Functional Languages

Project - Turtle Graphics I

05.11.2020

1 Get this turtle started...

- Clone the repository in github: `https://github.com/erdferkel-teaching/turtleWS20`
- There are stub projects for Haskell and F#. This document refers to the F# implementation. However, you can also do the Haskell version.
- This project is prepared for Visual Studio 2019 and should work with vscode. Please understand that we cannot provide ready to go solutions for all IDEs and tooling around! Ask during the lecture if you find any problems. The Haskell version is prepared via stack. The tooling should not be a problem, so please ask as early as possible in order not to get stuck in tooling problems.
- To build, follow the instructions in the readme files.
- For questions please consult the moodle forum.
- In the turtle project you can achieve 20 points, which is basis for the project part of grading.

The project structure is:

- `Program.fs` contains window and graphics setup code. Line 22 calls `PartOne.runTurtleProgram...` This function takes a turtle program and returns a list of points. This is the entry point of your implementation.
- `PartOne.fs` contains stubs and empty functions which should be implemented. They are marked with 'TODO'.

2 Modelling the domain (part of the codebase already)

Turtles may move forward by some distance, turn left and right. The following union type models this fact:

```
type Value = float // type alias for values in the turtle language
type Cmd =
    | Forward of Value // Move forward by Value
    | Left    of Value // Turn left by Value degrees
    | Right   of Value // Turn right by Value degrees
```

A turtle program constitutes of a list of such commands. Additionally, let us define type aliases for our domain.

```
type Program = list<Cmd>

type Vec2 = float * float
type Angle = float
```

In order to model the turtles state, we create a record as such:

```
type TurtleState =
{
    direction : Angle     // direction in degrees
    position : Vec2       // current position
    trail    : list<Vec2> // points produced so far
}
```

3 Implementing an interpreter (8 points)

The interpretation of a turtle command works as such: given a command and a current state, produce a new resulting state with the commands exectued. This can be formalized by a function of type: `TurtleState -> Cmd -> TurtleState`. Your task is to implement this interpreter for a list of commands, i.e. a turtle program:

```
let interpretTurtleProgram (s : TurtleState) (commands : Program) : TurtleState = your task...
```

Hint: You could start with a function which interprets a single command. Then extend the implementation to a list of commands using fold.

Make sure to test your implementation properly (there are already simple test functions as well as a more complex spiral program in the Examples module).

4 Parsing input commands from strings (8 points)

In the lecture we talked about functional parsers. In this project we would like to use an existing parser combinator library in order to accomplish a task. In this case we use *FParsec* [1] (in Haskell this would be parsec [2].

In the lecture we already looked at FParsec code. In essence, we need to write 5 one liners in order to parse our input language of the form:

⟨*cmd*⟩ ::= '`Forward(`' ⟨*float*⟩ '`)`'
| '`Left(`' ⟨*float*⟩ '`)`'
| '`Right(`' ⟨*float*⟩ '`)`'

⟨*cmd-list*⟩ ::= ⟨*cmd*⟩ '`;`' ⟨*cmd-list*⟩ | ⟨*cmd*⟩ '`;`'

The skeletons for the parsers are given in the repository. The concrete commands are straightforward: Parse the keyword, a parenthesis, a floating point number (`pfloat`) and a closing parenthesis. Next, `pCmd` is a choice between the previously defined primitive parsers. The tricky part comes with `pProgram`, which is a list of unknown length. The easiest way of parsing everything is by using the `manyTill` combinator, whereby we want to parse commands followed by semicolon until we end up applying parsecs `eof` parser (end of file, as discussed in lecture).

5 Files to submit

Submit your project till 19.12 as a zip file (without .git, bin or obj folders).

Happy coding.

References

[1] Fparsec documentation. `http://www.quanttec.com/fparsec/`. (Accessed on 11/19/2017).

[2] parsec: Monadic parser combinators. `https://hackage.haskell.org/package/parsec`. (Accessed on 11/19/2017).