# PLC: Homework 10 [75 points]

Due date: Friday, April 14, 10:30pm
3 point bonus if you turn in by Thursday, April 13, 10:30pm

## About This Homework

The goal of this homework is to experiment with internal verification in Agda, and to propose your final project. As usual, the first thing you should do for this homework is copy the files from the `hw/hw10` subdirectory of the class repository, to a new `hw10` subdirectory of your personal repository, similarly to previous homeworks.

### 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

**Partners are also allowed for the final project:** you may work by yourself or with one partner.

**All files should go in a hw10 subdirectory (which you create) of your personal repository.** Please call the subdirectory exactly that. Do not call it `Homework 4` or `Hw10` 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 `hw10` 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. 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.

### How To Get Help

You can post questions in the `hw10` 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

Please read Chapter 6.

# 2 Internal Verification [60 points]

The file `sorted-list.agda` defines an internally verified data type for sorted lists. A value of type `sorted-list l u` is a list which is guaranteed (statically, by type checking) to be in sorted order, and with all data between lower bound `l` and upper bound `u`.

1. Fill in the definition of `sorted-list-to-𝕃` so that you get out a simple list of the data in the input sorted-list [10 points].
2. Fill in the code for `relax-lb` and `relax-ub`, so that you get back a list with the same data as the input list, but with the lower bound, or upper bound, respectively, relaxed as indicated by the type. [10 points each]
3. Define a function for appending sorted lists. The type of your function may (in fact, should) include an assumption that the upper bound of the first list is less than or equal to the lower bound of the second list. You get 5 points just for the type of the function, and 10 more points for the code.
4. Define a function for inserting a piece of data into a sorted list (in the correct position to preserve being sorted). [15 points]
5. Fill in the definition of `insertion-sort` so that it returns a sorted version of the input list, by calling your function for inserting a piece of data into a sorted list. [5 points (just because we are over 60 points with this one)]

You can test your code with concrete data; see `test-sorted-list.agda` (though this cannot be loaded until you fill or comment out the code with holes in `sorted-list.agda`).

# 3 Project Proposal [25 points]

In a plain text file (not Word, not PDF) called `project-proposal.txt`, please tell us what you propose to do your project on. This file should have the following format, where you should explicitly write out the names of the sections as listed below:

- **Project title**: what is the title of your project
- **Project members**: list yourself or yourself and one partner
- **Summary**: what is the basic idea of what you are proposing to do
- **Technology**: what programming language(s) will you use, and/or any other software or libraries (including the IAL)
- **Milestone**: what will you get working for your progress report (due April 21)

- **Backup plan**: what is a backup plan in case your goals turn out to be too challenging and you cannot implement all of what you set out to do

Please limit yourself to 1 page (when printed out, for example) or a little more.

### 3.1 Sample projects

You can make up your own project completely (but then you have to ask me about it before you submit this homework), or else choose something following one of the ideas below. These still require you to come up with something specific, but hopefully will help point you in the right direction.

A few high-level guidelines:

- Your project should be to implement (code) something.
- It has to be written in a functional programming language (Agda, elisp, Haskell, OCaml, F#, functional part of Scala, Racket, etc.); if for some reason you need to use multiple languages, the functional language should be the primary one (majority of code).
- I am looking for a substantial amount of code, which is probably no less than 100 lines, unless you can argue it is very tricky.
- You will need to include demos in your final version, including sample inputs and some kind of generated outputs from your code (this is just in case we cannot manage to run your code for some software reason).

Here are some idea:

1. Write a rudimentary CSS pre-processor. You would have to write a grammar for let us say at least some core part of CSS (enough to write interesting styles) – though CSS does not seem to be too complicated to parse so maybe the whole language is reasonable. You could then add features like you find in Sass: variables which are in scope over a whole CSS file (or set of files, but that may be too ambitious), conditionals perhaps, local variables, perhaps some kind of functions or macros? The goal would be to read in an extended form of CSS and dump out actual CSS that can be processed by a browser.
2. Pick a functional language you do not know, and implement some algorithm or data structure in it that you have studied in this or another class. A reasonable example would be redoing hw9 (graph traversal for garbage collection) in one of these other languages, including the input/output.
3. Extend the xmlnav-mode.el example we worked on for hw8, to have additional functionality (which you should propose).
4. For the more mathematically minded: pick some **very simple** theorem in discrete math (maybe basic number theory?) that we do not have in the IAL, and prove it in Agda. Avoid theorems which require real numbers, as we do not have these in the IAL (and they are a bit tricky to deal with in languages like Agda).

5. Chapter 8 of the book walks through an example program written using Agda, the IAL, and gratr: a Huffman encoder and decoder. Read Chapter 8, and then do problems 2, 3, and 4 listed at the end of the chapter. These concern doing some internal and external verification about the Huffman encoding example. The code for the example is included with the IAL. You would be writing some additional proofs about the example.

6. For some computer language of interest to you (World of Warcraft configuration files?) write a grammar to parse them and then implement a tool that does something useful or interesting with them.