Hurtle Documentation

Alex Do - u5505454 - CS141 Coursework 2

To begin with, please take the time to demo the project to experience its full functionality. Inside of the test folder, you'll find a file called "spiraloctagons.hogo". I've documented and used all my key features here, so take a glance through and change numbers as you wish. I've also provided lines to uncomment to see different images. To run, go to toplevel, run "stack run", and the cmdline will prompt you to enter a filename: "test/spiraloctagons.hogo". You can slow it down by going into the "src/hatch/Hatch.hs" file and changing the "fps" variable.

Quick note for tests, I changed the file to accommodate my use of states (as previously it was designed for a "Parsec Void String", but I don't use that anymore (all to be explained!)).

Functionality

I believe the demo does an excellent job of speaking for itself; but to overview the main additions on top of the specification (and rules for the new language, which is my twist on Logo Turtle and the specification): commas are treated as newlines, each HogoCode element must be newlined in between (commas acceptable too), you can declare and use procedures (with parameters), procedures can't access the outer scope, you can have for loops that do access the outer scope, and repeats that do the same, you can declare variables, and have variables that are the sum, product, difference, or quotient of other variables and values, but can't use variables that don't yet exist, you can set pen width and colour, and finally, wherever you expect a value input, you can replace with a variable. I've also created custom errors and show instances to deal with the complexity of my project, but if I were to improve anything, it would be to tie together tokenization and syntax analysis so that they run concurrently. **For more detail, see the README.md in the project files.**

Program Architecture

The project has gone through significant changes throughout the last few weeks, but I believe its current form is nearly as refined as possible. In summary, a file is read to lowercase, passed into a tokenizer to form a list of tokens, which is passed into a second (Parsec _ [TOKENS a) for syntax analysis, followed by an interpreter to read the code.

In more detail, the tokenizer works by reading in keywords and characters from the file string (lowercased) and matching them to a type definition TOKENS. These span keywords like "to", "forward", and characters like ":", "[", ",", and so on, and finally can parse names. I defined a standard "Parsec Void String" for this and used default errors, but also implemented a specific show function for a list of TOKENS, formatted with arrows, padding, and an "eof" marker essentially. I also implemented an equality check for TOKENS that wasn't automatic because two of my tokens need a specifically different from standard check (check Types.hs for more detail). I defined a specific State for this to make coding the parser easier, but in hindsight it wasn't completely necessary for this simpler component. The "Tokenizer.hs" file's main details are that keywords are forced to end in a space or newline, and that undefined keywords are names and floats are values, where these 2 are checked at the very end to avoid early incorrect parsing.

Next, looking at syntax analysis, this went through a significantly detailed design process (not discussing now), but ended up with me firstly realising what a HogoCode actually was, being an operation that changed the state of the program, and secondly that a variable change counted as a state change (and a procedure declaration was not, but calling it would be). Indeed, the best way to keep changing variables, whilst updating procedures and the code itself, was to use a record to store a (strict - check Types.hs 157-164) map for variables and tables, and a list of HogoCodes. This also meant I could easily move out code and isolate portions by storing a previous state and inserting a blank one in the case of something like a procedure definition. At this point, I also created a recursive definition for Variables, as it made sense to use recursion to find a value if a chain of variables being references to other variables was created. Note I designed it such that you could never use an undefined variable, meaning at runtime all variables are guaranteed to be traceable to a float value. Also, (CodeGeneration.hs 213-281) I have procedure definitions that work by inserting placeholder values into a sub-HogoProgram and stored mapped to input parameter names in order to use parameters whilst inside of the procedure, as it reads from essentially a blank slate. This is complicated and better seen by looking at the code. At the very end, I have specific end cases for loops, procedures, and the whole file, and so must be separate.

Finally, the interpreter has the easy job as no rules have to be checked. (ShowHurtle.hs) I firstly don't use Gloss or Hatch lines, instead defining my own type so it can vary in width (easy maths explained in the file), and store the current state of the program in a type called HogoRun, which keeps record of the current image, position, pen state, and HogoProgram state. These are updated on the go - every time a code line is interpreted, the HogoCode list in HogoProgram is replaced with its tail, and the state of the run is updated appropriately (trivial, check code). Variables are traced back to float values via my getValue function recursively, which also handles arithmetic, and movement commands are interpreted differently depending on penState (to draw or just move).

Libraries

As I've already spoken about, I used the MTL in order to mess around with states, in particular StateT, which allows the same functionality of State, with the additional capabilities of wrapping yet another monad around the original. In this case, I've wrapped my parsers around my program states generally and wrapped all that in a state. I used containers for maps, which was necessary for fast access of variables and checking existence of a variable name, and for graphics I used Gloss, with minimal Hatch, since I wanted to have more control over shapes. I considered Brick, but it looked too hard to implement diagonal precisely angled lines with, so I didn't. Finally, for ease of debugging and aesthetic reasons, I used ANSI to colour text in the terminal (when I was procrastinating). Further standard libraries include Megaparsec for parsing, and bytestring for file case conversion.

Design Process

My initial design was quite similar to the final version, but did not separate tokenization from syntax analysis. Doing this meant I missed a lot of edge cases for input and made it incredibly tedious to write, as I had no clue where my parser was failing and what alternative (<|>) it took. By separating parsing and tokenization, it was incredibly easy to spot and correct errors, also avoiding repetition, and having to check fewer conditions during syntax analysis, for example multiple newlines in a row are parsed to the same newline token, and now I don't need to check for multiple newlines in my syntax analysis.

I also drew diagrams to link together variables, functions, and figure out how to design my Variable, HogoCode, and HogoProgram data type, as well as a good way to record procedures and variables, if they took a float or a reference (arithmetic combinations) of another variables. It turns out my original structure for HogoCode was fine and worked as intended, which was nice.