Technical Report for Assignment 2 - COMP1100

Aryan Odugoudar

Student Information

- **Name:** Aryan Odugoudar
- **Laboratory Time:** Friday 8:00am - 10:00am
- **Tutors:** Jess Allen and Madeleine Stewart
- **University ID:** u7689173

Contents

1 Introduction

This technical report provides an overview of the design, implementation, and testing of the Turtle Graphics program for COMP1100 Assignment 2. The program is used to translate a list of Haskell Commands (of type `TurtleCommand`) to Pictures using CodeWorld API

2 Analysis of the Program

The objective of the program is to generate a comprehensive solution by organizing the program into three primary files: `Main.hs`, `Turtle.hs`, and `TestPattern.hs`. Within `Turtle.hs`, there are functions responsible for generating command lists for different drawings, as well as the `runTurtle` function, which interprets these commands into a CodeWorld Picture. `TestPattern.hs` is specifically designed to create a textual graphical pattern used for testing the `runTurtle` function. `Main.hs` integrates the functions from `Turtle.hs` with user input to form the entire program.

`TurtleTest.hs` and `Testing.hs`, while included in the program files, do not contribute any functional components to the program. Instead, they are utilized for running unit tests, ensuring that the functions in Turtle.hs operate as intended.

2.1 Implementation Details

2.1.1 Turtle.hs

```
triangle ::  Double -> [TurtleCommand]
```

Generates a list of TurtleCommand that will draw an equilateral triangle with perimeter equal to the sidelength given.

`polygon :: Int -> Double -> [TurtleCommand]`

Generates a list of `TurtleCommand` that will draw a regular n-sided polygon, with perimeter equal to sidelength given.

`runTurtle :: [TurtleCommand] -> Picture`

Interprets a list of `TurtleCommand` into a CodeWorld Picture. Calls the helper functions `stateToPic, comToState and initialState`.

`initialState :: TurtleState`

This function is used to maintain the initial Position,Facing, TurtleCommand, Pen-Status and StepSize.

`comToState :: [TurtleCommand]-> TurtleState-> [TurtleState]`

A helper function for `runTurtle` that takes in a list of `TurtleCommand and TurtleState` as input and uses extra arguments in order to keep track of the cursor's position, orientation, speed, and pen state further generating a list of `TurtleState` to be used.

`stateToPic :: [TurtleState] -> Picture`

A helper function for `runTurtle` that takes in a list of `TurtleState` generated by `comToState` as input and generates picture according to the pen's situation

`data TurtleState`

A data type which tracks the turtle's pen, position, TurtleCommand, facing and step size.

`data PenStatus`

A data type which tracks the status of the Pen.

`tSquare :: Int -> Double -> [TurtleCommand]`

Funtion generates the necessary commands to draw an approximation to the boundary of the T-Square based on the given depth and sidelength. Used `corner` as a helper function.

`corner :: Int -> Double -> [TurtleCommand]`

It is a helper function for `tSquare` that is used to draw the corners using recursion.

3 Rationale and Reflection

In developing the Turtle Graphics program, several key design decisions were made to ensure the functionality and efficiency of the system. Below are the reasoning, assumptions, and reflections on the choices made during the implementation process:

3.1 Design Choices and Reasoning

1. **Use of Helper Functions:**

   I chose to implement different helper functions like `initialState, comToState and stateToPic` for `runTurtle` and `corner` for `tSquare` so that it enhances readability and makes it easier to extend or modify specific functionalities without affecting the entire system.

2. **Recursive Functions for Shapes:**

   Functions like `triangle` and `polygon` use recursion to generate shapes.

This recursive approach allows for drawing shapes of any number of sides without having to rewrite the drawing logic for each specific case. By recursively defining the shape drawing logic, the program achieves flexibility and scalability.

The `tSquare` function generates T-Square patterns using recursion. By breaking down the T-Square into smaller squares and connecting them, the function creates intricate patterns. This recursive approach aligns with the recursive nature of fractal patterns, allowing for the generation of complex and visually appealing designs.

3.2 Assumptions and Reflections:

1. **Recursive Depth in T-Square:**

   The T-Square generation assumes a positive depth for recursion. Higher depth values result in more intricate patterns. However, very high depths might lead to significant computational overhead. The choice of depth allows users to balance complexity and performance in their T-Square designs.

2. **Code Readability vs. Conciseness:**

   While the code uses recursion for flexibility, the readability of deeply nested recursive structures could be a concern. The decision to balance conciseness and readability was made, aiming to make the code expressive without sacrificing its understandability.

4 Testing

The functions in the program that were tested included the ones in `Turtle.hs`. The command list generating functions: `polygon, triangle and tSquare`

were tested using unit tests which were conducted using the testing files `TurtleTest.hs` and `Testing.hs`. However, the function `runTurtle` was tested visually using the testing file `TestPatterns.hs` and by observing the generated drawing on the CodeWorld canvas.

The unit tests conducted on `polygon, triangle and tSquare` functions involved varying argument values to ensure the accurate generation of command lists for all possible valid inputs. For example on `polygon`, tests were performed with different side lengths, verifying that the generated command list was correct for cases where the number of sides was greater than, equal to, or less than the perimeter. This comprehensive testing approach also included boundary values.

The functionality of the `runTurtle` function was visually tested using the textual graphic command list generated by `TestPattern.hs`. These tests confirmed the correctness of the function as the text graphic command list was accurately displayed as a CodeWorld Picture on the canvas. Additionally, further examinations were carried out using command lists from `polygon` (with varying numbers of sides) and `triangle` by employing predefined test keyboard inputs defined in `Main.hs`. These tests ensured the robustness and reliability of the implemented functions under different scenarios.

5 Conclusion

In conclusion, this technical report provides an overview of my implementation for Assignment 2 in COMP1100. It discusses the program's features, implementation details, design choices, assumptions, testing procedures, and style considerations.It successfully meets the requirements of the assignment as expected.