H01. Functional Sets

Problem statement

Sets are **unordered** collections of **unique** elements. There are several ways to store sets. One of them relies on **characteristic functions**. Such **functional sets** are especially useful if we expect many **insert/retrieve** operations and less **traversals** in our code.

A **characteristic function** of a set $A \subseteq U$ is a function $f : U \to \{0, 1\}$ which assigns $f(x) = 1$ for each element $x \in A$ and $f(x) = 0$ for each element $x \notin A$.

In our implementation, U will be the set of integers, hence we shall encode only **sets of integers**. Hence, the type of a set will be:

```
Int => Boolean
```

For instance, the set $\{1, 2, 3\}$ may be encoded by the anonymous function:

```
(x: Int) => (x == 1 || x == 2 || x == 3)
```

Also, the empty set can be encoded as:

```
(x: Int) => false
```

while the entire set of integers may be encoded as:

```
(x: Int) => true
```

1. Write a function `singleton` which takes an integer and returns **the set** containing only that integer:

```
def singleton(x: Int): Int => Boolean = ???
```

Note that `singleton` could have been equivalently defined as: `def singleton(x: Int)(e: Int): Boolean = ???` , however, the previous variant is more legible, in the sense that it highlights the idea that we are returning **set objects**, namely **characteristic functions**.

2. Write a function `member` which takes a set and an integer and checks if the integer is a member of the set. Note that `member` should be defined and called as a curry function:

```
def member(set: Int => Boolean)(e: Int): Boolean = ???
```

3. Write a function `fromBounds` which takes two integer bounds `start` and `stop` and returns the set $\{start, start + 1, \ldots, stop\}$. It is guaranteed that `start <= stop` (you do not need to check this condition in your implementation(.

```
def fromBounds(start: Int, stop: Int): Int => Boolean = ???
```

4. Write a function which performs the intersection of two sets:

```
def intersection(set1: Int => Boolean, set2: Int => Boolean): Int => Boolean = ???
```

5. Write the function which performs the union of two sets:

```
def union(set1: Int => Boolean, set2: Int => Boolean): Int => Boolean = ???
```

6. Write a function which computes the sum of all elements from a set, for given **bounds**. Use a tail-end recursive function:

```
def sumSet(start: Int, stop: Int, set: Int => Boolean): Int = {
  def auxSum(crt: Int, acc: Int): Int = ???
  ???
}
```

7. Generalise the previous function such that we can **fold** a set using any binary commutative operation over integers:

```
def foldSet(
             start: Int,            // bounds (inclusive)
             stop: Int,
             op: (Int, Int) => Int, // folding operation
             initial: Int,          // initial value
             set: Int => Boolean    // the set to be folded
           ): Int = ???
```

8. Implement a function `forall` which checks if all elements in a given range of a set satisfy a predicate (condition). (Such a condition may be that all elements from given bounds are even numbers).

```
def forall(
            start: Int, // start value (inclusive)
            stop: Int, // stop value (inclusive)
            condition: Int => Boolean, // condition to be checked
            set: Int => Boolean // set to be checked
          ): Boolean = ???
```

9. Implement a function `exists` which checks if a predicate holds for **some** element from the range of a set. Hint: it is easier to implement `exists` using the logical relation: $\exists x.\, P(X) \iff \neg\forall x. \neg P(X)$.

```
/* implement a function exists, using forall */
def exists(
            start: Int, // start value (inclusive)
            stop: Int, // stop value (inclusive)
            condition: Int => Boolean, // condition to be checked
            set: Int => Boolean // set
          ): Boolean = ???
```