Computer Science C24
University of Toronto Scarborough

Homework Exercise # 4

---

**Handing in and marking**

For all exercises/assignments in this course, you need to submit your solutions to the pencil-and-paper questions on crowdmark and your solutions to the programming questions on MarkUs. Your pencil-and-paper solutions will be marked with respect to correctness, clarity, brevity, and readability. Your code will be marked with respect to correctness, efficiency, program design and coding style, clarity, and readability. This exercise counts for 9% of the course grade.

---

**Question 1.** Haskell Data Types

In this question we are working on the module `Trees`. The starter file `Trees.hs` contains the following definition:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Begin by defining the following functions.

1. A function `countNodes` that returns the number of nodes in a given tree.
2. A *curried* function `forallNodes` that takes a predicate and a tree and returns true if and only if all the nodes of a given tree satisfy the predicate.
3. A *curried* function `existsNode` that takes a predicate and a tree and returns true if and only if at least one node of the given tree satisfies the predicate.
4. A function `inorder` that returns a list corresponding to the in-order traversal of the given tree.

Do you notice anything similar in the functions we have written? We can abstract the similar recursive structure of each of these functions into a higher-order function that works in a fashion similar to `fold`. We define the function **tfold** of type `tfold:: (a -> b) -> (a -> b -> b -> b) -> Tree a -> b` as follows:

```
tfold f g (Leaf x) = f x
tfold f g (Node x left right) = g x (tfold f g left) (tfold f g right)
```

The function `tfold f g t` takes the tree `t`, applies function `f` to all the leaves in `t`, and combines the resulting values for `t`'s branches using the function `g` (notice that `g` also depends on the value stored in each branch).

Rewrite each of the above functions using a single call to `tfold`. The new versions of our functions should be named `countNodes'`, `forallNodes'`, `existsNode'`, and `inorder'`.

**Question 2.** Haskell Type Classes

In this question we represent sets with the help of Haskell lists by defining the data type `Set` as follows:

```
data Set a = Set [a]
```

Some examples of sets are:

```
emptyset = Set []
singleton = Set [42]
s = Set [1,2,3]
s' = Set [3,1,2]
t = Set [3,2,1,4]
```

**a.** Make `Set` an instance of class `Show` with the following effect:

```
> emptyset
{}
> singleton
{42}
> s
{1,2,3}
> s'
{3,1,2}
> t
{3,2,1,4}
```

Filename: *SetShow.hs*.

**b.** We would like to be able to test `Set`s for equality. Note that we cannot simply `derive Eq`, since set equality differs from list equality by not taking the order of elements into consideration. Make `Set` an instance of `Eq` so that `==` on `Set`s tests for set equality. For example,

```
> s == s'
True
> s == t
False
```

Notice that you get `/=` "for free" from the definition of `Eq`. Filename: *SetEq.hs*.

**c.** We would like to define an ordering on `Set`s by using set inclusion. Notice that we cannot make `Set` an instance of `Ord`, since not every pair of sets is comparable. Define a class `POrd` that represents partial ordering. Instances of `POrd` support the following operations:

- the function `pcompare`, which returns values of type `POrdering` defined as follows:
  `data POrdering = PLT | PEQ | PGT | PIN deriving (Eq, Show)`
  The intended meaning of these values is: `PLT` – less than, `PEQ` – equals, `PGT` – greater than, and `PIN` – incomparable.
- the functions `lt` (less than), `gt` (greater than), `lte` (less than or equals to), `gte` (greater than or equals to), and `inc` (incomparable).

The class declaration should provide:

- a **default definition** of the function `pcompare` from the ordering operators `lt`, `gt`, etc., and
- **default definitions** of all the ordering operators from `pcompare`.

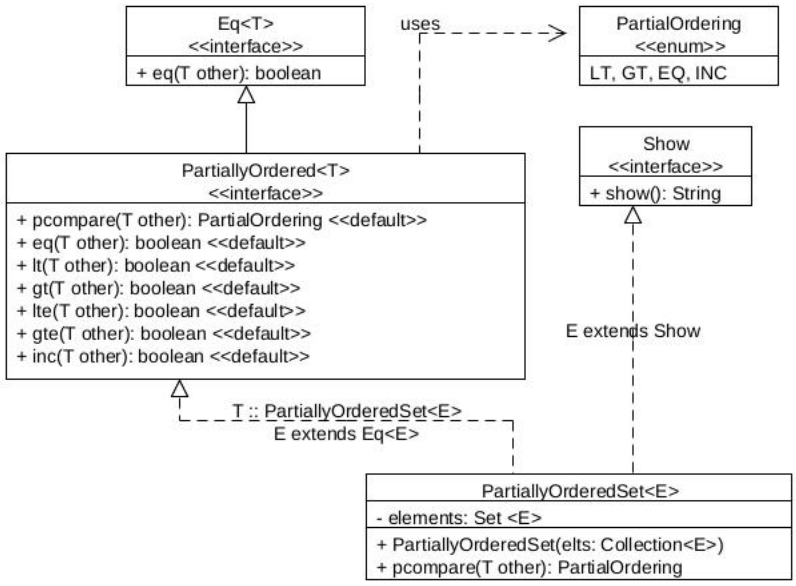
Filename: *POrd.hs*.

**d.** Make `Set` an instance of `POrd`. Provide the definition of the function `pcompare`, so that `pcompare s t` returns

- `PLT`, if the set `s` is less than the set `t` (i.e., it is a proper subset of `t`),
- `PEQ`, if the set `s` is equal to the set `t`,
- `PGT`, if the set `s` is greater than the set `t` (i.e., it is a proper superset of `t`), and
- `PIN`, if the sets `s` and `t` are incomparable.

The definitions of the ordering operators `gt`, `lt`, etc. should be **inherited from** `POrd`. Some examples:

```
> s `lt` t
True
> singleton `gte` emptyset
True
> singleton `lt` t
False
> singleton `inc` t
True
> s `lt` s'
False
> s `lte` s'
True
> pcompare s s'
PEQ
```




```
> pcompare s singleton
PIN
```

Filename: *SetPOrd.hs*.

**Question 3.** Exploring Programming Languages

In this question we implement a Java version of partially ordered sets, analogous to our earlier solution in Haskell. Study the given UML diagram and the provided starter code carefully. You may need to read up on the newer features of Java, such as *default interface methods, as well as on* advanced Java generics.

To make sure we learn as much as we can about modern Java in the process, we will not use constructs such as loops or recursion in this exercise. We will use Java **streams** instead.

Make sure all your classes/interfaces are **public**, all methods/fields have access modifiers exactly as in the diagram, all classes/interfaces and methods are named **exactly as specified**, and are in the **package set**.

Finally, make sure your code passes the `checkstyle` tool, linked from the course website. As always, make sure you are using the correct versions of all tools.

As always, make sure to fully test your code. The starter code provides **example** usage of the code you will develop, but **not proper testing** of the code — it is your responsibility to thoroughly test all code you submit.