are integers ≤ 10 that are the edge-lengths of a right triangle with perimeter 24.

- triples1
  ```
  = [(a,b,c) | a <- [1..10], b <- [1..10], c <- [1..10]]
  ```
- triples2
  ```
  = [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10]]
  ```

Finding Pythagorean Triples

Problem

Find all possible values of (a, b, c) such that a, b, and c are integers ≤ 10 that are the edge-lengths of a right triangle with perimeter 24.

- `triples1`
```
  = [(a,b,c) | a <- [1..10], b <- [1..10], c <- [1..10]]
```
- `triples2`
```
  = [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10]]
```
- `triples3`
```
  = [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10],
               a*a + b*b == c*c]
```


Finding Pythagorean Triples

Problem

Find all possible values of (a, b, c) such that a, b, and c are integers ≤ 10 that are the edge-lengths of a right triangle with perimeter 24.

- `triples1`
```
  = [(a,b,c) | a <- [1..10], b <- [1..10], c <- [1..10]]
```
- `triples2`
```
  = [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10]]
```
- `triples3`
```
  = [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10],
               a*a + b*b == c*c]
```

- `triples4`
```
  = [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10],
               a*a + b*b == c*c, a+b+c == 24]
```