CPSC 312

Functional and Logic Programming

November 17, 2015

Some Clarifications...

About list ranges and input range inside list comprehensions:

```
[x | x<-[1..10] ] = [1,2,3,4,5,6,7,8,9,10]
```

Specifying x in a list comprehension to be between 1 and 10 is NOT the same as specifying a restriction on a function's input, as in, for instance:

```
f x = div 2 x  -- x shouldn't be zero
```

Should you make that restriction explicit by writing something like this?

```
f x = div 2 x, x<-[1..]
```

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Reason 1)

you'll get a syntax error.

Haskell doesn't understand '<-' outside list comprehension (and some other places which you may see later...). Think of list comprehension as a special bubble inside which you can write such a statement. You can't write in the same syntax you write in there in a plain function environment.

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Reason 2) it makes no sense.

If you run div with zero you'll get an exception. That is the correct behaviour for your f and div. If you want to go the extra mile and return a nice message instead of an exception, you could use conditional expressions:

```
f x
  | x == 0 = “x can’t be zero, you f--- idiot”
  | otherwise =  div 2 x
```

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Except that you can't quite do so in this function, because the function returns numbers, not strings. your error message is a string. that's when you realize, why bother? let it return an exception. And like I said, that'd be sufficient.

You can also choose to throw your own exception using the built-in function error :: [Char] -> a

```
f x
  | x == 0 = error "x can't be zero, you f--- idiot"
  | otherwise =  div 2 x
```

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

in some cases, you may also choose to return a special value instead of an error message (depending on how the function is used, this may make more sense than stopping execution)

```
f x
  | x < 0 = 0
  | otherwise =  div 2 x
```

But in NO case, you explicitly specify a range for the function input. That's just nuts.

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Reason 3)

The command `x<-[0..]` CREATES a NEW VARIABLE, and uses it as an iterator over the given list. You can't write a list range like that for an input variable (which is not a free variable), even inside a list comprehension.

It would be like mixing up the assignment and the equality operators in imperative languages: `x := 2` vs `x == 2`

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Reason 3)

what if you wrote this?

```
f x list = [x | x<- list]
```

Some Clarifications...

`f x = div 2 x, x<-[1..]` ? NO, ABSOLUTELY NOT!!!

Reason 3)

what if you wrote this?

```
f x list = [x | x<- list]
```

The x inside the list comprehension is a different x than the input x. The input x will be effectively ignored. What about this?

```
f x list = [x] ++ [x | x<- list]
```

The scope of the x inside list comprehension is only the list comprehension.

Some Clarifincations...

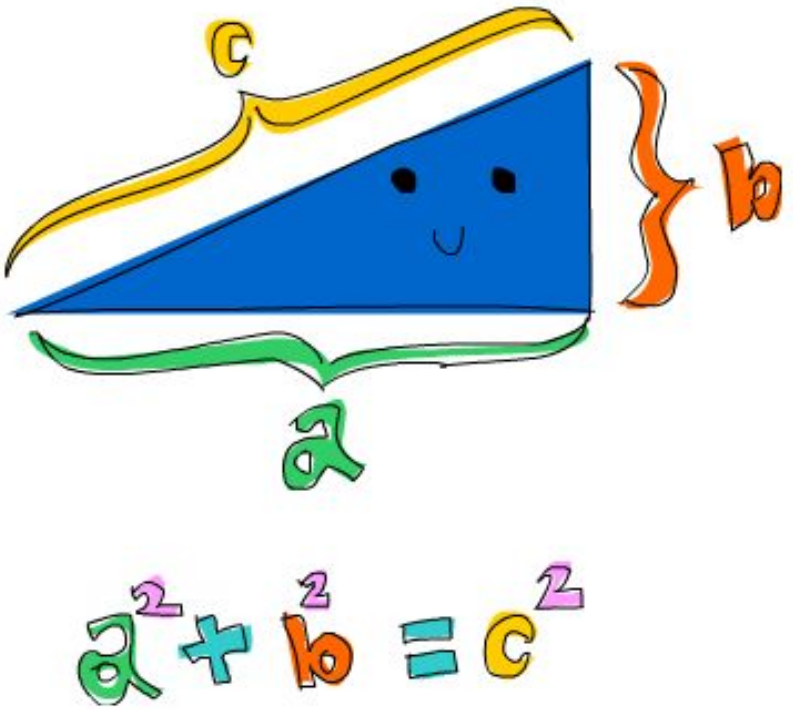
In your assignment, the only check you need to perform on input values is for base cases and recursive cases.

You don’t need to throw any exceptions/errors or handle illegal input at all.

List Comprehension

```
[(a,b,c) | a <- [1..10],
b<- [1..10], c<-[1..10],
a^2 + b^2 == c^2 ,
a+b+c == 24] = [(6,8,10),
(8,6,10)]
```

```
[(a,b,c) | a <- [1..10],
b<- [a..10], c<-[1..10],
a^2 + b^2 == c^2 ,
a+b+c == 24] = [(6,8,10)]
```



List Comprehension

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


> fac 4
[4*x | x <- fac 3]
[4*x | x <- [3*x | x<- fac 2] ]
[4*x | x <- [3*x | x <-[2*x | x<- fac 1]]]
[4*x | x <- [3*x | x <-[2*x | x<- [1]]]]
[4*x | x <- [3*x | x <-[2*1]]]
[4*x | x <- [3*2*1]]
[4*3*2*1]
[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 in the list has 1 e, word2 has 2 e's and word3, 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 with 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 ones are the answer?
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 subsets 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 one, will produce all possible answers not 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) ch 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) ch 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 word 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) ch newlist]
```

wordlist=[“table”, “roar”, “press”], `ch = ‘e’, n= 2`

word = “table”
newlist= [“roar”, “press”]
ws = [“press”]
result = [“table”, “press”]

word= “press”
newlist= [“table”, “roar”]
ws = [“table”]
result = [“press”, “table”]

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) ch newlist]
```

What about the base case? (remember, it 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) ch newlist]
```

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) ch newlist]
```

why not [] instead of [[]] ?

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) ch newlist]
```

why not [] instead of [[]] ?

and what about this base case?

```
ncharSubset_lc n ch []
```

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
      1. {word = "table", c = 1, newlist=["roar", "press"]
                ws<- ncharSubset 1 newlist}
      2. {word = "roar", c = 0, newlist= ["table", "press"]
                ws<- ncharSubset 2 newlist}
      3. {word = "press", c = 1, newlist=["table", "press"]
                ws<- ncharSubset 1 newlist}
```

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
```

1. {word = "table", c = 1, newlist=["roar", "press"]
   ws<- ncharSubset 1 newlist}

   ncharSubset 1 newlist= [word:ws | word <- ["roar", "press"] …

   1. {word= "roar",c > 0? No. throw away. move on to next word}
   2. {word= "press",c = 1, newlist=["roar"], ws<- ncharSubset 0 newlist}

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
```

1. {word = "table", c = 1, newlist=["roar", "press"]
   ws<- ncharSubset 1 newlist}

   ncharSubset 1 newlist= [word:ws | word <- ["roar", "press"] …

   ~~1. {word= "roar",c > 0? No. throw away. move on to next word}~~

2. {word= "press",c = 1, newlist=["roar"], ws<- [[]]}

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
```

1. {word = "table", c = 1, newlist=["roar", "press"]
   ws<- ncharSubset 1 newlist}

ncharSubset 1 newlist= [word:ws | word <- ["roar", "press"] …

~~1. {word= "roar",c > 0? No. throw away. move on to next word}~~

2. {word= "press",c = 1, newlist=["roar"], ws<- [[]]}

ncharSubset 1 ["roar", "press"] = ["press":[]]

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
      1. {word = "table", c = 1, newlist=["roar", "press"]
                ws<- [["press"]] }
```

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
      1. {word = "table", c = 1, newlist=["roar", "press"]
                ws<- [["press"]] }
          first result: "table":["press"]
```

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) ch newlist]
```

```
> ncharSubset_lc 2 'e' ["table", "roar", "press"]
[word:ws | word<-["table", "roar", "press"]
      1. {word = "table", c = 1, newlist=["roar", "press"]
                ws<- [["press"]] }
          first result: "table":["press"]
```

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) ch newlist]
```

```
> ncharSubset_lc 2 ‘e’ [“table”, “roar”, “press”]
[[“table”, “press”], [“press, “table”]]
```

```
> ncharSubset 3 ‘e’ [“table”, “roar”, “press”, “boot”, “team”, “sheet”]
[["table","press","team"],["table","team","press"],["table","sheet"],["press","
table","team"],["press","team","table"],["press","sheet"],["team","table","
press"],["team","press","table"],["team","sheet"],["sheet","table"],["sheet","
press"],["sheet","team"]]
```

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

Using a similar approach how would you define type Bool?

Type Definition

Using a similar approach how would you define type Bool?

data Bool = True | False

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 out how to define the necessary functions for your new type and Haskell uses its smartness 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)
```

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 out how to define the necessary functions for your new type and Haskell uses its smartness to do so. 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 = "Such a weak move!"
```

Type Definition

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

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

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

Type Definition

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

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

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

```
Person :: String -> Int -> People
```

Type Definition

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

Person :: String -> Int -> People
```

What about Rock, Paper, and Scissors? What's their nature?

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

Type Definition

What about Rock, Paper, and Scissors? What's their nature?

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

Also functions (everything is a function!) and also constructor functions (hence, capitalized), though these ones take no input (constact functions).

```
Rock :: People
Paper :: People
Scissors :: People
```

Type Definition

To make it clear what is meant by the String and Int fields, we can introduce a slight modification for readability:

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

type Name = String
type Age = Int
```

Type Definition

A less elegant (and less object oriented!) way of creating a similar type would be using tuples:

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

but this new way 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'd like to be able to sort a list of values of type '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 may not necessarily 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

```
class Eq a => Ord a where          -- meaning: a must be a member of Eq
                                   -- before it can be a member of Ord
```

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

class Eq a => **Ord** a where

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

If defined as

```
type People = (String,Int)
```

comparing two instances of people will follow the default ordering of tuples, which is based on the default ordering of its elements types, from left to right:

`(1,3)<(3,1)` -- first element on the right greater than first on the left

`(1,3)>(1,1)` -- first elements equal, therefore check seconds and so on

The default ordering of Characters is the alphabetical order:

a < z

(1,a) < (1,b)

but

(2,a) > (1,b)

Type Definition

In our custom ordering, we can choose to compare people only based on age or only based on name. Here's based on age, write the one based on name on your own:

```
instance Ord People where
    compare (Person _ age1) (Person _ age2) = compare age1 age2
    (Person _ age1) <= (Person _ age2) = age1 <= age2
```

Type Definition

In our custom ordering, we can choose to compare people only based on age or only based on name. Here's based on age, write the one based on name on your own:

```
instance Ord People where
    compare (Person _ age1) (Person _ age2) = compare age1 age2
    (Person _ age1) <= (Person _ age2) = age1 <= age2
```

What is (Person _ age1) in the function definition?

Type Definition

In our custom ordering, we can choose to compare people only based on age or only based on name. Here's based on age, write the one based on name on your own:

```
instance Ord People where
    compare (Person _ age1) (Person _ age2) = compare age1 age2
    (Person _ age1) <= (Person _ age2) = age1 <= age2
```

What is (Person _ age1) in the function definition?

Pattern matching! (Person x y) or a type constructor is yet another type of value that can act as a pattern in a function's input parameter.

Type Definition

How would you define an ordering for the Move type?

```
instance Ord Move where
    …?
```

Type Definition

How would you define an ordering for the Move type?

```
instance Ord Move where
    compare Rock Scissors = ?
    compare Rock Paper = ?
    compare Paper Scissors = ?
    compare Paper Rock = ?
    compare Scissors Paper = ?
    compare Scissors Rock = ?
```

Type Definition

How would you define an ordering for the Move type?

```
instance Ord Move where
    compare Rock Scissors = GT
    compare Rock Paper = LT
    compare Paper Scissors = LT
    compare Paper Rock = GT
    compare Scissors Paper = GT
    compare Scissors Rock = LT
```

Do you need to define all of these cases or, say, if you've already included `compare Rock Scissors = GT` can you expect Haskell to know `compare Rock Scissors = LT` ?

Type Definition

How would you define an ordering for the Move type?

```
instance Ord Move where
    compare Rock Scissors = GT
    compare Rock Paper = LT
    compare Paper Scissors = LT
    compare Paper Rock = GT
    compare Scissors Paper = GT
    compare Scissors Rock = LT
```

Do you need to define all of these cases or, say, if you've already included `compare Rock Scissors = GT` can you expect Haskell to know `compare Rock Scissors = LT` ?

No, you can't. When you're writing your own definitions, Haskell leaves you alone.

Type Definition

How would you define an ordering for the Move type?

```
instance Ord Move where
    compare Rock Scissors = GT
    compare Rock Paper = LT
    compare Paper Scissors = LT
    -- compare Paper Rock = GT
    -- compare Scissors Paper = GT
    -- compare Scissors Rock = LT

Perlude> compare Paper Rock
*** Exception: inclass.hs:(111,2)-(115,28): Non-exhaustive
patterns in function compare
```

Type Definition

This is the **minimal** definition that Haskell would ask you if you want it to work with all Ord functions:

```
instance Ord Move where
 compare Rock Scissors = GT
 compare Rock Paper = LT
 compare Paper Scissors = LT
 compare Paper Rock = GT
 compare Scissors Paper = GT
 compare Scissors Rock = LT
 Rock <= Paper = True
 Scissors <= Rock = True
 Paper <= Scissors = True
 Paper <= Rock = False
 Scissors <= Paper = False
 Rock <= Scissors = False
```

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.

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