# PLC: Homework 9 [75 points]

Due date: Friday, April 7, 10:30pm
3 point bonus if you turn in by Thursday, April 6, 10:30pm

## About This Homework

The goal of this homework is to implement a basic garbage-collection algorithm in Agda. As usual, the first thing you should do for this homework is copy the files from the `hw/hw9` subdirectory of the class repository, to a new `hw9` 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

**All files should go in a hw9 subdirectory (which you create) of your personal repository.** Please call the subdirectory exactly that. Do not call it `Homework 4` or `Hw9` 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 `hw9` 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 `hw9` 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

Review Sections 5.2, 5.4, and 5.5 of the book (already assigned back in hw6). We will be discussing these in class soon.

# Overview

In this homework you will write a tool called `simgc` that can simulate mark-and-sweep garbage collection on an input reference graph. You will have to parse in that graph, and then simulate mark-and-sweep in a step-by-step fashion on the graph. For each step of mark-and-sweep, you will emit a GraphViz file from your program. GraphViz is a graph description language, and there are tools that can render these files to reasonably nice-looking graphs. You can also render them on the web:

`http://www.webgraphviz.com`

By emitting these GraphViz files (and then rendering them using the existing GraphViz tools), you can create a sequence of images showing the step-by-step execution of mark-and-sweep on an input graph.

**Note** that unlike in previous homeworks, I am not going to walk you through a particular way to solve the problem. You will have to come up with a strategy yourself.

# 2 Parsing memgraph files [20 points]

The first part of implementing `simgc` is to be able to parse in files in `memgraph` format (a format I am proposing for this assignment). A sample file is `graph1.mem`. A `memgraph` file looks like:

1. optional whitespace, then the characters `ROOTS:` (with the colon).
2. whitespace, then a whitespace-separated list of `id`s, where an `id` is any sequence of one or more digits `0` through `9` or lowercase letter. So `x`, `cell`, `25`, and `x1y2` are all legal `id`s.
3. whitespace, then the characters `GRAPH:` (with the colon).
4. then a list of edges, where an edge has an `id`, then whitespace, then the characters `->`, then whitespace, and then a whitespace-separated list of `id`s.
5. then optional whitespace and a semicolon.

See `graph1.mem` for an example. The idea is that an edge like

```
a -> b c d ;
```

Represents three graph edges: from `a` to `b`, from `a` to `c`, and from `a` to `d`.

For this problem, please create a grammar file called `memgraph.gr` where the name of the grammar (first line of the file) is `memgraph` and also where the start symbol of the grammar is `strt`. Compile

your grammar using `gratr` as usual. The `simgc.agda` file you will modify subsequently assumes that you have done these steps. The parser gratr emits for your `memgraph.gr` grammar should be able to parse `graph1.mem` and similar examples.

You will receive full credit for this problem if the parser generated by `gratr` for your grammar can handle testcases like `graph1.mem`; we also may test to make sure your grammar is not too liberal in the sense that it accepts files it should not.

Note that for purposes of later problems, you may assume that all roots listed in the `ROOTS` part of a memgraph file are also used in the `GRAPH` part of the file.

## 3 The garbage collection [60 points]

Now you can try compiling the `simgc.agda` file I am providing, which also makes use of a file called `mg.agda`. **Update:** to use this `mg.agda` file, you need to do a subversion update of your IAL.

Your goal is to write a program that can simulate mark-and-sweep garbage collection on the graph you have parsed in in the previous problem. You should modify the `process-strt` function of `simgc.agda`, so that it returns a list of strings. Each string represents a snapshot of the garbage-collection algorithm running on the input graph. The code I am providing in `simgc.agda` takes care of printing these strings to `mem-*.gv` files.

For 40 points, you can just show the currently marked nodes and the next node that is to be processed (in either a depth-first or breadth-first traversal of the graph – either is ok). An example for `graph1.mem` is given in the files `mem-*.txt` in the `hw9` directory (though the code in `simgc` will print them to `mem-*.gv`, not `.txt`). Those files are generated using `mg-to-string` from the `mg.agda` file. This requires converting your parse tree to an `mg` value. In fact, my only official suggestion about how to structure your code is first to convert your parse tree to an `mg` structure, and then write a traversal function that takes an `mg` as input and returns a new one as output.

For 60 points, the snapshots should be GraphViz files showing the roots (as arrows from hidden nodes to the actual root nodes) and the graph. The next node to process should be double-circled. Marked nodes should be colored green. See the files `mem-*.gv` for examples. The files `mem-*.jpg` show the output you get when running the `dot` tool (part of the GraphViz package you can install on your computer if you wish) like this:

```
dot -Tjpg -o mem-0.jpg mem-0.gv
```

Of course, you have to write a function to print out a data structure (like `mg` as I am suggesting) in GraphViz format. This is somewhat involved.

The last snapshot should show the graph with garbage removed.