COMP3141 Assignment 1

Tortoise Graphics
Version 1.0

Liam O'Connor

Semester 1, 2018

Marking Total of 20 marks (20% of practical component)

Due Date Sunday, 22 April 2018, 23:55.

Late Penalty The maximum available mark is reduced by 10% if the assignment is one day late, by 25% if it is 2 days late and by 50% if it is 3 days late. Assignments that are late 4 days or more will be awarded zero marks. So if your assignment is worth 88% and you submit it one day late you still get 88%, but if you submit it two days late you get 75%, three days late 50%, and four days late zero.

Submission Instructions The assignment can be submitted using the '`give`' system.

To submit from a CSE terminal, type:

```
$ give cs3141 Tortoise TortoiseCombinators.hs
```

Overview

In this assignment, you will implement various extensions to a simple graphics drawing language, the Tortoise Graphics Language, embedded in Haskell. It is intended to give you experience writing Haskell programs, including functional programming idioms, as well as experience programming to algebraic specifications expressed as QuickCheck properties. These specifications will introduce you to concepts such as *monoids*, the distinction between *syntax* and *semantics*, and various notions of *composition*.

Provided Code

The provided code consists of a number of modules:

`Main.hs` contains a `main` function to save an example image to '`tortoise.png`'. It also includes a series of example graphics of increasing complexity. Initially, only the first example will work.

`Tests.hs` QuickCheck specifications for all functions you must implement, as well as any support code to run the tests, depending on your environment.

`TestSupport.hs` contains support code such as `Arbitrary` instances and alternative test data generation strategies for use when you are testing this assignment, and for when we are marking it.

`Tortoise.hs` contains the definitions for the *syntax* and *semantics* of the Tortoise Graphics Language.

`TortoiseGraphics.hs` contains a graphical backend (using the `rasterific` library) for the Tortoise Graphics Language, to actually visualise your graphics.

`TortoiseCombinators.hs` contains stubs for the additional functions you are required to implement.

Note: The **only** file you can submit is '`TortoiseCombinators.hs`', so make sure your submission compiles with the *original* versions of all other files.

The Tortoise Graphics Language

Simon the Tortoise has decided to take up line drawing as his latest hobby. Unfortunately for him, he lacks any artistic inspiration and is unable to dream up even the simplest picture. He is, however, very good at understanding Haskell programs[1]. To help Simon out, we've come up with a little language of drawing commands, and defined it as a Haskell data type (in '`Tortoise.hs`'):

```
data Instructions = Move Distance Instructions
                  | Turn Angle Instructions
                  | SetStyle LineStyle Instructions
                  | SetColour Colour Instructions
                  | PenDown Instructions
                  | PenUp Instructions
                  | Stop

type LineWidth = Int
data LineStyle = Solid LineWidth | Dashed LineWidth | Dotted LineWidth
type Angle = Integer -- Degrees
```

[1]Not unlike many other people named Simon.

```
type Distance = Integer -- Pixels
type Point = (Integer, Integer) -- (x, y)
data Colour = Colour { redC, greenC, blueC, alphaC :: Int }
```

The data type `Instructions` is the *syntax* of our Tortoise Graphics Language. With this language, we can do all the artistic thinking, and encode our pictures as `Instructions` for Simon to follow. When Simon moves, if the pen is *down*, he will also draw a line from his starting point to his ending point.

We define a `Picture` to be a series of `Lines`, drawn in order. A `Line` consists of a line style, a colour, a start point, and an end point[2]:

```
data Line = Line LineStyle Colour Point Point
type Picture = [Line]
```

Thus, Simon's job is simple: Given an initial state (consisting of a starting position, angle, colour etc.), follow the instructions to produce a `Picture` and a final state:

```
tortoise :: Instructions -> (TortoiseState -> (Picture, TortoiseState))
```

This `tortoise` function defines the *semantics*, or meaning, of our Tortoise Graphics Language. It maps *syntax* (`Instructions`) to a domain of *state transformers* that return a `Picture`. In computer science literature, the expression `tortoise i` would usually be written as $[\![\texttt{i}]\!]$, but we shall stick to our Haskell-based notation. We also define two utility functions that give the picture (resp. final state) produced for a given set of instructions if we start from the default `start` state:

```
tortoisePic :: Instructions -> Picture
finalState :: Instructions -> TortoiseState
```

With the basic Tortoise Graphics Language defined, we can now (in '`Main.hs`') produce syntax for a simple square:

```
square :: Distance -> Instructions
square s = Move s $ Turn 90
         $ Move s $ Turn 90
         $ Move s $ Turn 90
         $ Move s $ Turn 90
         $ Stop -- The dollar operator allows us to avoid nested parens.
```

You can use the provided `drawPicture` function (from '`TortoiseGraphics.hs`') to produce an image from a `Picture`, and `writePng` to save it to disk:

```
main = do
    writePng "tortoise.png" (drawPicture (tortoisePic (square 100)))
```

We have defined the basic language, but large drawings are very cumbersome to write directly. To remedy this, you must define a set of so-called *combinators*, functions that act on `Instructions` to let build bigger drawings out of smaller ones. Each of these combinators has been specified in '`Test.hs`'. You must implement them in '`TortoiseCombinators.hs`'.

[2]Note that we have defined equality on `Lines` to treat a line $A - B$ and a line $B - A$ as equal.

Sequential Composition (4 marks)

The first combinator you must implement is a way to take two sets of `Instructions` and combine them into one, one after another:

```
andThen :: Instructions -> Instructions -> Instructions
```

The specification for this function is given in the form of QuickCheck properties in '`Test.hs`'. As it is a composition operator, we expect it to be associative, and have a left and right identity with `Stop`:

$$\begin{aligned}
&\texttt{Stop `andThen` } i && = && i && \text{(andThen_left_id)}\\
&i \texttt{ `andThen` Stop} && = && i && \text{(andThen_right_id)}\\
&i_1 \texttt{ `andThen` } (i_2 \texttt{ `andThen` } i_3) && = && (i_1 \texttt{ `andThen` } i_2) \texttt{ `andThen` } i_3 && \text{(andThen_assoc)}
\end{aligned}$$

Algebraically, the above properties mean that the triple (`Instructions`, `andThen`, `Stop`) form a *monoid*. Monoids are very common in functional programming, and in programming in general. What other monoids can you think of?

Semantically, this combinator corresponds to the notion of *sequential composition* — that is, doing one thing after another. We define sequential composition of state transformers by running the first state transformer with the given state, then running the second with the output state of the first, and concatenating their outputs[3]:

```
comp :: (a -> (Picture, b)) -> (b -> (Picture, c))
     -> (a -> (Picture, c))
comp f g a = let (p , b) = f a
                 (p', c) = g b
             in (p ++ p', c)
```

Then, our correctness property for `andThen` can succinctly state the relationship between *syntactic* composition and *semantic* composition:

$$\texttt{tortoise } (i_1 \texttt{ `andThen` } i_2) \texttt{ start} \quad = \quad \texttt{comp } (\texttt{tortoise } i_1) \; (\texttt{tortoise } i_2) \texttt{ start} \qquad \text{(andThen\_compose)}$$

Marking Criteria	
Marks	Description
1	Left identity property passed
1	Right identity property passed
1	Associativity property passed
1	Semantic composition property passed
4	**Total**

[3] We use a more general type than necessary for `comp`. This helps the type system to aid us to get the implementation correct. In practice, all the type variables `a`, `b` and `c` will be instantiated to `TortoiseState`.

Bounded Looping (4 marks)

Our next combinator is for *bounded looping*, that is, repeating the same set of instructions a fixed number of times:

```
loop :: Int -> Instructions -> Instructions
```

The expression `loop` 0 i should be equivalent to `Stop`, as should `loop` n i for any negative n. Any positive n should produce the composition of n copies of i. For example, `loop` 3 i should be equivalent to i `` `andThen` `` i `` `andThen` `` i.

To define what we expect this combinator to do semantically, we `replicate` n times the state transformer for i, and then use the higher-order function `foldr` to compose them all together:

(loop_compose)

```
tortoise (loop n i) start  =  foldr comp nop (replicate n (tortoise i)) start
```

Here `nop` is the identity state transformer, that does not change the state and returns an empty picture (i.e. `tortoise Stop`). To get a better sense of how this works, it may be instructive to examine the following equational proof of the above property for the case where $n = 3$:

```
    foldr comp nop (replicate 3 (tortoise i)) start
=   foldr comp nop (tortoise i : tortoise i : tortoise i : []) start
=   (tortoise i `comp` tortoise i `comp` tortoise i `comp` nop) start
=   (tortoise (i `andThen` i) `comp` tortoise i `comp` nop) start
=   (tortoise (i `andThen` i `andThen` i) `comp` nop) start
=   tortoise (i `andThen` i `andThen` i `andThen` Stop) start
=   tortoise (i `andThen` i `andThen` i) start
=   tortoise (loop 3 i) start
```

This combinator should allow you to try some interesting pictures! For example, the `circlograph` and `squareograph` examples in '`Main.hs`' are classic examples of Tortoise graphics. We encourage you to try to express yourself creatively and come up with other pretty pictures, and share them with your friends.

Marking Criteria	
Marks	Description
2	Passes for positive n
1	Passes for negative n
1	Passes for zero n
4	**Total**

Invisibility (4 marks)

The next combinator you must implement is `invisibly`:

```
invisibly :: Instructions -> Instructions
```

As the name suggests, this function takes in some `Instructions` and produces a new set of `Instructions` that has the same effect on the tortoise state when starting from the initial state `start`, but does not produce any lines in the picture.

$$\texttt{tortoise}\ (\texttt{invisibly}\ i)\ \texttt{start} \quad = \quad ([], \texttt{finalState}\ i) \qquad (\texttt{invisibly\_sems})$$

The `PenUp` and `PenDown` constructors govern whether or not to draw lines. While the pen is up, no lines are drawn, and while the pen is down, the `Move` constructor will draw a line as well as move the tortoise. The complication when implementing this combinator is that the given `Instructions` may contain these pen-controlling constructors. You may find it helpful to start by only handling the cases with simple `Move` and `Turn` constructors, and gradually adding more constructors while keeping tests passing.

To assist you in testing your program, you may use the provided `newtype`s that restrict test data generation to some subset of constructors, `MoveTurnOnly` and `NoPenControl`. These types have different `Arbitrary` instances and therefore will produce different test data. If you wish to run tests using these subsets, simply change the property definition from, for example:

```
prop_invisibly_sems i = ...
```

to:

```
prop_invisibly_sems (MoveTurnOnly i) = ...
```

Make sure you remove these changes and test with the full set of constructors after you have implemented it, however! You are only allowed to submit '`TurtleCombinators.hs`', so any changes you make to '`Test.hs`' will not be submitted.

Hint: You may find it helpful to define a separate helper function that takes additional arguments, and define `invisibly` in terms of that helper function. It is also useful to note that the pen starts *down* in the initial state `start`. You should only need to make one pass through the given `Instructions`.

Marking Criteria	
Marks	Description
1	Passes with just `Move, Turn`
1	Additionally passes with `SetStyle, SetColour`
2	Additionally passes with `PenUp, PenDown` (i.e. everything)
4	**Total**

Retracing Backwards (4 marks)

The next combinator you are to implement is called `retrace`:

```
retrace :: Instructions -> Instructions
```

If a set of instructions i goes from state `start` to state σ and produces picture p, then the instructions `retrace` i will go from state σ to `start` and produce the picture `reverse` p — that is, the same lines, but in reverse order.

(retrace_sems)

$$\texttt{tortoise}\ (\texttt{retrace}\ i)\ (\texttt{finalState}\ i) \quad = \quad (\texttt{reverse}\ (\texttt{tortoisePic}\ i), \texttt{start})$$

Like the previous combinator, you may find it helpful to define `retrace` with a separate helper function that takes additional arguments. Also like before, you may wish to start by retracing simple `Instructions` such as those containing just `Move` and `Turn` constructors, before moving on to more complicated ones involving style, colour and pen state. You may find it useful to make use of previously-defined combinators such as `andThen` in your implementation, but be warned: `andThen` is $\mathcal{O}(n)$ time complexity. Using it here can easily make your `retrace` function $\mathcal{O}(n^2)$, which is unacceptably slow for many real images. To remedy this, you will need to introduce an *accumulator*. To illustrate this concept, here is a slow, $\mathcal{O}(n^2)$ implementation of `reverse` on lists:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ x
```

And here is a linear-time version using an *accumulator* parameter `a`:

```
reverse :: [a] -> [a]
reverse l = go l []
  where go [] a = a
        go (x:xs) a = go xs (x:a)
```

Note that we avoid calling the $\mathcal{O}(n)$ time `(++)` function on each list element in the accumulator version, and thus reduce the overall complexity from quadratic to linear. One mark is on offer here for writing a linear-time version of `retrace`. You will also find some examples (such as the `flowers circlograph` example) that do not perform in reasonable time given a quadratic-time `retrace`. If you have a working, but slow version of `retrace` already implemented, I would recommend developing the faster version guided by an additional QuickCheck property such as:

```
prop_retrace_same i = slowRetrace i == retrace i
```

Marking Criteria	
Marks	Description
1	Passes with just `Move, Turn`
1	Additionally passes with `SetStyle, SetColour`
1	Additionally passes with `PenUp, PenDown` (i.e. everything)
1	$\mathcal{O}(n)$ complexity; no quadratic blowup (or worse).
4	**Total**

Overlaying Images (4 marks)

The final combinator you must implement is called `overlay`. It takes in a list of `Instructions` and produces a single set of `Instructions`:

```
overlay :: [Instructions] -> Instructions
```

If instructions i_1, i_2 and i_3 produce images p_1, p_2 and p_3 respectively, then the expression `overlay` $[i_1, i_2, i_3]$ returns instructions that produce the combined picture where p_1 is drawn, then p_2, then p_3, such that p_3 appears "on top" of p_2, which is in turn "on top" of p_1. The `overlay` function should ensure that the tortoise returns to the initial state `start` after drawing any images. If the provided list is empty, the resultant instructions should draw nothing.

$$\begin{aligned} \texttt{finalState}\ (\texttt{overlay}\ is) &= \texttt{start} && \text{(overlay\_state)} \\ \texttt{tortoisePic}\ (\texttt{overlay}\ is) &= \texttt{concatMap}\ \texttt{tortoisePic}\ is && \text{(overlay\_pic)} \end{aligned}$$

Here we use the higher-order function `concatMap` to express what we mean about overlaid pictures. Here is a proof sketch for the second property, for our three `Instructions` i_1, i_2 and i_3:

$$\begin{aligned} & \texttt{concatMap}\ \texttt{tortoisePic}\ [i_1, i_2, i_3] \\ = \ & \texttt{concat}\ (\texttt{map}\ \texttt{tortoisePic}\ [i_1, i_2, i_3]) \\ = \ & \texttt{concat}\ [\texttt{tortoisePic}\ i_1, \texttt{tortoisePic}\ i_2, \texttt{tortoisePic}\ i_3]) \\ = \ & \texttt{tortoisePic}\ i_1\ \texttt{++}\ \texttt{tortoisePic}\ i_2\ \texttt{++}\ \texttt{tortoisePic}\ i_3 \\ = \ & \texttt{tortoisePic}\ (\texttt{overlay}\ [i_1, i_2, i_3]) \end{aligned}$$

Tip: You can use the combinators you have previously defined such as `invisibly` and `retrace` to implement `overlay` very succinctly.

Once you have implemented `overlay`, you can now generate all of the example images in '`Main.hs`', including the complex `circlographograph` example.

Marking Criteria	
Marks	Description
1	Empty list works
1	State is preserved
2	Images are concatenated
4	**Total**

Compiling and Building

This project has a number of dependencies, specifically the `rasterific` graphics library, the `JuicyPixels` image library, the `QuickCheck` testing library and the test framework called `tasty`. For CSE machines, we have already a configured a package database on the course account that should build the assignment without difficulty using the standard Haskell build tool `cabal`. For students using their own computer, we instead recommend the alternative build tool `stack`, available from the Haskell Stack website at `https://www.haskellstack.org`. We have provided a `stack` configuration file that fixes the versions of each of our dependencies to known-working ones. If you use `stack` to set up your toolchain, you should have minimal compatibility difficulties regardless of the platform you are using. *If you are using versions of GHC or Haskell build tools supplied by your Linux distribution, these are commonly out of date or incorrectly configured. We cannot provide support for these distributions.*

If you are using a Mac computer, you may be interested in using the Haskell for Mac IDE. The Lecturer in Charge has access to special (free) student licenses for COMP3141 students, so contact the lecturer for more information.

Detailed, assignment-specific instructions for each build tool are presented below.

On CSE Machines

Enter a COMP3141 subshell by typing `3141` into a CSE terminal:

```
$ 3141
newclass starting new subshell for class COMP3141...
```

From there, if you navigate to the directory containing the assignment code, you can build the assignment by typing:

```
$ cabal build
```

To run the program from '`Main.hs`', which saves an image `tortoise.png`, type:

```
$ ./dist/build/Tortoise/Tortoise
```

To run the program from '`Tests.hs`', which contains all the QuickCheck properties, type:

```
$ ./dist/build/TortoiseTests/TortoiseTests
```

To start a `ghci` session for the `Tortoise` program, type:

```
$ cabal repl Tortoise
```

Similarly, `cabal repl` can be used with `TortoiseTests`. Lastly, if for whatever reason you want to remove all build artefacts, type:

```
$ cabal clean
```

For `stack` users

Firstly, ensure that GHC has been setup for this project by typing, in the directory that contains the assignment code:

```
$ stack setup
```

If `stack` reports that it has already set up GHC, you should be able to build the assignment with:

```
$ stack build
```

This `build` command will, on first run, download and build the library dependencies as well, so be sure to have an internet connection active. To run the program from '`Main.hs`', which saves an image `tortoise.png`, type:

```
$ stack exec Tortoise
```

To run the program from '`Tests.hs`', which contains all the QuickCheck properties, type:

```
$ stack exec TortoiseTests
```

To start a `ghci` session for the `Tortoise` program, type:

```
$ stack repl Tortoise:exe:Tortoise
```

Similarly, `stack repl` can be used with `TortoiseTests`:

```
$ stack repl Tortoise:exe:TortoiseTests
```

Lastly, if for whatever reason you want to remove all build artefacts, type:

```
$ stack clean
```

For Haskell for Mac users

The provided Haskell for Mac code bundle should include everything you need. The tests are run via the playground of `Tests.hs`. Furthermore, the playground of `Main.hs` includes an example of visualising images directly within Haskell for Mac.

Marking and Testing

All marks for this assignment are awarded based on *automatic marking scripts*, which are comprised of several QuickCheck properties (based on, but not exactly the same as, the QuickCheck properties given in this assignment spec). Marks are not awarded subjectively, and are allocated according to the criteria presented in each section.

Barring exceptional circumstances, the marks awarded by the automatic marking script are *final*. For this reason, *please* make sure that your submission compiles and runs correctly on CSE machines. We will use similar machines to mark your assignment.

A dry-run script that runs the tests provided in the assignment code will be provided. When you submit the assignment, please make sure the script does not report any problems.

Late Submissions

Unless otherwise stated if you wish to submit an assignment late, you may do so, but a late penalty reducing the maximum available mark applies to every late assignment. The maximum available mark is reduced by 10% if the assignment is one day late, by 25% if it is 2 days late and by 50% if it is 3 days late. Assignments that are late 4 days or more will be awarded zero marks. So if your assignment is worth 88% and you submit it one day late you still get 88%, but if you submit it two days late you get 75%, three days late 50%, and four days late zero.

Extensions

Assignment extensions are only awarded for serious and unforeseeable events. Having the flu for a few days, deleting your assignment by mistake, going on holiday, work commitments, etc do not qualify. Therefore aim to complete your assignments well before the due date in case of last minute illness, and make regular backups of your work.

Plagiarism

Many students do not appear to understand what is regarded as plagiarism. This is no defense. Before submitting any work you should read and understand the UNSW plagiarism policy `https://student.unsw.edu.au/plagiarism`.

All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. In this course submission of any work derived from another person, or solely or jointly written by and or with someone else, without clear and explicit acknowledgement, will be severely punished and may result in automatic failure for the course and a mark of zero for the course. Note this includes including unreferenced work from books, the internet, etc.

Do not provide or show your assessable work to any other person. Allowing another student to copy from you will, at the very least, result in zero for that assessment. If you knowingly provide or show your assessment work to another person for any reason, and work derived from it is subsequently submitted you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep your work private until submissions have closed.

If you are unsure about whether certain activities would constitute plagiarism ask us before engaging in them!