# Technical Report
# COMP1100 Assignment 2

Jacob Bos
ANU u7469354

May 4, 2022

Lab: Tuesday 11am

Tutor: Abhaas Goyal

Word-count beyond cover page at $\leq$ 1250 words

## Contents

# 1 Introduction

The program detailed herein is an implementation of Fransson's *QR World* cellular automata and graphical representation in Haskell with complimentary unit tests.

# 2 Documentation

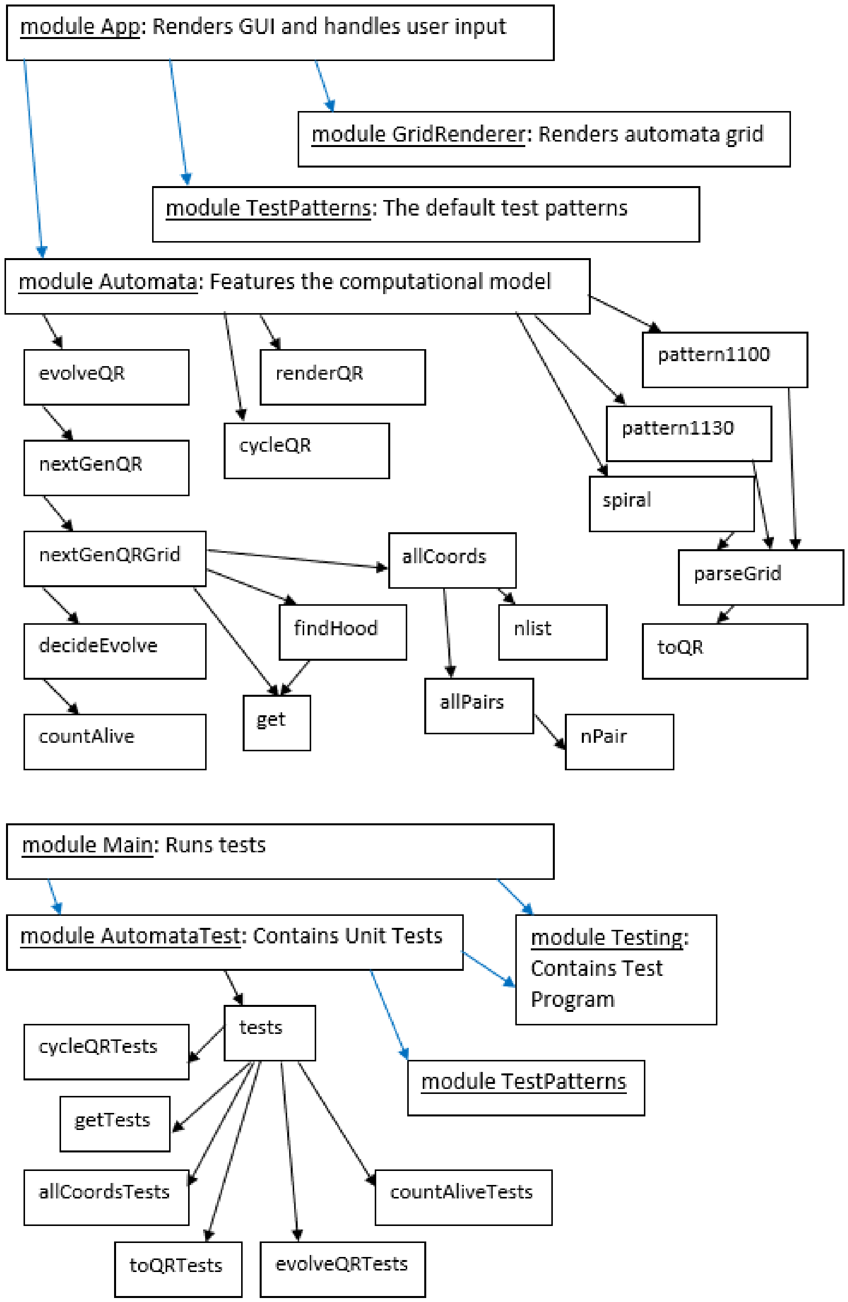
## 2.1 Design Documentation and Technical Decisions

**Task 1** defined the type of `QRCell` as either `Dead` or `Alive`. An if then else (ITE) statement is used for `toQR` to convert values in a textual representation to useful values with `'A'` becoming `Alive::QRCell` and anything else returning `Dead`. `cycleQR` swaps a cell's state upon cursor clicks using a case statement. `renderQR` renders each cell as a codeWorld picture. `get` retrieves the cell's value for a given `GridCoord`, returning `Nothing` for nonsensical arguments. `allCoords` generates a row-major list of all coordinates in an $(a \times b)(a, b > 0)$ grid returning an error for nonsensical arguments of $a, b \ngtr 0$ as par specifications. Otherwise it calls 3 helper functions. `nList` generates an ascending list from 0 to (a-1). `nPair` then pairs each value in `nList` with some integer. `allPairs` then uses `nList`to create one list from (0,0) to (a-1,b-1).

**Task 2** consists of two primary functions, `nextGenQR` which parses the automata through one iteration, and `evolveQR` which iterates the automata $n$ times. `nextGenQR` calls `allCoords` and `nextGenQrGrid`, which is the main function handling the grid evolution. `nextGenQrGrid` recurses through the `allCoords` list with `get` retrieving the state at each position and `findHood` retrieving a list of the neighbouring states. `decideEvolve` updates the state of the cell according to the QRWorld rules. This new state is then prepended to the recursive call on the rest of the `allCoords` list. `allCoords` is guarded to return an error for nonsensical grid dimensions avoiding irrational program operation. A guarded case recursion was chosen for `countAlive` to only count alive elements. `decideEvolve` was chosen to use a case to direct the function to guards based on the number of alive neighbours determined by `countAlive` to then decide how to change the state. `findHood` uses `get` to retrieve a four element list of `[Maybe QRCell]` to give the states of the neighbouring cells. `decideEvolve` then calls `countAlive` to then make a decision about what each cell state should evolve to depending on how many alive neighbours it has. To do this it cases on the state of the cell and is then guarded by the number of alives to evolve the state properly. `countAlive` just uses a case and nested guard recursion to sort through the list of neighbouring states and returns the number that are alive. `evolveQR` recurses down to a base of 1 from a natural $n$ applying `nextGenQR` to itself $n$ times.

## 2.2 Program Design / Structure

The following is a module and function dependency graph for the program and test program:



The program's broad structure was dictated by the specifications with modules and key functions already named and structured. The above graph mostly shows the functions developed by the author and their helper functions. The author broke `allCoords` up into helper functions for ease of understanding, allowing a clear step by-step construction of the list with `allCoords` just calling the helper functions that do each construction.

It was chosen that `nextGenQR` would have a helper `nextGenQRGRid` because the former was unable to recurse through a list of `allCoords` but the latter could be made to. This ability was important as it made it easier to maintain the order of the state list and aided the implementation of `findHood` which was needed to make the evolution decision for each cell. Designing `findHood` to return a list rather than a 4 element double was chosen as it allowed `countAlive` to use recursion which Haskell is optimised for.

It was chosen that the main test function would concatenate separate test functions of type `[Test]` together to allow easier documentation via comments and function naming.

### 2.3 Assumptions

Whilst writing `get` it was assumed that attempts to retrieve a point outside the grid should return `Nothing :: Maybe QRCell` as doing so eased implementation of `findHood` and `countAlive` far more than returning an error would. It was initially assumed that nonsensical inputs to `allCoords` should return an empty list but this was revised to return an error as specified.

## 3 Testing

**Unit tests** were divided into 6 groups, each testing a particular function and/or their helpers. `cycleQRTests` is a fully comprehensive test group for `cycleQR` indicating its correctness. `toQR` is tested against tests two typical cases and an edge case. The tests for `get` cover most possible edge cases and also some typical cases as documented in the `AutomataTest` file. The tests for `allCoords` covers some expected inputs to both the main function and helpers. There were no edge case tests written as such cases are written to return an error and there was no provision to test if errors are returned.

`evolveQR` was tested with three unit tests which also test `nextGenQR` due to `evolveQR`'s dependency. The first two check if the 1100 pattern gets to an alternating steady state after 12 evolutions and the third if the 1130 pattern eventually reached steady state as specified. All tests passed indicating program correctness. All these tests are documented with comments in `AutomataTest.hs`

**GUI tests** focussed on the behaviour of functions which handle direct user interaction. `get` and `cycleQR` were tested by clicking cells with various states and checking if the appropriate cell switched state. `decideEvolve` was tested by observing the evolution of a single cell in various neighbourhoods compared against the rules of QRWorld. All tests passed indicating correctness.

# 4 Reflection

## 4.1 Design Choices

Program development followed progressed linearly parallel to ordering of the specifications. Design decisions were made with both functionality and style in focus, making proper use of Haskell's recursive propensity. Consequently, program is quite readable and supplemented by effective documentation comments.

ITE statements were used whenever it was necessary to check for one case and return a single answer for anything else. Guard and case recursion was selected based on whether the function needed to iterate n times or traverse a list. For `allPairs` and `nList` which iterate n times, (++) allowed proper order of the list as n counts down but the list counts up. When traversing a list (:) was used to maintain the input ordering such as for `nPair`.

An algebraic datatype was chosen for `QRCell` as it was more descriptive of the program's meaning than boolean values. `renderQR` uses a piecewise definition to improve style. Guards were chosen for `get` to protect against retrieving elements outside the automata grid. It was chosen for `nextGenQR` to call `nextGenQrGrid` so that the helper functions could be called appropriately and allow for a recursion through the list of `allCoords`.

## 4.2 Reflection

If they were to re-develop the program the author believes the helper `nPair` could be rewritten or removed by using `map` and `(\y x -> (x,y))`. However, they would make no changes to the structure which was largely dictated largely by specifications.

The program was within the author's technical abilities and so no significant problems encountered in development however, they suspect that there is a cleaner way to write `allCoords` as it is rather convoluted, however, other than what's mentioned above, inspiration escapes them. The author had some trouble defining the type `QRCell` but came to a good solution under closer reading of the specifications which allowed the rest of the program to develop smoothly. Consequently, they did not need to collaborate with others in any significant way.