# COMP4075/G54RFP
# Real-world Functional Programming:
# Coursework Part II

Autumn, Academic Year 2019/20

Henrik Nilsson
School of Computer Science
University of Nottingham

November 5, 2019

## 1 Introduction

As a reminder, the assessed coursework for the module COMP4075/G54RFP (COMP4075 from now on) consists of two parts. They are essentially programming exercises directly related to the content covered in the lectures. (There is also an optional project, COMP4095, for those who wish to go deeper.) The weights of the three parts are as follows:

- Part I: 15 %
- Part II: 35 %

Overall, the coursework is thus worth 50 % of the COMP4075 mark.

The COMP4075 coursework is to be carried out *individually*. You are welcome to discuss the coursework with friends, in the COMP4075 Moodle forum, with the module team, etc., but, in the end, you must solve the problems on your own and demonstrate that you have done so by being able to explain your solutions as well as their wider context.

## 2 Practicalities

For basic practical information on how to work with Haskell on the School's computers, and issues specific to various platforms such as Windows and Linux, see Part I of the coursework.

However, for some of the tasks here you may have to install additional packages using Cabal. To get started, ensure the list of available packages is up to date:

```
cabal update
```

You can list what packages are available; e.g.

```
cabal list
cabal list --installed
```

To install a package, do:

```
cabal install <package-name>
```

However, installing large packages, at least on the School's servers, sometimes fails with errors "failed to create OS thread" (which can be hard to see as this then causes lots of follow-on errors obscuring the initial cause). The problem appears to be that certain resource limits are configured in a quite conservative way on the School's servers. To avoid this problem, limit the number of concurrent processes that Cabal spawns. The most conservative is to set the limit to 1, but this means no parallelism and thus possibly lengthy installation times:

```
cabal install -j1 <package-name>
```

Setting the limit to 3 also appears to work and might be an acceptable compromise.

For general assistance, Cabal provides built-in help:

```
cabal help
cabal help <subcommand>
```

# 3 Submission and Assessment

For information about deadlines, see the module web page. For Part II of the coursework, the following has to be submitted by the deadline:

- A brief written report as specified below.
- The source code of all solutions.

The submission is electronic:

- Electronic copy of the report (PDF). The file should be called `psyxyz-report-partII.pdf`, where `psyxyz` should be replaced by your University of Nottingham user ID.
- Archive of the source code hierarchy (gzipped TAR, or zip). The archive should be called `psyxyz-src-partII.tgz` or `psyxyz-src-partII.zip`, where `psyxyz` again should be replaced by your University of Nottingham user ID, and it should contain a single top-level directory containing all the other files.

The written report should be structured by task. For each task:

- *Brief* comments about the key idea of the solution, how it works, and any subtle aspects; a *few* sentences to a couple of paragraphs would usually suffice.
- Answers to any theoretical questions.
- Unless specified otherwise, the code you wrote or added, with enough context to make an incomplete definition easy to understand. Thus, in cases where you have extended given code, you do not need to include what was given, except small excerpts to provide context if necessary. Indeed, if the give code is lengthy, you are encouraged to keep what you include in the report to a minimum.
- Anything extra that the task specifically asks for.

As for part I, solutions will be assessed on correctness and style. See the Part I description for details. However, for tasks with a large weight, fractional correctness and style marks may be used for a more fine-grained assessment.

# 4 Tasks

**Task II.1** (Weight 40 %)

Implement a deadlock-free solution to the Dining Philosophers problem using Software Transactional Memory (STM). See

```
https://en.wikipedia.org/wiki/Dining_philosophers_problem
```

for the problem statement.

Each philosopher should be represented by a thread (`forkIO`), given a name and, to allow us to see what is going on, announce (print to the terminal) when they are hungry, eating, and thinking, stating their name as well in each case (e.g. `Socrates is hungry`). They should eat and think for a period of time decided at random (import `System.Random` and use `threadDelay`).

In addition to explanations of the design and implementation, and the code itself, include enough sample output to provide evidence that your solution is working.

Finally, discuss your STM solution in relation to the two classical solutions outlined in the Wikipedia article (*Resource Hierarchy Solution* and *Arbitrator Solution*). Explain in particular why your solution is free from deadlocks, and any pros and cons of using STM compared to the classical solutions.

While the solution to this task will be rather "imperative" in style, it is still possible to deploy neat functional programming techniques and appropriate types to obtain an elegant overall solution, and this will be taken into account is the assessment for style.

**Task II.2** (Weight 60 %)

Implement a simple "electronic calculator" using the *Threepenny* GUI framework: `https://wiki.haskell.org/Threepenny-gui` (this will be covered in a lecture shortly). It should look and behave in a reasonably standard way: see section 1 of `https://en.wikipedia.org/wiki/Calculator`.

You will likley have to install the *Threepenny* GUI package:

```
cabal install threepenny-gui
```

However, do check the section on using Cabal above, in particular if the installation fails for some reason. Also, if you are running on the School's Linux servers and if you want interact with your application from a browser running on your local machine, you will have to set up a tunnel:

```
ssh -fN -L8023:127.0.0.1:8023 severn
```

At the very least, the calculator should:

- Handle at least 10-digit integers
- Support addition, subtraction, multiplication, and division
- Allow the sign to be changed (+/-)
- Allow the calculator to be reset (C) as well as clearing the last entry (CE)

A functioning and reasonably looking basic calculator will get half the marks.

For full marks, the calculator should additionally:

- Support calculations with decimal fractions (decimal point)
- Have a memory and associated functions for store and recall
- Support standard precedence rules among the arithmetic operations along with parentheses for grouping (e.g. the result of entering 1+3*4 = should be 13 and the result of entering $5 + 3 * (5 - 9) =$ should be $-7$)
- Have a clearly structured implementation making use of appropriate functional programming techniques and types (e.g. a state transition function embodying the logical core, Functional Reactive Programming or other ways to manage events and effecs to avoid the proverbial "call-back soup").

You might find the classic Shunting-yard Algorithm due to Edsger Dijkstra to be helpful for handling precedence:

`https://en.wikipedia.org/wiki/Shunting-yard_algorithm`

Another possibility is to simply gather all input as a string and parse it according to a grammar imparting the desired precedence among the operators. This tend to be how more modern calculators, including typical mobile apps, work. The parsing could happen repeatedly as the input is being entered, with an evaluation after each successful parse, or only at the end when the result is demanded.

For this task, the written report itself need only include enough code to illustrate the key ideas and aspects of the implementation. However, two or three screen dumps showing your calculator in action should be included.