Elm Practice Questions 2

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

Week of Oct. 9^{th}, 2015

Type Declarations

1. Elm features a handful of built in types (*Int*,*Bool*,*Char*,etc). Sometimes we wish to give types more descriptive names to add *context* to our type signatures, to do this we have the **type alias** keyword

```
type alias Pos2D = (Float,Float)
type alias Name = String
```

Type Declarations

1. Elm features a handful of built in types (*Int*, *Bool*, *Char*, etc). Sometimes we wish to give types more descriptive names to add *context* to our type signatures, to do this we have the **type alias** keyword

```
type alias Pos2D = (Float,Float)
type alias Name = String
```

2. Remember from the last slideset, without the **alias** keyword we must construct types differently using **value constructors**. These types are known as **Abstract Data Types**, like this type for constructing a *List* of *Int*

```
type List = Cons Int List | Nil
```

Parameterized Abstract Data Types

1. Consider the built in *List* type, it can have many different types of values, not just Int. To accomplish this with out *List* type, we add a parameter like so

```
type List a = Cons a List | Nil
```

Parameterized Abstract Data Types

1. Consider the built in *List* type, it can have many different types of values, not just Int. To accomplish this with out *List* type, we add a parameter like so

```
type List a = Cons a List | Nil
```

2. Now our *List* can hold any types of values. How would you create a parametrized *Binary Tree*?

Parameterized Abstract Data Types

1. Consider the built in *List* type, it can have many different types of values, not just Int. To accomplish this with out *List* type, we add a parameter like so

```
type List a = Cons a List | Nil
```

2. Now our *List* can hold any types of values. How would you create a parametrized *Binary Tree*?

```
type BTree a = Node a BTree BTree | Leaf a
```

Polymorphism

1. The real power behind *parameterized data types* shines through when we define functions to act on them. Consider the following function which returns the length of a *List*

```
length list = case list of
                Cons x list' -> 1 + length list'
                Nil -> 0
```

2. What is the type of this function?

Polymorphism

1. The real power behind *parameterized data types* shines through when we define functions to act on them. Consider the following function which returns the length of a *List*

```
length list = case list of
                Cons x list' -> 1 + length list'
                Nil -> 0
```

2. What is the type of this function?

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

What are the implications of this?

Mapping

1. *Higher Order Functions* are functions that take other functions as arguments!
2. One of the most important *Higher Order Functions* is **map**, which applies a function to every element in a data structure. Consider map as we would define it on lists

Mapping

1. *Higher Order Functions* are functions that take other functions as arguments!
2. One of the most important *Higher Order Functions* is **map**, which applies a function to every element in a data structure. Consider map as we would define it on lists

```
map : (a -> b) -> List a -> List b
map f list = case list of
                x::xs -> (f x) :: map f xs
                [] -> []
```

Mapping

1. *Higher Order Functions* are functions that take other functions as arguments!
2. One of the most important *Higher Order Functions* is **map**, which applies a function to every element in a data structure. Consider map as we would define it on lists

```
map : (a -> b) -> List a -> List b
map f list = case list of
                 x::xs -> (f x) :: map f xs
                 [] -> []
```

3. What is the result of **map (+1) [1,2,3,4]**?

Mapping

1. *Higher Order Functions* are functions that take other functions as arguments!
2. One of the most important *Higher Order Functions* is **map**, which applies a function to every element in a data structure. Consider map as we would define it on lists

```
map : (a -> b) -> List a -> List b
map f list = case list of
                 x::xs -> (f x) :: map f xs
                 [] -> []
```

3. What is the result of **map (+1) [1,2,3,4]**?

```
map (+1) [1,2,3,4] == [2,3,4,5]
```

Folding

1. Many list functions can be defined using the following simple pattern of recursion

```
f list = case list of
            x::xs -> x (OP) f xs
            []    -> id
```

Folding

1. Many list functions can be defined using the following simple pattern of recursion

```
f list = case list of
              x::xs -> x (OP) f xs
              []    -> id
```

2. **(OP)** represents any *binary* function
3. **(id)** is some value of the same type as in the list, usually chosen to be the *identity* of **(OP)**

Folding

For Example:

```
sum list = case list of
           x::xs -> x + f xs    -- OP = +
           [] -> 0              -- id = 0

product list = case list of
               x::xs -> x * f xs -- OP = *
               [] -> 1           -- id = 1

and list = case list of
           x::xs -> x && f xs   -- OP = &&
           [] -> True           -- id = True
```

Folding

1. **Folds** are functions that encapsulate this pattern of recursion given **(OP)** and **(id)** and a *List* as arguments.
2. We can define a **right fold** like so

```
foldr f v list = case list of
                   x::xs -> f x (foldr f v xs)
                   [] -> v
```

3. Why is this a **right** fold? What would a **left** fold look like?

Folding

1. **Folds** are functions that encapsulate this pattern of recursion given **(OP)** and **(id)** and a *List* as arguments.
2. We can define a **right fold** like so

```
foldr f v list = case list of
                   x::xs -> f x (foldr f v xs)
                   [] -> v
```

3. Why is this a **right** fold? What would a **left** fold look like?

```
foldl f v list = case list of
                   x::xs -> f (foldl f v xs) xs
                   [] -> v
```

Folding

It helps to thinks of folds in the following manner

```
sum [1,2,3]

 = foldr (+) 0 [1,2,3]

 = foldr (+) 0 (1::(2::(3::[])))

 = 1+(2+(3+0))    -- replace (::) with (+)

 = 6
```

Lambda Expressions

1. **Lambda Expressions** (also known as Anynomous Functions) allow you to define a nameless function to pass as an argument to a *higher-order function*
2. For example, instead of

```
add x y = x + y
sum list = foldr (add) 0 list
```

Lambda Expressions

1. **Lambda Expressions** (also known as Anynomous Functions) allow you to define a nameless function to pass as an argument to a *higher-order function*
2. For example, instead of

```
add x y = x + y
sum list = foldr (add) 0 list
```

3. We write

```
sum list = foldr (\x y -> x+y) 0 list
```

4. This is even more useful then it may first appear, consider the fact that lambda expressions are defined in **local scope**

Problem 1

Refer to the Elm Practice 1 Slides, do Problems 4,5,6 (length, reverse, flatten) using folds from List (import List and use foldl or foldr)

Problem 1

Refer to the Elm Practice 1 Slides, do Problems 4,5,6 (length, reverse, flatten) using folds from List (import List and use foldl or foldr)

Solution

```
length list = foldl (\x ys -> ys + 1) 0 list

reverse list = foldl (\x ys -> [x] ++ ys) [] list

flatten list = foldl (\x ys -> ys ++ x) [] list
```

Problem 2

Try defining **map** using a **fold**. (**Hint**: use a lambda expression when calling fold)

Problem 2

Try defining **map** using a **fold**. (**Hint**: use a lambda expression when calling fold)

Solution

```
map f list = foldr (\x rest -> (f x) :: rest) [] list
```

Bonus Thought For Algorithm Buffs: try using foldl instead of foldr. What happens? Why?

Problem 3

Define a type for representing Trees (**NOT a Binary Tree**, but one with an arbitrary amount of children).
It should be parameterized to hold any types of values

Problem 3

Define a type for representing Trees (**NOT a Binary Tree**, but one with an arbitrary amount of children).
It should be parameterized to hold any types of values

Solution

```
type Tree a = Node a (List (Tree a))

exampleTree = Node 1 [Node 2 [Node 5 [Node 8 []]],
                      Node 3 [Node 6 []],
                      Node 4 [Node 7 []]
                      ]
```

Extra Challenge: Try drawing exampleTree

Problem 4

Define a **map** function for your Tree type (hint, use the List map in your function)

```
mapT : (a -> b) -> Tree a -> Tree b
```

Problem 4

Define a **map** function for your Tree type (hint, use the List map in your function)

```
mapT : (a -> b) -> Tree a -> Tree b
```

Solution

```
mapT f tree = case tree of
                Tree x [] -> Tree (f x) []
                Tree x ts -> Tree (f x) (map (mapT f) ts)
```

Problem 5

Define a **fold** function for your Tree type (hint, use the List map and foldr in your function)

Problem 5

Define a **fold** function for your Tree type (hint, use the List map and foldr in your function)

Solution

```
foldT f v tree = case tree of
      Node x [] -> f x v
      Node x ts -> f x <| foldr f v (map (foldT f v) ts)
```

Problem 6

Define a **fold** function for your Tree type (**Hint**: use the List map and foldr in your function)

Problem 6

Define a **fold** function for your Tree type (**Hint**: use the List map and foldr in your function)

Solution

```
foldT f v tree = case tree of
        Node x [] -> f x v
        Node x ts -> f x <| foldl f v (map (foldT f v) ts)
```

Bonus Thought For Algorithm Buffs: is your fold **breadth** first or **depth** first?

Problem 7

Define **length**, **sum**, **product** functions for your Tree (**Hint**: use your fold)

Problem 7

Define **length**, **sum**, **product** functions for your Tree (**Hint**: use your fold)

Solution

```
treeLength tree = foldT (\_ ys -> 1 + ys) 0 tree

treeSum tree = foldT (+) 0 tree

treeProduct tree = foldT (*) 1 tree
```