Assignment 1

EXERCISE:
RSLogo

Change Log

- Assignment Released (2024-03-02)
- Added clarification on how much to add to heading for each direction, to avoid test case issues (2024-03-25)

RSLogo

Logo is a programming language derived from lisp and other programming languages. Many older programmers' first experience of programming was writing Logo. One of Logo's main features is its "turtle": a mechanism for picking up and putting down a pen, and walking around to draw things.

Your task in this assignment will be to build a Logo interpreter, which writes to a .svg or .png image. You will be provided with some sample programs, which you are expected to implement correctly. If you pass these tests, and make a good-faith attempt to accurately implement the described features, you will get the marks for that section. There will be no surprising "hidden tests". Should you have a question about the correct functionality of this program, and the tests don't specifically answer it; you can assume it's undefined behaviour which you can implement however you wish (without purposeful malice).

The goals of this assignment are:

1. To get you to practice designing and structuring a larger Rust program.
2. To focus as much as possible on skills and design patterns that would actually be used when solving modern programming problems (i.e. writing a pull request, writing re-usable code, etc.)
3. To have fun creating an aesthetic and interesting application, and connecting with the history of programming.

We want to also be explicit about what the goals aren't:

1. To assess your ability to write large amounts of useless cruft solely for us to mark you on. Where you're writing code, we have a reason for it. Where you're writing text, it's because we want to genuinely understand your thinking.
2. To assess your skills at programming language design. Our Logo dialect has been explicitly designed to be as simple as possible, and we'll explain how to parse it (and give some hints on how to structure your interpreter).

Importantly: *the big challenge of this assignment is how to design your program*. There are no complicated algorithms (the most complicated thing you may need to do is some recursion, or building a stack... and that's one of the last stages), and no "tricks" you've got to search for; but it will be difficult to have all the functionality fit "neatly".

Part of this assignment involves submitting a `mark_request.txt` file. This gives you an opportunity to talk about both design excellence, and limitations in your design. If you can identify limitations in your design, we'll give you some marks even where we would have otherwise taken them away, since noticing bad design and learning from it is part of the challenge of the assignment.

An Introduction to the logo language:

A logo program consists of lines of text; which are then split up into tokens (whitespace-separated strings). Lines starting with `//` (i.e. two slashes on their own) or empty lines should be ignored as comments.

A token can be one of three things:

- A procedure: this is like a function, which can either be part of the interpreter, or implemented within the logo file. For example, the `MAKE` procedure sets a variable to a given value. Some procedures take arguments and some don't. In later stages of the assignment, we'll also see procedures which return a value.
- A variable: this is a token prefixed by a `:`, which means you should look up a variable with that name. For example, `:MYVARIABLE` might have the value `"42"`
- A value: this is prefixed with a `"`, and indicates a raw string. All values in logo are stored as strings.

Procedures in Logo always take a fixed number of arguments. For example, `MAKE` always takes two arguments.

Values in Logo are always strings. Some strings are special though: the exact (i.e. case-sensitive) strings `TRUE` and `FALSE` are interpreted by some operations as booleans. Similarly, If the `parse::<f32>` function can parse the string into a floating-point number; then the string can be used as a number.

You can choose how you want to handle errors in the language. You can either pre-check for errors in the program text; or you can just throw an error when you get to an irrecoverable error. The only requirements when you encounter an error are that you:

- Return with an exit code other than 0.
- Print a descriptive error message. We suggest that the error should include a relevant line (though since it's not always easy to say "what line did the error occur on", this isn't a hard requirement).

You must make sure that every token is used. For example, the `PENDOWN` command which you'll learn about takes no arguments. If you were to write `PENDOWN 3`, that would be an error, since the `3` means nothing.

An Introduction to Unsvg

In this assignment you will use the crate unsvg to generate SVG or PNG image. That crate contains two useful features:

- `unsvg::Image` which represents an image. It has methods like `draw_simple_line` which allow you to draw your image. **NOTE:** `draw_simple_line` returns the end coordinates of the line. You do not need to do any maths to work out where the line ends.
- `unsvg::get_end_coordinates`, which returns where a line drawn from a given point would end.

How your program will work

You will produce a program called `rslogo`, which always takes four arguments:

- a logo program file (these usually end in `.lg`).
- the path to the output SVG or PNG file (must end in `.svg` or `.png`)
- the height the image should be.
- the width the image should be.

We have provided you with starter code. You should use this code as a starting point for your assignment.

You can also run the command `6991 rslogo` to play with a sample solution.

Your program should:

1. Read the logo program.
2. Work line by line, parsing then executing the program.
3. Print an error, and then exit with a return code other than 0 if there's an issue.
4. If there are no issues, write an SVG or PNG using the unsvg crate.

The starter code deals with the basics of command-line arguments and importing the `unsvg` crate. You must use the `unsvg` crate to produce your images.

Design Excellence

Skip this section unless you're sure you want full marks in the assignment!

Part of the goal of this course is to get you to think about how to write truly great code. To this end, to get 100% in your "idiomatic design" component, your code requires at least one aspect of "design excellence": something that makes your design stand out from the crowd.

If you want to get full marks in this assignment (and it's OK if you don't!) then as you go through each part of the assignment below, think about whether you can do one of these tasks:

- Make your errors beautiful. This means:
  - Your errors should be descriptive, and should include the text of the line that has a problem.
  - Your errors should provide a hint for what might be wrong, where it's feasible to do so.
  - Your errors should be colourful, making it obvious what the text is, and what the issue is. Take inspiration from the Rust error messages if you can.
  - You could (but don't have to) highlight where in the line the issue is, like Rust does.
  - You may want to consider making use of one of the following diagnostic reporting crates:
    - miette
    - ariadne

    - chumsky
    - annotate-snippets
    - codespan-reporting

  This is "Design Excellence" because it requires structuring your program in a way that you can easily report errors; and making your program more usable.
- Get 80% test coverage for your application. You could use a library like tarpaulin to calculate your test coverage. Consider approaching this assignment in a Test-Driven Development style if you're aiming for this design excellence. This is design excellence because it requires you to think about how to test your program, and to design a testable program.
- Build a parser for `rslogo` that uses a parser combinator library (e.g. nom, winnow, chumsky).
- Create a facility for programmers to add a language extension. The idea is that they should only need to change like 1 or 2 lines of code, and add a useful command (for example, a REPEAT command that's like a while loop; or add trigonometric functions). Additionally, you should document all the public interfaces that are available to a programmer. You can use `#[warn(missing_docs)])` to identify any interfaces that should be documented.
- Build your program so it is as close to zero-copy as possible. This is design excellence because it requires you to think about how to write a performant program, and to think about how to avoid allocating memory.
- Write, and create a Pull Request for, a meaningful contribution to the `unsvg` library (can be documentation, a new feature, or a bug fix). Before doing this, file an issue so we know what you're planning on working on, and create a forum post requesting approval (so we can determine whether your plan is "meaningful" enough to award marks). This is design excellence because we think that a willingness and knowledge of how to contribute features to a library is a really important skill for a programmer to have, and generally improves the quality of the downstream library. **You must file the issue before Friday Week 6, so we have time to review it**.
- (hard) build a transpiler for `rslogo` that converts `rslogo` code to another language (e.g. Python, JavaScript, C, Rust) in order to run it. This isn't necessarily design excellence, but it'll force you into some interesting design challenges that are worth exploring.
- We will happily add more design excellence things here, however you need to run them past us first. If you have an idea, please make a forum post reqeuesting approval!

Note that when deciding if you've done "design excellence", markers won't be following strict guidelines: a reasonable, good-faith attempt at one of these tasks is enough. If you're not sure, ask us!

The Tasks To Complete

Part 1: Turtle Control (20%)

Completing up to this section means you'll be able to get a maximum of 20% for the Idiomatic Design and Functional Correctness sections

The first thing we'll get working in this language is controlling the "turtle". The turtle is like an invisible pen that can write on your image. It exists at particular coordinates. It starts "up" (i.e. not drawing), but when you put it "down", then moving it should draw a line. The pen also has a colour, which starts as white. The turtle starts facing straight up, in the center of the screen (note that if the image has odd dimensions, the turtle can be at a floating-point location).

The turtle can go off the image. The turtle leaving the image should not cause an error. You do not need to do any bounds-checking, or any special behaviour for the turtle leaving the image.

In this stage, you'll be required parse the following commands:

- `PENUP`
- `PENDOWN`
- `FORWARD [numpixels:f32]`
- `BACK [numpixels:f32]`
- `LEFT [numpixels:f32]`
- `RIGHT [numpixels:f32]`
- `SETPENCOLOR [colorcode:f32]`
- `TURN [degrees:f32]`
- `SETHEADING [degrees:f32]`
- `SETX [location:f32]`
- `SETY [location:f32]`

`PENUP` and `PENDOWN`

As described previously, the turtle can be in two states: "up" or "down". If the turtle is "up" (which is the state it starts in), then it does not do any drawing. It can move around without affecting the image. If the turtle is "down", it should draw a line wherever it travels.

`FORWARD "[numpixels]`, `BACK "[numpixels]`, `LEFT "[numpixels]`, `RIGHT "[numpixels].`

These commands move the turtle. Initially, "forward" means "up the screen", and "back" means "down the screen". Once you implement the turn/set-heading commands, these commands are relative to the direction the turtle is facing. The argument to this command must be a number. If you are unable to convert the argument to a number, you should print an error and exit with a non-zero exit code.

Turning left specifically means adding 270 degrees to your current heading and proceeding in that direction. Turning right means adding 90 degrees to your heading and proceeding in that direction. Going back means adding 180 degrees to your heading and proceeding in that direction.

If the turtle is "down", remember that these should also draw a line.

Note that the turtle is able to go off the image. You do not need to do any bounds-checking, or any special behaviour for the turtle leaving the image. numpixels can also be negative, which means the turtle "reverses" in that direction (i.e. `FORWARD "-10` goes back 10 pixels)

`SETPENCOLOR "[colorcode]`

The pen starts by drawing white lines. This changes what color the pen draws.

The pen color can be one of 16 options. They are specified in the `COLORS` array of `unsvg`. For example, `0` is black and `15` is grey.

```
SETPENCOLOR "15

SETPENCOLOR "Yeet
... (prints an error about an invalid color)
```

It should be an error if the number is not an integer.

`TURN "[degrees]` and `SETHEADING "[degrees]`

These commands can turn the turtle by a particular integer number of degrees, or set the heading to a particular direction. The heading 0 is heading up the image, 90 is heading right; 180 is heading south, and 270 is heading left. This means that a positive argument to `TURN` will turn the turtle clockwise, and a negative argument will turn it counter-clockwise.

Note that internally, you shouldn't normalise the heading. That is, if the turtle is facing 370 degrees, you should report the turtle as facing 370 degrees, not 10 degrees.

Having anything other than an integer as the `degrees` here should cause an error.

`SETX "[position]` and `SETY "[position]`

These commands set the X and Y position of the turtle respectively. The position can be any floating point. Note that even if the turtle is "down", these commands should not draw anything.

Part 2: Variables and Queries (20%)

Completing up to this section means you'll be able to get a maximum of 40% for the Idiomatic Design and Functional Correctness sections

Variables

The `MAKE` command in Logo is a procedure used to create and assign variables to values within the Logo programming language. When you use `MAKE`, you specify a variable name followed by a value. Logo then associates the variable with the provided value, effectively storing it for later use. This enables programmers to store and manipulate data, facilitating the creation of more complex algorithms and procedures. For example, `MAKE "x "10` would assign the variable `x` to have the value `10`, allowing subsequent commands to reference and manipulate this value throughout the program's execution.

Additionally, you will need to implement the `ADDASSIGN` command. This is the equivalent of `+=` in most programming languages. It should be given the name of an existing variable, a value to add, and try to add the two together. It should be an error for the variable to not exist. It should also be an error if either argument is not a number.

Variables have no scope, so once you create them they are available anywhere. Variables can also be overwritten.

While in the sample solution, variables can have any name (like being plain numbers), we won't be testing this behaviour. You should ensure that any valid rust variable name is also valid in your implementation; but you don't have to validate this.

You can then use a variable anywhere you could use a raw value. This includes using variables as the value in a `MAKE` statement . For example, these two programs should be equivalent:

```
FORWARD "6
```

```
MAKE "DISTANCE "3
ADDASSIGN "DISTANCE :DISTANCE
FORWARD :DISTANCE
```

Queries

Additionally, you should support "queries", which are procedures which "return" a value. They act very similarly to variables; except they don't use `:`s at the start, and they are set automatically by the program. The queries are:

- `XCOR`: return the current x coordinate.
- `YCOR`: return the current y coordinate.
- `HEADING`: return the current heading.
- `COLOR`: return the color of the pen, as an number.

Part 3: IFs, WHILE, `[ ]` (20%)

Completing up to this section means you'll be able to get a maximum of 60% for the Idiomatic Design and Functional Correctness sections

One of the fundamental concepts in Logo is conditional execution and looping. To enhance the capabilities of your Logo interpreter, you will implement two important commands: `IF` and `WHILE`. These commands will help users to create more complex and dynamic drawings and programs.

In this section, you'll only be required to implement the `IF EQ` and `WHILE EQ` commands. In a future section, we will explore more types of comparisons.

`IF EQ` Command

The `IF EQ` command introduces conditional execution in Logo. It allows you to specify two values, and if those two values are the same (i.e. they are identical strings), a specific block of code will be executed. If they are not the same, the block of code should not execute.

In our dialect of Logo, this is what the command will look like:

```
IF EQ <value1> <value2> [
  <line of code>
  <line of code>
]
```

- `<value1>` and `<value2>`: These are the two values you'll be checking whether are the same or different. They could be raw values, queries, or variables.
- `<line of code>`: These are lines of code which should execute if the condition is true. Note that this may contain more `IF` statements (or `WHILE`s).

WHILE EQ Command

The `WHILE EQ` command is identical to `IF EQ`, except that at the end of the block, you should check the conditions again. If the conditions are again equal, you should repeat the block. The block should be repeated continuously until the condition is false.

```
WHILE EQ <value1> <value2> [
  <line of code>
  <line of code>
]
```

Part 4: Implementing Maths and Comparisons using a Stack (20%)

Completing up to this section means you'll be able to get a maximum of 80% for the Idiomatic Design and Functional Correctness sections

In the world of computer programming and mathematical notation, there are multiple methods to represent and evaluate mathematical expressions. One method used in programming languages like Logo is known as **Polish Notation**. Polish Notation, also referred to as prefix notation, differs from the more traditional infix notation by placing operators before their operands. In this section, we will delve into the concept of Polish Notation, explore the idea of a stack, and introduce several fundamental mathematical operations in Logo, including GT, LT, EQ, NE, +, -, *, and /.

In Logo, to write the equivalent of Rust's `3 + 4`, you should write: `+ "3 "4`. This is kind of like calling `+(3, 4)`, where the `+` function adds two numbers together.

A more complicated expression in Logo would be `EQ * "3 "4 * + "3 "3 "2`. Rewriting this in the style above, this is: `==(*(3, 4), *(+(3, 3), 2))`. Rewriting it again, it's `3 * 4 == (3 + 3) * 2`, which evaluates to `TRUE`.

The full list of operators in Logo are:

- `EQ <arg1> <arg2>`: returns `TRUE` if `arg1` is equal to `arg2`, and `FALSE` otherwise. neither a number, a boolean nor a word, raise an error.
- `NE <arg1> <arg2>`: returns `TRUE` if `arg1` is not equal to `arg2`, and `FALSE` otherwise. If either `arg1` or `arg2` is neither a number, a boolean nor a word, raise an error.
- `GT <arg1> <arg2>`: returns `TRUE` if `arg1` is greater than `arg2`, and `FALSE` otherwise. If either `arg1` or `arg2` is not a number, raise an error.
- `LT <arg1> <arg2>`: returns `TRUE` if `arg1` is less than `arg2`, and `FALSE` otherwise. If either `arg1` or `arg2` is not a number, raise an error.
- `AND <arg1> <arg2>`: returns `TRUE` if `arg1` and `arg2` are both `TRUE`, and `FALSE` otherwise. If either `arg1` or `arg2` is not a boolean, raise an error.
- `OR <arg1> <arg2>`: returns `TRUE` if `arg1` or `arg2` is `TRUE`, and `FALSE` otherwise. If either `arg1` or `arg2` is not a boolean, raise an error.
- `+ <arg1> <arg2>`: returns the result of adding `arg1` and `arg2`. If either `arg1` or `arg2` is not a number, raise an error.
- `- <arg1> <arg2>`: returns the result of subtracting `arg2` from `arg1`. If either `arg1` or `arg2` is not a number, raise an error.
- `* <arg1> <arg2>`: returns the result of multiplying `arg1` and `arg2`. If either `arg1` or `arg2` is not a number, raise an error.
- `/ <arg1> <arg2>`: returns the result of dividing `arg1` by `arg2`. If either `arg1` or `arg2` is not a number, raise an error. Additionally, if `arg2` is zero, raise an error to avoid division by zero.

You should implement all of these operations such that they can be used anywhere a raw value or variable could be used. Note that the arguments to these operations may themselves be more operations; values; or functions.

particularly, you will need to implement stack operations on `IF` and `WHILE`.

Also note that if the line doesn't make sense (i.e. you don't provide all the arguments) you should raise an error.

Part 5: Logo Defined Procedures (20%)

Completing up to this section means you'll be able to get a maximum of 100% for the Idiomatic Design and Functional Correctness sections

All good programming languages allow you to break down your code into smaller pieces. Logo allows this through Procedures. These are analogous to functions or subroutines in other programming languages. They allow for the encapsulation of a series of Logo commands into a single named unit, making the code more organized, reusable, and easier to manage.

The syntax for this in Logo is to have a line starting with `TO`, then name the procedure, and list out its arguments. Procedures could have any number of arguments (including 0 arguments). The procedure ends on the first line after that has the command `END`.

```
TO DoSquare "arg1
  FORWARD :arg1
  LEFT :arg1
  BACK :arg1
  RIGHT :arg1
END
```

Here are some important notes which make your life easier:

- Procedures in our Logo will always be defined on a line above where they're used.
- A procedure can call other procedures, however a procedure will never call itself (i.e. procedures will not be recursive).
- Procedures will never be defined inside `[]` (i.e. they won't be conditionally defined).
- Procedures will never be defined inside other procedures.
- You do not need to worry about scope within procedures: at the end of the procedure, the arguments to the procedure should not be changed or deleted.
- Procedures will never be created with the same name.

Common Questions

How can I design this assignment?

There are two major approaches, both will work; however both have their own challenges.

1. line-by-line

This is the more "obvious" approach: you make a while loop which starts on line 1, and executes each line in turn. As you go, you'll need to store more data and more state, and you'll need to do things like change what line you're reading next in order to support if statements.

2. parse then execute

This approach is more like how modern programming languages approach executing a program. In this approach, you convert the text into a datastructure called an "Abstract Syntax Tree". Then, rather than reading the text of the Logo code directly; you build your system so it can read this tree and understand it.

This means that the line:

```
MAKE "VarName + 3 4
```

Might be represented in your program as:

```
MakeCommand(
  String::from("VarName"),
  Box::new(
    Add(Box::new(Value(3)), Box::new(Value(4))
  )
)
```

This can often result in a neater program, but it requires more effort and pre-planning.

How much should I plan before starting the assignment?

This is a large program, and one way of approaching the task is to make a plan for how you'll implement all the functionality first. Another way of approaching the task is to just take it step by step, rewriting your code as you go.

Both approaches will work for this assignment, and we expect both will take a similar amount of time. Pre-planning takes effort, and then you may find your plan doesn't work as expected. Just diving in to the programming may mean you have to rewrite swathes of code when a requirement you didn't fully understand comes along.

We suggest a happy middle: think about what approach you're going to take, and read through the assignment. But don't spend more than 30 mins planning your approach: you'll learn more from doing than from just thinking.

Can I use AI?

It's 2023, so many programmers are using AI assistants. In this assignment, you're permitted to use AI. In fact, rather than telling you "don't use it", our answer is: "here's how to use it".

Use 1: For help with a concept

If you've forgotten something, maybe try asking ChatGPT what the thing was. Remember: AI can't fact-check the answer, but for simple questions it's OK. See if you can confirm its answer through a non-AI source.

Use 2: For pattern matching

If you've written a couple lines of code that work for `EQ`, you can get Copilot (or ChatGPT) to write the same lines but for `NE`, `GT`, etc.

Use 3: For a skeleton

If you're writing some code, and you want to get a match block, type `match` and then Copilot will usually fill in the context for you. Note that Rust-Analyzer can do this without an error-rate that exists in AI assistants, but will simply leave match arms as `todo!()`. AI assistants could fill out simple cases.

Use 4: For tests

Usually writing a comment is enough for an AI to generate you a pretty reasonable test for any functionality you include.

Other Information

Submission

See the instructions down the bottom of the page.

Using Other Crates

We are happy for you to use any crate that has been published on crates.io under three conditions:

> The crate must not have been authored by anyone else in the course.
> The crate must have at least 1000 downloads, excluding the last 30 days.
> The crate must not impose license restrictions which require you to share your own code.

If you are in doubt (or think these restrictions unfairly constrain you from using a reasonable crate), ask on the course forum.

Marking Scheme

There are 3 things on which you will be marked:

- Mechanical Style (10% of the total marks for this assignment.)
- Functional Correctness (40% of the total marks for this assignment times the percentage of the assignment completed)
- Idiomatic Design (50% of the total marks for this assignment, times the percentage of the assignment completed)

And a detailed analysis is shown below:

1. Mechanical Style (10%):

We will look at your crates, and make sure they:

- Compile, with no warnings or errors.
- Raise no issues with `6991 cargo clippy`.
- Are formatted with `rustfmt` (you can run `6991 cargo fmt` to auto-format your crate).
- Have any tests written for them pass.

If they do all of the above, you get full marks. Otherwise, we will award partial marks. This is meant to be the "easy marks" of programming.

2. Functional Correctness (40%):

You should pass the provided test cases. We will vary the test case very slightly during marking, to ensure you haven't just hard-coded things; but we're not going to do anything that's not just changing around some commands and re-ordering things.

3. Idiomatic Design (50%):

Your code should be well designed. This is where we will spend most of our time when marking. To help you, we have provided "design excellence" suggestions, which are ideas to make your design really excellent. You don't have to do them, but they would be good ways of getting a great design.

The following list of properties will be marked in your program:

- Code is written with the reader in mind, so it's organised properly, using somewhat reasonable names.
- Functions accept reasonable arguments for what they're doing.
- Large parts of code are not repeated without good reason.
- You don't just have "one long main function" style code.
- Types are used appropriately to express data in the program.
- The design does not impose unnecessary constraints on either the caller or callee through borrowing, lifetimes or ownership.

- Uses traits sensibly to add expressiveness and avoid unnecessary code.
- Data structures used are appropriate to store data.
- Functions perform error handling; cases that are expected do not panic.
- Code is sensibly organised, and split into appropriate modules.
- Documentation, where provided, is correct and readable.
- (optional) External crates are used effectively to achieve the above goals.
- (optional) Where code is designed in a sub-optimal way, comments about how to improve it are made under "Design Limitations".

IMPORTANT: your marks for the assignment **are not** the percentage of tests which you pass. We'll scale the tests to fit in with the weights described above.

You *must* complete the checklist of the `mark_request` faithfully. if you do not fill in the file, you may receive reduced Idiomaitc Design Marks.

Your mark will be calculated based on the feedback you have received:

100% of available marks	Very little negative feedback is given on the above criteria; and a design excellence suggestion is implemented.
85% of available marks	Some minor comments are made about some of the above criteria. Above this mark, one design excellence suggestion will have been implemented.
75% of available marks	Major comments are made about one or two criteria, with multiple small comments in different areas.
65% of available marks	Major comments are made about three or more criteria.
50% of available marks	Many areas have major comments made.
below 50% of available marks	Assignments in this category are likely written as "translations from C", and ignore many Rust features and design patterns.

Note that the following penalties apply to your total mark for plagiarism:

0 for the assignment	Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 for the assignment	Submitting any other persons work. This includes joint work.
0 FL for COMP6991	Paying another person to complete work. Submitting another persons work without their consent.

Formal Stuff

Assignment Conditions

- **Joint work** is **not permitted** on this assignment.

  This is an individual assignment.

  The work you submit must be entirely your own work. Submission of any work even partly written by any other person is not permitted.

  The only exception being if you use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from a site such as Stack Overflow or other publicly available resources. You should attribute the source of this code clearly in an accompanying comment.

  Assignment submissions will be examined, both automatically and manually for work written by others.

  Do not request help from anyone other than the teaching staff of COMP6991.

  Do not post your assignment code to the course forum.

  Rationale: this assignment is an individual piece of work. It is designed to develop the skills needed to produce an entire working program. Using code written by or taken from other people will stop you learning these skills.

- The use of **code-synthesis tools** is permitted on this assignment, however beware -- the code it creates can be subtly broken or introduce design flaws. It is your job to figure out what code is good. Your code is your responsibility. If your AI assistant blatantly plagiarises code from another author which you then submit, you will be held accountable.

  Rationale: this assignment is intended to mimic the real world. These tools are available in the real world. However, you must be careful to use these tools cautiously and ethically.

- **Sharing, publishing, distributing** your assignment work is **not permitted**.

  Do not provide or show your assignment work to any other person, other than the teaching staff of COMP6991. For example, do not share your work with friends.

Do not publish your assignment code via the internet. For example, do not place your assignment in a public GitHub repository. You can publish Workshops or Labs (after they are due), but assignments are large investments for the course and worth a significant amount; so publishing them makes it harder for us and tempts future students.

Rationale: by publishing or sharing your work you are facilitating other students to use your work, which is not permitted. If they submit your work, you may become involved in an academic integrity investigation.

- **Sharing, publishing, distributing your assignment work after the completion of COMP6991** is **not permitted** .

  For example, do not place your assignment in a public GitHub repository after COMP6991 is over.

  Rationale: COMP6991 sometimes reuses assignment themes, using similar concepts and content. If students in future terms can find your code and use it, which is not permitted, you may become involved in an academic integrity investigation.

Violation of the above conditions may result in an academic integrity investigation with possible penalties, up to and including a mark of 0 in COMP6991 and exclusion from UNSW.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted - you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

If you have not shared your assignment, you will not be penalised if your work is taken without your consent or knowledge.

For more information, read the UNSW Student Code, or contact the course account.

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 7 Wednesday 22:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.
