PLC: Homework 4 [75 points]

Due date: Wednesday, February 22nd, 10:30pm
3 point bonus if you turn in by Tuesday, February 21st, 10:30pm

About This Homework

The goal of this homework is to gain experience working with lists and other data structures in Agda.

The first thing you should do for this homework is copy the files you will be modifying, from the `hw/hw4` subdirectory of the class repository, to a new `hw4` subdirectory of your personal repository, similarly to previous homeworks. These files are `bintree.agda` and `list-todo.agda`.

There is extra credit possible for this assignment, so you can earn over 75 points (but 75 is a perfect score).

Partners Allowed

For this homework, you may work by yourself or with one partner (no more). See the instructions for `hw2` for how to create the `ack.txt` and `partner.txt` files that are required if you work with a partner

All files should go in a hw4 subdirectory (which you create) of your personal repository. Please call the subdirectory exactly that. Do not call it `Homework 4` or `Hw4` or other variations, or we will penalize you 5 points.

How to Turn In Your Solution

Make sure you have done a "Subversion add" on the `hw4` directory you created in your personal repository, and on the `.agda` files you have to submit for this assignment, and then do a "Subversion commit" on your personal repository directory to submit them. The files you will be submitting are `nat-todo.agda`. You can commit as many times as you want up to the deadline. If you commit after the deadline, we will use the last existing version before the deadline. Work submitted after the deadline will not count.

How to Check Your Solution

As for previous homeworks, you might want to make sure you have correctly submitted your solution via Subversion. Again, just go to the URL for your personal repository using a web browser, log in with your HawkId and password, and you can see what has been submitted.

Make sure each of the files you submit can be checked by Agda without any holes, yellow highlighting, or red highlighting. If any file does not check this way, we will penalize you 5 points total (if multiple files do not check, we will still just penalize you 5 points). Any

problems you do not solve should simply be removed from the `.agda` files, so that there are no holes (question marks) in the files.

How To Get Help

You can post questions in the `hw4` section on Piazza, or elsewhere on Piazza. See the course's Google Calendar, linked from Piazza, for the locations and times for office hours, including evening Skype office hours for Prof. Stump.

1 Reading

Read Chapter 4 of the book.

2 Functional programming with lists [40 points]

In the file `list-todo.agda`, write code for the following problems.

1. Write a recursive definition of function `take`, so that `take n l` will return the first `n` elements of `l`. If your code is correct the yellow highlighting will disappear from `take-test`. [5 points]
2. Write a recursive definition of function `drop`, so that `drop n l` will return the elements of `l` after the first `n` elements (so the first `n` elements of `l` will be dropped). If your code is correct the yellow highlighting will disappear from `drop-test`. [5 points]
3. Write a recursive definition of `splitAt` so that `splitAt n l` will return a pair of lists, where the first list in the pair is the list of the first `n` elements of `l`, and the second list is the rest of the elements of `l` after the first `n` of them. [5 points]

   Please note that we do not want you to define `splitAt` in terms of `drop` and `take`, nor vice versa. We will not give points for solutions that define one in terms of the others.
4. Prove the `splitAt-thm` which says that `splitAt` returns the pair of results you would get by calling `take` and `drop`. [5 points]
5. Fill in the definition of `filter-pos`, which is supposed to take a list and a predicate on indices in the list, and return the list consisting just of those elements whose indices are accepted by the predicate. I found I needed to write a helper function for this one. [10 points]
6. Fill in the definition of function `grid`, so that `grid x y` returns a list of all pairs of numbers `(n , m)` where `n` is less than or equal to `x` and `m` to `y`. The order of the pairs does not matter. [10 points]

3 Programming and proving for binary trees [40 points]

In `bintree.agda`, there is a definition of binary trees with data at the nodes, as an Agda datatype.

1. Fill in the definition of function `size` which should compute the size of a binary tree, where leaves are considered to have size 1 and for nodes, you add 1 to the sum of the sizes of the subtrees. If your definition is correct, the yellow highlighting for `test-size` in the file should disappear. [5 points]

2. Fill in the definitions of `in-order` and `pre-order` to implement in-order and pre-order traversals, respectively, of the binary tree. (See Wikipedia for "tree traversal" if you do not remember what these are.) Your traversals should take a binary tree and convert it to a list using the appropriate order of adding elements to the list. If your code is right, the yellow highlighting will disappear from the tests `test-in-order` and `test-pre-order`. [15 points]

3. Here is a theorem just about lists, which is a warm-up for the next one. Fill in the proof of `list-member-map`, which says that if an element `a` is a member of list `l` using one equality function `eqA`, and if you map a function `f` over the list, then as long as you check list membership in the new list using an equality function `eqB` for which `f` preserves equalities (if `eqA` holds of some elements of type `A`, then `eqB` will hold of the elements you get by applying `f` to those elements of type `A`); in this case, `f a` will still be a member of the mapped list.

   I found I needed to use the `keep` idiom for this problem (see Section 4.3.4 of the book, or class discussion Thursday, Feb. 16). I also needed to use the theorem called ||-tt from `bool-thms.agda` in the IAL. [10 points]

4. Fill in the proof of `pre-in-thm`, which says that if you can find element `x` in the list you get by pre-order traversal of a binary tree, then you can also find it in the list you get by in-order traversal.

   For this theorem, I found I needed to use the keep idiom and ||-tt as in the previous problem. I also needed `list-member-++` from `list-thms2.agda`. [10 points]