Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
15.01.2020
Deadline: 25.1.2020, 23:59

Functional Programming and Verification

Sheet 10

Tutorial Exercises

Exercise T10.1 Repeated IO

As a warmup, write a function `ioReadPrintInt :: IO ()` that uses `readMaybe` to read an `Int`. Then, print the given number or print an error message if `readMaybe` failed.

Write a function

```
ioLoop :: (a -> Maybe b) -> IO a -> IO b
```

The invocation `ioLoop` f a should repeat the IO action a until f accepts its value (i.e. does not return `Nothing`). It should then return the result of f.

Use `ioLoop` and `readMaybe` from `Text.Read`, to write an IO action

```
getInt :: IO Int
```

that reads lines from stdin until it gets a line that represents a valid integer. It should then return this integer. You can read a line from stdin using the function `getLine` from `System.IO`.

Exercise T10.2 Guessing Game

In this exercise you will implement the following game: The program selects a random number in the range `[0..100]` and asks the user to guess the number. If the user's guess is incorrect, the program tells them whether the actual number is larger or smaller and waits for another guess. If the user guesses correctly, the program prints the number of guesses made.

Hints:

- Use `getInt` from the previous exercise to read numbers from the user.
- Use `randomRIO` from `System.Random` to get a random number in a certain range.

Exercise T10.3 Action, Action, Action!

In this exercise, you are tasked to execute multiple IO actions in sequence.

- Implement a function `ioSequence :: [IO a] -> IO [a]` that for a list of io actions `ios`, executes those actions sequentially and collects their results in a list.
- Use the previous function to define a function `ioSumInts :: IO Int` that first reads a number `n` and then reads in `n` additional numbers. Print those numbers and return their sum.

  Remark: a more general version of the above function already exists in `base` and is called `sequence`.

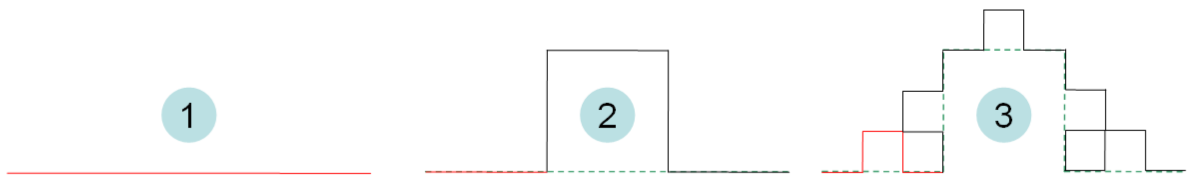

Figure 1: Three iterations of the quadratic Koch curve (type 1)

Homework

You need to pass all tests to collect a coin.

Exercise H10.1 Draw me like one of your Functional Programs

> **Note:** On this exercise sheet, there is a task which requires you to use `IO`. The test system uses a mocked version of `IO` which defines the most important `IO` functions from `Prelude` and `System.IO`. Usually, there is no difference in the behaviour of your program between your local Haskell installation and the submission system. The usability of `IO` functions from other modules cannot be guaranteed however. As always, the submission is only graded if it compiles on the test server. Any problems should be brought to our attention immediately.

This exercise shall be devoted to the beauty of recursion, expressed by us, the artists, in the form of formal grammars and cute, drawing turtles.

A Lindenmayer system (L-system) consists of an alphabet, a start pattern, and rewrite rules. A L-system can recursively be executed. In each step, it takes a pattern and applies all applying rewrite rules to it. It then takes the resulting pattern and recurses.[1]

The L-system's alphabet can be interpreted as commands for a turtle graphics's turtle and hence be visualised. If you haven't heard of turtle graphics before, checkout the brief overview on Wikipedia. Basically, it is a cute turtle holding a pencil and drawing graphics as you command. How amazing is that? btw. turtles are so much better and cuter than penguins!

In the template function `apply`, we define the following characters as turtle commands:

- `F`: move 10 steps forward
- `G`: move 10 steps forward
- `+`: turn 30° to the right
- `-`: turn 30° to the left
- `*`: turn 15° to the right
- `~`: turn 15° to the left

[1] L-systems are a form of formal grammars, in case you're familiar with them. If not, look forward to your lecture on theoretical computer science.

Any other character will not affect the turtle's position. However, you are free to extend this table in any way you like (but do not change the already given rules)! The turtle indeed can do more than just spinning and moving. It can even change its pen colour and branch off in different directions (see Wettbewerb notes below).

Example: The following L-system can be used to model the Quadratic Koch curve (type 1), where `u` turns 90° to the left and `d` turns 90° to the left (see Figure 1):

```
alphabet: F, u, d
start pattern: uF
rewrite rule: F -> FuFdFdFuF
first three iterations:
1. uF
2. uFuFdFdFuF
3. uFuFdFdFuFuFuFdFdFuFdFuFdFdFuFdFuFdFdFuFuFuFdFdFuF
```

To help with your testing, and to give some inspiration for your own L-systems, we also provided two with L-systems in the template.

We model L-systems using the data types `LSystem` and `Rule`, which are defined as follows:

```
data Rule = Char :->: String
data LSystem = LSystem {
                start :: String,
                rules :: [Rule]
              }
```

We assume that the symbols used in our L-system are of type `Char`. A `Rule` defines what non-empty sequence of symbols (i.e. which `String`) a specific symbol is expanded to. LSystems consist of the start pattern `start` and a list of Rules.

Note: Our L-systems are defined using record syntax. You can access any of the fields of an L-system by using the field name as a function, e.g. `start l :: String` where `l :: LSystem`.

a) Write a function `findRule :: [Rule] -> Char -> Rule` that retrieves the rule with the given character on the left-hand side. If there are multiple matching rules, it should return the first one. If there is no matching rule, the symbol should not be modified and hence you should return a rule that acts as the identity function.

b) Write a function `expandLSystem :: LSystem -> Integer -> String` that expands (recurses on) a given L-system for a given number of times and returns the resulting sequence of symbols.

At this point, you can test and marvel at your implementation by calling the `display` function in the Display module, passing it your L-system and the number of recursion steps. Using the Up and Down keys on your keyboard, you can increase respectively decrease the number of recursion steps. The result of the turtle's laborious drawing should then be revealed to you. You can also use the `savePngs` function in the Display module to save images of the recursive drawing steps. There are also two example L-systems that you can check out.

You need to run a suitable GLUT version (for example, freeglut) to be able to execute the `display` function.

On MacOS Big Sur, you might also need to use a specific display server. The following setup might help you in case you are having problems:

1. brew install freeglut
2. brew install xquartz
3. restart your Mac or, alternatively, open xquartz and execute the program from there

If you are still having troubles, let us know on Zulip and share your fixes there.

Now manually entering L-systems by hand into GHCi or hard-coding them is not a solution if you want to boast in front of your relatives. They want some user-interface! Well, to their great luck, we just learnt about IO and will use this power.

Write a function `update :: LSystem -> IO LSystem` that receives an L-system, that checks for new innput with `hReady stdin` and, if there is any, reads a line and interprets it. This should be repeated as long as there is input to read. A command should be interpreted as follows:

1. `start <string>` updates the start pattern of the L-system to the given string.
2. `rule <symbol> -> <string>` inserts a rule at the beginning and, if applicable, deletes the old rule corresponding to that symbol. If there is an error parsing the rule, print `Error parsing rule` and a newline instead.
3. `clear` returns an empty L-system.
4. `print` prints the passed L-system and a newline and returns the passed L-system again.
5. In all other cases, print `Error parsing command` and a newline and return the passed L-system.

After implementing `update`, you can run `stack run display` and enter commands as defined above into the console. The visualization will update itself accordingly.

Wettbewerb **Note:** Wettbewerb deadline extended to 01.02.2021, 23:59
Now it is your turn to get creative. We will conduct a *Schönheitswettbewerb*. Your task is to use Haskell to generate a pixel image, a vector image, a video,... anything visual really! as long as it uses customary formats (e.g. PNG, SVG, MP4)

Needless to say, you can, for example, use and extend the code from the previous exercise for this purpose and submit the prettiest, fanciest, or coolest L-Systems you can think of. You can find some tips for this further below. But we also allow you to do something completely different and utilise third-party libraries, even the ones we have not listed as part of the normal homework. You can find some graphics libraries here.

In any case, besides your code, make sure to include your created piece of art in your artemis repository (if you create large videos, include a link to some filehosting service). As usual leave some comments to the MCs to understand what you were aiming for and how to build your project using the `{-MCCOMMENT ...-}` tag.

The generated piece of art will be graded by aesthetics (nice picture) and technology (impressive code). As an inspiration, have a look at previous iterations of this competition exercise: 2014/15 2019/20

If you go with our L-systems framework, here are some more tips. Extend the `apply` function. It is supposed to call one of the functions that implement Turtle commands (`move`, `turn`, `changeColor`, `sit`, `branch`, `endbranch`) form `Turtle.hs`.

1. Use different colours. For this, think of a new encoding for your desired colour change and call the `changeColor` function in `apply`. `changeColor` expects a `Pen` as an argument – see the `Turtle` module for a definition of that datatype. There are some colours already defined in that module. You can use those build your own ones. Do not change the Turtle.hs file.
2. Make use of branching: This time, you need to set up two encodings to mark the beginning and end of the sequence you want to have executed as a branch. The Turtle is designed to return to the position and angle of the branching point after drawing the branch.

   As an example, take a look at the following L-System that produces a binary tree:

   ```
   alphabet: F, G, u, d, [, ]
   start pattern: F
   rewrite rules: F -> G[uF][dF], G -> GG
   symbol meanings:
     F/G: move forward
     u/d: turn right/left 45°,
     [: start branch
     ]: end branch
   ```

Credits: This exercise is based on a tutorial by the University of Edinburgh and got polished and extended by our tutors. Special thanks to all of them!

Art is not what you see, but what you make others see.

— Edgar Degas