Turtle Graphics and L-systems
Informatics 1 – Functional Programming: Tutorial 7

Due: The tutorial of week 9 (13/14 Nov.)

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please let your tutor know if you cannot attend.

Turtle graphics

Turtle graphics is a simple way of making line drawings.[1] The turtle has a given location on the canvas and is facing in a given direction. A command describes a sequence of actions to be undertaken by a turtle, including moving forward a given distance or turning through a given angle.

Turtle commands can be represented in Haskell using an algebraic data type:

```
type Distance = Float
type Angle = Float
data Command = Go Distance
             | Turn Angle
             | Sit
             | Command :#: Command
```

The last line declares an infix data constructor. We have already seen such constructors in Tutorial 5, where we used them for the binary connectives of propositional logic. While ordinary constructors must begin with a capital letter, infix constructors must begin with a colon. Here, we have used the infix constructor `:#:` to join two commands.

Thus, a command has one of four forms:

- `Go d`, where `d` is a distance — move the turtle the given distance in the direction it is facing. (**Note:** distances are not expected to be negative.)
- `Turn a`, where `a` is an angle — turn the turtle anticlockwise through the given angle.

[1] This exercise in based on a similar exercise used at Imperial College. See `http://el.media.mit.edu/logo-foundation/logo/turtle.html` for more on turtle graphics.

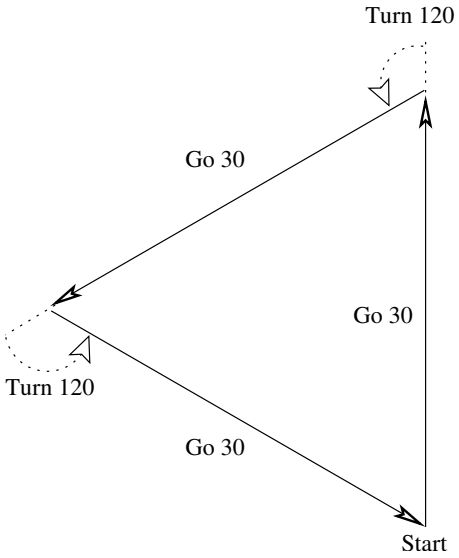

Figure 1: Drawing a triangle with turtle commands

- `Sit` — do nothing: leaves the turtle's position and direction unchanged.
- `p :#: q`, where `p` and `q` are themselves commands — execute the two given commands in sequence.

For instance, to draw an equilateral triangle with sides of thirty units, we need to order the turtle to move forward three times, turning 120° between moves:

```
Go 30 :#: Turn 120 :#: Go 30 :#: Turn 120 :#: Go 30
```

(See Figure 1.)

Viewing paths

You can view a turtle's path by typing

```
*Main> display path
```

where `path` is an expression of type `Command`. This will open a new graphics window and draw the turtle graphic. For example,

```
*Main> display (Go 30 :#: Turn 120 :#: Go 30 :#: Turn 120 :#: Go 30)
```

draws the triangle described above.

When you close the graphics window, GHCi will exit as well, so you will need to re-load your code to draw another picture.

Note: to prevent emacs from freezing, you should close the graphics window before restarting GHCi.

Equivalences

Note that `:#:` is an associative operator with identity `Sit`. So we have:

```
p :#: Sit        =  p
Sit :#: p        =  p
p :#: (q :#: r)  =  (p :#: q) :#: r
```

We can omit parentheses in expressions with `:#:` because, wherever they are placed, the meaning remains the same. In this assignment, when we say that two commands are *equivalent* we mean that they are the same according to the equalities listed above.

However, to evaluate an expression Haskell has to place parentheses; if you ask it to show a command, it will also show where it has placed them:

```
*Main> Sit :#: Sit :#: Sit
Sit :#: (Sit :#: Sit)
```

Exercises

1. In this first exercise we will explore the equivalence of turtle commands and convert them into lists and back.

   (a) Write a function

   ```
   split :: Command -> [Command]
   ```

   that converts a command to a list of individual commands containing no `:#:` or `Sit` elements. For example,

   ```
   *Main> split (Go 3 :#: Turn 4 :#: Go 7)
   [Go 3, Turn 4, Go 7]
   ```

   (b) Write a function

   ```
   join :: [Command] -> Command
   ```

   that converts a list of commands into a single command by joining the elements together. For example,

   ```
   *Main> join [Go 3, Turn 4, Go 7]
   Go 3 :#: Turn 4 :#: Go 7 :#: Sit
   ```

   As in all our examples, the result can be any command equivalent to the given command.

   (c) Note that two commands are equivalent, in the sense of the equivalence laws above, if `split` returns the same result for both.

   ```
   *Main> split ((Go 3 :#: Turn 4) :#: (Sit :#: Go 7))
   [Go 3, Turn 4, Go 7]
   *Main> split (((Sit :#: Go 3) :#: Turn 4) :#: Go 7)
   [Go 3, Turn 4, Go 7]
   ```

   Write a function `equivalent` that tests two commands for equivalence. Give both its type and definition.

   (d) Write two QuickCheck properties to test `split` and `join`. The first should check that `join (split c)` is equivalent to `c`, where `c` is an arbitrary command. The second should check that the list returned by `split` contains no `Sit` and `(:#:)` commands. You need to give the type as well as the definition of both test properties.

2. Using the above translation from lists, we will write a function to draw regular polygons.

   (a) Write a function

   ```
   copy :: Int -> Command -> Command
   ```

   which given an integer and a command returns a new command consisting of the given number of copies of the given command, joined together. Thus, the following two commands should be equivalent:

```
copy 3 (Go 10 :#: Turn 120)
Go 10 :#: Turn 120 :#: Go 10 :#: Turn 120 :#: Go 10 :#: Turn 120
```

(b) Using `copy`, write a function

```
pentagon :: Distance -> Command
```

that returns a command which traces a pentagon with sides of a given length. The following two commands should be equivalent:

```
pentagon 50
```

and

```
Go 50.0 :#: Turn 72.0 :#:
Go 50.0 :#: Turn 72.0 :#:
Go 50.0 :#: Turn 72.0 :#:
Go 50.0 :#: Turn 72.0 :#:
Go 50.0 :#: Turn 72.0
```

(c) Write a function

```
polygon :: Distance -> Int -> Command
```

that returns a command that causes the turtle to trace a path with the given number of sides, of the specified length. Thus, the following two commands should be equivalent:

```
polygon 50 5
pentagon 50
```

Hint: You may need to use the Prelude function `fromIntegral` to convert an `Int` to a `Float`.

3. Next, we will approximate a spiral, by making our turtle travel increasing (or decreasing) lengths and turning slightly in between. Our function `copy` is of no help here, since the distance our turtle needs to travel changes after each corner it takes. Therefore, your spiral function will have to be recursive. It's type signature should be as follows:

```
spiral :: Distance -> Int -> Distance -> Angle -> Command
```

Its parameters are

- `side`, the length of the first segment,
- `n`, the number of line segments to draw,
- `step`, the amount by which the length of successive segments changes, and
- `angle`, the angle to turn after each segment.

To draw such a spiral, we draw `n` line segments, each of which makes angle `angle` with the previous one; the first should be as long as `segment` and thereafter each one should be longer by `step` (or shorter, if `step` is negative).

Thus, the following two commands should be equivalent:

```
spiral 30 4 5 30
```

```
Go 30.0 :#: Turn 30.0 :#:
Go 35.0 :#: Turn 30.0 :#:
Go 40.0 :#: Turn 30.0 :#:
Go 45.0 :#: Turn 30.0
```

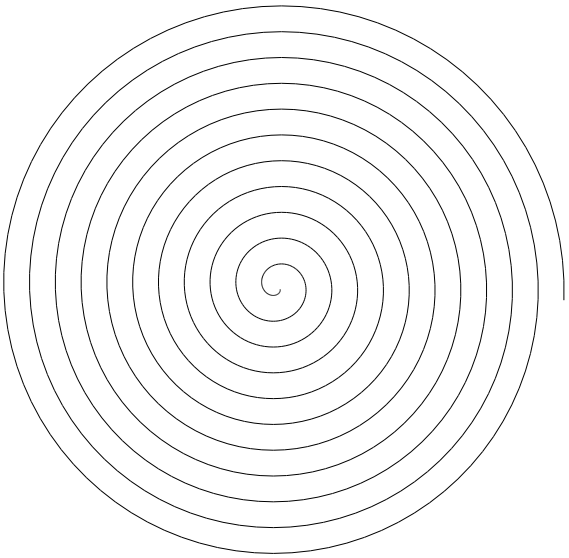

Figure 2: A spiral (`spiral 0.1 1000 0.1 4`)

Note: your recursion should definitely stop after `n` steps (the second parameter), but you will also need to keep in mind that line segments should not become negative in length.

Sample output is shown in Figure 2.

4. Besides the equalities we saw earlier, we might also want to consider the following ones:

```
Go 0  =  Sit
Go d :#: Go e  =  Go (d+e)
Turn 0  =  Sit
Turn a :#: Turn b  =  Turn (a+b)
```

So the `Sit` command is equivalent to either moving or turning by zero, and any sequence of consecutive moves or turns can be collapsed into a single move or turn (as long as moves have a non-negative distance!).

Write a function:

```
optimise :: Command -> Command
```

which, given a command `p`, returns a command `q` that draws the same picture, but has the following properties:

- `q` contains no `Sit`, `Go 0` or `Turn 0` commands, unless the command is equivalent to Sit.
- `q` contains no adjacent `Go` commands.
- `q` contains no adjacent `Turn` commands.

For example:

```
*Main> optimise (Go 10 :#: Sit :#: Go 20 :#:
                 Turn 35 :#: Go 0 :#: Turn 15 :#: Turn (-50))
Go 30.0
```

You can use `split` and `join` to make your task easier. (If your version of `join` adds a `Sit` command, you will need to define a new version which does not.)

Branching and colours

So far we've only been able to draw linear paths; we haven't been able to branch the path in any way. In the next section, we will make use of two additional command constructors:

```
data Command = ...
             | GrabPen Pen
             | Branch Command
```

where `Pen` is defined as:

```
data Pen = Colour Float Float Float
         | Inkless
```

These give two additional forms of path.

- `GrabPen p`, where `p` is a pen: causes the turtle to switch to a pen of the given colour. The following pens are predefined:

  ```
  white, black, red, green, blue :: Pen
  ```

  You can create pens with other colours using the Colour constructor, which takes a value between 0 and 1.0 for each of the red, green and blue components of the colour. The special `Inkless` pen makes no output; you can use `Inkless` to create disjoint pictures with a single command.

- `Branch p`, where `p` is a path: draws the given path and then returns the turtle to direction and position which it had at the *start* of the path (rather than leaving it at the end). Pen changes within a branch have no effect outside the branch.

  To see the effect of branching, draw the following path.

  ```
  let inDirection angle = Branch (Turn angle :#: Go 100) in
     join (map inDirection [20,40..360])
  ```

Introduction to L-Systems

The Swedish biologist Aristid Lindenmayer developed *L-Systems* to model the development of plants.[2]

An L-System consists of a *start pattern* and a set of *rewrite rules* which are recursively applied to the pattern to produce further increasingly complex patterns. For example, Figure 3 was produced from the "triangle" L-System:

angle:	90
start:	`+f`
rewrite:	`f` → `f+f-f-f+f`

Each symbol in the string generated by an L-System represents a path command: here, `+` and `-` represent clockwise and anticlockwise rotation and `f` represents a forward movement. Which symbols represent which commands is a matter of convention.

In this system, only the symbol `f` is rewritten, while the `+` and `-` symbols are not. The rewriting replaces the straight lines with more complex figures.

[2]For more on L-Systems, see `http://en.wikipedia.org/wiki/L-System`. A book, *The Algorithmic Beauty of Plants*, contains beautiful color illustrations produced by L-Systems; it is available online at `http://algorithmicbotany.org/papers/#abop`

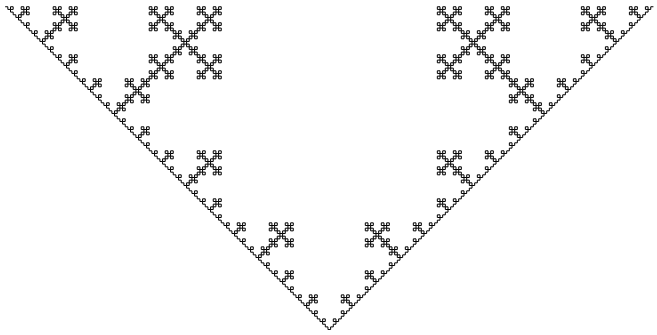

Figure 3: Triangle L-System output

Here is how to generate a picture with an L-System. Begin with the start pattern. Then apply the rewrite rule some number of times, replacing the character on the left by the sequence on the right. For instance, applying the above rule three times gives the following strings in successive steps:

Step	**Pattern**
0	`+f`
1	`+f+f-f-f+f`
2	`+f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f`

3	`+f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f`
	`+f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f`
	`-f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f`
	`-f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f`
	`+f+f-f-f+f+f+f-f-f+f-f+f-f-f+f-f+f-f-f+f+f+f-f-f+f`

Note that you could continue this process for any number of iterations.

After rewriting the string the desired number of times, replace each character that remains by some drawing commands. In this case, replace `f` with a move forward (say, by 10 units), replace each `+` by a clockwise turn through the given angle, and replace each `-` by an anticlockwise turn through the given angle.

Converting L-Systems to functions that return turtle commands is straightforward. For example, the function corresponding to this "triangle" L-System can be written as follows:

```
triangle :: Int -> Command
triangle x  =  p :#: f x
  where
  f 0      = Go 10
  f x      = f (x-1) :#: p :#: f (x-1) :#: n :#: f (x-1)
             :#: n :#: f (x-1) :#: p :#: f (x-1)
  n        = Turn 90
  p        = Turn (-90)
```

Study the above definition and compare it with the L-System definition on the previous page. The above definition is included in `LSystem.hs`, so you can try it out by typing (for instance):

```
display (triangle 5)
```

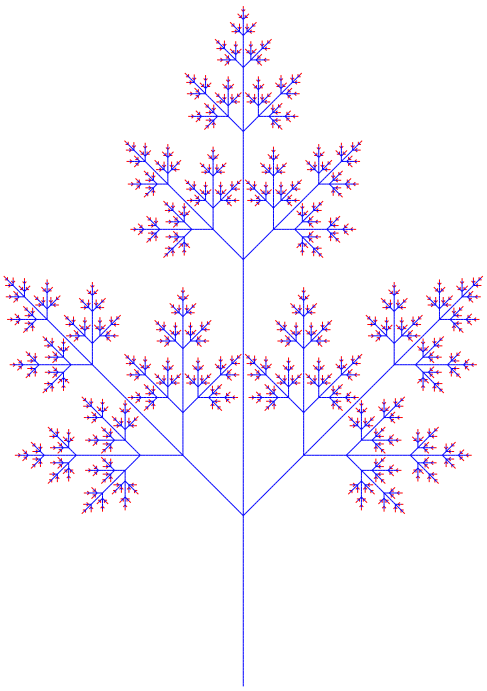

Figure 4: Tree L-System output

A couple of things are worth noting. The symbols from the system that are rewritten are implemented as functions that take a "step number" parameter—in this case, only `f` is rewritten. When we have taken the desired number of steps, the step number bottoms-out at 0, and here `f` is just interpreted as a drawing command. The symbols that are not rewritten are implemented as variables, such as `n` and `p`. In general, there will be one definition in the `where` clause for each letter in the L-System.

A rewrite rule for the L-System may contain clauses in square brackets, which correspond to branches. For example, here is a second L-System, that uses two letters and branches.

angle:	45
start:	`f`
rewrite:	`f` → `g[-f][+f][gf]`
	`g` → `gg`

Here is the corresponding code (also included in `LSystem.hs`.

```
tree :: Int -> Command
tree x  =  f x
  where
  f 0       = GrabPen red :#: Go 10
  f x       = g x :#: Branch (n :#: f (x-1))
              :#: Branch (p :#: f (x-1))
              :#: Branch (g (x-1) :#: f (x-1))
  g 0       = GrabPen blue :#: Go 10
  g x       = g (x-1) :#: g (x-1)
  n         = Turn 45
  p         = Turn (-45)
```

A picture generated by this definition is shown in Figure 4. Here we use different pens to draw the

segments generated by different symbols: this is not part of the description of the L-system, but it generates prettier pictures.

Exercises

5. Write a function `arrowhead :: Int -> Command` implementing the following L-System:

   angle: 60
   start: `f`
   rewrite: `f` → `g+f+g`
   `g` → `f-g-f`

6. Write a function `snowflake :: Int -> Command` implementing the following L-System:

   angle: 60
   start: `f--f--f--`
   rewrite: `f` → `f+f--f+f`

7. Write a function `hilbert :: Int -> Command` implementing the following L-System:

   angle: 90
   start: `l`
   rewrite: `l` → `+rf-lfl-fr+`
   `r` → `-lf+rfr+fl-`

   Note: Not all of the symbols here need to move the turtle. Check your result against the pictures at `http://en.wikipedia.org/wiki/Hilbert_curve` and adjust the final values (e.g. `r 0 = ...`) until it looks like those.

Optional Material

Just for fun, here are more L-Systems for you to try.

- Peano-Gosper:

  angle: 60
  start: `f`
  rewrite: `f` → `f+g++g-f--ff-g+`
  `g` → `-f+gg++g+f--f-g`

- Cross

  angle: 90
  start: `f-f-f-f-`
  rewrite: `f` → `f-f+f+ff-f-f+f`

- Branch

  angle: 22.5
  start: `g`
  rewrite: `g` → `f-[[g]+g]+f[+fg]-g`
  `f` → `ff`

- 32-segment

  angle: 90
  start: `F+F+F+F`
  rewrite: `F` → `-F+F-F-F+F+FF-F+F+FF+F-F-FF+`
  `FF-FF+F+F-FF-F-F+FF-F-F+F+F-F+`