CPSC 312
Functional and Logic Programming

November 10, 2015

Assignment

I'll upload your first Haskell homework by tonight. I've extended it to include most recent lecture materials so it's a little longer than usual.

You will have about 10 days to finish it.

There may be one more Haskell assignment, or I may skip it and jump straight to the project.

From the previous lectures

Natural recursion

Tail recursion

Multiple recursion

Today: Lists!

Lists

List is a collection of an arbitrary number of values or items all of the same type.
[ ] is the empty list.

List is the fundamental data structure of functional programming languages. It can represent arrays, trees, graphs, or anything that you want.

Just like in Prolog, Haskell Lists are inspired by Lisp concepts and implementations of lists.
The only difference is that in Haskell list members must be of the same type. (why?)

Lists

```
[1,2,3]           -- ok
[7.1, 8.2, 4.5]     -- ok
[1, 2.0, 3, 4.7]    -- this works too. 1 and 3 will be
understood as Float.
['a','b','c']       -- ok, and note that a list of
characters is a string.
“abc”         -- this is the same as previous one and
also a list
['a',1,'c']      -- nope, that’s an error:

<interactive>:3:6:
    No instance for (Num Char) arising from the literal
‘1’
    In the expression: 1
    In the expression: ['a', 1, 'c']
    In an equation for ‘it’: it = ['a', 1, 'c']
```

Lists

As for indicating types of lists, it works like this:

```
[1,2,3] :: [Int]  -- the list [1,2,3] is a list of Int
types
[7.1, 8.2, 4.5] :: [Float]
['a','b','c'] :: [Char]
“abc” :: [Char]
```

The empty list is a member of every list type.

Lists can have lists as members in which case the nested members should also all be of the same type

```
[[1],[2,3]] :: [[Int]] -- A list of lists of integers
[[1],[[2]]] -- Error [Int] /= [[Int]]
[[1],[‘a’]] -- Error [Int] /= [Char]
```

Common List Operations

The "standard prelude" (the library called "Prelude.hs") contains a lot of useful list processing functions. These are just a few. More are given in your book (section 6.2)

`:` is the constructor function (equivalent of '|' in Prolog)

It takes one item and a list and returns a list consisting of the item followed by the elements of the first list

```
Prelude> ‘a’:“bc”
“abc”
```

Common List Operations

“A list is either an empty list or it has a head and a tail, which itself is a list”. This definition still holds.

There’s an empty list at the root of every list:

```
Prelude> 1:[]
[1]
Prelude> 2:[1]
[2,1]
Prelude> 2:1:[]
[2,1]
Prelude> 2:1 -- Nope. That’s an error.
```

Common List Operations

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

```
Prelude> “my ” ++ “name”
“my name”
Prelude> [1] ++ [2,3]
[1,2,3]
```

!! (member selection) :: [a] -> Int -> a

```
Prelude> [1,2,3] !! 1      -- the list index starts at
zero
2
Prelude> [1,2,3] !! 4
*** Exception: Prelude.!!: index too large
```

Common List Operations

head returns the first element of a list passed as a parameter.
(It is the equivalent of H in [H|T] in Prolog)

```
head [1,2,3]    ~> 1
head "abcde"  ~> 'a'
head []    ~> exception
```

Common List Operations

tail returns the list consisting of all but the first element of a list passed as a parameter. (It is the equivalent of T in [H|T] in Prolog.)

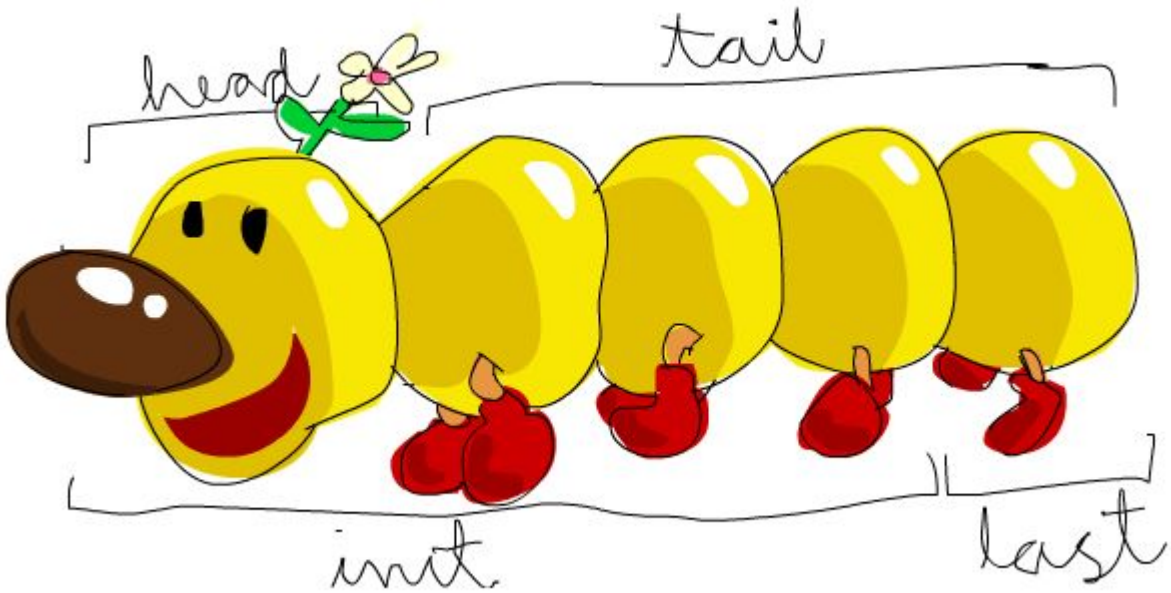
```
tail [1,2,3]  ~> [2,3]
tail "abcde"  ~> "bcde"
tail []    ~> exception
```

Common List Operations

last the last member
init all of the list's members except for the last ones

```
Prelude> last [1,2,3]
3
Prelude> init [1,2,3]
[1,2]
```

```
last []
or
init []
~> exception
```



Common List Operations

elem x l returns True if x is an element of the list l.
(It is the equivalent of member in Prolog.)

```
elem 2 [1,2,3]  ~> True
elem 'c' "abcde"~> True
```

elem is often used in infix notation:

```
2 `elem` [1,2,3] ~> True
```

Common List Operations

null l returns True if l is an empty list, False otherwise.

See p. 126 and 127 of textbook for:

```
reverse
length
take
drop
sum
product
```

and more.

(you'll be expected to know all the functions on those two pages for this course)

Common List Operations

zip list1 list2 takes a pair of lists (not necessarily of the same type) and returns a list of pairs

```
zip [1,2] [3,4]     ~> [(1,3),(2,4)]
zip [1,2] [3,4,5]   ~> [(1,3),(2,4)]
zip [1,2,3] [4,5]   ~> [(1,4),(2,5)]
zip "ab" "cd"    ~> [('a','c'),('b','d')]
```

(does this remind you of a Prolog function that you know?)

Common List Operations

zip list1 list2 takes a pair of lists (not necessarily of the same type) and returns a list of pairs

```
zip [1,2] [3,4]     ~> [(1,3),(2,4)]
zip [1,2] [3,4,5]   ~> [(1,3),(2,4)]
zip [1,2,3] [4,5]   ~> [(1,4),(2,5)]
zip "ab" "cd"    ~> [('a','c'),('b','d')]
```

Remember what data type the parentheses represent?

Tuples revisited

A tuple represents another kind of collection, in a way the opposite of lists:
fixed number of elements, but of possibly different types

So far we've only seen tuples in the form of pairs. Type declaration for pairs is like this:

```
(1,3) :: (Int,Int)
('a','c') :: (Char,Char)
```

And for a list of pairs:

```
[(1,3),(2,4)] :: [(Int,Int)]
[('a','c'),('b','d')] :: [(Char,Char)]
```

Tuples revisited

Haskell provides two built-in selector functions for pairs:

```
> fst (3,7)
3
> snd (3,7)
7
```

(but nothing stops you from writing your own)

Operations defined for tuples do not work on lists.

Operations defined for lists do not work on tuples.

Think of tuples as structs.

Tuples revisited

Tuples can have any number of elements.
Say we want to organize the info about a rental car as a tuple:

```
("Ford","Taurus",5,29.95) :: (String,String,Int,Float)
```

which means a Ford Taurus accommodates 5 passengers and rents for $29.95 per day

and a list of cars would have this type

```
[(String,String,Int,Float)]
```

Tuples revisited

We can give tuple types names like this:

type RentalCar = (String,String,Int,Float)

and the type of a list of rental cars can be named like this

type AvailableCars = [RentalCar]

the keyword **type** introduces a synonym for an existing type. (It does not define a new type)

Exercise

Use your recursion expertise (and other built-in functions: ':', head, tail,...) to implement these three functions.

```
reverse [1,2,3] ~> [3,2,1]
reverse "abcde" ~> "edcba"

length [1,2,3]  ~> 3
length []     ~> 0
length "abc"    ~> 3

elem 2 [1,2,3]  ~> True
elem 'c' "abcde"~> True
```

Type Declaration revisited

Here's an implementation of length

```
mylength l
    | null l      = 0
    | otherwise         = 1 + mylength (tail l)
```

What's the type of this function?

Type Declaration revisited

We could write:

```
mylenInt :: [Int] -> Int
mylenInt inlist
   | null inlist     = 0
   | otherwise       = 1 + mylenInt (tail inlist)

mylenStr :: String -> Int
mylenStr inlist
   | null inlist     = 0
   | otherwise       = 1 + mylenStr (tail inlist)
```

and so on.

Type Declaration revisited

We could write:

```
mylenInt :: [Int] -> Int
mylenInt inlist
   | null inlist     = 0
   | otherwise       = 1 + mylenInt (tail inlist)

mylenStr :: String -> Int
mylenStr inlist
   | null inlist     = 0
   | otherwise       = 1 + mylenStr (tail inlist)
```

and so on.

But we know if we don't add type declaration at all, the function will work with any type. How do we say to Haskell we want the function to work with all types?

Polymorphic data types

Haskell allows us to define polymorphic functions using a polymorphic type declaration. So instead of saying that length takes a list of Ints or a list of Chars as a parameter, we can say that it takes a list of any type:

```
mylength :: [a] -> Int
mylength l
    | null l = 0
    | otherwise   = 1 + mylength (tail l)
```

Polymorphic data types

The type *variable* `a` indicates that the list can consist of elements of arbitrary type (but all elements must still have the same type).
(it doesn't have to be `a`, but it does have to begin with a lower case letter -- because it’s a variable like all variables)

```
mylength :: [a] -> Int
mylength l
    | null l = 0
    | otherwise   = 1 + mylength (tail l)
```

Polymorphic data types

Let's try another.

With what we know so far, this should work:

```
myelem :: a -> [a] -> Bool
myelem item inlist
    | null inlist          = False
    | item == head inlist = True
    | otherwise            = myelem item (tail inlist)
```

Polymorphic data types

But when we compile this function, we get an error:

```
No instance for (Eq a)
     arising from a use of `=='
   In the expression: item == head inlist
   In a stmt of a pattern guard for
                    an equation for `myelem':
     item == head inlist
   In an equation for `myelem':
       myelem item inlist
         | null inlist = False
         | item == head inlist = True
         | otherwise = myelem item (tail inlist)
```

Type Classes

What does that mean? Haskell has done some type checking, and it's figured out an important constraint on the types that type variable `a` could represent --- whatever `a` is, it had better be a type where the == operator makes sense

...known in Haskell as *belonging to the* **Eq** *class*.

Type Classes

In other words, our type declaration isn't complete.

If we're going to use a polymorphic type and an equality (or inequality) test, we must make it very clear to Haskell that not just any type will work here.
It has to be a type whose values can be compared using the equality (==) operator.

How do we do that?

Type Classes

The Eq class contains those types, and by asserting that the values of `a` must belong to this class we constrain the types of values that can replace `a` and make Haskell type police happy.

We hadn't made that constraint explicit before (who knew?) so we should do that now:

```
myelem :: (Eq a) => a -> [a] -> Bool
myelem item inlist
    | null inlist             = False
    | item == head inlist     = True
    | otherwise               = myelem item (tail inlist)
```

Type Classes

Built-in members of the Eq class include:

```
Int
Float
Bool
Char
[Char] or String
```

And many more.
Chapter 13 of your textbook talks about Type Classes.

we will also come back to it shortly, but before that, a reminder...

Automatic type checking

How did Haskell know we needed to assert the Eq class constraint?

It did because it's smart. So smart, in fact, that it can in most cases infer the type declaration from the function definition itself, so that you don't have to write an explicit type declaration for a function. That means this will work just as well:

```
myelem item inlist
    | null inlist             = False
    | item == head inlist     = True
    | otherwise               = myelem item (tail inlist)
```

Automatic type checking

So why did we get that error? Because the type declaration that was inferred by Haskell didn't match up with our explicit type declaration...Haskell's type declaration was better. (Just Haskell's way of letting us know it's smarter than we are.)

It's called Hindley-Milner automatic type inference. It's based on unification. (yes, that’s Prolog’s unification)

Questions?

Another practice problem

Implement (++) this time with type declaration:

```
myappend list1 list2 = ...
```

Syntactic Variation

Using guards:

```
nonzero n
   | n == 0.0  = False
   | otherwise = True
```

Using if-then-else:

```
nonzero n = if n == 0 then False else True
```

You can use if-then-else to re-write all the function you saw/wrote just now. Just try to keep it readable when using if-then-else. Nested if's aren't especially readable.

Pattern Matching

We can also write the same function using pattern matching:

```
nonzero 0 = False
nonzero _ = True
```

This is easy, since instead of writing one complicated definition, we write the function as a series of simple definitions with the same name, but with different patterns in the parameter list.

Pattern Matching

We can also write the same function using pattern matching:

```
nonzero 0 = False
nonzero _ = True
```

Just like Prolog, When Haskell sees a function call, it starts looking at the appropriate function definitions top to bottom, and chooses the first one it sees with a pattern in the parameter list that matches the value of the argument being passed via the function call. (There is no backtracking here)

Pattern Matching

We can also write the same function using pattern matching:

```
nonzero 0 = False
nonzero _ = True
```

If Haskell is asked to evaluate `nonzero 0` it looks at that first line, says "the argument that was passed to the function matches the literal pattern 0" and returns (or replaces the function call with) `False`.

Pattern Matching

We can also write the same function using pattern matching:

```
nonzero 0 = False
nonzero _ = True
```

When Haskell evaluates `nonzero 3` it looks at that first line, says "the argument doesn't match the literal pattern 0" and continues to look for a definition with the same name and a pattern that does match the argument.

Pattern Matching

We can also write the same function using pattern matching:

```
nonzero 0 = False
nonzero _ = True
```

When Haskell evaluates `nonzero 3` it looks at that first line, says "the argument doesn't match the literal pattern 0" and continues to look for a definition with the same name and a pattern that does match the argument.

Haskell then sees the second line, says "the argument matches the wildcard pattern _ " and returns `True`.

Pattern Matching

We can also write the same function using pattern matching:

```
nonzero 0 = False
nonzero _ = True
```

Underscore (_) in Haskell means any pattern. It can also be used for names of variables that are not to be re-used…just like in Prolog[1].

[1] Usually for the purpose of documentation and readability you would want to give a variable a name according to its role, even if it isn't re-used; adding _ to the beginning of a variable's name in this case can make it anonymous (i.e. unused) without losing its name…again, just like Prolog.

Pattern Matching

Patterns can be:

- a literal value -- 24, 'x', "abc", True
  any argument value matches if it is equal to the literal value

Pattern Matching

Patterns can be:

- a literal value -- 24, 'x', "abc", True
  any argument value matches if it is equal to the literal value

- a variable -- x, foo, list1
  any argument value will match a variable

Pattern Matching

Patterns can be:

- a literal value -- 24, 'x', "abc", True
  any argument value matches if it is equal to the literal value

- a variable -- x, foo, list1
  any argument value will match a variable

- a wild card -- _
  any argument value will match the wild card

Pattern Matching

Patterns can be:

- a literal value -- 24, 'x', "abc", True
  any argument value matches if it is equal to the literal value

- a variable -- x, foo, list1
  any argument value will match a variable

- a wild card -- _
  any argument value will match the wild card

- a tuple pattern -- $(p_1,p_2,...,p_n)$
  an argument of form $(v_1,v_2,...,v_n)$ will match this if each v_k matches p_k

Patterns

Example: how do you write a function that returns the third element of a three-element tuple? (It would be a selector, like `fst` and `snd`)

Patterns

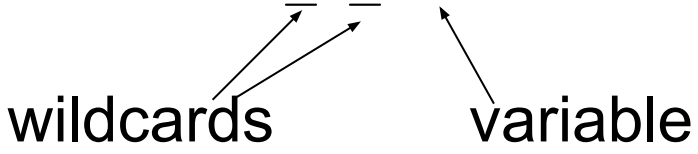
Example: how do you write a function that returns the third element of a three-element tuple? (It would be a selector, like `fst` and `snd`)

```
third (_,_,z) = z
```

Patterns

Example: how do you write a function that returns the third element of a three-element tuple? (It would be a selector, like `fst` and `snd`)

```
third (_,_,z) = z
```



The parameter is a tuple pattern consisting of a variable and two wildcards.

Patterns can also be...

...lists!
More accurately, they can be constructor patterns which are matched to the structure of a list passed as an argument. (It's called a constructor pattern because it contains a cons operator).

The general form of this pattern is

```
(<headoflist>:<tailoflist>)
```

Anything that can be a pattern (i.e. a variable, a wildcard, or a literal, or even another constructor) can go into the <headoflist> or the <tailoflist>.

Constructor patterns

In typical cases, each is represented with a variable, unless the list is empty:

“A constructor pattern over lists will either be `[]` or will have the form `(p:ps)` where `p` and `ps` are themselves patterns.”

..If the input list’s not empty, then the head of the list will be unified/matched with `p` and its tail with `ps`.

Constructor patterns

Using constructor patterns, can you implement these fundamental Haskell list functions?
(try to do it on your own before peaking at the next slide!)

```
head :: [a] -> a
head ? = ?

tail :: [a] -> [a]
tail ? = ?

null :: [a] -> Bool
null ? = ?
null ? = ?
```

Constructor patterns

Using constructor patterns, we could write definitions for the most fundamental Haskell list functions:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs

null :: [a] -> Bool
null [] = True
null (_:_) = False
```

This is in fact how these functions are defined in the Haskell prelude.

Another Example

`elem:` take an element and a list; return true if element is in the list, return false otherwise.

Another Example

`elem:` take an element and a list; return true if element is in the list, return false otherwise.

How is this?

```
myelem item [] = False
myelem item (item:ys) = True
myelem item (_:ys) = myelem item ys
```

Another Example

`elem:` take an element and a list; return true if element is in the list, return false otherwise.

How is this?

```
myelem item [] = False
myelem item (item:ys) = True
myelem item (_:ys) = myelem item ys
```

No! Haskell says: "`Conflicting definitions for ‘item’`"

Why? What did we do?

Another Example

`elem:` take an element and a list; return true if element is in the list, return false otherwise.

How is this?

```
myelem item [] = False
myelem item (item:ys) = True
myelem item (_:ys) = myelem item ys
```

We used ‘item’ twice in the pattern.

Haskell doesn't allow multiple uses of the same pattern on the left hand side. Sorry.

Another Example

`elem:` take an element and a list; return true if element is in the list, return false otherwise.

What about this?

```
myelem item [] = False
myelem item (x:ys) = x == item
myelem item (_:ys) = myelem item ys
```

Another Example

`elem:` take an element and a list; return true if element is in the list, return false otherwise.

What about this?

```
myelem item [] = False
myelem item (x:ys) = x == item
myelem item (_:ys) = myelem item ys
```

Another thing Haskell doesn't allow is overlapping patterns. The two last patterns overlap. This will also not work (not to mention that the logic of implementation is incorrect too).

Exercise

Define each of these built-in functions once using guards (or if-then-else) and once with pattern matching:

`reverse`

`length`

`elem`

`append`

`sum` (this function takes a list and return the sum of all its elements)

`product` (this function takes a list and return the product of all its elements)

`insert` (take a number and an order list of numbers and insert the number into the list at its right place)

Questions?

More List Fun

List comprehensions

List ranges

List Ranges

There is an easy shorthand to express ranges of numbers and other values in a list, without having to write every value:

```
[1..10]  ~> [1,2,3,4,5,6,7,8,9,10]
['a'..'i'] ~> "abcdefghi"
```

Or:

```
[1,3..9]  → [1,3,5,7,9]
['a', 'c'.. 'n'] → "acegikm"
```

List Ranges Syntax

- `[n..m]`
  - If n<m, equals the list [n, n+1, n+2, … ,m]
  - If m<n, it is an empty list.
  - if m==n, it is [n]
- `[n,p..m]`
  - If n<m, equals a list of values with `p-n` as the incremental (or decremental) step. The last member of the list will be the last value that's less than m.
  - for m<n it will be an empty list
  - for m==n, it is [n]

List Ranges Syntax

- In this case: `[n,p..m]`

the last member of the list will be the last member that's reached through (p - n) steps and is less than m, but not necessarily m:

```
[1,3..10] ~> [1,3,5,7,9]
['a', 'd'.. 'i'] ~> "adg"
```

p. 110 of textbook has more examples

More Examples

```
[4,3..0]   -- a decreasing range

[4..0]     -- not a decreasing range. []

[0,5..]    -- [0,5,10,15,20,...] an infinite list

[1..]      -- the infinite list of natural numbers

[1,1..]    -- an infinite list of 1's

[2,2..2]   -- also an infinite list of 2's

[0.1,0.3..1]
-- you should use floating point numbers in ranges with a caution.
-- their values are not accurately represented which can lead
-- to strange results
-- what Haskell outputs for the above range:
-- [0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Infinite Lists

```
[0,5..]    -- [0,5,10,15,20,...] an infinite list

[1,1..]    -- an infinite list of 1's

[2,2..2]   -- also an infinite list of 2's

[1..]      -- the infinite list of all natural positive numbers
```

By definition, an infinite list is a list that never terminates.
How is it possible to have such an infinite structure in a program and expect the program to terminate?

Infinite Lists

```
[0,5..]    -- [0,5,10,15,20,...] an infinite list

[1,1..]    -- an infinite list of 1's

[2,2..2]   -- also an infinite list of 2's

[1..]      -- the infinite list of all natural positive numbers
```

By definition, an infinite list is a list that never terminates.
How is it possible to have such a structure in a program and expect the program to terminate?
The answer is because of Haskell's Laziness in evaluation.

```
Infinite Lists can be quite useful in programming with Haskell.
```

Infinite Lists

```
[0,5..]    -- [0,5,10,15,20,...] an infinite list

[1,1..]    -- an infinite list of 1's

[2,2..2]   -- also an infinite list of 2's

[1..]      -- the infinite list of all natural positive numbers
```

Remember we said that Haskell is lazy which means it leaves functions as "thunks" or expressions that are ready to be evaluated, but doesn't necessarily evaluate them, until they are invoked.

```
Infinite Lists can be quite useful in programming with Haskell.
```

Infinite Lists

Consider this example (from Learn You a Haskell):

write a function that returns the first 24 multiples of 13.

Two ways to do this:

using ranges, like this: `[13,26..13*24]`

Or, use infinite lists: `take 24 [13,26..]`

Because of Lazy evaluation, only the first 24 members will be evaluated and returned and the program will terminate.

In Haskell, infinite Lists can be used in innovative ways to facilitate programming.

Infinite Lists

What happens if you type an infinite list into ghci?

```
Prelude> [1..]
```

Infinite Lists

What happens if you type an infinite list into ghci?

```
Prelude> [1..]
```

That's a program that won't terminate. If you do that you'll have to manually terminate it.

Infinite Lists

What happens if you type an infinite list into ghci?

```
Prelude> [1..]
```

That's a program that won't terminate. If you do that you'll have to manually terminate it.

There are other ways, besides ranges, to create an infinite list:

```
take 10 (cycle [1,2,3]) ~> [1,2,3,1,2,3,1,2,3,1]

take 10 (repeat 5)  ~> [5,5,5,5,5,5,5,5,5,5]
```

Back to ranges

What type of data can be used in a range?

Back to ranges

What type of data can be used in a range?

I casually said “numbers and other values”. Can it be any value?

Back to ranges

What type of data can be used in a range?

I casually said “numbers and other values”. Can it be any value?

Like, `[“apple”, “orange”..“cucumber”]` ?

Back to ranges

What type of data can be used in a range?

I casually said “numbers and other values”. Can it be any value?

Like; `[“apple”, “orange”..“cucumber”]` ?

No, it can’t. Haskell will blows up at this:

```
No instance for (Enum [Char])
```

Back to ranges

What type of data can be used in a range?

I casually said “numbers and other values”. Can it be any value?

Like; `[“apple”, “orange”..“cucumber”]` ?

If you give the first example to ghci, it blows up:

```
No instance for (Enum [Char])
```

What’s that, `Enum [Char]`?

Back to ranges

What type of data can be used in a range?

I casually said “numbers and other values”. Can it be any value?

Like; `[“apple”, “orange”..“cucumber”]` ?

If you give the first example to ghci, it blows up:

```
No instance for (Enum [Char])
```

What’s that, Enum [Char]?

Let’s experiment to find out..

Back to ranges

We already know that Haskell's automatic type checker is smarter than us. So we'll ask Haskell to find out.
I want to know what data types are allowed by Haskell to appear in a range. I'll write this function to a file, save, and load in ghci:

```
make_range a b c = [a,b..c]
```

without providing the type declaration.

Once loaded (without errors), I'll query ghci for its type:

```
Prelude>:t make_range
```

Back to ranges

We already know that Haskell's automatic type checker is smarter than us. So we'll ask Haskell to find out.
I want to know what data types are allowed by Haskell to appear in a range. I'll write this function to a file, save, and load in ghci:

```
make_range a b c = [a,b..c]
```

without providing the type declaration.

Once loaded (without errors), I'll query ghci for its type:

```
Prelude>:t make_range
make_range :: Enum t => t -> t -> t -> [t]
```

Enum

```
make_range a b c = [a,b..c]

Prelude>:t make_range
make_range :: Enum t => t -> t -> t -> [t]
```

This is just like Eq, isn't it?

This means Enum is another type class.

Meaning: The t's can be anything as long they belong to the Enum class.

Enum

The Enum class is a class for all types whose values can be enumerated in a certain order.

To put it simply, anything can appear in a list range as long as the values in its type can be enumerated.

Natural numbers naturally have this capacity. So do letters of alphabet. But things that don't have a 'natural' enumeration can also belong to this class.

How can a type belong to this class? Who determines that?
How can a type belong to any class?

Type Classes

Haskell's Type classes are a lot like Interfaces in Java.

Each Type Class has a set of *abstract* key functions, meaning they have a predetermined type declaration but no implementation. In order to be allowed membership to the club, a data type must implement those key functions for its own members.

The signature function of the Eq class was the equality check (==). Any type that's of Eq type (such as the ones we saw before) have a type specific implementation for this function.

Type Classes

The Enum class has functions like

```
toEnum :: Int -> a
fromEnum :: a -> Int
```

that map values in a given type to the sequence of integer numbers or vice versa (and by doing so, create an enumeration among any set of arbitrary values).

It also has

```
succ :: a -> a
pred :: a -> a
```

that return the successor or predecessor of a value.

Type Classes

See the documentation in Prelude for more information on Enum: https://downloads.haskell.org/~ghc/latest/docs/html/libraries/base-4.8.1.0/Prelude.html#t:Enum

For any of the known data types (such as Char, Bool, etc.), the Prelude documentation gives a list of all the type classes they belong to.

Type Classes

The Bool data type, for instance,
has these listed for it in Prelude.
So Bool too, is a member of Enum.

⊟ Instances

```
Bounded Bool
Enum Bool
Eq Bool
Data Bool
Ord Bool
Read Bool
Show Bool
Ix Bool
Generic Bool
FiniteBits Bool
Bits Bool
Storable Bool
type Rep Bool
type (==) Bool a b
```

Type Classes

The Bool data type, for instance,
has these listed for it in Prelude.
So Bool too, is a member of Enum.

```
Prelude>fromEnum True
1
Prelude>fromEnum False
0

Prelude>make_range True False True
[True]
```

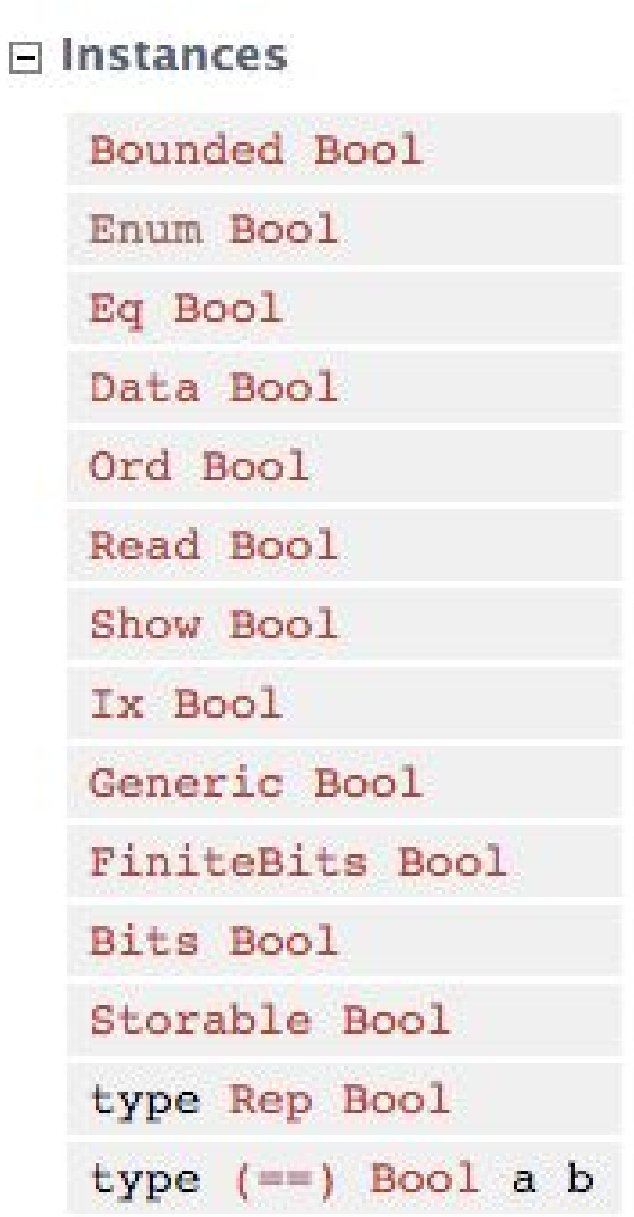
⊟ Instances

```
Bounded Bool
Enum Bool
Eq Bool
Data Bool
Ord Bool
Read Bool
Show Bool
Ix Bool
Generic Bool
FiniteBits Bool
Bits Bool
Storable Bool
type Rep Bool
type (==) Bool a b
```

Type Classes

Note: For this course, you only need to know about the type classes that we talk about in the class. Most of the ones in this picture we won't talk about.

In addition to Enum and Eq, we have also seen Num. Num encompasses all the basic numerical types (e.g. Float and Int).
we may also talk about Show and Read and maybe Bounded.



List Comprehension

An example of Set comprehension from Math:

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

A set that contains the first ten even natural numbers.

List Comprehension

An example of Set comprehension from Math:

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

A set that contains the first ten even natural numbers.

Haskell equivalent of it is called a list comprehension:

```
[x*2 | x <- [1..10]]
```

the part before the pipe is the generator function

the part after, express the input range and the potential constraints over the input. Note the use of list ranges to express the input domain.

* Page 113 and 114 of your book covers list comprehension.

List Comprehension

```
addPairs pairList = [ m+n | (m,n) <- pairList ]
```

which, BTW, means pairList needs to be a list of tuples and m and n be numbers.

```
addPairs :: Num a => [(a,a)] -> [a]
```

List Comprehension

```
addPairs pairList = [ m+n | (m,n) <- pairList ]

Prelude> addPairs [(1,2), (3,5), (4,1)]
?
```

List Comprehension

```
addPairs pairList = [ m+n | (m,n) <- pairList ]

Prelude> addPairs [(1,2), (3,5), (4,1)]
[3, 8, 5]
```

List Comprehension

```
addPairs pairList = [ m+n | (m,n) <- pairList ]
```

Now, let's add a constraint:

```
addPairs2 pairList = [ m+n | (m,n) <- pairList, m<n ]
```

this constraint is on the input range; so, only those elements in the input (pairList) will be passed to the generator that meet this constraints.

What would the result of this be now?

```
Prelude> addPairs2 [(1,2), (3,5), (4,1)]
```

List Comprehension

```
addPairs2 pairList = [ m+n | (m,n) <- pairList, m<n ]
```

this constraint is on the input range; so, only those elements in the input (pairList) will be passed to the generator that meet this constraints.

What would the result of this be now?

```
Prelude> addPairs2 [(1,2), (3,5), (4,1)]
```

`[3,8]` -- (4,1) does not meet the condition of m<n, so it's discounted

Any conditional expression can go in place of m<n in the formula.

Any function that return a list can sit in place of the input range.

List Comprehension

One of the uses of list comprehensions is filtering a list and producing a subset of its members. Here's another example:

```
digits :: String -> String

digits st = [ch | ch<-st, isDigit ch ]
```

List Comprehension

But filtering is not its only use. List comprehension is a very easy way to manipulate existing values from one or multiple sources into a new representation. From one source:

```
toUpperString st = [ toUpper x | x <- st]

>toUpperString “sara”

“SARA”


oddOrEven l = [ if x `mod` 2 == 0 then “even” else “odd”
| x <- l]

>oddOrEven [2,3,4]

["even","odd","even"]
```

Exercise

1. Use list comprehension to write a function that takes an int value and returns a list of its divisors:

```
divisor 12 ~> [1,2,3,4,6,12]
```

2. Use list comprehension (and the built-in function `sum`) to write a function that takes a value (of any type) and a list of values (of the same type) and returns the number of occurrences of that value in the list:

```
matches 2 [1,2,3,2,5,2] ~> 3
```

```
matches 1 [2,3,4] ~> 0
```

List Comprehension

From multiple sources: i.e. list comprehension to produce Cartesian products or nested loops:

```
[(i,j) | i <- [1..5], j <- [1..5]] ~> [(1,1),(1,2),(1,3),
(1,4), (1,5),(2,1),(2,2),(2,3),(2,4),(2,5),(3,1),(3,2),(3,3),
(3,4), (3,5),(4,1),(4,2),(4,3),(4,4),(4,5),(5,1),(5,2),(5,3),
(5,4)]
```

```
[(i,j) | i <- [1..5], j <- [1..5], i+j = 5] ~> [(1,4),(2,3),
(3,2),(4,1)]
```

```
[(i,j) | i <- [1..5], j <- [i+1..5]] ~> [(1,2),(1,3),(1,4),
(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
```

```
[(i,j) | i <- [1..5], j <- [i+1..5], i+j = 5] ~> [(1,4),(2,3)]
```

List Comprehension

From multiple sources: i.e. list comprehension to produce Cartesian products or nested loops:

```
[(i,j) | i <- [1..5], j <- [1..5]] ~> [(1,1),(1,2),(1,3),
(1,4), (1,5),(2,1),(2,2),(2,3),(2,4),(2,5),(3,1),(3,2),(3,3),
(3,4), (3,5),(4,1),(4,2),(4,3),(4,4),(4,5),(5,1),(5,2),(5,3),
(5,4)]
```

```
[(i,j) | i <- [1..5], j <- [1..5], i+j = 5] ~> [(1,4),(2,3),
(3,2),(4,1)]
```

```
[(i,j) | i <- [1..5], j <- [i+1..5]] ~> [(1,2),(1,3),(1,4),
(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
```

```
[(i,j) | i <- [1..5], j <- [i+1..5], i+j = 5] ~> [(1,4),(2,3)]
```

Note that the 4th produces the same result as 2nd, except without repetitions..

List Comprehension

Recursive calls are also allowed from inside a list comprehension. Here's a (somewhat tortured) implementation of the factorial function:

```
fac n = [n*x| x <- if n ==1 then [1] else fac (n-1)]
```

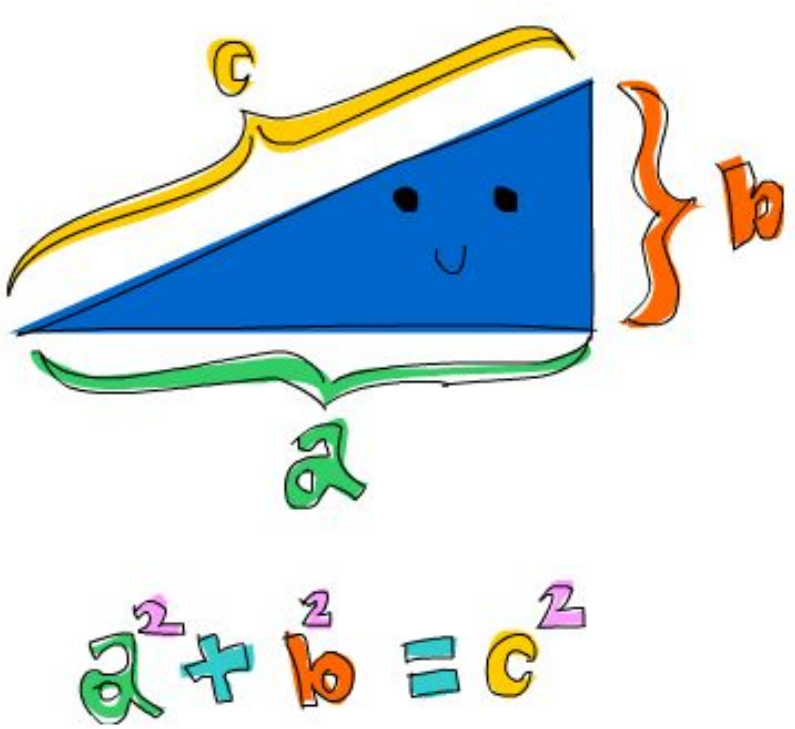
Do you see how it works?

List Comprehension

To show the wide scope of possibilities with list comprehension, here's an interesting example from Learn You a Haskell:

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

Use list comprehension to implement a function that calculates the result.



List Comprehension

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

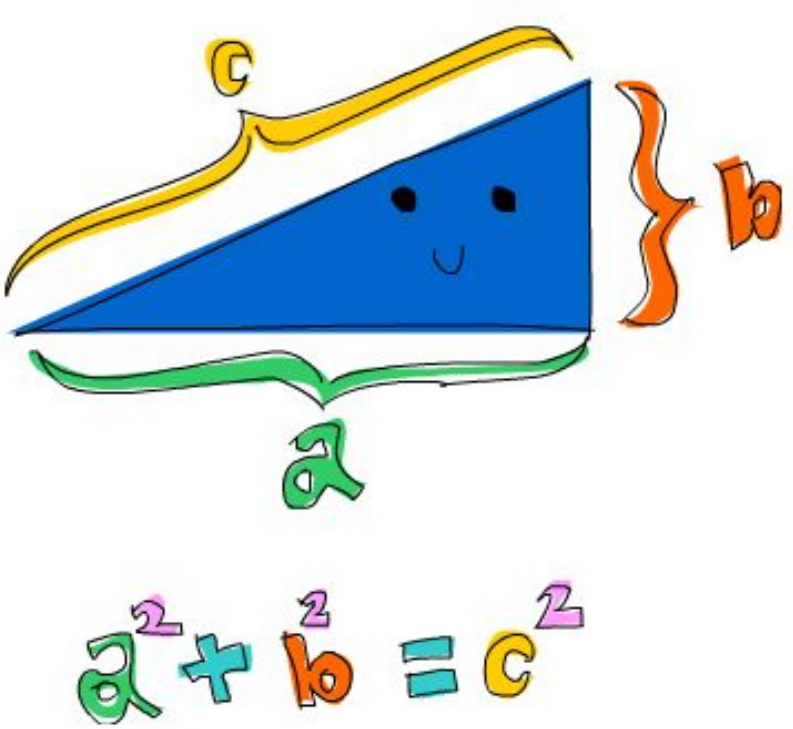
Answer:

there are three sides, each equal to or smaller than 10. So the input ranges are:

a <- [1..10]

b <- [1..10]

c <- [1..10]



List Comprehension

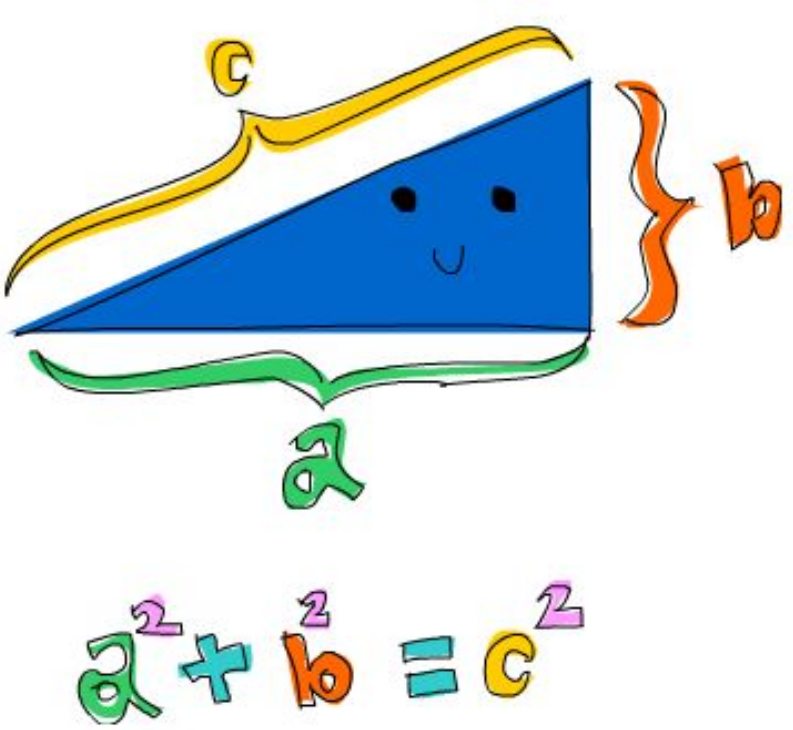
Question: “which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?”

Answer:

two constraints exists on these ranges:

1) that the three numbers have to form a right triangle together.

2) that the perimeter of the triangle has to be 24.

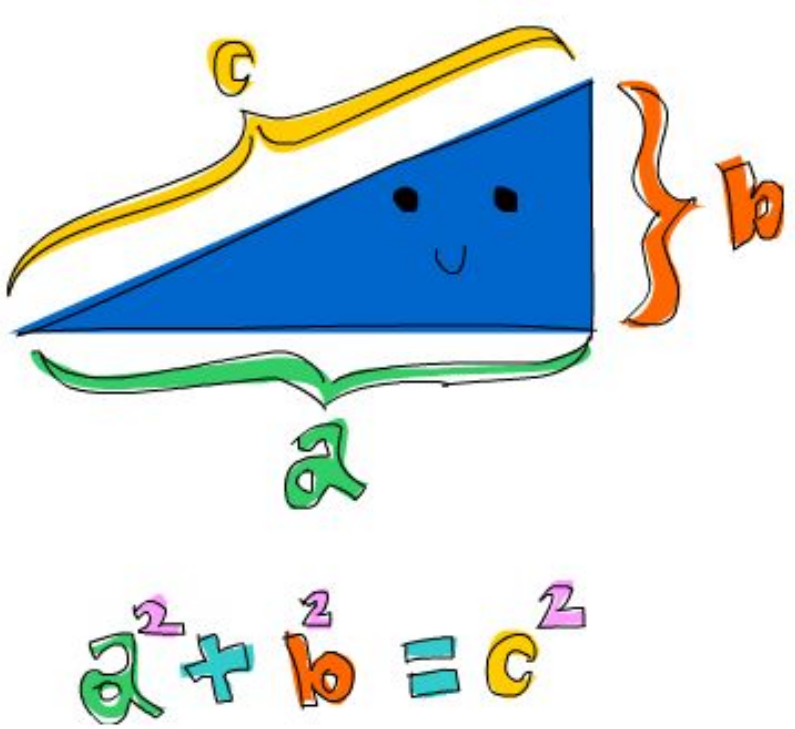


List Comprehension

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

Answer:

How to express “that the three numbers have to form a right triangle together?”



List Comprehension

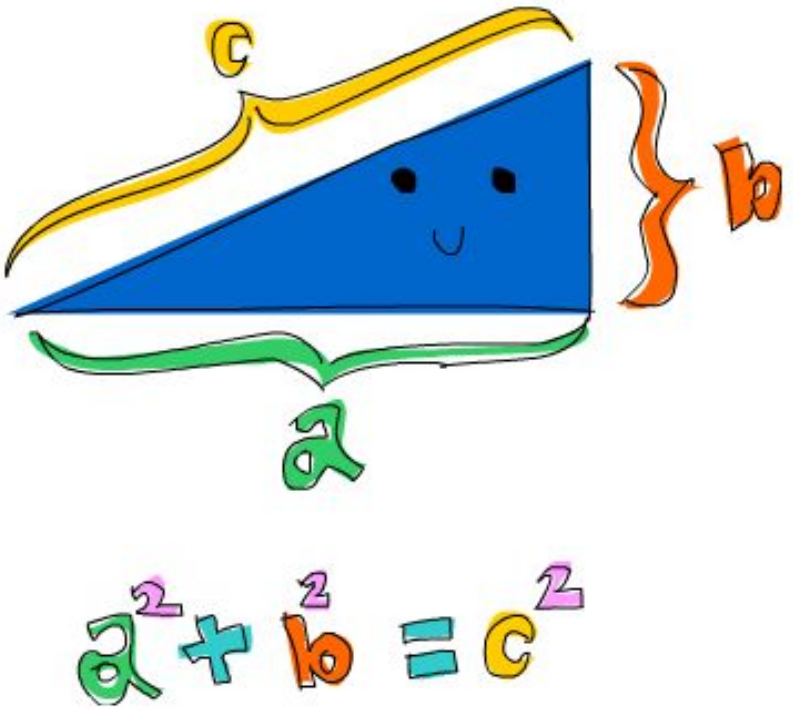
Question: “which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?”

Answer:

How to express “that the three numbers have to form a right triangle together?”

We know the pythagorean theorem holds for all right trangles. So all we need to assert for this part is:

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

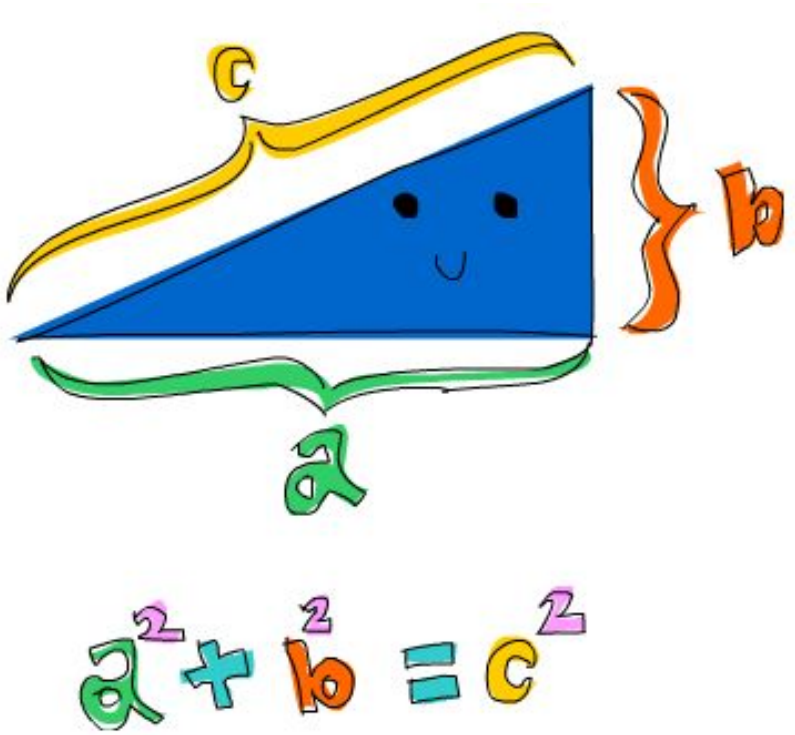


List Comprehension

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

Answer:

What about the second constraint? the perimeter must be 24?



List Comprehension

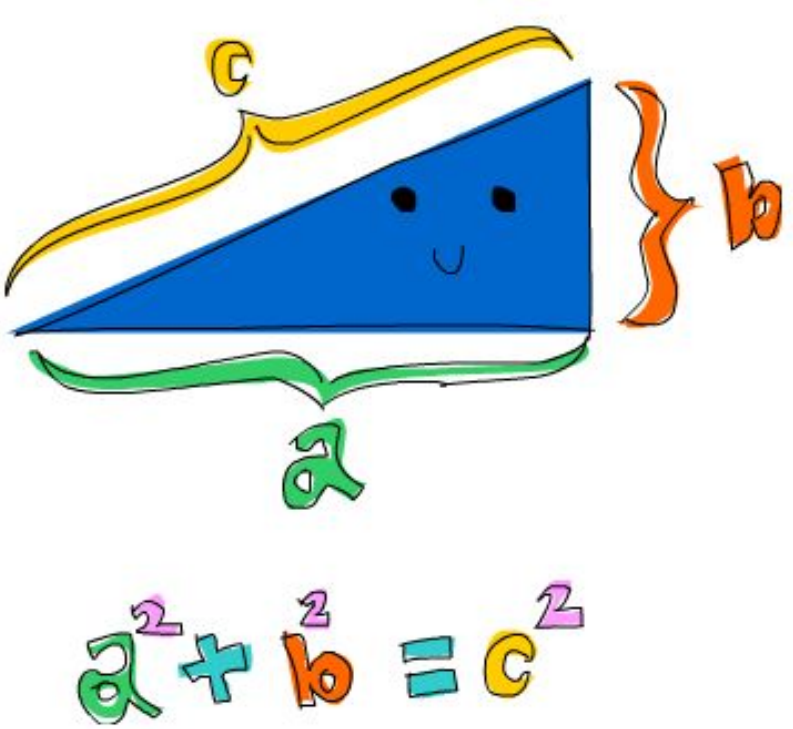
Question: “which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?”

Answer:

What about the second constraint? the perimeter must be 24?

perimeter = a + b + c = 24

adding it all to the list comprehension..



List Comprehension

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

Answer:

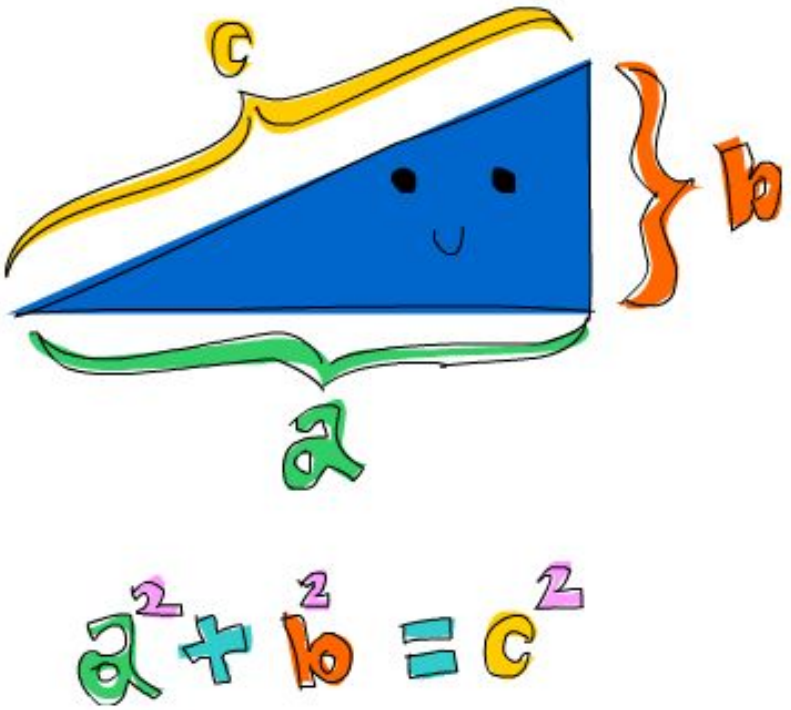
```
[(a,b,c) | a <- [1..10],

b<- [1..10], c<-[1..10],

a^2 + b^2 == c^2 ,

a+b+c == 24]
```

→ the answer is `[(6,8,10),(8,6,10)]`

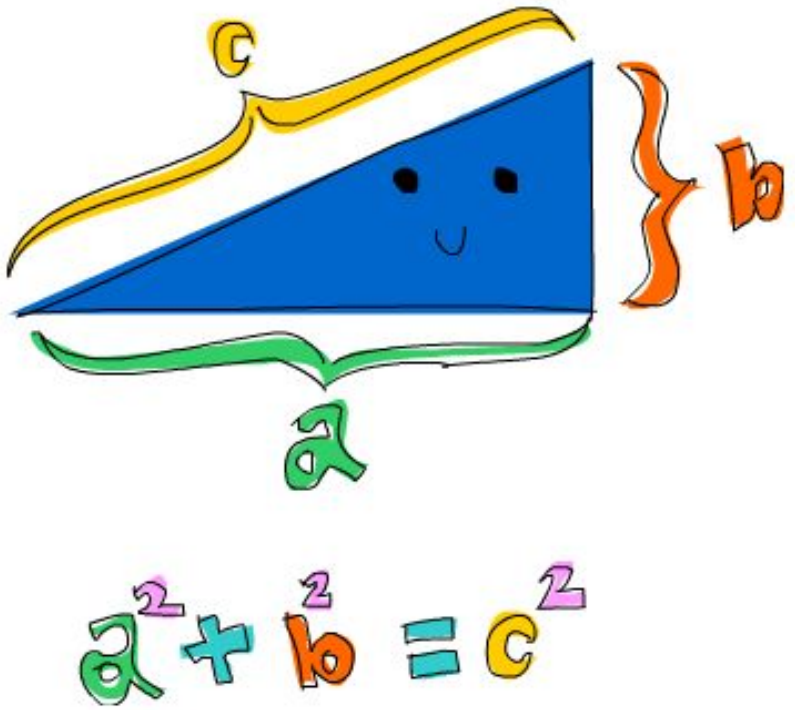


List Comprehension

Answer:

```
[(a,b,c) | a <- [1..10],
b<- [1..10], c<-[1..10],
a^2 + b^2 == c^2 ,
a+b+c == 24]
```

Question: Can you change this implementation to only produce unique results? Either (6,8,10) or (8,6,10), but not both. (go back a few slides for inspiration if you can't think of the answer)



List Comprehension

Side Note:

Did this approach to solving the problem remind you at all of a previous problem we'd solved in this class?

Instead of imperatively constructing the solution, we expressed the range of possibilities as well as the constraints on this range and left it to the compiler to find the solution.

This is another example of non-deterministic programming that we did in Prolog (remember the zebra puzzle?).

List Comprehensions are great tools for this type of problem solving.

Questions?

We have covered (most of) chapters 5, 6 and 7.