Functional Programming Skills
Assignment 1

Arthur Nunes-Harwitt

Each question is worth 10 points.

Explicitly write the type of each function.
Following each function there should be commented out expressions to test the function.

1. Write a function `second` that takes a list as input. It should return the second element of the list. Do *not* use any type class. Do not worry about error checking. The time complexity should be $\mathcal{O}(1)$.
   Example:
   `second [4,5,6]` $\leadsto$ `5`

2. Write a predicate (i.e., a function that returns true or false) `singleton` that takes a list as input. It should return `True` if the list has exactly one element, and `False` otherwise. Do *not* use any type class. The time complexity should be $\mathcal{O}(1)$.
   Examples:
   `singleton []` $\leadsto$ `False`
   `singleton [5]` $\leadsto$ `True`
   `singleton [5,6]` $\leadsto$ `False`

3. Haskell has the function `concat` that takes a list of lists and concatenates all of them. (The type of `concat` is actually a little more complicated.) Write a structurally recursive function `catAll` that also concatenates a list of lists. Do *not* use any type class.
   Examples:
   `catAll []` $\leadsto$ `[]`
   `catAll [[1,2], [3,4,5]]` $\leadsto$ `[1,2,3,4,5]`
   `catAll ["a", "bc", "d", "efg"]` $\leadsto$ `"abcdefg"`

4. Write a function `index` that has inputs `x` and `lst`, where `lst` is a list of elements of the same type as `x`; the type of `x` should have equality. (Use the `Eq` type class.) It should return `Just` n, where n, an `Int`, is the zero based location of the first occurrence of `x` in `lst`, or `Nothing` if there is no occurrence. (Remember that `Just` and `Nothing` are constructors for the `Maybe` type.) The time complexity should be $\mathcal{O}(n)$.
   Example:
   `index 'x' "qrsxyz"` $\leadsto$ `Just 3`
   `index 'x' "qrsyz"` $\leadsto$ `Nothing`

5. Consider the following function.

```
evenSquares :: [Integer] -> [Integer]
evenSquares lst = [x*x | x <- lst, even x]
```

Example:
`evenSquares [1..10]`$\leadsto$`[4,16,36,64,100]`

This function is implemented using a list comprehension. Write a function `evenSquares'` that does the same thing, but does *not* use list comprehensions. Use `map` and `filter` instead.

6. Write a function `insertionSort` that has input `lst`, where `lst` is a list of elements that can be compared. (Use the `Ord` type class.) It should return a list of the same length and with the same elements as `lst` but they should be in ascending order. The insertion-sort algorithm is structurally recursive: sort the tail of the list and then insert the head of the list in its proper place. Write `insertionSort` and its helper function `insert` using *explicit recursion*. The time complexity should be $\mathcal{O}(n^2)$.
Example:
`insertionSort [5,1,4,3,2,6,5]` $\leadsto$ `[1,2,3,4,5,5,6]`

7. Implement insertion-sort again. Write `insertionSortH` using `foldr` instead of explicit recursion. (You should make use of `insert` from the previous question.)

8. Write a function `perm` that has input `lst`. It should return a list of all the permutations of `lst`. You should *not* restrict the type variable with a type-class. (HINT: You will need at least one helper function.)
Examples:
`perm [2,3,4]`$\leadsto$`[[2,3,4],[3,2,4],[3,4,2],[2,4,3],[4,2,3],[4,3,2]]`
`perm "abc"`$\leadsto$`["abc","bac","bca","acb","cab","cba"]`

9. This problem has several parts.

   (a) Define a new data structure `Peano` to represent numbers in unary. It consists of two choices: `Zero` or `S`. The choice `S` has one part of type `Peano`.

   (b) Use structural recursion to define the function `add`, which implements addition for Peano numbers. (You may *not* use conversion functions in your implementation.)
   Example:
   `add (S (S Zero)) (S (S (S Zero)))`$\leadsto$`S (S (S (S (S Zero))))`

   (c) Use structural recursion to define the function `mult`, which implements multiplication for Peano numbers. (You may *not* use conversion functions in your implementation.)
   Example:
   `mult (S (S Zero)) (S (S (S Zero)))`$\leadsto$`S (S (S (S (S (S Zero)))))`

   (d) Using pattern matching and the multiplication operator defined above, write the factorial function `fact` for Peano numbers. (You may *not* use conversion functions in your implementation.)
   Example:
   `fact (S (S (S Zero)))`$\leadsto$`S (S (S (S (S (S Zero)))))`

Graduate Problems/Undergraduate Extra Credit

1. (a) Define a function `meaning` specified as follows.

$$
\begin{array}{rcl}
[\![\mathsf{Zero}]\!] & = & \lambda s.\lambda z.z \\
[\![(\mathsf{S}\ n)]\!] & = & \lambda s.\lambda z.(s\ ([\![n]\!]\ s\ z))
\end{array}
$$

   (b) Use `meaning` to write a function `fromPeano` that converts Peano numbers to Haskell integers.

2. Here we will write code so that we will be able to write an FP-like definition of factorial. Recall the FP formulation.

```
zero = (= @ [id, %0])
decr = (- @ [id, %1])
fact = (zero -> %1; * @ [id, fact@decr])
```

   We can express the same idea in Haskell as follows.

```
zero = equals . (pair id (const 0))
decr = difference . (pair id (const 1))
fact = imp zero (const 1) (product . (pair id (fact . decr)))
```

   - The FP `@` is function composition, which is written in Haskell as `.`.
   - The FP `id` is the identity function, which is written in Haskell as `id`.
   - The FP `%` is the constant function constructor, which is written in Haskell as `const`.
   - The FP `*` is the multiplication operator, which is written in Haskell as `product`.

   There are still several functions that need to be written.

   (a) Write a function `equals` that takes a list and returns a Boolean indicating whether or not all the elements of the list are the same.

   (b) Write a function `pair` that takes two functions and returns a function that maps a value to lists of length two, where the elements of the list are the functions applied to the value.

   (c) Write a function `difference` that takes a non-empty list of numbers and subtracts all the elements in the tail from the first element.

   (d) Write a function `imp` that takes three functions, the first of which returns a Boolean, and returns a function that maps a value to a conditional involving those functions.