Programming Assignment: Functional Sets

You have not submitted. You must earn 8/10 points to pass.

Deadline Pass this assignment by September 25, 11:59 PM PDT

Instructions

My submission

Discussions

Attention: You are allowed to submit **an unlimited number of times** for grade purposes. Once you have submitted your solution, you should see your grade and a feedback about your code on the Coursera website within 10 minutes. If you want to improve your grade, just submit an improved solution. The best of all your submissions will count as the final grade.

In this assignment, you will work with a functional representation of sets based on the mathematical notion of characteristic functions. The goal is to gain practice with higher-order functions.

Download the funsets.zip handout archive file and extract it somewhere on your machine. Write your solutions by completing the stubs in theFunSets.scala file.

Write your own tests! For this assignment, we don't give you tests but instead the FunSetSuite.scala file contains hints on how to write your own tests for the assignment.

Representation

We will work with sets of integers.

As an example to motivate our representation, how would you represent the set of all negative integers? You cannot list them all... one way would be so say: if you give me an integer, I can tell you whether it's in the set or not: for 3, I say 'no'; for -1, I say yes.

Mathematically, we call the function which takes an integer as argument and which returns a boolean indicating whether the given integer belongs to a set, the *characteristic* function of the set. For example, we can characterize the set of negative integers by the characteristic function (x: Int) => x < 0.

Therefore, we choose to represent a set by its characteristic function and define a type alias for this representation:

```
1  type Set = Int => Boolean
```

Using this representation, we define a function that tests for the presence of a value in a set:

```
1  def contains(s: Set, elem: Int): Boolean = s(elem)
```

2.1 Basic Functions on Sets

Let's start by implementing basic functions on sets.

1. Define a function **singletonSet** which creates a singleton set from one integer value: the set represents the set of the one given element. Now that we have a way to create singleton sets, we want to define a function that allow us to build bigger sets from smaller ones.
2. Define the functions **union**,**intersect**, and **diff**, which takes two sets, and return, respectively, their union, intersection and differences. **diff(s, t)** returns a set which contains all the elements of the set ***s*** that are not in the set ***t***.
3. Define the function **filter** which selects only the elements of a set that are accepted by a given predicate *p*. The filtered elements are returned as a new set.

2.2 Queries and Transformations on Sets

In this part, we are interested in functions used to make requests on elements of a set. The first function tests whether a given predicate is true for all elements of the set. This forall function has the following signature:

```
1  def forall(s: Set, p: Int => Boolean): Boolean
```

Note that there is no direct way to find which elements are in a set. *contains* only allows to know whether a given element is included. Thus, if we wish to do something to all elements of a set, then we have to iterate over all integers, testing each time whether it is included in the set, and if so, to do something with it. Here, we consider that an integer x has the property -1000 <= x <= 1000 in order to limit the search space.

1. Implement **forall** using linear recursion. For this, use a helper function nested inforall.
2. Using **forall**, implement a function **exists** which tests whether a set contains at least one element for which the given predicate is true. Note that the functions