Elm Practice Questions 1

CS 1JC3
Created By Curtis D'Alves - curtis.dalves@gmail.com

Week of Sept 25^{th}, 28^{th}, 29^{th} 2015

Functions

1. Elm is a *functional* language , so we know using functions is an important part of Elm. Considering a simple function to add two numbers as defined in Elm

```
add : Int -> Int -> Int
add x y = x + y
```

Functions

1. Elm is a *functional* language , so we know using functions is an important part of Elm. Considering a simple function to add two numbers as defined in Elm

```
add : Int -> Int -> Int
add x y = x + y
```

2. **add** is the name of are function, and it takes two **arguments** x and y, which are of **type** Int (whole numbers). A few things to note

Functions

1. Elm is a *functional* language , so we know using functions is an important part of Elm. Considering a simple function to add two numbers as defined in Elm

```
add : Int -> Int -> Int
add x y = x + y
```

2. **add** is the name of are function, and it takes two **arguments** x and y, which are of **type** Int (whole numbers). A few things to note
   1. Now instead of writing $5 + 6$ in our code, we can now write **add 5 6**
   2. The **type signature** isn't necessary (Elm can *infer* the type of functions you write, they are mostly for the readers benifit)

Types

1. A **type** is a name for a collection of related values. For example, Elm has a built in type **Bool** which is composed of two values: **True** or **False**. You can define the **Bool** type in Elm yourself like so

```
type Bool = True | False
```

Types

1. A **type** is a name for a collection of related values. For example, Elm has a built in type **Bool** which is composed of two values: **True** or **False**. You can define the **Bool** type in Elm yourself like so

```
type Bool = True | False
```

2. Types in Elm defined with the **type** keyword consist of a **data constructor** (i.e **Bool**) and **value constructors** (i.e **True** and **False**).

Types

1. **Value constructors** can take other types as arguments, consider the following

```
type ListI = Cons Int ListI | Nil
```

2. In this example **ListI** is a data constructor and **Cons** and **Nil** are value constructors. Can you use **ListI** to alternatively represent the List value $[1, 2, 3]$?

Types

1. **Value constructors** can take other types as arguments, consider the following

```
type ListI = Cons Int ListI | Nil
```

2. In this example **ListI** is a data constructor and **Cons** and **Nil** are value constructors. Can you use **ListI** to alternatively represent the List value $[1, 2, 3]$?

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

3. **Cons** takes two arguments, an **Int** (whole number) and another **ListI**

Lists

1. A List is a sequence of values of the **same type**. You can have lists of any type, even another list!

```
[[1.0,2.0,3.0],[4.0,5.0,6.0]]
```

Lists

1. A List is a sequence of values of the **same type**. You can have lists of any type, even another list!

```
[[1.0,2.0,3.0],[4.0,5.0,6.0]]
```

2. The above is a List of Lists of Float (decimal numbers). We specify it's type in elm as **List (List Float)**

Lists

1. A List is a sequence of values of the **same type**. You can have lists of any type, even another list!

```
[[1.0,2.0,3.0],[4.0,5.0,6.0]]
```

2. The above is a List of Lists of Float (decimal numbers). We specify it's type in elm as **List (List Float)**
3. Consider this other example, a List of **Form** that you could render to the screen with the function **collage**

```
[filled red <| circle 50,
 text <| formString "Hello"]
```

Super Important List Functions

1. The *infix* function $(::) : a- > \text{List a}- > \text{List a}$ takes a value and puts it into a list of the same value, i.e

```
1 :: [2,3]          =     [1,2,3]
True :: [False]     =     [True,False]
(circle 50) :: [] =     [circle 50]
```

Super Important List Functions

1. The *infix* function (::) : $a->$ List $a->$ List a takes a value and puts it into a list of the same value, i.e

```
1 :: [2,3]          =     [1,2,3]
True :: [False]     =     [True,False]
(circle 50) :: [] =    [circle 50]
```

2. The *infix* function (++) : List $a->$ List $a->$ List a takes two lists and put them together, i.e

```
[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]
```

Super Important List Functions

1. The *infix* function $(::) : a- > \text{List a}- > \text{List a}$ takes a value and puts it into a list of the same value, i.e

```
1 :: [2,3]          =    [1,2,3]
True :: [False]     =    [True,False]
(circle 50) :: [] =    [circle 50]
```

2. The *infix* function $(++) : \text{List a}- > \text{List a}- > \text{List a}$ takes two lists and put them together, i.e

```
[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]
```

3. **Note:** there are plenty of useful List functions in the List module, but you need to *import* it by adding the following at the top of your code

```
import List exposing (..)
```

Pattern Matching

1. Frequently, you will need to write functions that use the *case* construct to *pattern match* a variable to a value, for example

```
not : Bool -> Bool
not b = case b of
          True -> False
          False -> True
```

Pattern Matching

1. Frequently, you will need to write functions that use the *case* construct to *pattern match* a variable to a value, for example

```
not : Bool -> Bool
not b = case b of
            True -> False
            False -> True
```

2. Pattern Matching works on all types, but be careful to cover all possible values of that type. Whats wrong with the following function?

```
dumb : Int -> Bool
dumb x = case x of
            1 -> True
            2 -> False
```

Pattern Matching

1. The _ and **otherwise** keyword be used to as a match for everything (Note: order is important in when pattern matching)

```
smart : Int -> Bool
smart x = case x of
                1 -> True
                otherwise -> False
```

Pattern Matching

1. The _ and **otherwise** keyword be used to as a match for everything (Note: order is important in when pattern matching)

```
smart : Int -> Bool
smart x = case x of
            1 -> True
            otherwise -> False
```

2. We can use pattern matching to pull values out of a **value constructor**. Consider the following function that sums the last data type form before

```
sumList xs = case xs of
                Cons x list -> x + sum list
                Nil -> 0
```

Pattern Matching With Lists

1. Pattern matching is extremely important for *traversing* lists. Consider the following function that sums a list of **Int**

```
sum : List Int -> Int
sum nums = case nums of
             (x::xs) -> x + sum xs
             [] -> 0
```

Pattern Matching With Lists

1. Pattern matching is extremely important for *traversing* lists. Consider the following function that sums a list of **Int**

```
sum : List Int -> Int
sum nums = case nums of
             (x::xs) -> x + sum xs
             [] -> 0
```

2. Consider how this function is evaluated for the example sum [1,2]

```
sum [1,2]
  => 1 + sum [2]
  => 1 + 2 + sum []
  => 1 + 2 + 0 ... you can figure out the rest
```

Good Advice

1. Go to `http://elm-lang.org/docs` and study the links Syntax, Core Language, Model The Problem
2. Reference the core libraries for useful functions while programming (`http://package.elm-lang.org/packages/elm-lang/core/2.1.0/`)
3. There aren't many practice problems available for Elm, so look up Haskell practice problems on google (the solutions will be very similar as Elm is based on Haskell)
4. Take notes during lectures, bring them to drop in centre with questions (times on avenue)

Problem 1

Write a function **last** that takes a list and returns it's last element

```
last : List a -> a
```

Problem 1

Write a function **last** that takes a list and returns it's last element

```
last : List a -> a
```

Solution

```
last list = case list of
              [x] -> x
              (x::xs) -> last xs
```

Problem 2

Write a function **butLast** that takes a list and returns it's last **but one** element

```
butLast : List a -> a
```

Problem 2

Write a function **butLast** that takes a list and returns it's last **but one** element

```
butLast : List a -> a
```

Solution

```
butLast list = case list of
                (x::y::[]) -> x
                (x::xs) -> butLast xs
```

Problem 3

Write a function **kElement** that takes a list and returns it's k^{th} element

```
kElement : Int -> List a -> a
```

Problem 3

Write a function **kElement** that takes a list and returns it's k^{th} element

```
kElement : Int -> List a -> a
```

Solution

```
kElement k (x::xs) = case k of
                       1 -> x
                       _ -> kElement (k-1) xs
```

Problem 4

Write a function **length** that returns the number of elements in a list (the length of the list)

```
length : List a -> Int
```

Problem 4

Write a function **length** that returns the number of elements in a list (the length of the list)

```
length : List a -> Int
```

Solution

```
length list = case list of
                  [] -> 0
                  (x::xs) -> 1 + length xs
```

Problem 5

Write a function **reverse** that reverses a list

```
reverse : List a -> List a
```

Problem 5

Write a function **reverse** that reverses a list

```
reverse : List a -> List a
```

Solution

```
reverse list = case list of
               (x::xs) -> reverse xs ++ [x]
               [] -> []
```

Problem 6

Write a function **flatten** that takes a list of lists and *iten* flattens it into one list

```
flatten : List (List a) -> List a
```

Problem 6

Write a function **flatten** that takes a list of lists and *iten* flattens it into one list

```
flatten : List (List a) -> List a
```

Solution

```
flatten lists = case lists of
                    (l::ls) -> l ++ flatten ls
                    [] -> []
```

Problem 7

Write a function **drop** that removes the first **n** elements of a list (assume the list $\geq n$)

```
drop : Int -> List a -> List a
```

Problem 7

Write a function **drop** that removes the first **n** elements of a list (assume the list $\geq n$)

```
drop : Int -> List a -> List a
```

Solution

```
drop k (x::xs) = case k of
                   1 -> xs
                   _ -> drop (k-1) xs
```

Problem 8

Write a function **splits** that splits a list into two

```
split : List a -> (List a,List a)
```

Problem 8

Write a function **splits** that splits a list into two

```
split : List a -> (List a,List a)
```

Solution

```
split list = let
    split' (ls,(r::rs)) = if length ls >= length (r::rs)
                           then (ls,r::rs)
                           else split' (r::ls,rs)
  in split' ([],list)
```

Problem 9

Write a function **slice** that extracts the i^{th} to j^{th} elements from a list (assume those elments always exist)

```
slice : Int -> Int -> List a -> List a
```

Problem 9

Write a function **slice** that extracts the i^{th} to j^{th} elements from a list (assume those elments always exist)

```
slice : Int -> Int -> List a -> List a
```

Solution

```
slice list i k = let
    slice' (l1,l2) x y =
        case (l1,l2,x,y) of
            (ss,_,0,0) -> ss
            (ss,l::ls,0,k) -> slice' (ss++[l],ls,0,k-1)
            (ss,l::ls,i,k) -> slice' (ss,ls,i-1,k)
  in slice' ([],list) i k
```

Problem 10

Write a function **removeAt** that removes the k^{th} element of a list (assume it exists)

```
removeAt : Int -> List a -> List a
```

Problem 10

Write a function **removeAt** that removes the k^{th} element of a list (assume it exists)

```
removeAt : Int -> List a -> List a
```

Solution

```
removeAt k list = case (k,list) of
                    (1,l::ls) -> ls
                    (i,l::ls) -> l :: removeAt (i-1,ls)
```

Problem 11

Write a function **rotate** that rotates a list n places to the left, i.e rotate 2 $[1,2,3,4,5] = [3,4,5,1,2]$

```
rotate : Int -> List a -> List a
```

Problem 11

Write a function **rotate** that rotates a list *n* places to the left, i.e rotate 2 $[1, 2, 3, 4, 5] = [3, 4, 5, 1, 2]$

```
rotate : Int -> List a -> List a
```

Solution

```
rotate n xs = case (xs,n) of
              (ys,0) -> ys
              (y:ys,n) -> rotate (n-1) (ys ++ [y])
```

Problem 12

Write your own type to represent binary trees with String labeled nodes and create a tree with 4 nodes

Problem 12

Write your own type to represent binary trees with String labeled nodes and create a tree with 4 nodes

Solution

```
type BTree = Node String BTree BTree | Leaf

tree = Node "A" (Node "B" Leaf Leaf)
                (Node "C" (Node "D" Leaf Leaf) Leaf)
```

Problem 13

Write a function **flattenT** that flattens your tree into a list

```
flattenT : BTree -> List String
```

Problem 13

Write a function **flattenT** that flattens your tree into a list

```
flattenT : BTree -> List String
```

```
flattenT tree =
    case tree of
      BTree s t1 t2 -> s :: flattenT t1 ++ flattenT t2
      Leaf -> []
```

Problem 14

Ever heard of Peano Numbers?
They're this weird way of representing Natural numbers (whole numbers > 0) that computer scientists care about for some reason.

Peano numbers consist of a **zero value** and a function **successor** that takes a Peano number and and returns another one (it's sucessor $(+1)$).

Create a type in Elm to represent Peano numbers

Problem 14

Ever heard of Peano Numbers?
They're this weird way of representing Natural numbers (whole numbers > 0) that computer scientists care about for some reason.

Peano numbers consist of a **zero value** and a function **successor** that takes a Peano number and and returns another one (it's sucessor $(+1)$).

Create a type in Elm to represent Peano numbers

Solution

```
type Peano = Succ Peano | Zero
```

Problem 15

Write functions to add and subtract the Peano numbers you defined

Problem 15

Write functions to add and subtract the Peano numbers you defined

Solution

```
add p1 p2 = case (p1,p2) of
                (x,Zero) -> x
                (x,Succ y) -> add (Succ x) y
sub p1 p2 = case (p1,p2) of
                (x,Zero) -> x
                (Zero,y) -> Zero
                (Succ x, Succ y) -> sub x y
```

Problem 16

Write a function to convert a Peano number to an Int

```
peanoToInt : Peano -> Int
```

Problem 16

Write a function to convert a Peano number to an Int

```
peanoToInt : Peano -> Int
```

Solution

```
peanoToInt ps = case ps of
                  Zero -> 0
                  Succ p -> 1 + peanoToInt p
```

Problem 17

Write a function to convert an Int number to a Peano number (assume the Int is ≥ 0)

```
intToPeano : Int -> Peano
```

Problem 17

Write a function to convert an Int number to a Peano number (assume the Int is ≥ 0)

```
intToPeano : Int -> Peano
```

Solution

```
intToPeano k = case k of
                 0 -> Zero
                 i -> Succ (intToPeano (i-1))
```