Assignment 2

Solving Sudokus

The deadline for Assignment 2 is **11pm on Sunday 12 October (Canberra time, sharp).** Late submissions **will not be marked** unless you have an **approved extension**. Extensions can only be applied for through the Extension App. Students with Education Access Plans still need to use this App.

Use the "Submit" button on the top-right corner to submit your code to our server. You must submit at least once before the deadline. You can submit as many times as your want before the deadline; only your latest submission counts.

Usual academic misconduct standards apply to Assignment 2. You must work on this assignment yourself. Note that, unlike Task 0 in Assignment 1, there is no part of this assignment for which collaboration is acceptable. Plagiarism detection tools are highly advanced nowadays. **Do not take the risk**.

Introduction

Sudoku is a type of logic puzzle, invented in New York in the 1970s and later popularised under its current name in Japan, in which players try to place the digits 1 to 9 into each cell of a 9x9 grid according to certain rules. In this assignment you will write code that analyses Sudoku puzzles, with the aim of producing solutions that can both be read by humans and run by a computer.

The ideas behind this assignment, such as descriptions of the techniques which can be used to solve simple puzzles, can be found in many different places, but we were particularly influenced by the website Sudoku9x9.com.

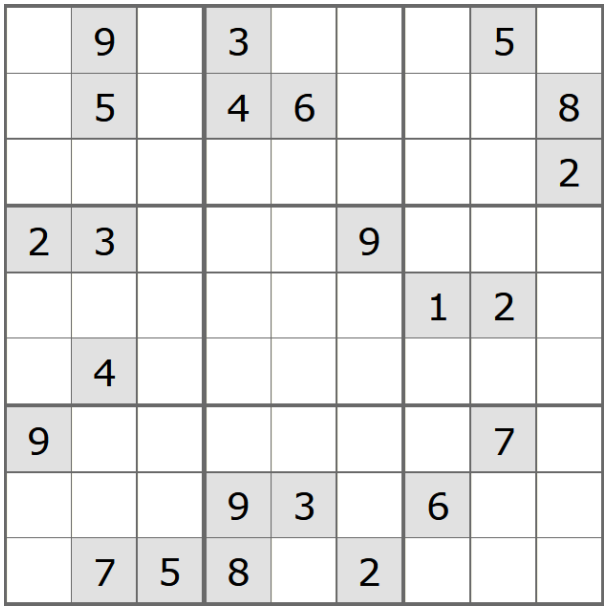
If you look at the file `Sudoku.hs` you will see that we model a Sudoku grid as a polymorphic list of lists of values, i.e.

```
data Sudoku a = Sudoku [[a]]
```

You will not change this type definition, although you are welcome to add `deriving` clauses.

The intended meaning of an element of this type is that its nth (for example, first) element is itself a list, containing all the values in the nth *row*. We will expect that every well-formed element of this type is a list of length exactly 9, whose every member is itself a list of length exactly 9, but there is nothing in the type definition that enforces that, let alone the more complicated rules of Sudoku. You will write code that checks these conditions yourself.

We have provided two example Sudokus, both corresponding to puzzle number #850878221 on Sudoku9x9.com: the first is the puzzle itself, i.e.



with the digit `0` used to indicate a blank space, and the second is its unique correct solution with no blank spaces. We have also implemented a `show` function so that when you type

```
ghci> example
```

you will see the example Sudoku displayed in the terminal in an easy-to-read format.

Assignment Overview

Two mostly incomplete Haskell files are provided for you to edit: `Sudoku.hs` and `TestSudoku.hs`. Either can be loaded in `ghci` so that you may interact with them. `Sudoku.hs` will contain your code that analyses Sudokus, suggests next moves towards their solution, and solves them. `TestSudoku.hs` will contain your code for unit tests to check the correctness of your code. No tests will be provided for you by us. You will also add a third file to your assignment directory, called `Report.pdf`, which will be your technical report about your assignment.

Your total mark is out of 100. Breakdown:

- **Correctness of `Sudoku.hs` code:** 60 marks total, breaking down as:
  - Pencilled Sudoku: 6 marks
  - Cell Logic: 12 marks
  - Validating Sudokus: 12 marks
  - Suggesting Moves: 12 marks
  - Solving Sudokus: 12 marks
  - Advanced Techniques: 6 marks
- **Unit tests in `TestSudoku.hs`:** 10 marks
- **Code style in `Sudoku.hs` and `TestSudoku.hs`:** 10 marks
- **Technical report:** 20 marks

Your code should be able to run, even if you do not complete all parts of the assignment. It is expected that you submit code that can be successfully loaded in `ghci` without throwing errors. Code that does not do this,

for either `Sudoku.hs` and `TestSudoku.hs`, will receive **heavy** mark deductions. If you have a partial solution that you cannot get running, you should comment out the parts that cannot run and discuss them in your report.

Part 1.1: Pencilled Sudoku (6 marks)

We think it is likely that you will find the functions you are asked to write, in this and other preliminary parts of the assignment, useful as helper functions for later parts of the assignment. However, if you solve the later parts in some other way, not using these functions, that is fine. You should still include these functions so that you can be marked for these parts, and this will not be considered as 'dead code'.

If Sudoku grids are always filled by the digits 1 to 9, why did we implement them as a polymorphic type? Because we are going to work with our Sudoku puzzles using two different points of view, for which we will use different instantiations of the `Sudoku` type.

The first is `Sudoku Int`, where every cell (element of the list of lists) contains an `Int`. We will expect that the only `Int`s that appear will be from `1` to `9`, along with `0` to indicate a blank cell, although there is no code that enforces this as yet.

The second view, which we will call a *pencilled* Sudoku, is `Sudoku [Int]`.

Yes, this is a list of lists of lists! This is likely to be confusing at first. Learning how to effectively use the functions already written for you in the Prelude, and perhaps in `Data.List`, will help you to manage this profusion of lists.

Here every cell contains a list of `Int`s, which we will understand as the `Int`s that might possibly appear in that cell. For example, if we know that a cell must contain exactly the value `6`, that cell in a pencilled Sudoku will be `[6]`. On the other hand, if the only thing we know about that cell is that it does *not* contain the value `6`, then that cell will be `[1,2,3,4,5,7,8,9]`.

- Complete the function `pencil`, which maps a Sudoku of `Int`s to a pencilled Sudoku, by mapping every value from `1` to `9` to the singleton list with that value (e.g. `6` to `[6]`), and every cell containing `0` to the list of all possibilities, `[1,2,3,4,5,6,7,8,9]`, to indicate that we do not (yet) have any idea which value is correct for that cell. You may handle an input that is not between `0` to `9` in any way you think is appropriate.
- Complete the function `unpencil`, which maps a pencilled Sudoku to a Sudoku of `Int`s, by mapping every singleton list to its unique value, and any list with multiple possibilities to `0`. You may handle impossible values (such as empty lists) in any way you think is appropriate.

Part 1.2: Cell Logic (12 marks)

As we work with our puzzles we will need to refer to specific locations inside the Sudoku grid by their coordinates. This part will help to build the logic of how cell coordinates relate to each other.

A `Cell` is a type synonym for a pair of `Int`s. We will expect all such `Int`s will be between `0` and `8`.

In this part, you may handle any `Int` that is not in this range in any way you think is appropriate. These `Int`s will refer to a position in the grid and should not be confused with the value that is placed inside a cell. The cell `(0,0)` is understood to refer to the *top left* cell of the Sudoku. The first `Int` refers to the horizontal axis and the second to the vertical axis, so for example `(1,0)` refers to the cell to the *right* of `(0,0)`, whereas `(0,1)` refers to the cell *below* `(0,0)`.

- Complete the functions `sharesRow`, `sharesCol`, and `sharesBox`, each of which is given a `Cell` and returns a list of all `Cell`s that share the input's row, column, or 3x3 box respectively, not including the input `Cell` itself.

To explain in more detail, if we run `sharesRow (1,3)` we should receive the list

`[(0,3),(2,3),(3,3),(4,3),(5,3),(6,3),(7,3),(8,3)]`

(in any order) which corresponds to the coordinates of the cells sharing the row with the cell in the `1`st column (recalling that we start counting at `0`) and `3`rd row:

	2		7				4	
4		8	6					
		3			5	1		
6	4		8			3	7	
1						8		2
	9				3		5	
7	6			4				
			2	5				9
		1	9	8				6

`sharesCol` is similar, but should return the `Cell` coordinates above and below the input. To understand `sharesBox` we need to know that Sudoku grids are split into nine 3x3 boxes, so we should produce a list containing all the `Cell`s lying in the input's box, other than the input. For example `sharesBox (1,3)` will return the coordinates of all `Cell`s in the left-middle box except for `(1,3)` itself:

	2		7				4	
4		8	6					
		3			5	1		
6	4		8			3	7	
1						8		2
	9				3		5	
7	6			4				
			2	5				9
		1	9	8				6

Although we are using the Sudoku grid as visual inspiration here, note that this part does not require interaction with the `Sudoku` type and has nothing to do with the values held in any cell.

Part 1.3: Validating Sudokus (12 marks)

Call a `Sudoku Int` *partially valid* if

- It has nine rows, each containing nine `Int`s;
- All its `Int`s are between `0` and `9`;
- Each row has each value from `1` to `9` appearing *at most once*;
- Each column has each value from `1` to `9` appearing *at most once*;
- Each 3x3 box has each value from `1` to `9` appearing *at most once*.

The `example` Sudoku is partially valid. Call a `Sudoku Int` *valid* if it is partially valid and contains no blank spaces (indicated by `0`). The `example` Sudoku is not valid, but `solvedExample` is.

- Complete the functions `partiallyValid` and `valid` that tell us whether a `Sudoku Int` is (partially) valid.

Part 1.4: Suggesting Moves (12 marks)

Given an incomplete Sudoku, we would like our program to produce a recommended next move, if one can be found, to lead us towards to the solution. Because these moves often work by rejecting possible values for cells, we will need to work with pencilled Sudokus in this part. You are welcome, in this part, to assume without checking that any Sudoku we take as input to a function is well formed (i.e., that its unpencilled equivalent is partially valid).

While exhaustively crunching through every possible solution is one way to approach solving a Sudoku by computer, this is not how we will approach this assignment. Instead we will write a program than can recognise certain useful moves that humans can learn and use. Easy and intermediate Sudoku (what Sudoku9x9.com class as *Kindergarten*, *Elementary*, and *High School* levels) can be solved by a combination of just two such techniques: **Naked Singles** and **Hidden Singles**. We have defined for you a `Move` type whose elements are each one of these sorts of moves, along with some accompanying information.

- Complete the function `suggestMove` which analyses a pencilled Sudoku and suggests an applicable `NakedSingle` or `HiddenSingle` move, or returns `Nothing` if no such move is possible. There may in fact be more than one move available, but this function should return at most one.

A **Naked Single** move can be suggested when we have a cell that has been completed (has only one possible value), but that value also appears, once or more, in a cell or cells that share its row, column, and/or box. We would like to erase the completed cell's value from all those row/column/box sharing cells. The code that does the erasing will not be written until the next part. The job we want `suggestMove` to do is to return `NakedSingle c`, where `c` is the coordinates of the completed `Cell`. Note that no naked single move should be reported if the value of the completed cell does not appear elsewhere in its row, column, or box.

For example, the cell `(1,3)` in `pencil example` has value `[3]`, but there are many cells in `pencil example` with lists that include `3`, including some that share their row, column, or box with `(1,3)`.

Eventually we will wish to delete `3` from all those `Cell`s. Therefore `suggestMove (pencil example)` might return `nakedSingle (1,3)`. Note that there are many other such moves available for `pencil example`, so your implementation might suggest something different but still correct.

A **Hidden Single** move exists when a value has been excluded from every `Cell` in either a row, a column, or a box, except for one. Then that must be the `Cell` where that value goes! The actual insertion of that value will be coded later. We want `suggestMove` to return `HiddenSingle v c`, where `v` is the value that should be inserted, and `c` is the `Cell` coordinates where it should go. Note that no hidden single move should be reported if we already knew that `c` can only have value `v`.

For example, consider the bottom left cell of this puzzle:

1 5 8	7	1 5	1 4 6 8	9	4 6 8	3	2	1 4 8
1 2 3 8 9	1 2 3 8 9	4	1 7 8	1 2 7 8	2 7 8	6	5	1 7 8 9
1 2 8 9	1 2 8 9	6	5	3	2 4 7 8	1 4 7 9	7 8 9	1 4 7 8 9
2 3 7 9	2 3 6 9	2 3	3 4 6 7 8 9	6 7 8	1	5	6 7 8 9	2 4 7 8 9
4	1 6 9	1 5 9	2	5 6 7 8	6 7 8 9	1 7 9	6 7 8 9	3
1 2 3 5 7 9	1 2 3 6 9	8	3 4 6 7 9	5 6 7	3 4 6 7 9	1 4 7 9	6 7 9	1 2 4 7 9
6	5	2 3	1 3 8	1 2 8	2 3 8	7 9	4	7 9
2 3 9	2 3 9	7	3 6 9	4	2 3 6 9	8	1	5
1 8 9	4	1 9	1 7 8 9	1 7 8	5	2	3	6

(Here the small digits correspond to the list of possible values, while large digits are used where only one value is possible). There may appear to be several places where `8` could be in the leftmost column, and in the bottom row, but there is only one place where `8` could be in the bottom-left box. So while the bottom-left cell might appear to be compatible with values `1`, `8`, or `9`, in fact the detection of the hidden single in the box tells us that it must be `8`. Therefore `suggestMove` might return `HiddenSingle 8 (0,8)`.

Part 1.5: Solving Sudokus (12 marks)

While suggesting one useful `Move` might in some cases be enough for us to get unstuck on a tricky puzzle, we will push forward towards a completely automated solution.

- Complete the function `applyMove` to apply one `Move` to a pencilled Sudoku, producing a new pencilled Sudoku. You may assume the move is legal (i.e. that it comes from a correctly coded `suggestMove` function). If the `Move` is `NakedSingle c` you should delete every value that

matches the contents of `c` from the cells that share a row, column, and/or box with `c`. If the `Move` is `HiddenSingle v c` you should change only one `Cell`, overwriting the contents of `c` with the single value `[v]`.

- Complete the function `solutionList` which, given a pencilled Sudoku, produces a list of `Move`s which, if applied in order from the left of the output list to the right, will continue until no more `Moves`s can be detected.
- Complete the function `solve` which, given a `Sudoku Int`, outputs the `Sudoku Int` which is the result of solving it as far as possible using our `Move`s. Given an easy or intermediate Sudoku that is correctly designed, the output should be valid, but note that harder Sudokus (requiring `Move`s other than naked and hidden singles) will get stuck with some blank spaces remaining. If the Sudoku is in certain ways not correctly defined it may be that it stops being partially valid, if it ever was. Your `solve` function should detect this and throw an error.

Part 1.6: Advanced Techniques (6 marks)

You should not attempt this part unless you have mastered all other parts of the assignment. It is a fair amount of extra work for only a small amount of extra marks, and is designed to stretch top students. If you do decide to attempt it, we recommend saving a copy of your Haskell code for the simpler sections so that you can revert to submitting that if this section does not work out for you.

Naked Singles and Hidden Singles do not suffice for solving the 'College' level Sudokus on Sudoku9x9.com. For this, you must extend the `Move` type to include three new options: Naked Pairs, Hidden Pairs, and Locked Candidates. See descriptions on the website's techniques page, or search for yourself for resources, including videos, to understand these. You will then have to adapt every function that interacts with the `Move` type, most notably `suggestMove` and `applyMove`.

We will not award any marks to any attempts at techniques that go beyond College to 'Graduate' level, but note that to solve these very hardest Sudokus on the website you would need to further extend `Move` to include Naked Triplets and Quads, Hidden Triples and Quads, X-Wings, XY-Wings, and XYZ-Wings.

Part 2: Unit Tests (10 marks)

The file `TestSudoku.hs` contains some infrastructure to help you run a list of unit tests against your program. A `Test` type is defined that allows you to give each test a descriptive name as a `String`, and then specify what the actual test is as a `Bool`; a typical example might be checking that the output of some function on a certain input is `==` to what it ought to be. Lists of `Test`s are defined by us as an instance of `Show`, so that typing

```
ghci> test
```

will print out a positive result if everything passes, or announce each failing test if there are any (although if any of your tests cause your program to crash with an error, note that the later tests will *not* run).

- Complete the function `test`, deleting the rather silly tests that are currently sitting there as an example, and replacing them with your own. All the functions you were asked to write for Part 1 should be tested, along with any helper functions you defined for yourself.

Some hints:

- Try writing tests before you write code. Then work on your code until the tests pass. Then define some more tests and repeat. This technique is called *test-driven development*.
- The expected values in your test cases should be easy to check by hand. If the tested code comes up with a different answer, then it's clear that the problem is with the tested code and not the test case.
- Sometimes it is difficult to check an entire data structure returned by a function. In this case, consider focusing on some feature of the result that is easier to test. For example, instead of checking an entire Sudoku, perhaps you can check it has the correct value in a certain cell.
- If you find yourself checking something in `ghci` ask yourself: "should I make this into a unit test?". The answer is often "yes".
- If you are finding it difficult to come up with sensible tests, it is possible that your functions are doing too many things at once. Try breaking them down into smaller functions and writing tests for each.
- Putting functions into `where` clauses can be neat style, but can also make them harder to test. Consider 'promoting' such functions out of their `where` clause if you believe they require testing.
- If you want to write tests involving equality, or inequality, for a type, write `deriving Eq` under the type definition. If the type is already deriving something else, e.g. `Show`, then you can add the ability to test for equality to this by using parentheses and commas, e.g. `deriving (Show, Eq)`.
- It is not possible to test for a call to `error` using the tools provided in this course.

Part 3: Code Style (10 marks)

The code you wrote in both `Sudoku.hs` and `TestSudoku.hs` should exhibit good style, as described in lectures and labs.

Part 4: Technical Report (20 marks)

Write a concise technical report about your program.

MAXIMUM word count: 1500 — Your report must not have **more** than 1500 words. (In fact, the fewer words the better, provided you can get your ideas across.)

Marks for the report break down as

- Documentation (what you did): **6**
- Reflection (why you did it): **6**
- Testing (how you tested): **5**

- Style (the report presentation): **3**

Your report must be in PDF format, uploaded to your assignment directory, and named `Report.pdf`. Otherwise, it may not be marked, or will be marked but with a penalty.

The report must have a **title page** with the following items:

- Your name
- Your laboratory time and tutor
- Your university ID number

Content and Structure

Your audience is the tutors and lecturers, who already understand the basic concepts of programming and Haskell, and know the specification of this assignment. There is no need, for example, to explain how recursion works, or what a pencilled Sudoku is.

Your report should give a broad overview of your program, but focus on the specifics of what *you* did and why. Below is a potential outline for your report and some points you might discuss.

Introduction

If you wish to do so you can write an introduction. In it, give:

- A brief overview of your program:
  - how it works; and
  - what it is designed to do.

Documentation

The purpose of this section is to describe your program to the reader, both in detail and at a high level.

Talk about what features your program actually has. We know what we asked for, but what does your program actually let a user do? How does your program work as a whole?

How does it achieve this? Let us know how each individual function works and how they work together to solve particular design goals.

A successful report will demonstrate conceptional understanding of all relevant functions, and depicts a holistic view of program structure through discussion of what it is and hour it works.

Reflection

The purpose of this section is to describe the design decisions you made while writing the program.

Tell us the reasoning behind the design you detailed above. Tell us the assumptions you made about how the program might be used. Why did you solve the problems the way you did? Why did you

write the functions you wrote?

This is a *critical reflection* not a personal one. You're explaining the justification and reasoning behind the choices you made. You are welcome to discuss different design choices you considered and abandoned, but do not get too distracted by giving us a narrative account of your personal feelings during your work.

A successful report will give a thorough explanation of the process followed to reach a final design, including relevant reasoning and assumptions / influences.

Testing

This purpose of this section is to give the reader confidence that your program has been thoroughly tested.

Tell us how you tested the program as a whole to ensure correctness. Tell us in detail how you tested individual functions to ensure correctness. Discussion of your unit tests file will certainly form part of this section, but was there anything else you did to build your confidence in the correctness of your program?

A successful report will demonstrate evidence of a *process* that checked most, if not all, of the relevant parts of the program through testing. Such a report would combine this with some discussion of *why* these testing results prove or justify program correctness.

Style

A successful report should have excellent structure, writing style, and formatting. Write professionally, use diagrams where appropriate but not otherwise, ensure your report has correct grammar and spelling.

This is a list of **suggestions**, not requirements. You should only discuss items from this list if you have something interesting to write.

Things to Avoid

- Line by line explanations of large portions of code. (If you want to include a specific line of code, be sure to format as described in the "Format" section below.)
- Pictures of code or screenshots from Ed.
- Content that is not your own, unless cited.
- Grammatical errors or misspellings. Proof-read it before submission.
- Informal language - a technical report is a professional document, and as such should avoid things such as:
    - Unnecessary abbreviations (atm, wrt, ps, and so on), emojis, and emoticons; and
    - Stories / recounts of events not relevant to the development of the program.
- Irrelevant diagrams, graphs, and charts. Unnecessary elements will distract from the important content. Keep it succinct and focused.

- While many students like to use generative AI when writing, remember that your AI does not understand your code and does not understand your reasons for constructing your code in the way you did, and so you will not be able to outsource your report writing to AI. Also, many AIs write in a padded-out verbose style, while you are trying to be concise and focus on what is important.

If you need additional help with report writing, ANU Academic Skills have resources to help.

Format

You are not required to follow any specific style guide (such as APA or Harvard). However, here are some tips which will make your report more pleasant to read, and make more sense to someone with a computer science background.

- Colours should be kept minimal. If you need to use colour, make sure it is absolutely necessary.
- If you are using graphics, make sure they are *vector* graphics (that stay sharp even as the reader zooms in on them).
- Any code, including type/function/module names or file names, that appears in your document should have a mono-spaced font (such as Consolas, Courier New, Lucida Console, or Monaco)
- Other text should be set in serif fonts (popular choices are Times, Palatino, Sabon, Minion, or Caslon).
- Do not use underscore to highlight your text.

If you need to cite any sources that helped with the completion of your assignment, do so in your report. In particular, use these instructions if you need to acknowledge the use of generative AI.

You **must cite** any work that is not your own. You can only and will only be marked on work that you produced yourself.

Communicating

Do not post your code publicly on Ed Discussions or via any other means. Posts on Ed Discussions can trigger emails to students, depending on how they have configured their notifications, so if by mistake you post your code publicly, others will have access to your code and you may be held responsible for plagiarism.

Once again, and we cannot stress this enough: **do not post your code publicly** . If you need help with your code, post it *privately* to the instructors.

When brainstorming with your friends, **do not view each others' code**. There might be pressure from your friends, but this is for both your and their benefit. Anything that hints at plagiarism will be investigated and there may be serious consequences.

Sharing concepts and sketches is perfectly fine, but sharing should stop before you risk handing in suspiciously similar solutions.

The use of generative AI tools is permitted in this course, with appropriate citation.

Course staff will not look at assignment code unless it is posted **privately** in Ed Discussions, or shared in a drop-in consultation.

Course staff will typically give assignment assistance by asking questions, directing you to relevant exercises from the labs, or definitions and examples from the lectures.

Submission Checklist

- You have fully read and understand the entire assignment specification.
- In addition to the files provided to you to edit, you have uploaded a file called `Report.pdf` containing your technical report.
- You have fully cited any and all work (including generative AI) in your report.
- Your program compiles and runs in the Ed Lessons environment.