CPSC 312

Functional and Logic Programming

November 12, 2015

Some Clarification on the Project

hand in:

project1 for team submissions

p1individual for individual submissions

Q1:

1. write at least 3 rules.

2. your knowledge base has to load using load_rules.

Some Clarification on the Project

Q3:

1. Regard the example questions as types or templates, not exact sentences.

what is it ?

what does it VERB ?

Does it VP ?

Is it NP ?

I don’t want you to only parse “is it a brown swan?”, but sentences of this type: is it a flat nose? does it swim? etc.

Recap from last session

Lists: basic concepts, important built-in functions

We tried to implement some built-in functions with recursion

Then we saw constructor patterns

Which allowed us to write recursive functions using pattern matching

List Ranges

Type Classes

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]

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]
```

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

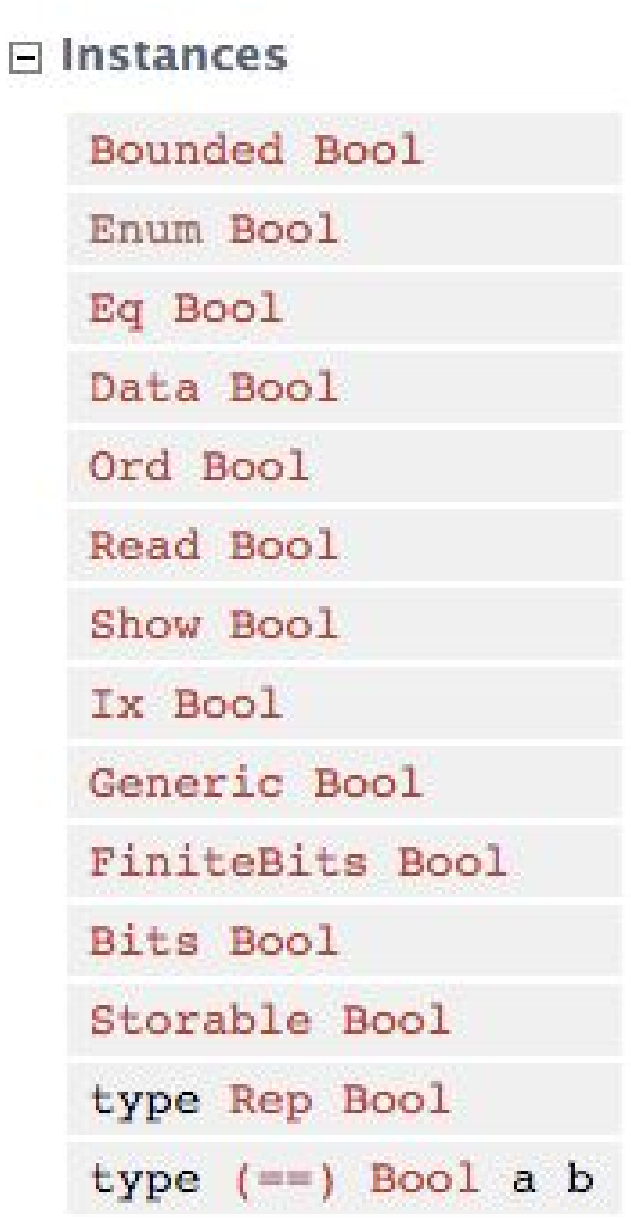
```
succ :: a -> a
pred :: a -> a
```

that return the successor or predecessor of a value.

Type Classes

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

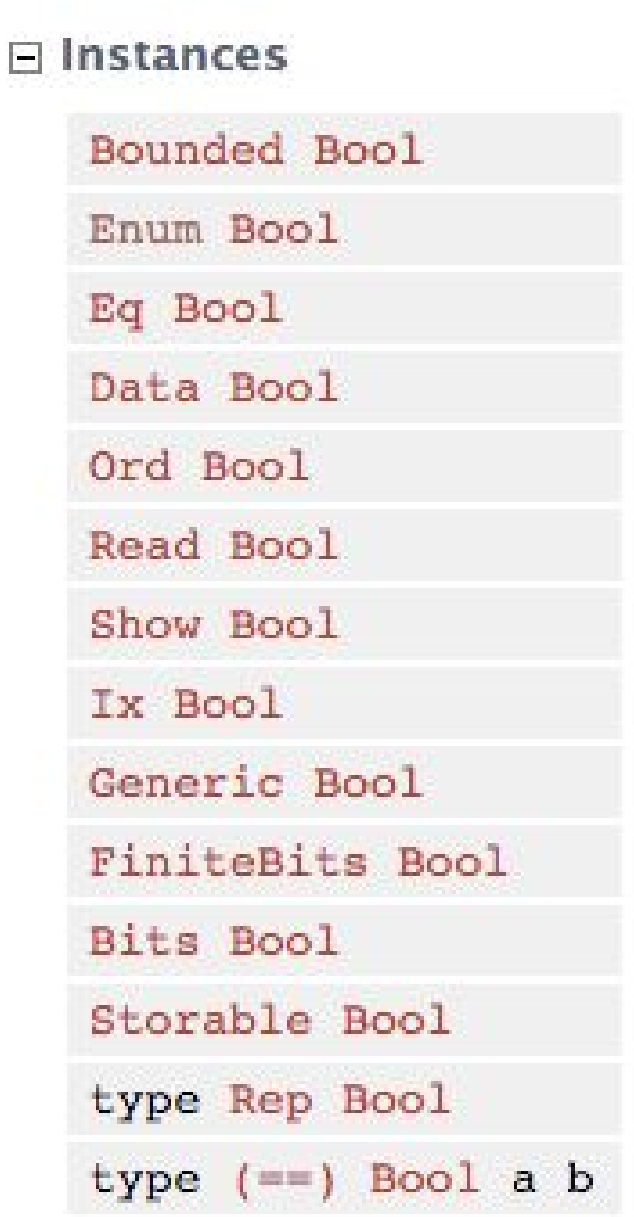
```
Prelude>fromEnum True
1
Prelude>fromEnum False
0
Prelude>toEnum 1 :: Bool
True

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



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.



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)

List Recursion

`reverse`

How do we find a recursive solution?

List Recursion

`reverse`

How do we find a recursive solution? Induction.

`reverse [1,2,3] = [3,2,1] ?`

`reverse [2,3] = [3,2] ?`

`reverse [3] = [3]`

base case: `reverse [x] = [x]`

`going back up from the base case, can you observe a pattern from one step to the next?`

List Recursion

`reverse`

How do we find a recursive solution? Induction.

`reverse [1,2,3] = [3,2,1] ?`

`reverse [2,3] = [3,2] ?`

`reverse [3] = [3]`

base case: `reverse [x] = [x]`

going back up from the base case, can you observe a pattern from one step to the next?

if we have the result of `reverse [2,3]` how do we get `reverse [1,2,3]` from it?

List Recursion

`reverse`

How do we find a recursive solution? Induction.

`reverse [1,2,3] = [3,2,1] ?`

`reverse [2,3] = [3,2] ?`

`reverse [3] = [3]`

base case: `reverse [x] = [x]`

recursive step: `reverse (x:xs) = reverse xs ++ [x]`

List Recursion

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

List Recursion

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Now, let's do it without pattern matching…

the logic is the same, the only difference would be in how we check for conditions.

```
reverse inlist
  | -- base case condition check
  | otherwise = -- the recursive step
```

List Recursion

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse inlist
  | inlist == []  = [] -- base case condition check
  | otherwise = reverse (tail inlist) ++ (head inlist)
-- the recursive step
```

List Recursion

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse inlist
  | inlist == []  = [] -- base case condition check
  | otherwise = reverse (tail inlist) ++ (head inlist)
-- the recursive step
```

Is this complete?

List Recursion

```
reverse_pm :: [a] -> [a]
reverse_pm [] = []
reverse_pm (x:xs) = reverse xs ++ [x]


reverse :: [a] -> [a]
reverse inlist
  | inlist == []  = [] -- base case condition check
  | otherwise = reverse (tail inlist) ++ (head inlist)
-- the recursive step
```

It is now.

List Recursion

```
append list1 list2
```

List Recursion

`append list1 list2`

Induction:

append `[1,2] [3,4] = [1,2,3,4] ?`

append `[2] [3,4] = [2,3,4] ?` somewhat easy

append `[] [3,4] = [3,4]` very easy. we'll start with this

base case: `append [] list2 = list2`

List Recursion

`append list1 list2`

Induction:

append `[1,2] [3,4] = [1,2,3,4] ?`

append `[2] [3,4] = [2,3,4] ?` somewhat easy

append `[] [3,4] = [3,4]` very easy. we'll start with this

base case: `append [] list2 = list2`

recursive step: look for patterns...if we knew the result of append [2] [3,4], how would we calculate the result of append [1,2] [3,4] ?

List Recursion

`append list1 list2`

Induction:

append `[1,2] [3,4] = [1,2,3,4] ?`

append `[2] [3,4] = [2,3,4] ?` somewhat easy

append `[] [3,4] = [3,4]` very easy. we'll start with this

base case: `append [] list2 = list2`

recursive step: look for patterns...if we knew the result of append [2] [3,4], how would we calculate the result of append [1,2] [3,4] ? append the head of list1 to the result.

List Recursion

`append list1 list2`

Induction:

append `[1,2] [3,4] = [1,2,3,4] ?`

append `[2] [3,4] = [2,3,4] ?` somewhat easy

append `[] [3,4] = [3,4]` very easy. we'll start with this

base case: `append [] list2 = list2`

recursive step: look for patterns...if we knew the result of append [2] [3,4], how would we calculate the result of append [1,2] [3,4] ? append the head of list1 to the result.

List Recursion

`append list1 list2`

Induction:

append `[1,2] [3,4] = [1,2,3,4] ?`

append `[2] [3,4] = [2,3,4] ?` somewhat easy

append `[] [3,4] = [3,4]` very easy. we'll start with this

base case: `append [] list2 = list2`

recursive step: `append (l:ls) list2 = l:(append ls list2)`

List Recursion

```
append :: [a] -> [a] -> [a]
append [] list2 = list2
append (l:ls) list2 = l:(append ls list2)
```

Could we have done the induction differently?

why is the base case

```
append [] list2 = list2
```

and not

```
append list1 [] = list1
```

?

List Recursion

Different Induction:

`append [1,2] [3,4] = [1,2,3,4] ?`

`append [1,2] [4] = [1,2,4] ?`

`append [1,2] [] = [1,2] base case`

but if take this approach, what would be the recursive step?

say we have the list that's the result of appending [1,2] and [4]. How do we then build append of [1,2] and [3,4] from that list?

It means, we'd need to build [1,2,3,4] from [1,2,4].

We'd have to insert the head of list2 at the index (length list1 + 1).

We must implement length and insert before we can do that. It's difficult, but it's not impossible.

List Recursion

Another different Induction:

`append [1,2] [3,4] = [1,2,3,4] ?`

`append [1,2] [3] = [1,2,3] ?`

`append [1,2] [] = [1,2] base case`

instead of taking the head off the second list, we can take the last item off at each recursive/inductive step.

If we knew the result of appending [1,2] to [3], could we use it to build the result of appending [1,2] to [3,4] ?

This means building [1,2,3,4] from [1,2,3].

It would mean taking the last item of list2 and appending it to the end of [1,2,3]...wait, did I say append? But that's what I'm implementing now!

List Recursion

Another different Induction:

`append [1,2] [3,4] = [1,2,3,4] ?`

`append [1,2] [3] = [1,2,3] ?`

`append [1,2] [] = [1,2] base case`

instead of taking the head off the second list, we can take the last item off at each recursive/inductive step.

If we knew the result of appending [1,2] to [3], could we use it to build the result of appending [1,2] to [3,4] ?

This means building [1,2,3,4] from [1,2,3].

Okay, then, it would mean inserting the last element of list2 at the last position in [1,2,3]. Better, still need to implement insert first.

List Recursion

Back to the first induction:

append `[1,2] [3,4] = [1,2,3,4] ?`

append `[2] [3,4] = [2,3,4] ?` somewhat easy

append `[] [3,4] = [3,4]` very easy. we'll start with this

base case: `append [] list2 = list2`

recursive step: `append (l:ls) list2 = l:(append ls list2)`

So this is not the only way, but it's certainly the easiest way, because the only other function we need to use is the list constructor. The base case we choose to include is also consistent with our method of induction.

List Recursion

```
append :: [a] -> [a] -> [a]
append [] list2 = list2
append (l:ls) list2 = l:(append ls list2)
```

What about the non pattern-matching way?

Once again the logic is the same. We merely translate the syntax above into a guard or if-then-else type statement.

List Recursion

```
append_pm :: [a] -> [a] -> [a]
append_pm [] list2 = list2
append_pm (l:ls) list2 = l:(append ls list2)

append_ite list1 list2 =
   if list1 == [] then list2
   else (head list1):(append_ite (tail list1) list2)

append list1 list2
   | list1 == []  = list2
   | otherwise = (head list1):(append (tail list1) list2)
```

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)]
```

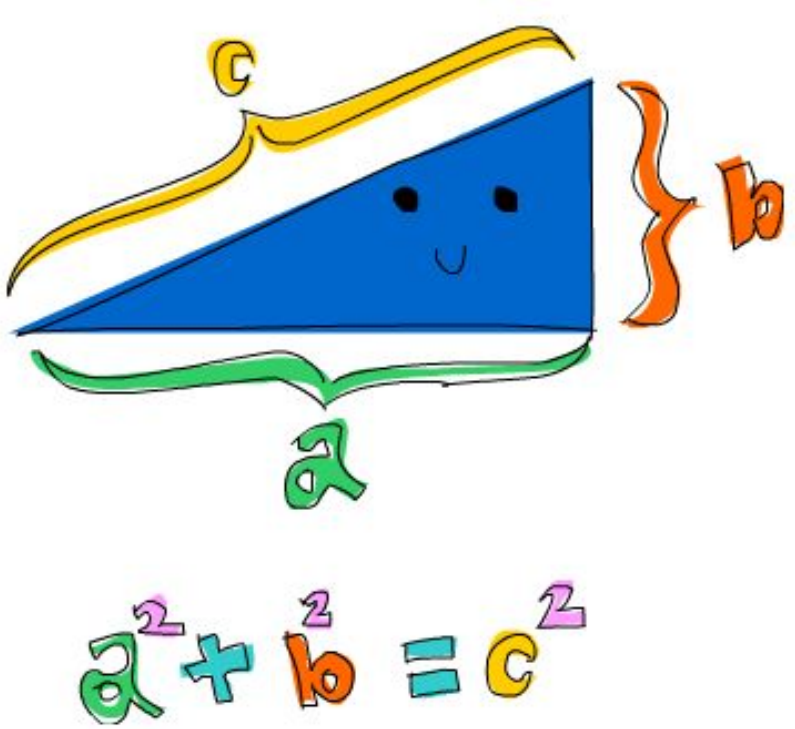
Note that the 4th produces the same result as 2nd, except without repetitions..

List Comprehension

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?”

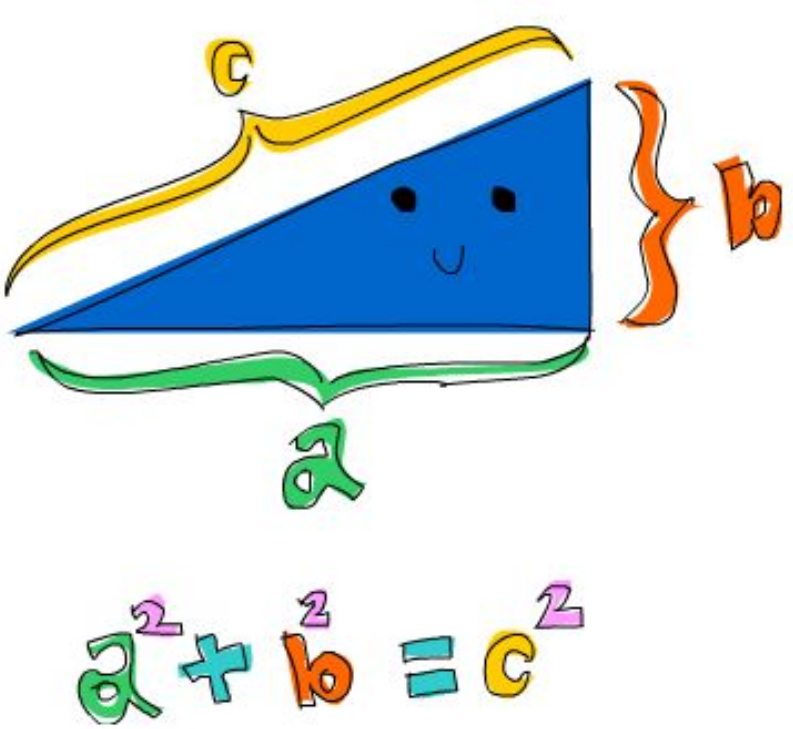
Answer:

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

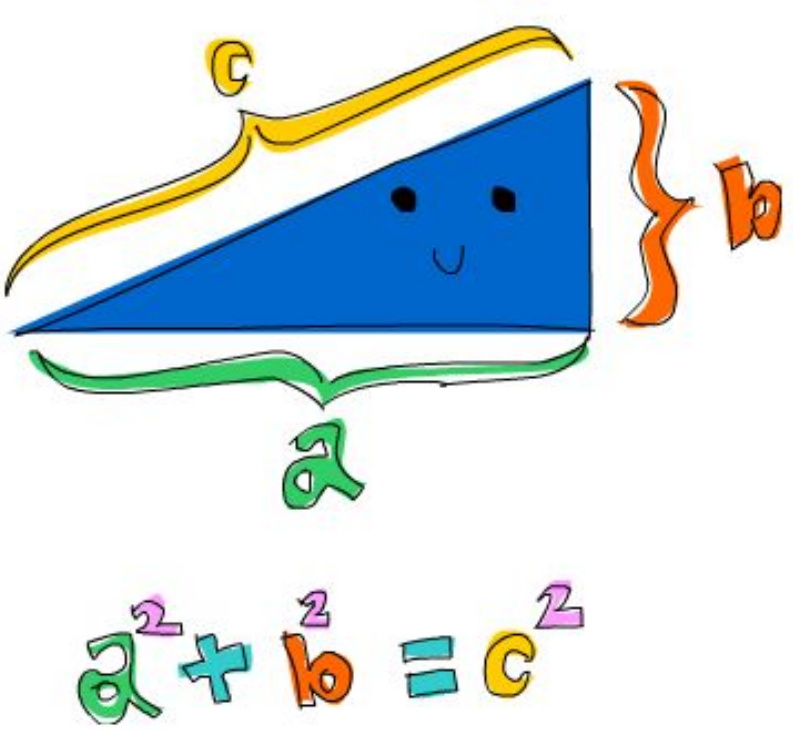
Question: “which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?”

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.

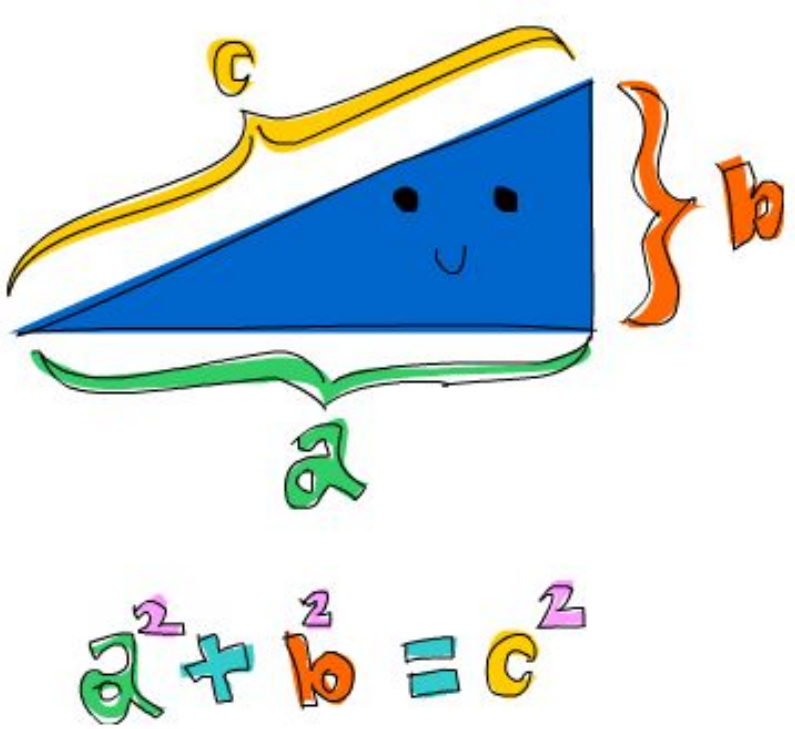


List Comprehension

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

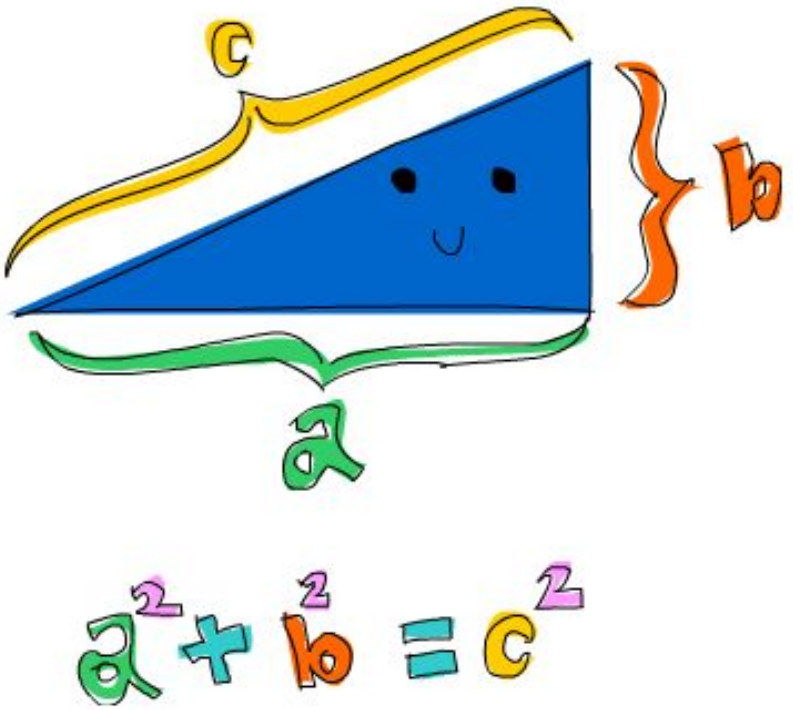
Question: “which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?”

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:

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

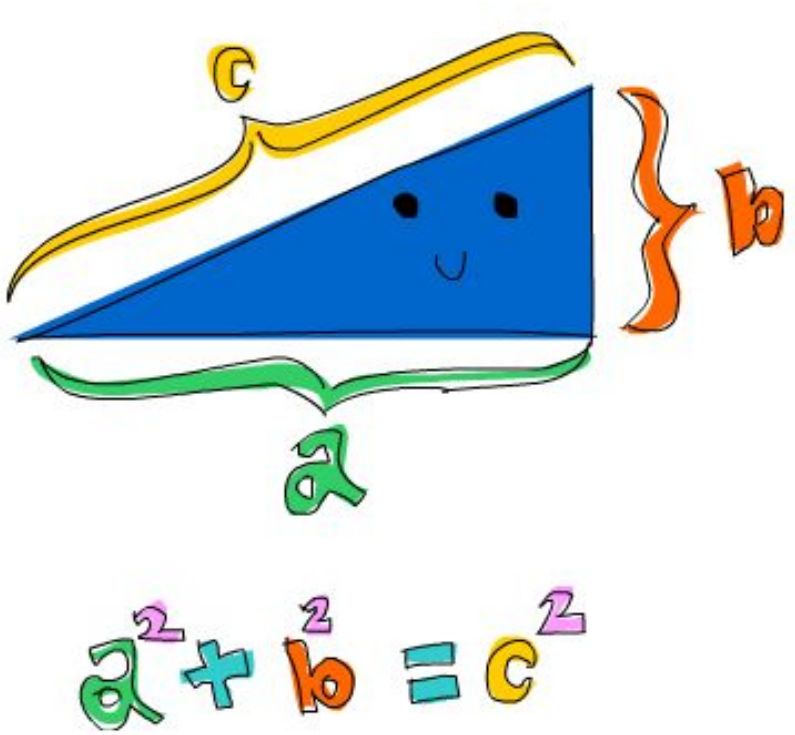


List Comprehension

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

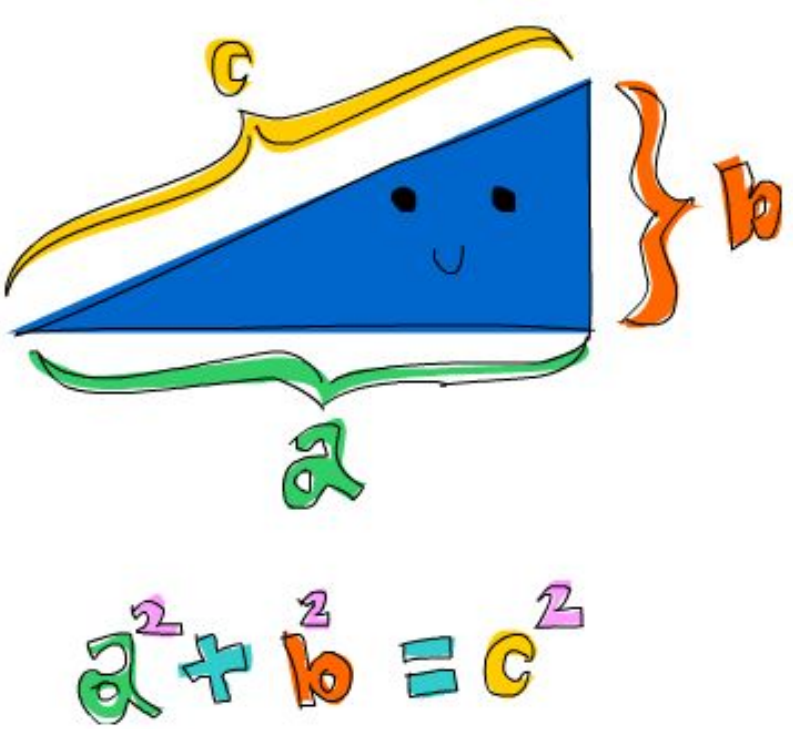
Question: “which right triangle that has integers for all sides and all sides equal to or smaller than 10 has a perimeter of 24?”

Answer:

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

perimeter = a + b + c = 24

adding it all to the list comprehension..



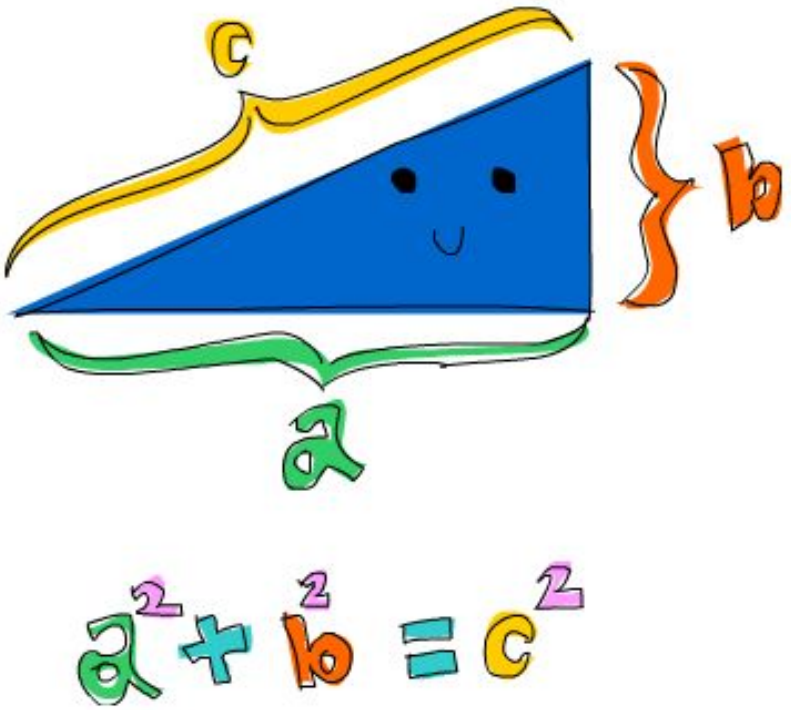
List Comprehension

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)]`

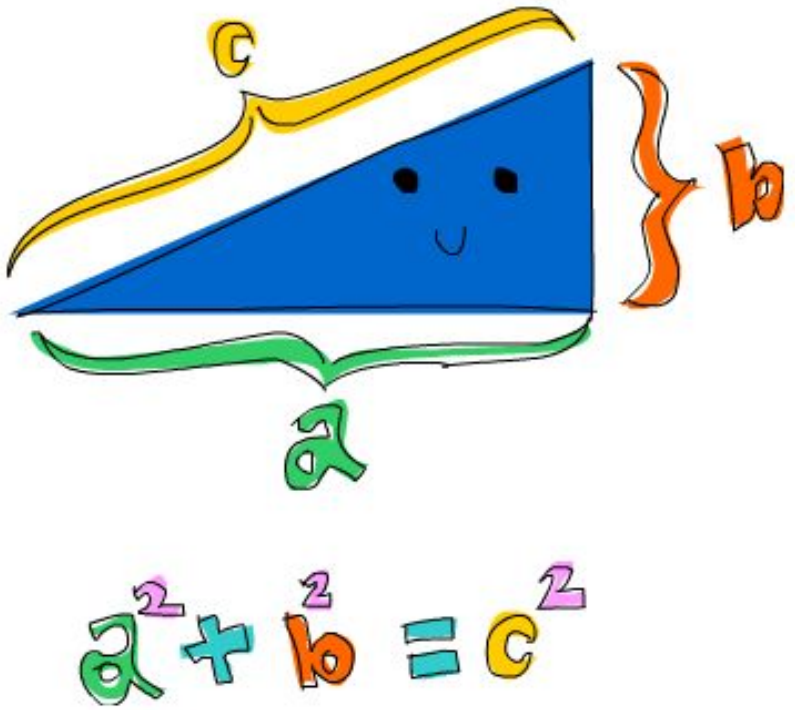


List Comprehension

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.

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

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?

Prelude> fac 4
[24]

List Comprehension

Question: Given a list of words, a character, and a number, select a subset of the words, in which the total number of occurrences of the character is the given number and every word has at least one occurrence.

List Comprehension

Question: Given a list of words, a character, and a number, select a subset of the words, in which the total number of occurrences of the character is the given number and every word has at least one occurrence.

list = [“table”, “roar”, “press”, “boot”, “team”, “sheet”]
ch = ‘e’
n = 3

List Comprehension

Question: Given a list of words, a character, and a number, select a subset of the words, in which the total number of occurrences of the character is the given number and every word has at least one occurrence.

list = [“table”, “roar”, “press”, “boot”, “team”, “sheet”]
ch = ‘e’
n = 3

possible answers = [“table”, “press”, “team”]; [“table”, “sheet”]; [“press”, “sheet”]; [“team”, “sheet”]

List Comprehension

list = [“table”, “roar”, “press”, “boot”, “team”, “sheet”]

ch = ‘e’

n = 3

possible answers = [“table”, “press”, “team”]; [“table”, “sheet”];
[“press”, “sheet”]; [“team”, “sheet”]

Can we implement this with recursion?

List Comprehension

list = [“table”, “roar”, “press”, “boot”, “team”, “sheet”]

ch = ‘e’

n = 3

possible answers = [“table”, “press”, “team”]; [“table”, “sheet”]; [“press”, “sheet”]; [“team”, “sheet”]

Can we implement this with recursion?

maybe.

List Comprehension

Strategy based on list recursion:

Pick the first word from the list, keep it if it the character is in it.

call the function again with the tail, the character and the original number minus the number of occurrences of the character in head.

List Comprehension

```
ncharSubset n ch inlist
    | -- count of ch in the head of inlist > 0  = ?
    | otherwise  = ncharSubset n ch (tail inlist)
```

List Comprehension

```
ncharSubset n ch inlist
    | c > 0  = head:(ncharSubset (n-c) ch (tail inlist))
    | otherwise  = ncharSubset n ch (tail inlist)
    where c = count ch (head inlist)

-- count: an aux function, returns the # of occurrences
of a char in a word
count:: Char -> [Char] -> Int

base case?
```

List Comprehension

```
ncharSubset 0 _ _ = []   -- is this enough?
ncharSubset n ch inlist
    | c > 0  = head:(ncharSubset (n-c) ch (tail inlist))
    | otherwise  = ncharSubset n ch (tail inlist)
    where c = count ch (head inlist)


-- an aux function, returns the # of occurrences of a
char in a word
count:: Char -> [Char] -> Int
```

List Comprehension

```
ncharSubset _ _ [] = []
ncharSubset 0 _ _ = []
ncharSubset n ch inlist
    | c > 0  = head:(ncharSubset (n-c) ch (tail inlist))
    | otherwise  = ncharSubset n ch (tail inlist)
    where c = count ch (head inlist)


-- an aux function, returns the # of occurrences of a
char in a word
count:: Char -> [Char] -> Int
```

List Comprehension

```
ncharSubset _ _ [] = []
ncharSubset 0 _ _ = []
ncharSubset n ch inlist
    | c > 0  = head:(ncharSubset (n-c) ch (tail inlist))
    | otherwise  = ncharSubset n ch (tail inlist)
    where c = count ch (head inlist)
```

```
Prelude>ncharSubset 3 'e' ["table", "roar", "press", "boot",
"team", "sheet"]
["table","press","team"]
```

what about the other answers?

List Comprehension

```
ncharSubset _ _ [] = []
ncharSubset 0 _ _ = []
ncharSubset n ch inlist
    | c > 0  = head:(ncharSubset (n-c) ch (tail inlist))
    | otherwise  = ncharSubset n ch (tail inlist)
    where c = count ch (head inlist)
```

Prelude>ncharSubset 4 ‘e’ [“table”, “roar”, “press”, “boot”, “team”, “sheet”]

["table","press","team","sheet"]

not quite right..

List Comprehension

```
ncharSubset _ _ [] = []
ncharSubset 0 _ _ = []
ncharSubset n ch inlist
    | c > 0  = head:(ncharSubset (n-c) ch (tail inlist))
    | otherwise  = ncharSubset n ch (tail inlist)
    where c = count ch (head inlist)
```

```
Prelude>ncharSubset 3 'e' ["roar", "press", "boot", "team",
"sheet"]
["press","team","sheet"]
```

not quite right..

List Comprehension

The list recursion strategy is tied to the order of elements in the list.

If word1 has 1 e, word2 has 2 e's and word3 has 3 e's

and the user asks for 5 total e's, the answer clearly needs to be word2 and word3, but by the time we get to word2 or word3, word1 is already selected.

...and there's no backtracking.

List Comprehension

The list recursion strategy is tied to the order of elements in the list.
If word1 has 1 e, word2 has 2 e's and word3 has 3 e's
and the user asks for 5 total e's, the answer clearly needs to be word2 and word3, but by the time we get to word2 or word3, word1 is already selected.
...and there's no backtracking.

there are possible workarounds. One is to use list comprehension recursion instead of regular recursion.
Here, it helps to remember that list comprehensions can work like nested loops.
what's the benefit of a nested loop in this situation?

List Comprehension

[“table”, “roar”, “press”]
    [“roar”, “press”]

[“table”, “roar”, “press”]
    [“table”, “press”]

[“table”, “roar”, “press”]
    [“table”, “roar”]

the outer loop loops over the entire list, while the loop inside loops over the remainder of the list.

This way, each of the elements of the list get to be picked as first.

n! possible ordering of the n list elements will be generated representing all possible orderings.

List Comprehension

[“table”, “roar”, “press”]
    [“roar”, “press”]

[“table”, “roar”, “press”]
    [“table”, “press”]

[“table”, “roar”, “press”]
    [“table”, “roar”]

But which subsets do we need?
well, remember, filtering is what list comprehensions are great at.

the outer loop loops over the entire list, while the loop inside loops over the remainder of the list.
This way, each of the elements of the list get to be picked as first.
n! possible ordering of the n list elements will be generated representing all possible orderings.

List Comprehension

So, instead of backtracking (which doesn't exist in Haskell), we generate all possible orders of selecting elements in, then test to see which orderings fit our solution.

By now, you should be noticing that quite subtly, we've shifted into a "generate-and-test", i.e. nondeterministic, mode of problem solving.

This solution unlike the previous ones, will produce all possible answers rather than just one.

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
outer loop?,
constraint?,
inner loop? ]
```

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
word <- wordlist,
constraint?,
inner loop? ]
```

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
word <- wordlist,
constraint?,
inner loop? ]

count ch word = …
count should be greater than 1
```

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
word <- wordlist,
let c = count ch word,
c > 0,
inner loop? ]

count ch word = sum [1 | c <- word, c==ch]
```

inner loop?

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
word <- wordlist,
let c = count ch word,
c > 0,
inner loop? ]

count ch word = sum [1 | c <- word, c==ch]
```

inner loop: repeat the same actions on the rest of the list.

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
word <- wordlist,
let c = count ch word,
c > 0,
let newlist= delete word wordlist,
inner loop? ]

count ch word = sum [1 | c <- word, c==ch]
delete word wordlist = [w | w<-wordlist, w/=word]
```

inner loop: repeat the same actions on the rest of the list.

List Comprehension

```
ncharSubset_lc n ch wordlist = [ ? |
word <- wordlist,
let c = count ch word,
c > 0,
let newlist= delete word wordlist,
ws <- ncharSubset_lc (n-c) newlist]

count ch word = sum [1 | c <- word, c==ch]
delete word wordlist = [w | w<-wordlist, w/=word]
```

what's the generator function? how should the two loop cursors be mixed to produce the final result?

List Comprehension

```
ncharSubset_lc n ch wordlist = [ word:ws |
word <- wordlist,
let c = count ch word,
c > 0,
let newlist= delete word wordlist,
ws <- ncharSubset_lc (n-c) newlist]

count ch word = sum [1 | c <- word, c==ch]
delete word wordlist = [w | w<-wordlist, w/=word]
```

word is an iterator over a list of words and ws an iterator over a list of subsets of words (partial solutions). word:ws for each instance of w and ws will complete the partial solutions.

List Comprehension

```
ncharSubset_lc n ch wordlist = [ word:ws |
word <- wordlist,
let c = count ch word,
c > 0,
let newlist= delete word wordlist,
ws <- ncharSubset_lc (n-c) newlist]

count ch word = sum [1 | c <- word, c==ch]
delete word wordlist = [w | w<-wordlist, w/=word]
```

What about the base case? (note that this is still a recursive function).

List Comprehension

```
ncharSubset_lc 0 _ _ = [[]]
ncharSubset_lc n ch wordlist = [ word:ws |
word <- wordlist,
let c = count ch word,
c > 0,
let newlist= delete word wordlist,
ws <- ncharSubset_lc (n-c) newlist]

count ch word = sum [1 | c <- word, c==ch]
delete word wordlist = [w | w<-wordlist, w/=word]
```

List Comprehension

```
ncharSubset_lc 0 _ _ = [[]]
ncharSubset_lc n char wordlist = [word:words | word<-
wordlist,
let c =count char word, c > 0,
let newlist=delete word wordlist,
words <- (select (n-c) char newlist)]
```

Questions?

Type Definition (section 5.3)

Here are some type synonym declarations we saw before

```
type RentalCar = (String, String, Int, Float)
type AvailableCars = [RentalCars]
type String = [Chars]
```

Type Definition

Here are some type synonym declarations we saw before

```
type RentalCar = (String, String, Int, Float)
type AvailableCars = [RentalCars]
type String = [Chars]
```

Here's a simple custom type definition:

```
data Move = Rock | Paper | Scissors
```

`Move` is the name of this new type

and it's values are `Rock, Paper` and `Scissors.`

(from the Rock, Paper, Scissors game domain)

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
       deriving (Eq, Show)
```

The values of this class now can be tested for equality and can be printed.

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
       deriving (Eq, Show)
```

The values of this class now can be tested for equality and can be printed.

How does this work exactly?

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
       deriving (Eq, Show)
```

The values of this class now can be tested for equality and can be printed.

How does this work exactly?

The short answer is magic.

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
       deriving (Eq, Show)
```

The values of this class now can be tested for equality and can be printed.

How does this work exactly?
The short answer is magic.
The long answer is that you trust Haskell to figure how to define the necessary function for you new type and Haskell uses its smartness to do it.

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
       deriving (Eq, Show)
```

The values of this class now can be tested for equality and can be printed.

How does this work exactly?
The short answer is magic.
The long answer is that you trust Haskell to figure how to define the necessary function for you new type and Haskell uses its smartness to do it. So the long answer is also magic.

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
       deriving (Eq, Show)
```

If you don’t want to trust Haskell however, because you think you’re smarter, you’re free to do so.

Type Definition

Adding new types to type classes can be done as easily as:

```
data Move = Rock | Paper | Scissors
        deriving (Eq, Show)
```

If you don't want to trust Haskell however, because you think you're smarter, you're free to do so.

instance Show Move where

show Rock = "Rock"

show Paper = "Paper"

show Scissors = "What a weak move!"

Type Definition

Defining a slightly less type involves the use of “constructors” and a potentially number of fields:

```
data People = Person String Int
        deriving (Eq, Show)
```

The type is called People. The (constructor) function ‘Person’ takes two arguments of type String and Int and returns an object of type People.

Type Definition

To make it clear what is meant by the String and Int fields, we can introduce this modification:

```
data People = Person Name Age
           deriving (Eq, Show)

type Name = String
type Age = Int
```

The type is called ‘People’. The (constructor) function ‘Person’ takes two arguments of type String and Int and returns an object of type People.

Type Definition

To make it clear what is meant by the String and Int fields, we can introduce this modification:

```
data People = Person Name Age
           deriving (Eq, Show)

type Name = String
type Age = Int
```

Person :: String -> Int -> People

Type Definition

A less elegant (and less object oriented!) way of creating a simialr type would have been using tuples:

```
type People = (Name,Age)
```

but this gives your program more flexibility:

```
data People = Person Name Age
            deriving (Eq, Show)

type Name = String
type Age = Int
```

Type Definition

For instance, let's say you want to have a way to sort a list of these 'People'. Maybe you want an alphabetical sort or you want a sort based on age. The tuple representation can give you an ordering but it's not necessarily to be the ordering you need for your program.

Nothing stops you from defining your own ordering however.

```
data People = Person Name Age
          deriving (Eq, Show)
```

Type Definition

For instance, let's say you want to have a way to sort a list of these 'People'. Maybe you want an alphabetical sort or you want a sort based on age. The tuple representation can give you an ordering but it's not necessarily to be the ordering you need for your program.

Nothing stops you from defining your own ordering however.

```
data People = Person Name Age
          deriving (Eq, Show, Ord)
or:
instance Ord People where
 ....
```

Type Definition

The Ord class: (From Prelude)

Minimal complete definition

compare | (<=)

Methods

compare :: a -> a -> Ordering

(<) :: a -> a -> Bool

(<=) :: a -> a -> Bool

(>) :: a -> a -> Bool

(>=) :: a -> a -> Bool

max :: a -> a -> a

min :: a -> a -> a

Type Definition

The Ord class: (From Prelude)

Minimal complete definition

compare | (<=)

Methods

compare :: a -> a -> Ordering

(<) :: a -> a -> Bool

(<=) :: a -> a -> Bool

(>) :: a -> a -> Bool

(>=) :: a -> a -> Bool

max :: a -> a -> a

min :: a -> a -> a

```
data Ordering = LT | GT | EQ
```

Type Definition

So in order to have a custom way to order our new type, the minimal work we need to do is implement these two functions:

```
instance Ord People where
    compare person1 person2 = ... (returns LT, GT, or EQ)
    (<=) person1 person2 = ...
```

Type Definition

Mixing the two approaches we've seen so far we can define a multi-faceted type like this:

```
data Shape = Circle Float |
             Rectangle Float Float
             deriving (Eq, Ord, Show)
Circle :: Float -> Shape
Rectangle :: Float -> Float -> Shape

area :: Shape -> Float
area (Circle r) = pi*r*r
area (Rectangle h w) = h*w
```

Questions?

We have covered chapters 5, 6 and 7.

To learn more about Algebraic Types (what we just saw) and/or Type Classes, you can read chapter 13 and 14 of your book.