Instructions

Homework 3

This assignment deals primarily with types and type classes in Haskell, as well as a Prolog exercise.

The questions 1 to 6 all have to be solved with Haskell, 7 to 10 in Prolog.

1. Suppose we have the following type of binary search trees with keys of type k and values of type v:
2. ```
   data BST k v = Empty |
                  Node k v (BST k v) (BST k v)
   ```

   Write a val function that returns Just the stored value at the root node of the tree. If the tree is empty, val returns Nothing.

   ```
   val :: BST k v -> Maybe v
   ```

   Example:

   ```
   > val Empty
   Nothing
   > val (Node 0 23 Empty Empty)
   Just 23
   ```

   (5 points)

3. Write a size function that returns the number of nodes in the tree.

   ```
   size :: BST k v -> Int
   ```

   Example:

   ```
   > size Empty
   0
   > size (Node 0 23 Empty (Node 0 42 Empty Empty))
   2
   ```

   (5 points)

4. Write an ins function that inserts a value v using k as key. If the key is already used in the tree, it just updates the value, otherwise it will add a new node while maintaining the order of the binary search tree.

   ```
   ins :: (Ord k) => k -> v -> BST k v -> BST k v
   ```

   Here, the (Ord k) in the type assignment means that the operators <,> and == can be applied to values of type k.

   ```
   > val (ins 23 42 Empty)
   Just 42
   > val (ins "key" "new" (ins "key" "old" Empty))
   ```
   ```

```
Just "new"
> size (foldl (\acc e -> ins e e acc) Empty [1..10])
10
```

(15 points)

5. Make BST an instance of the Show type class. You will have to implement the show function such that the result every node of the tree is shown as "(leftTree value rightTree)".

```
6. instance (Show v) => Show (BST k v) where
7.   show Empty = ""
     show (Node k v left right) = ...
```

Here, (Show v) => means that show can be applied to values of type v.

```
> Node "foo" "bar" Empty Empty
("bar")
> let lst = [(9,"ipsum"),(12,"sit"),(7,"Lorem"),(13,"amet"),(10,"dolor")]
> foldl (\acc (x,y) -> ins x y acc) Empty lst
> (("Lorem")"ipsum"(("dolor")"sit"("amet")))
```

(5 points)

8. Suppose we have the following type

```
9. data JSON = JStr String
10.          | JNum Double
11.          | JArr [JSON]
             | JObj [(String, JSON)]
```

Make JSON an instance of the Show type class. You will have to implement the show function such that the output looks like normal JSON.

```
instance Show JSON where
  show j = ...
```

Example usage:

```
> JArr [JStr "12", JNum 23.0]
["12",23.0]
> JArr [JStr "12", JNum 23.0, JObj [("foo",JNum 23),("bar",JNum 42)]]
["12",23.0,{"foo":23.0,"bar":42.0}]
```

Hint: Use intercalate of the module Data.List to join a list of strings with commas

(10 points)

12. Suppose we have a type class for everything that can be converted to and from JSON.

```
13. class Json a where
14.   toJson :: a -> JSON
      fromJson :: JSON -> a
```

Make Double and lists of Json-things members of the type class Json. You will have to implement toJson and fromJson for each of these types.

```
instance Json Double where ...
instance (Json a) => Json [a] where ...
```

Example usage:

```
> toJson [1..4]
[1.0,2.0,3.0,4.0]
> 2.0 `elem` fromJson (JArr [JNum 2])
True
```

(10 points)

15. Get the SWI-Prolog compiler or go to the Linux lab which has it pre-installed.

    Create a file **hw3.pl** with the following facts:

    ```
    father(al, bud).
    father(al, kelly).
    mother(peggy, kelly).
    mother(peggy, bud).
    mother(martha, peggy).
    ```

    Then start Prolog with either pl or swipl, load the file and test that it works.

    ```
    $ pl
    ?- [hw3].
    ?- father(X,bud).
    X = al.
    ```

    (5 points)

16. Now, in the file, write a predicate

    ```
    grandma(X,Y) :- ...
    ```

    such that

    ```
    ?- grandma(martha, kelly).
    yes.
    ?- grandma(X, bud).
    X = martha.
    ```

    Note: Your rule should **only** depend on the `mother` and `father` predicates without referring to concrete names like martha or kelly. These are just examples but your code needs to work with **any**people as long as they are connected with `mother` and `father` predicates.

    (5 points)

17. Using recursion, write a predicate

    ```
    descendants(X,Y) :- ...
    ```

which is true if Y is a direct or indirect descendant of X.

```
?- descendants(X,bud).
X = peggy ;
X = al ;
X = martha.
```

(10 points)

18. Write a predicate

```
siblings(X,Y) :- ...
```

which is true if X and Y have a common parent but are not the same person.

```
?- siblings(bud,kelly).
yes.
?- siblings(kelly,kelly).
no.
```

(10 points)

19. NFA-Simulator in Prolog

The following facts describe a non-deterministic finite automaton with four states q0,q1,q2 and q3 which accepts the regular language "ab|aa*"

```
20. %    ASCII-ART for the NFA:
21. %
22. %   (q0) ---a--> (q1) ---b--> (q2*)
23. %    |
24. %    a
25. %    |
26. %    V  / --<-- \
27. %   (q3*)      a
28. %       \ -->-- /
29.
30. %  Transition relation:
31. transition(q0,q1,a).
32. transition(q1,q2,b).
33. transition(q0,q3,a).
34. transition(q3,q3,a).
35.
36. %  Accepting states:
37. accepting(q2).
    accepting(q3).
```

Write a predicate

```
accepts(State, InputList) :- ...
```

that is true if a series of transitions starting in State and processing the symbols (a,b,c) in the InputList leads to an accepting state.

```
?- accepts(q0, [a,b]).
yes
?- accepts(q1, [b]).
yes
?- accepts(q0, [a,a,a]).
yes
?- accepts(q0, [a,a,b]).
no
?- accepts(q0, [a,X]).
X = b ;
X = a.
```

Hint: Also try out `accepts(q0,X)`.

Note: As for question #6, your code should only use the `transition` and the `accepting` predicates **without** ever referring to the concrete states in the example (q0, q1, etc.).

(20 points)