CS225 Programming Languages: Homework 2

Due Date: 10/14/21 by 11:59PM

Submission: Your submission should be an executable OCaml code file. Please complete the template `hw2-template.ml`, replacing `template` with your last name in your submission file. All submissions must be made electronically in Blackboard.

Problem 1 ***(15 points).*** Using pattern matching, write a function called `dayname` with type `int -> string`, such that `dayname n` evaluates to the name of the 0-indexed `n`th day of the week– i.e., such that:

```
dayname 0  ⇓  "Monday"
dayname 1  ⇓  "Tuesday"
dayname 2  ⇓  "Wednesday"
dayname 3  ⇓  "Thursday"
           ⋮
```

As long as you account for the seven days of the week, your match need not be exhaustive on `int`s, but this should be specified in the comments (which must be provided to document your code).

Problem 2 ***(15 points).*** Let a *month offset* be the number of days of the week after Monday that a given month begins; for example, February 1 falls on a Saturday this year, so its month offset is 5. Using the `dayname` function defined in the previous problem and the `List.map` function (which is the built-in version of `map` discussed in class), write a function called `add_daynames` that takes a month offset and a list of dates d of that month, and returns another list consisting of pairs (d, n), where n is the day of the week corresponding to d. We can specify this function as follows:

```
(*
  add_daynames : int -> int list -> (int * string) list
  in : a month offset o, list of calendar dates
       l = [d1,...,dk]
  out : list [(d1,dn1),...,(dk,dnk)], where for all
        i in {1,...,k},  dni = dayname di given o
*)
```

For example:

```
               add_daynames 5 [1;2;3;10;11]
                            ⇓
   [(1, "Saturday"); (2, "Sunday"); (3, "Monday");
      (10, "Monday"); (11, "Tuesday")]
```

Hint: If you `map`, `add_daynames` can be written on a single line of code.

Problem 3 ***(15 points).*** Another useful polymorphic list function is `member`, with the following type signature:

```
member :  'a -> 'a list -> bool
```

The specification of `member` is as follows:

```
(*
   member : 'a -> 'a list -> bool
   in : value v, list l
   out : true iff v is in l
*)
```

In other words, `member` checks whether a given element is a member of a list. For example:

member 1 [1;5;7] $\Downarrow$ true
member 'a' ['c';'d';'b'] $\Downarrow$ false

In this problem, you are to define `member` as a recursive function according to these specifications.

Problem 4 ***(15 points).*** Write a recursive function `exists` with the following specification:

```
(*
    exists :  ('a -> bool) -> 'a list -> bool
    in :  a predicate p on values v :  'a
    out :  a function f :  'a list -> bool such
           that f(l) is true iff p(x) for some x in l
*)
```

Problem 5 ***(15 points).*** Imagine that we define english and american measurement datatypes, using variants, as follows:

```
type emeasures = Meter of float | Liter of float |
                 Centigrade of float
type ameasures = Feet of float | Gallon of float |
                 Fahrenheit of float
```

Given these datatypes, define a function `conversion` that matches the following specification:

```
(*
  conversion : emeasures -> ameasures
  in : english measurement x
  out : conversion of x into corresponding american
        measurement
*)
```

Problem 6 ***(25 points).*** Let the `tree` datatype be as defined in class:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Now, you may recall working with binary search trees in previous classes, where keys were integers and lookup and insert involved comparing integers and ran in time logarithmic in the size of the tree. However, just as we can generalize our definition of trees with polymorphism, we can also generalize our notion of binary search trees. In this problem, you will define insert and lookup functions that work on any sort of binary search tree. To begin, we define the notion of a *total ordering* as follows:

Definition 1.1 *Let* `lt` *be a binary relation on pairs of type* τ*– that is, we suppose that* `lt` *is a function with type* `(`τ `*` τ`) -> bool`*. Then* `lt` *is a* total order *on* τ *iff:*

1. `lt(x,y)` $\Downarrow$ `true` *and* `lt(y,z)` $\Downarrow$ `true` *imply that* `lt(x,z)` $\Downarrow$ `true`
2. *If* $x \neq y$ *then either* `lt(x,y)` $\Downarrow$ `true` *or* `lt(y,x)` $\Downarrow$ `true`
3. *There exists no* x *such that* `lt(x,x)` $\Downarrow$ `true`

For example, the operator < is a total order on `int`. Now, we can define the *BST* (binary search tree) property as follows:

Definition 1.2 *Let* `lt` *be a total order on* τ*; then for any* `t` *such that* `t : τ tree`*, we say that* `t` *possesses the BST property iff for every node* `n` *in* `t`*, if* `v` *is the value stored at* `n`*, then for every value* `vl` *in the left subtree of* `n` *we have* `lt(vl,v)` $\Downarrow$ `true`*, and for every value* `vr` *in the right subtree of* `n` *we have* `lt(v,vr)` $\Downarrow$ `true`*.*

For example, since < is a total ordering on type `int`, the following tree possesses the BST property:

```
Node(
    Node(
        Leaf,
        8,
        Leaf
    ),
    10,
    Node(
        Node(Leaf,15,Leaf)
        42,
        Leaf
    )
)
```

Your task is to complete the following functions `lookup` and `insert` according to specifications; *your implementation of each should run in* $\log n$ *time on a balanced tree possessing* n *nodes (assuming that* `lt` *runs in constant time)*:

```
(*
  lookup : ('a * 'a -> bool) -> 'a -> 'a tree -> bool
  in : total order lt, element x, tree t possessing
       BST property
  out : true iff x is in t
*)
let rec lookup lt x t = <COMPLETE ME>

(*
  insert : ('a * 'a -> bool) -> 'a -> 'a tree -> 'a tree
  in : total order lt, element x, tree t possessing
       BST property
  out : tree t' which is t with x inserted such that t'
        possesses BST property
*)
let rec insert lt x t = <COMPLETE ME>
```

For example, letting `t` be the example tree above, we can insert and lookup values as follows:

```
lookup (fun (x,y) -> x < y) 8 t
insert (fun (x,y) -> x < y) 24 t
```

Problem 7 ***(15 points: Graduate Students Only).*** Research the fold functionality on lists, and use either `List.fold_left` or `List.fold_right` to redefine both `member` and `exists`, so that neither is declared recursively. The following links are excellent resources:

https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/hop/fold_right.html

https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/hop/fold_left.html