

Turtle Graphics and L-systems  
Informatics 1 – Introduction to Computation  
Functional Programming Tutorial 7

Cooper, Heijltjes, Lehtinen, Melkonian, Sannella, Scott, Vlassi-Pandi, Wadler, Yallop

**Week 8**  
**due 4pm Tuesday 8 November 2022**  
**tutorials on Thursday 10 and Friday 11 November 2022**

You will not receive credit for your coursework unless you attend the corresponding tutorial session. Please email [kendal.reid@ed.ac.uk](mailto:kendal.reid@ed.ac.uk) if you cannot join your assigned tutorial.

**Good Scholarly Practice:** Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

<http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct>.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

# 1 Turtle graphics

Turtle graphics is a simple way of making line drawings. 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 6, 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. Distances must be non-negative.
- `Turn a`, where `a` is an angle — turn the turtle anticlockwise for a positive angle, clockwise for a negative angle.
- `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  $120^\circ$  between moves (see Figure 1):

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

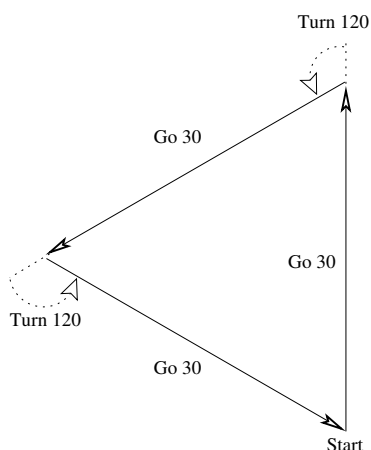


Figure 1: Drawing a triangle with turtle commands

## 1.1 Viewing paths

You can view a turtle's path by typing, for instance,

```
*Main> display pathExample
```

where `pathExample` is an expression of type `Command`. This will output an SVG image as `output.html`, which you can view in your browser. For instance, `pathExample` is already defined for you as:

```
pathExample = Go 30 :#: Turn 120 :#: Go 30 :#: Turn 120 :#: Go 30
```

and draws the triangle described above.

If you ask Haskell to show a command, it will show where it has placed the parentheses.

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

### Exercise 1

We will write a function to draw regular polygons.

(a) Write a function

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

which given a non-negative integer and a command returns a new command consisting of the given number of copies of the given command, joined together. If the requested number of copies is zero, return `Sit`.)

Thus, the following two commands should be equivalent:

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

and

```
(Go 10 :#: Turn 120) :#:
(Go 10 :#: Turn 120) :#:
(Go 10 :#: Turn 120) :#:
Sit
```

(b) Write a function

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

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

```
polygon 50 5
```

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) :#:
Sit
```

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

## 1.2 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])
```

## 2 Introduction to L-Systems

The Swedish biologist Aristid Lindenmayer developed *L-Systems* to model the development of plants.<sup>1</sup>

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 2 was produced from the “triangle” L-System:

```
angle:    90
start:    +f
rewrite:  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 by the given angle 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.

---

<sup>1</sup> 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>

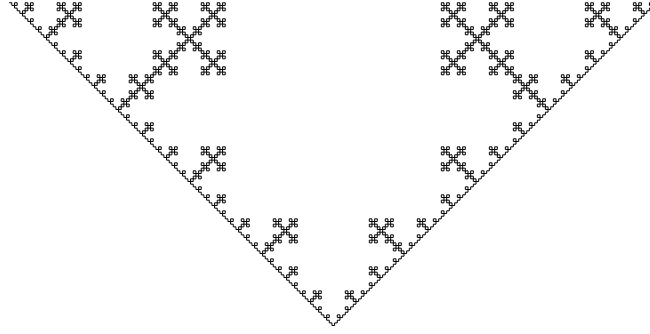


Figure 2: Triangle L-System output

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	<code>+f</code>
1	<code>+f+f-f-f+f</code>
2	<code>+f+f-f-f+f+f+f-f-f+f-f-f+f-f-f+f-f-f+f-f-f+f</code> <code>+f+f-f-f+f+f+f-f-f+f-f-f+f-f-f+f-f-f+f-f-f+f</code> <code>+f+f-f-f+f+f+f-f-f+f-f-f+f-f-f+f-f-f+f-f-f+f</code>
3	<code>-f+f-f-f+f+f+f-f-f+f-f-f+f-f-f+f-f-f+f-f-f+f</code> <code>-f+f-f-f+f+f+f-f-f+f-f-f+f-f-f+f-f-f+f-f-f+f</code> <code>+f+f-f-f+f+f+f-f-f+f-f-f+f-f-f+f-f-f+f-f-f+f</code>

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)
```

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

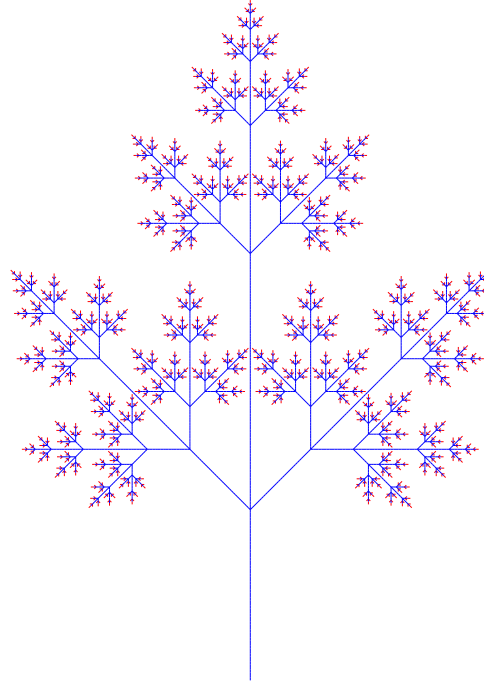


Figure 3: Tree L-System output

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-1) :#: 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 3. Here we've chosen **f** to stand for a red line segment, and **g** to stand for a blue line segment.

### Exercise 2

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

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

### Exercise 3

Write a function `sierpinski :: Int -> Command` implementing the following L-System:

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

### Exercise 4

**(Not marked, just for fun!)** Write a function `hilbert :: Int -> Command` implementing the following L-System:

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

In this case, after the L-system is expanded, each `f` is taken to stand for forward motion, while each remaining `l` and `r` stands for no motion at all.

### Exercise 5

**(Not marked, just for fun!)** Write a function `dragon :: Int -> Command` implementing the following L-System:

```
angle:    90
start:    l
rewrite:  l → l+rf+
          r → -fl-r
```

In this case, after the L-system is expanded, each `f` is taken to stand for forward motion, while each remaining `l` and `r` stands for no motion at all.

### 3 Optional Material

Following the Common Marking Scheme, a student with good mastery of the material is expected to get 3/4 points. This section is for demonstrating exceptional mastery of the material. It is optional and worth 1/4 points.

#### Exercise 6

This exercise explores converting commands 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) In terms of the path drawn, `:#:` is an associative operator with identity `Sit`. So we have:

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

We will say that two commands are *equivalent* if they are the same according to the equalities listed above. Note that two commands are equivalent 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`. Property `prop_split_join` should check that `join (split c)` is equivalent to `c`, where `c` is an arbitrary command. Property `prop_split` should check that the list returned by `split` contains no `Sit` and `(:#:)` commands.

#### Exercise 7

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.

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
```

*Hints:* 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. Remember that `Go` does not take negative arguments.

## Challenge: The 2022 Inf1A FP Programming Competition

You are invited to enter this year's Inf1A programming competition. The prizes are Amazon vouchers—and glory! Depending on quality of entries, we may award several prizes.

Some entries from previous years are online:

<https://homepages.inf.ed.ac.uk/wadler/fp-competition-2019/>  
<https://uoe-inf1a-2020.github.io/FP-competition-2020/>

The competition is optional and unassessed. The prize will go to the best picture generated by a Haskell program. You may use turtle commands or any other Haskell graphics package, and you may generate your picture with an L-System or with any other technique. You may generate still images or use animation or interaction. You may enter alone or with a group of other students.

Entries are judged on the quality of the picture, the quality of the code, and the quality of the documentation. A cool idea explained well is more likely to win. Be creative!

Submit code and image(s), and list everyone who contributed. Document any ideas or code taken from elsewhere. E-mail your submission as follows:

To: Younwoo Jeong <younwoo11650@gmail.com>  
Subject: 2022 Inf1A FP Competition

The judges will be Don Sannella, Younwoo Jeong, and Dee Yeum.

Entries are due **4pm Monday 21 November 2021**. Prizes will be handed out at the final FP lecture.