

TurtleGraphics

several authors, adapted by Andres Löh

Difficulty: relatively easy

The topic of this task is to write a simple interpreter for a simple version of the Logo programming language. More information about Logo can for instance be found in the Wikipedia article at

[http://en.wikipedia.org/wiki/Logo_\(programming_language\)](http://en.wikipedia.org/wiki/Logo_(programming_language))

The focus of this assignment is the use of IO in Haskell. Parts of the assignment deal with reading a file and drawing in a window.

Logo

The Logo language is a very simple language to control a turtle (that happens to carry a pen around). The turtle begins in one particular position (say, in the middle of a window), facing in one particular direction.

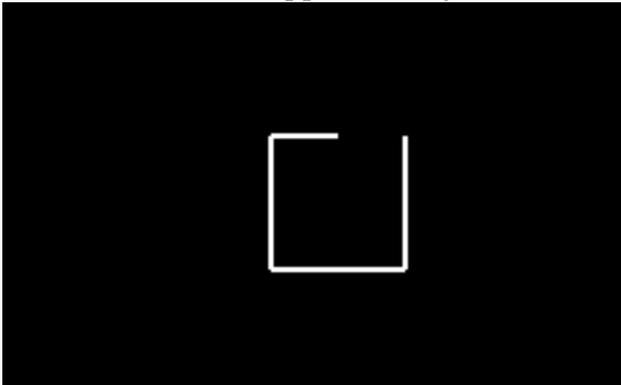
You can now give commands to the turtle, of the following form:

| Command | Effect |
|----------------------------------|---|
| <code>forward <num></code> | The turtle steps forward by <code>num</code> steps. If the turtle is currently drawing, then a line is drawn along the turtle's path. |
| <code>back <num></code> | The turtle steps back by <code>num</code> steps. If the turtle is currently drawing, then a line is drawn along the turtle's path. |
| <code>right <angle></code> | The turtle turns right (i.e., clockwise) by <code>angle</code> degrees. |
| <code>left <angle></code> | The turtle turns left (i.e., counterclockwise) by <code>angle</code> degrees. |
| <code>penup</code> | Stop drawing. |
| <code>pendown</code> | Start drawing. |

Here is an example:

```
forward 50
right 90
forward 50
right 90
penup
forward 25
pendown
forward 25
right 90
forward 50
```

The result should look approximately as follows:



The task

The program should read a Logo program from disk, open a window and paint the results of the program in the window. Finally, the program should wait for a keypress before the window is closed again.

For drawing the resulting picture, we are using the SOE graphics library, an extremely simple (and not very powerful) graphics interface originating from the Haskell book called "The Haskell School of Expression" by Paul Hudak – hence the name SOE. There are several implementations of that library, so you have a choice between (at least) two versions. One implemented in the HGL package on top of the X11 libraries, another one implemented in the gtk2hs package on top of Gtk.

Depending on the library you choose, you have to import a module. If you want to use the HGL variant, simply place the line

```
import Graphics.SOE
```

in your module header. If you choose the gtk2hs variant, use

```
import Graphics.SOE.Gtk
```

instead. *Warning: Do not import both libraries at the same time. That will lead to name clashes, as both libraries export the same interface. Choose one.* The functions you need will

be mentioned in the exercises. Note nevertheless that you can use the GHCi commands `:browse` to browse all definitions in a module, and `:info` and `:type` to get information and types of identifiers.

Steps

1. Write a datatype `Command` that represents a single Logo command. I.e., define one constructor per command.
2. Write a parser that turns a string into a list of commands:

```
parseLogo :: String -> [Command]
```

It is possible to do this using parser combinators, but the Logo language is so simple that this is not really needed. You can use `lines` to split text into lines, `words` to split a line into multiple words at blanks, and then read where appropriate to convert strings into numbers.

The function may just fail if the string contains an illegal Logo program.

3. Now, write a function that takes a filename, reads the file and then parses it:

```
getLogo :: FilePath -> IO [Command]
```

Note that the following type synonym is predefined in the prelude:

```
type FilePath = String
```

4. Define a type that represents the state of the turtle. Depending on your preference, this can be a type synonym, thus

```
type TurtleState = ...
```

or a datatype

```
data TurtleState = ...
```

The state has three components: the current position, the current angle where the turtle is facing, and whether the turtle is currently drawing or not.

5. Define the initial state of the turtle. Let's code the window size as a constant for the moment, for instance

```
windowSize :: Size  
windowSize = (600, 600)
```

The type synonym `Size` is defined by the `SOE` library as follows:

```
type Size = (Int, Int)
```

The turtle should initially be in the middle of the window, face up and be drawing.

```
initialState :: TurtleState
```

6. Write a function that executes a single command and transforms the state. There are different ways of sophistication in which this can be achieved. The simplest approach is probably to directly draw in the window using the

```
drawInWindow :: Window -> Graphic -> IO ()
```

function. You then define

```
processCommand :: Window -> Command -> TurtleState -> IO TurtleState
```

that receives the window, the command, and the original state as arguments.

An alternative that is somewhat nicer is not to draw immediately, but to accumulate a `Graphic` that can be drawn later. This has the advantage that `processCommand` does not use `IO` and does not need the current window as a parameter.

```
processCommand :: Command -> TurtleState -> (Graphic, TurtleState)
```

Now, if you are already familiar with the state monad, this is just an instance of it. So yet another option is to use the type:

```
processCommand :: Command -> State TurtleState Graphic
```

Choose whatever you like best or find easiest. Use the following functions from the SOE library

```
line      :: Point -> Point -> Graphic
withColor :: Color -> Graphic -> Graphic
```

Note that you will have to calculate the new position from the old position using `sin` and `cos`, and that you will have to convert angles, because the `left` and `right` commands are parameterized by degrees (between 0 and 360) whereas `sin` and `cos` expect radians (between 0 and 2π). The constant π is available as `pi` in Haskell.

7. Try to define a function that processes multiple commands. The type of this function depends on the type that you have chosen for `processCommand`, but it will be something like

```
processCommands :: ... [Command] ... -> ...
```

i.e., there should be a list of commands among the inputs.

If you do not draw directly into the window, but assemble values of type `Graphic`, you may find the function

```
overGraphics :: [Graphic] -> Graphic
```

useful that simply combines different graphics (in this case, lines) into a single graphic.

8. Write a function that takes a list of commands, opens a window, processes the commands given the initial state, waits for a keypress, then closes the window. Here is a template for the function:

```
runLogo :: [Command] -> IO ()
runLogo cmds = runGraphics $
  do
    w <- openWindow "Logo" (600, 600)
    ... processCommands ... cmds ... initialState ...
    ...
    getKey w
    closeWindow w
```

How to invoke `processCommands` depends on the type. If `processCommands` does not draw directly, you have to extract the resulting `Graphic` and draw it using `drawInWindow`.

9. Now, combine everything with the parser. Write

```
runFile :: FilePath -> IO ()
```

that reads the file using `getLogo` and subsequently invokes `runLogo`.

10. We are done. A final option is to create a proper main function that uses the command line argument as file name.

```
main = do
  args <- getArgs
  runFile (head args)
```

This will fail if there are no command line arguments. Note that `getArgs` is not defined in the prelude, but requires you to

```
import System.Environment
```

at the top of your module.

Bonus exercises

There are lots of extensions possible for our little language. However, keep in mind that the possibilities of the SOE library are quite limited, and many more advanced graphical features may require switching to a different graphics library.

11 (bonus). Add a possibility to draw in different colors.

12 (bonus, medium). Create a possibility to save the current state and return to it later, in a nested, stack-like way, using commands `push` and `pop`.

13 (bonus, medium). Add loop constructs such that for example

```
repeat 4
  forward 50
  right 90
end
```

draws a square. The main difficulty here is that parsing becomes less straight-forward. Note that you can also only implement everything but the parsing, starting from extending the `Command` datatype. Logo programs are rather simple to write in Haskell.

14 (bonus, medium). Taking the hint from the previous exercise, devise a nice embedded domain-specific language in Haskell in order to create values of type `[Command]`.

15 (bonus, difficult). Extend the Logo language with variables.

Any other ideas are also fine. Use your imagination.