Australian National University

» Assignments » Assignment 1: Shapes

In this assignment, you will build a Haskell program that uses the CodeWorld API to draw colourful shapes on the screen, including lines, polygons, rectangles, circles and parallelograms.

This assignment is worth **10%** of your final grade.

Deadlines: Part A: Sunday March 21, 2021, at 11:00pm Canberra time *sharp* Part B: Sunday April 4, 2021, at 11:00pm Canberra time *sharp*

Indicative marks and feedback for Part A will be returned in week 6.

Required Knowledge

If you have finished the Week 3 lab, you should be able to complete Part A.

If you have finished the Week 4 lab, you should be able to complete the majority of the assignment. Some parts require recursion over lists, which is covered in the Week 5 lab.

Overview of Tasks

	COMP1100 Marks	COMP1130 Marks
Task 1: Helper Functions	20 marks	10 marks
Task 2: Rendering Shapes	30 Marks	25 Marks
Task 3: Handling Events	30 Marks	25 Marks
Task 4: Mirrors	5 Marks	5 Marks
1130 Extensions	0 Marks	30 Marks
Technical Report	15 marks	25 marks

Part A of the assignment requires you to complete Task 1. **Part B** of the assignment requires you to complete all assigned tasks. As you complete each task (or even each function within a task), you should commit and push your work with a sensible commit message.

The purpose of Part A is to give you an opportunity to collect feedback on your code and your progress in the course, and for us to give you an indicative mark for your work so far. This will be returned to you before the census date. Part A will be re-marked alongside your Part B submission, giving you a final mark for the assignment.

Getting Started

1. Fork the assignment repository and create a project for it in VSCodium, following the same steps as in Lab 2. The assignment repository is at https://gitlab.cecs.anu.edu.au/comp1100/comp1100-assignment1.
2. Add **our** version of the repository as a *remote* called `upstream`. This allows us (the tutors) to provide additional fixes if they are required. You do this by doing the following:
   - Go to the command palette in VSCode (or VSCodium) by pressing `Ctrl + Shift + p`
   - Type `git remote`
   - Click **Git: Add Remote**
   - Enter `upstream` into the box for the remote name
   - Put the following URL as the remote url: `https://gitlab.cecs.anu.edu.au/comp1100/comp1100-assignment1.git`.

Overview of the Repository

Most of your code will be written to Haskell files in the `src/` directory. We are using the model-view-controller pattern to structure this assignment. Each file is called a *module*, and we use modules to group related code together and separate unrelated code.

Model.hs

The *model* is a data type that describes the state of the running program. The program will move to new states (new values of type `Model`) in response to user actions, as defined by the *controller*.

View.hs

The *view* turns the *model* into something that can be shown on the screen; in this project, that is the CodeWorld `Picture` type.

Controller.hs

The *controller* considers user input (and other events), along with the current *model*, and uses that to decide what the new *model* should be.

Other Files

- `tests/ShapesTest.hs` contains some *unit tests* - simple checks that verify small parts of your program are working correctly. You are not required to write tests for this assignment, but you might find it useful to do so.
- `tests/Testing.hs` is a small testing library used by `tests/ShapesTest.hs`. You are not required to understand it for this assignment.
- `app/Main.hs` ties your functions together into the final program that runs. You are not required to understand it.
- `comp1100-assignment1.cabal` tells the cabal build tool how to build your assignment. You are not required to understand this file, and we will discuss how to use cabal below.
- `Setup.hs` tells cabal that this is a normal package with no unusual build steps. Some complex packages (that we won't see in this course) need to put more complex code here. You are not required to understand it.

Overview of Cabal

`cabal` is the build tool for Haskell programs and libraries. It provides several useful commands:

- `cabal v2-build`: Compile your assignment.
- `cabal v2-run shapes`: Build your assignment (if necessary), and run the `shapes` program.
- `cabal v2-repl comp1100-assignment1`: Run the GHCi interpreter over your project. This gives you the same ghci environment you use in labs, but with the assignment code loaded. (Aside: REPL is the name for interactive sessions like GHCi - it stands for read-eval-print loop. Many modern languages have REPLs.)
- `cabal v2-test`: Build and run the tests. Tests will abort on the first failure, or the first call to a function that is still `undefined`.

You should execute these cabal commands in the **top-level directory** of your project: `~/comp1100/assignments/assignment1` (i.e., the directory you are in when you launch the VSCodium Terminal for your project).

Overview of the Program

You use a web browser to interact with the `shapes` program that you launched with `cabal v2-run shapes`. Once you have completed the assignment, it will respond to the following actions:

Action	**Effect**
`Esc` (key)	Clear the canvas
`M` (key)	Display a mystery image
`C` (key)	Cycle colour (of shape to draw)
`T` (key)	Cycle tool (type of shape to draw)
`Backspace`/`Delete` (key)	Remove the last added shape
`D` (key)	Print the current `Model` to the terminal (useful for testing)
Click-drag-release (mouse)	When not drawing polygons or parallelograms, draw a shape between the click-point and release-point.
Click (mouse)	When drawing polygons or parallelograms, add a point.
`Spacebar` (key)	When drawing a polygon, finish drawing the polygon, adding it to the canvas. Otherwise, nothing.
`X` (key)	Flip the canvas about the x-axis
`Y` (key)	Flip the canvas about the y-axis

Task 1: Helper Functions (COMP1100: 20 marks, COMP1130: 10 marks)

The easiest way to solve a large problem is often to break it apart into smaller, easier problems. Programming is the same. In this task you will write some helper functions that will make future tasks easier. You can test your implementations by running `cabal v2-test`.

The functions you need to write for this task are:

- `toolToLabel` in `src/View.hs`. This function should return instructions for the user on how to use each `Tool`, according to the following table:

Tool	Label
`LineTool`	`"Line... click-drag-release"`
`PolygonTool`	`"Polygon... click 3 or more times then spacebar"`
`RectangleTool`	`"Rectangle... click-drag-release between opposite corners"`
`CircleTool`	`"Circle... click-drag-release from opposite points on the circumference"`
`ParallelogramTool`	`"Parallelogram... click three points, each adjacent to the next"`
`SquareTool`	`"Square... click-drag-release between midpoints of opposite sides"`

Note: At the time this assignment is released, the course will have only briefly covered lists. You do not need to manipulate lists to write `toolToLabel`; you can use a blank pattern (`_`) to ignore them.

- `nextColour` in `src/Controller.hs`. This function should return the next colour in our set of `ColourName`s:

Argument	Result
`Black`	`Purple`
`Purple`	`Blue`
`Blue`	`Green`
`Green`	`Yellow`
`Yellow`	`Orange`
`Orange`	`Red`
`Red`	`Black`

- `nextTool` in `src/Controller.hs`. This function implements tool-switching, but should not change `Tool` if the user is halfway through an operation:
  - If the tool is not holding a point (that is, a non-`PolygonTool` tool holding `Nothing` (in all fields) or a `PolygonTool` holding the empty list `[]`), select the next tool in the following sequence: `Line` -> `Polygon` -> `Rectangle` -> `Circle` -> `Parallelogram` -> `Square` -> `Line` -> `...`.
  - If there is a `Point` stored in the given tool (because it's holding a `Just` value or the list in `PolygonTool` is non-empty), return the argument unchanged.
  - If this is unclear, study the `nextToolTests` in `test/ShapesTest.hs`.

Note: At the time this assignment is released, the course will have only briefly covered lists. You can write the `PolygonTool` case for `nextTool` without using list recursion. Use `[]` to match an empty list. In a subsequent case, give the entire list a name like `points` to match any nonempty list (or find a way to use the blank pattern!).

Part A ends here.

Submitting Part A

Your submission for Part A should include implementations of `toolToLabel`, `nextColour`, and `nextTool` that compile without warnings and pass the tests run by `cabal v2-test`. You are welcome to continue working on Part B of your assignment and committing and pushing changes, so long as the code continues to compile and the tests continue to pass.

Part B begins…

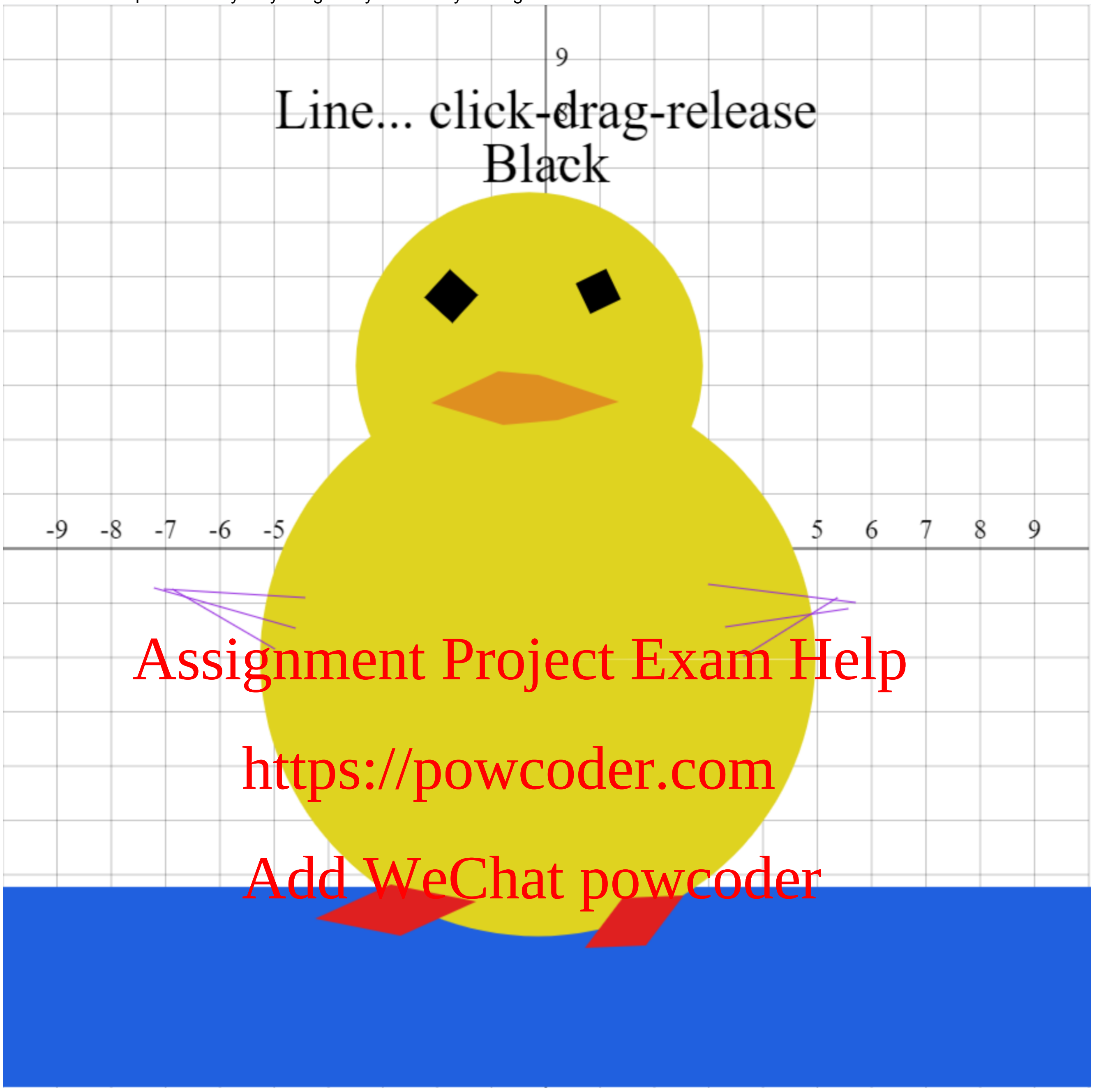
Task 2: Rendering Shapes (COMP1100: 30 marks, COMP1130: 25 marks)

In `src/View.hs`, `modelToPicture` converts your `Model` type into a CodeWorld `Picture`, so that it can be displayed on the screen. It currently does not work, because `colourShapesToPicture` is `undefined`. In this task you will fill in that missing piece, building up a function to convert the `[ColourShape]` from your `Model` into a `Picture`. You can test these functions individually by using `cabal v2-repl comp1100-assignment1`, using `drawingOf` to show small pictures on the screen. You can also test everything as a whole by launching the program with `cabal v2-run shapes` and pressing the `M` key to show the mystery image. The functions you need to write for this task are all in `src/View.hs`:

- `colourNameToColour`: This function turns your `ColourName` type from the model into a CodeWorld `Colour`. You should check the CodeWorld documentation for information on colours.
- `shapeToPicture`: This function turns your `Shape` type into a CodeWorld `Picture`. You will need to consider the constructors for `Shape` individually, and work out the best way to turn each one into a `Picture`. Here are some hints to help you along:
    - CodeWorld has no function to draw a single line segment. It does have a function to draw a line made of multiple segments - `polyline`.
    - Polygons, rectangles, squares, circles and parallelograms should be drawn as solid (filled) `Picture`s.
    - Most of CodeWorld's functions draw individual shapes centred on the origin - `(0, 0)`. You will need to figure out how to slide (translate) the generated `Picture` so it winds up where it's supposed to go. Drawing diagrams will help. The `abs` function will also help - it computes the absolute value of its argument (i.e., `abs x == x` if `x > 0`, and `abs x == negate x` otherwise).
    - While CodeWorld has no function *for* parallelograms, it does have a function that *can* draw parallelograms.
    - Squares are specified by opposite midpoints on two of the square's edges. Convince yourself that this uniquely defines a square, and then think of the possible ways you could turn it into a CodeWorld picture.

    moving onto others. If you have written code to render `Rectangle`s but nothing else, you can create a drawing out of `Rectangle`s only, and check that your other functions work, before returning to finish `shapeToPicture`.
- `colourShapeToPicture`: This function should render the `Shape` and colour it using the Colour that corresponds to the given `ColourName`.
- `colourShapesToPicture`: This function should turn every `ColourShape` in a list into a single `Picture`. You will need to recurse over the input list. If you have not yet completed Lab 5, you may want to work on other parts of the assignment and come back to this.

- Here is an example of the mystery image for you to test yours against:



Task 3: Handling Events (COMP1100: 30 marks, COMP1130: 25 marks)

It is now time to tackle `handleEvent` in `src/Controller.hs`. CodeWorld calls this function whenever something interesting happens (like a key press, a pointer press, or a pointer release). This function is called with two arguments:

- The `Event` that just happened, and
- The current `Model` at the time the `Event` happened.

`handleEvent` then returns a new `Model` for the program to use moving forward.

(Aside: Elm is a functional programming language that uses a similar pattern to build front-end web applications that are compiled to JavaScript.)

Let's trace a simple interaction. If the user wants to draw a *red line* by clicking on the screen at coordinates $(1, 1)$ and releasing the mouse at coordinates $(2, 2)$. starting at a blank canvas, the state would transition as follows, starting with the initial model:

1. `Model [] (LineTool Nothing) Black`

2. The user presses "C" to change the colour from black to red:

```
Model [] (LineTool Nothing) Red
```

3. The user presses the mouse button at $(1,1)$ changing the state to

```
Model [] (LineTool (Just (1.0,1.0))) Red
```

4. The user releases the mouse button at $(2,2)$ changing the state to

```
Model [(Red,Line (1.0,1.0) (2.0,2.0))] (LineTool Nothing) Red
```

Note that the `Tool` and the `ColourName` do not reset to the default values after a shape has been drawn. However, the `Maybe Point` inside the tool reverts to `Nothing`.

Task 3.1: Handling Mouse Input

CodeWorld provides a few different event constructors for mouse input, but the ones we're interested in here are `PointerPress` for when the user clicks, and `PointerRelease` for when the user releases the mouse button.

When a `PointerPress` event arrives, you will need to store it in the current `Tool`. For everything except `PolygonTool`, you will store it in the `Maybe Point` argument. For `PolygonTool`, you will add it to the list of vertices. For the third `PointerPress` event accepted by the `ParalellogramTool`, you should construct the appropriate shape and add it to the `[Shape]` in the `Model`.

When a `PointerRelease` event arrives, we can ignore it for `PolygonTool` and `ParalellogramTool`. We'll finish polygons using the spacebar in Task 3.2, and `Paralellograms` are mentioned above. For everything else, a `PointerRelease` will mean the end of a click-drag-release action, so you should construct the appropriate shape and add it to the `[Shape]` in the `Model`. You should also remove the starting point from the current `Tool`, so that future shapes draw properly too.

Task 3.2: Handling Key Presses

To handle keyboard input, CodeWorld provides a `KeyPress` event. This case is already present in the assignment skeleton, because we have implemented some keyboard functionality already. In the “Overview of the Program” section, we listed the full set of keyboard commands that your program will respond to. You need to implement the missing functionality for these keys:

Key	Effect
`C`	Cycle colour (of shape to draw)
`T`	Cycle tool (type of shape to draw)
`Backspace`/`Delete`	Remove the last added shape
`Spacebar`	Finish drawing a polygon, adding it to the canvas.

If you have made it this far, you should not need to write a lot of code to implement these. A few hints:

- Think back to Task 1.
- `Backspace`/`Delete` with no shapes drawn should not crash the program.
- Nor should any other unexpected input. Try to test all unexpected” cases.

Task 4: Mirrors (COMP1100/1130: 5 Marks)

Note that this task will also require lab 5, it is more difficult than the previous tasks, and it is not worth many marks. For 1100 students especially, you may wish to begin writing your report before attempting this task.

For this task you will add functionality to take a mirror image of any drawing. i.e., you will be able to flip the canvas about the y axis - a standard mirror image - and about the x axis - as if looking into the reflection on a pool of water below the drawing.

Key	Effect
`X`	Flip all shapes in the canvas about the x axis
`Y`	Flip all shapes in the canvas about the y axis

In order to add this functionality, you will also need to create a function which recurses through the `[ColourShape]` and alters each shape's position (and in the case of squares, paralellograms and polygons, orientation) so that they are flipped about the appropriate axis. Note that for a flip about the x axis, $(x,y) \mapsto (x,-y)$ and for a flip about the y axis, $(x,y) \mapsto (-x,y)$

Task 5: 1130 Extensions (COMP1130 Only: 30 Marks)

COMP1100 students are welcome to attempt extensions so long as that does not inhibit the function of tasks 1-4, but will not attract marks for completing them.

COMP1130 students will need to complete *two* of the *six* extensions in the pool below. If you attempt more than two, you must indicate which two you wish to be marked. Otherwise, we will mark the the first extensions you have completed in the order they appear in this document.

If you change the behaviour of tool or colour switching, you may want to adjust the tests in `tests/ShapesTest.hs' so they continue to pass. They *must* continue to compile without errors.

Note: Extension functionality must be executed through the codeworld API just like the rest of the assignment. Interactions should be received through the `handleEvent` function, which you will need to modify. If you need to add fields to the model, please do so after the existing ones: `data Model = Model [ColourShape] Tool ColourName YourField1 YourField2 ...`

Task 5.1

Allow the user to delete multiple shapes using the mouse. The user should enter some kind of "mouse delete mode" and then delete multiple shapes at once by pressing the mouse button and dragging a rectangle around them. When the user releases the mouse, any shape that is *wholly* within the rectangle should be deleted.

If you wish, you may implement a clipboard (cut, copy and paste) instead of just delete. If you do, make sure to clearly document how this can be used in your report.

Task 5.2

Add a free-hand drawing tool. When using this tool, moving the mouse with the button pressed should draw short lines or curves, as if a pen were being dragged across the screen. When the button is released, the program should stop drawing.

Task 5.3

Add the ability to select and move a shape using the keyboard. How you select shapes, how you move them, and how you enter and leave the "keyboard movement" mode is up to you. But make sure you document how this should be done clearly in your report.

Task 5.4

Preview shapes as the user draws them. If the user has started to draw a shape, then you should show the shape currently being drawn using a hollow (not solid) rectangle, polygon, circle, or square. For polygons and parallelograms, you should not give a preview until at least two points have been saved.

Task 5.5

Extend Task 4 so that you can reflect about any line that crosses through the origin ($(0, 0)$). You should give a convinient way to select such a line and to action the reflection. After completing this exercise, you will have created the functionality of task 4 twice, this time more generally. As such, after successfully completing this task, you may remove your task 4 code.

Task 5.6 (Difficult)

Each time a new shape is added to the canvas, check whether its perimeter intersects with that of other shapes. If it does, mark the point(s) of intersection with a small x. When a shape is removed, the x's at any intersections with that shape should also be removed.

Technical Report (COMP1100: 15 marks, COMP1130 25 marks)

You should write a concise technical report explaining your design choices in implementing your program. The **maximum word count is 1000** for COMP1100 students and **1500** for COMP1130 students. This is a *limit*, not a *quota*; concise presentation is a virtue.

Once again: These are not required word counts. They are the **maximum number of words** that your marker will read. If you can do it in fewer words without compromising the presentation, please do so.

Your report must be in PDF format, located at the root of your assignment repository on GitLab and named `Report.pdf`. Otherwise, it may not be marked, or will be marked but with a penalty. You should double-check **on GitLab** that this is typed correctly.

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

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

Content and Structure

Your audience is the tutors and lecturers, who are proficient at programming and understand most concepts. Therefore you should not, for example, waste words describing the syntax of Haskell or how recursion works. After reading your technical report, the reader should thoroughly understand what problem your program is trying to solve, the reasons behind major design choices in it, as well as how it was tested. Your report should give a broad overview of your program, but focus on the specifics of what *you* did and why.

Remember that the tutors have access to the above assignment specification, and if your report *only* contains details from it then you will only receive minimal marks. Below is an potential outline for the structure of your report and some things you might discuss in it.

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.
- If you have changed the way the controls work, or added something that may make your program behave unexpectedly then it would be worth making a note of it here.

This section is particularly relevant to more complicated programs.

Content
Talk about why you structured the program the way you did. Below are some questions you could answer:

- Program design
    - Describe what each relevant function does conceptually. (i.e. how does it get you closer to solving the problems outlined in this assignment spec?)
    - How do these functions piece together to make the finished program? Why did you design and implement it this way?
    - What major design choices did *you* make regarding the functions that *you've* written and the overall structure of your program?
- Assumptions
    - Describe assumptions you have made about how a user might use the program and how this has influenced your design decisions.
- Testing
    - How did you test individual functions?
        - Be specific about this - the tutors know that you have tested your program, but they want to know *how*.
        - Describe the tests that prove individual functions on their own behave as expected (i.e. testing a function with different inputs and doing a calculation by hand to check that the outputs are correct).
    - How did you test the entire program? What tests did you perform to show that the program behaves as expected in all (even unexpected) cases?
        - Again, be specific - did you check that you can draw shapes from left to right? What about right to left? A circle and a square are each defined by opposite points on their perimeter - did you check that a circle fits perfectly within a square drawn from the same two input points?
        - Have you checked edge cases (this is a computer science term that refers to unexpected or unlikely inputs that may cause a program to crash or behave in strange ways)? It is not likely that someone would try to change the tool halfway through drawing a shape, but it is essential that it has behaviour defined for that scenario. Describe similar tests that you have done to ensure the program can handle all inputs.
- Inspiration / external content
    - What resources did you use when writing your program (e.g., published algorithms)?
    - If you have used resources such as a webpage describing an algorithm, be sure to cite it properly at the end of your report in a 'References' section. References do not count to the maximum word limit.

Reflection

Discuss the reasoning behind your decisions, rather than *what* the decisions were. You can reflect on not only the decisions you made, but the process through which you developed the final program:

- Did you encounter any conceptual or technical issues?
    - If you solved them, describe the relevant details of what happened and how you overcame them.
    - Sometimes limitations on time or technical skills can limit how much of the assignment can be completed. If you ran into a problem that you could not solve, then your report is the perfect place to describe them. Try to include details such as:
        - theories as to what caused the problem;
        - suggestions of things that might have fixed it; and
        - discussion about what you did try, and the results of these attempts.

- What would you have done differently if you were to do it again
    - What changes to the design and structure you would make if you wrote the program again from scratch?
- Are parts of the program confusing for the reader? You can explain them in the report (in this situation you should also make use of comments in your code).
- If you collaborated with others, what was the nature of the collaboration? (Note that you are only allowed to collaborate by sharing ideas, not code.)
    - Collaborating is any discussion or work done together on planning or writing your assignment.
- Other info
    - You may like to briefly discuss details of events which were relevant to your process of design - strange or interesting things that you noticed and fixed along the way.

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 in a technical report

- 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 VSCodium.
- 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, btw, 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.

If you need additional help with report writing, the academic skills writing centre has a peer writing service and writing coaches.

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 monospaced 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).
- When available, automatic *ligatures* should be activated.
- Do not use underscore to highlight your text.
- Text should be at least 1.5 spaced.

Communicating

Do not post your code publicly, either on Piazza or via other forums. Posts on Piazza trigger emails to all students, 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 share code**. There might be pressure from your friends, but this is for both your and their benefit. Anything that smells of plagiarism will be investigated and there may be serious consequences.

Sharing ideas and sketches is perfectly fine, but sharing should stop at ideas.

Course staff will not look at assignment code unless it is posted **privately** in piazza.

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

Before the assignment is due, course staff will not give individual tips on writing functions for the assignment or how your code can be improved. We will help you get unstuck by asking questions and pointing you to relevant lecture and lab material. You will receive feedback on you work when marks are released.

Submission Checklist

Once you have finished your assignment, and preferably 24 hours prior to the deadline, you should make sure that:

- You have fully read and understand the entire assignment specification.
- Your work has been pushed to GitLab.
- Your program compiles and runs, including the `cabal v2-test` test suite.
- Your program works on the lab machines - if the program does not work on the lab machines, it might fail tests used by the instructors.
- You have proof-read and spell-checked your report.
- The report is in PDF format, located at the root of your project on GitLab and named `Report.pdf`. That capital `R` is important - Linux uses a case-sensitive file system. Otherwise, it may not be marked.