UNIVERSITY OF THE WEST INDIES
Department of Mathematics and Computer Science
COMP3652–Language Processors
Sem I, 2017
Lecturer(s): Prof. Daniel Coore

The FRACTAL 2.0 Specification

Introduction

A fractal is a geometric object that has non-integer dimension. Fractals are commonly depicted as self-similar shapes at all scales, but this is not the only type of fractal there is. The key feature of a fractal is that the (infinite) sum of the measures (lengths, areas, or volumes) of all of the replicas is different from the measure of the original shape. The language FRACTAL provides a simple way to create images composed of self-similar fractal approximations.

One simple way to think of a fractal is to think of it being generated in discrete levels, each successively higher level being a better approximation to the actual fractal shape. A *generator* indicates how a level n fractal's approximation is generated from its level $n-1$ approximations, which are represented at a smaller scale. For the simplest types of fractals, a level 0 fractal is just a line segment. So the approximation at level 1 can often serve as a geometric representation of the generator, since we can usually interpret each line in the level 1 approximation as a segment along which the fractal should be drawn at a smaller scale. Figure 1 illustrates this process for a simple fractal, called the *Gosper* curve.

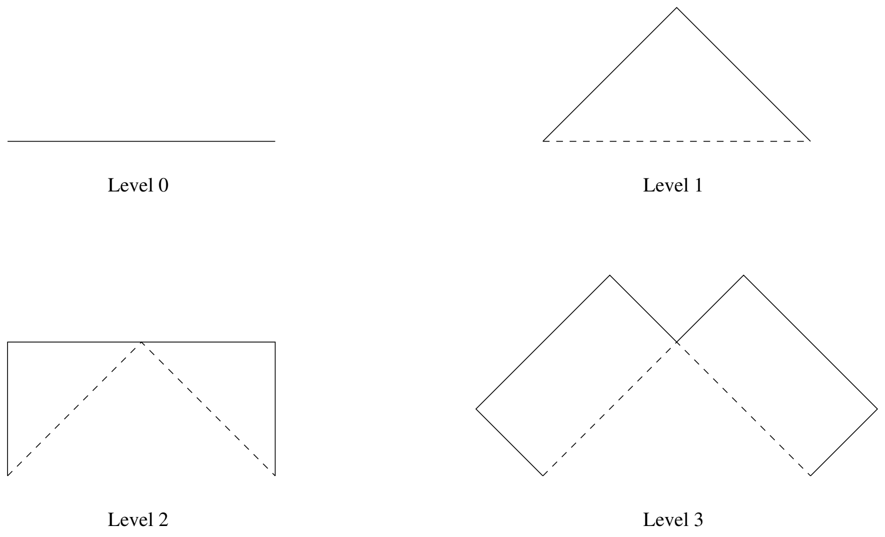


Figure 1: A level n fractal is made up of level $n-1$ fractals arranged according to the generator. A level 0 fractal is a straight line. This example illustrates the Gosper "C" curve. It is created by placing at right angles, two copies of itself each scaled down by $\frac{1}{\sqrt{2}}$ (the level 1 fractal illustrates the generator).

This view of the generator can usually inform us about how to define the generator. For example, the Gosper generator requires that two level $n-1$ fractal approximations are placed at right angles to each other so that the end points of the two fractals touch. They are oriented so that the other ends of the two fractal approximations lie along the line segment specified for the level n fractal to be drawn. To see this more clearly, observe how the rendered Gosper curve generator is a right isosceles triangle oriented so that its hypotenuse is aligned with the intended direction of the fractal. From that, we can deduce that the level $n-1$ fractals must each be drawn at the scale $\frac{1}{\sqrt{2}}$ relative to the level n fractal.

The language FRACTAL is a graphical language that enables its users to quickly generate fractals, and to combine them in interesting ways. It uses the *turtle graphics* system, first introduced in a programming language called LOGO, which was created by Seymour Papert in the 1970's, as a language that could be used to teach children programming. Although FRACTAL will use many primitive commands from LOGO, it is not a superset of LOGO.

In LOGO, there is a single turtle with a pen, which may be in either the up or down position. Whenever the pen is down, the turtle leaves a mark upon the screen, as it moves. The turtle always has a position on the screen, and a bearing. It can be given commands to move forward, or backward by a given distance, and to turn either left or right by a given angle. Each turtle command typically does only one of those actions.

For example, the Gosper curve could be defined by the following FRACTAL code:

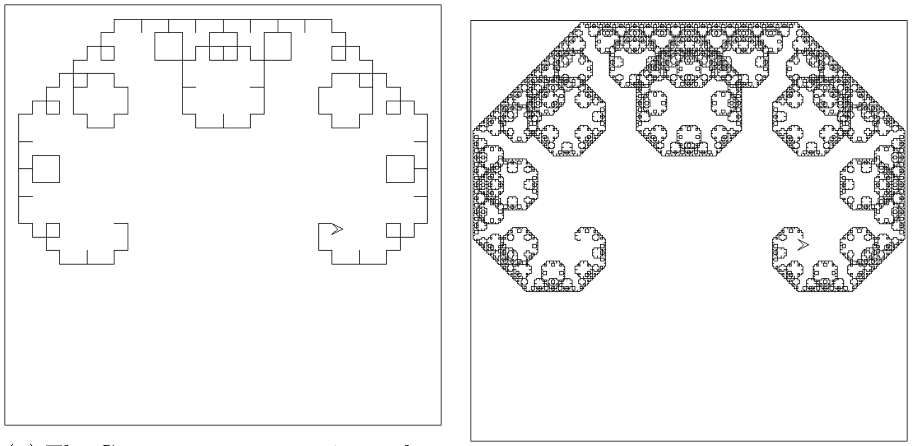
```
def gosper fractal  // make a fractal called gosper
       (0.707): // with replicas at 0.707 (1/√2) scale
   left 45      // turn left by 45 degrees
   self         // draw a scaled replica
   right 90     // turn right 90 degrees
   self         // draw a scaled replica
   left 45      // point turtle in original direction
   end           // end of fractal definition body
```

To display a fractal, we use the `render` command. It takes an optional argument specifying the level of approximation of the fractal, and a required argument specifying the scale of the fractal. In FRACTAL optional command parameters are always provided in square brackets immediately after the command keyword. For example, calling `render[8](200) gosper` after defining `gosper` above would render the Gosper curve at level 8, along a horizontal line segment 200 units (turtle steps) long (see Figure 2a). In contrast, leaving out the level optional argument causes the `render` command to reduce the scale until the current distance to be rendered over is less than 2 steps. Thus, the effect of calling simply `render(200) gosper` is to render `gosper` "in full" (as far as the resolution will permit) along a line segment 200 steps long (see Figure 2b). By default, the turtle starts at the centre of the display (coordinate (0, 0)) and the display is 600 turtle steps wide and 600 turtle steps high.

Primitive Statements and Expressions

A FRACTAL program is a sequence of statements, each of which typically causes some side effect, such as directly manipulating the turtle (by using a turtle command) or defining a new fractal generator.

Expressions in FRACTAL are either arithmetic, logical or fractal combinations, which denote, respectively, numerical, boolean or fractal values. Numerical values, in turn, can be either integer or real valued. Integers may always be supplied where reals are expected, but it is an error to pass



(a) The Gosper curve approximated at level 8, rendered along a line 200 steps long. This was obtained by invoking `render[8](200) gosper` (after moving the turtle 100 steps back from its home position).

(b) The Gosper curve, rendered along a line 200 steps long. This was obtained by invoking `render(200) gosper` (after moving the turtle 100 steps back from its home position).

Figure 2: The difference between rendering an approximation of a fractal to a certain level, and rendering the "full" fractal.

a real where an integer was expected. Fractal values may be passed only where fractal values are expected.

Arithmetic expressions are combined using the usual operators (+, -, /, *) with the usual order of precedence. Logical operators include the comparators, >, <, =, as well as the boolean operators `and, or, not`. All the comparators have a lower precedence than any arithmetic operator, and `not` is higher than `and`, which in turn, is higher than `or`.

In the tables of special forms below, the variable n is used to denote an integer (arithmetic) expression, the variable r is used to denote a real valued expression, and the variable f is used to denote a fractal valued expression. All parentheses and brackets appearing in the tables are to be taken as part of the syntax of the statement or expression being documented.

Statements

The statements that directly manipulate the turtle, which have all been inherited from LOGO, are listed in Table 1. Table 2 lists other statements, such as forms that affect how fractals are created and rendered.

Fractal Valued Expressions

The primary way to denote a fractal is to define one using the `fractal` special form. A fractal's definition starts with the keyword `fractal`, followed immediately by the scale factor enclosed within parentheses, then a colon, and a non-empty sequence of statements, and ends with the keyword `end`. The sequence of statements, collectively called the *body* of the fractal, is made up of statements that may occur at the top level, or one additional one called `self`.

The keyword `self` is valid only within a fractal generator's body and denotes the current fractal reduced by the fractal's scale factor. It is by reference to the `self` special form that a fractal generator reproduces itself. A reference to `self` at the top level should generate a runtime error.

Loops

The `repeat` command is adapted from the original LOGO version and is the only (current) form that permits iterated computation. The `repeat` command may be used both at the top level as well as within the body of fractal definitions. It can also contain calls to render fractals within its body. So, the combination of `repeat` and fractal forms have the potential to generate some very intricate forms.

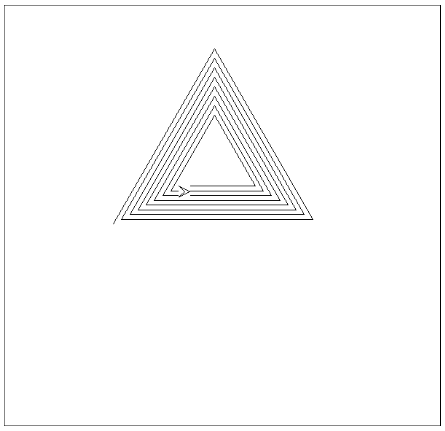
The two figures shown in Figures 3a and 3b illustrate the figures obtained by the code using the `repeat` command, as shown below:

Listing 1: Triangle spiral

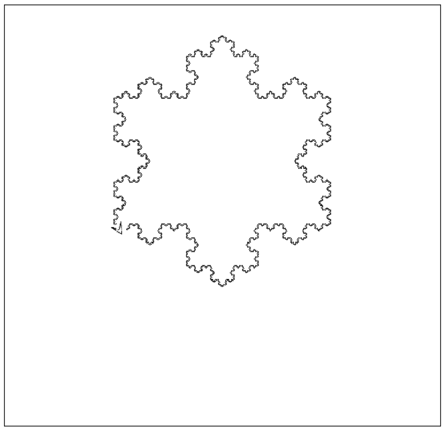
```
repeat[i] 8:
    left 60 fd (200 - 15 * i)
    right 120 fd (200 - 15 * i - 5)
    rt 120 fd (200 - 15 * i - 10)
    rt 180
    end
```

Listing 2: The snowflake curve

```
left 60
repeat 3:
    render(200) koch rt 120
end
```

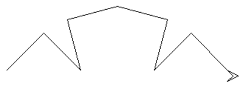


(a) A triangular spiral generated using `repeat` and turtle commands (after moving the turtle 100 steps back from its home position).



(b) The snowflake curve, obtained by using `repeat` to draw the Koch curve 3 times along the edge of an equilateral triangle.

Figure 3: Illustrations of the use of the `repeat` command.


(a) The generator for `gosper @ koch`.

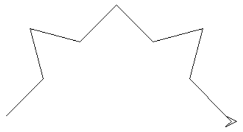
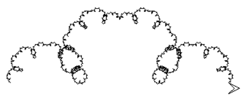

(b) The generator for `koch @ gosper`.

Figure 4: The rendering of the fractal generator for the result of composing `gosper` and `koch` in either order.


(a) The fractal represented by `gosper @ koch`.

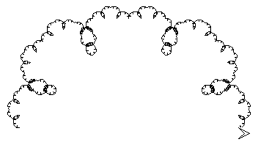

(b) The fractal represented by `koch @ gosper`.

Figure 5: The fractal rendering for the result of composing `gosper` and `koch` in either order.

Means of Combination

The language FRACTAL provides two different combinators for fractal values. A fractal may be composed onto another to produce a single new fractal, or a fractal may be sequenced with another one. The composition of two fractals is a kind of interleaving of generators so that one generator is applied to the path of the other generator. Table 3 summarises all of the special forms used to define and combine fractal valued expressions.

When two fractals are composed, the generator for the resulting fractal is the figure that would be obtained by rendering the generator of the first one along the generator of the second. Semantically, this can be considered to rendering the second fractal, but references to `self` will represent the overall fractal, while references to `self` in the first fractal will represent the second. The generators and fractals resulting from composing `gosper` and `koch` in both orders are illustrated, respectively, in Figures 4a, 4b, 5a, 5b.

Sequencing two fractals is simply constructing a generator by starting the second where the first ends. Figures 6a and 6b illustrate how the generators for the result of a sequence expression look, and Figures 7a and 7b illustrate the corresponding fractals when fully rendered.

Note that by using the `save` and `restore` commands, it is possible to create fractal shapes that have more than one "tip". For example, consider the definition of a fractal tree, as below:

```
def tree fractal (0.5):
  fd 1.0
  save
  left 45 self
  restore
  right 45 self
  end
```

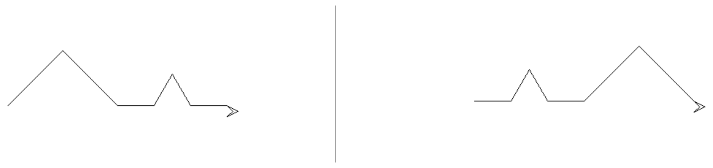

(a) The generator for `gosper ! koch`.
(b) The generator for `koch ! gosper`.

Figure 6: The rendering of the generators for the result of sequencing `gosper` and `koch` in either order.

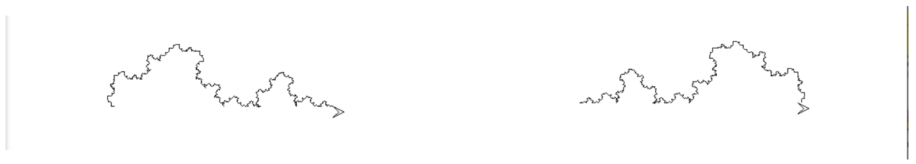

(a) The fractal represented by `gosper ! koch`.
(b) The fractal represented by `koch ! gosper`.

Figure 7: The fractal rendering for the result of sequencing `gosper` and `koch` in either order.

Both the generator and the fractal for `tree` are illustrated in Figures 8a and 8b to help you visualise it. The semantics of the sequence command, as described above, imply that the second fractal will be drawn at only one of the tips (specifically, the one at which the `end` command occurs). This might seem counter-intuitive from the point of view of the art of the images, although perfectly sensible from the semantic description. (How might it be possible to alter the semantics of sequence – or create a new combinator – so that the following fractal would indeed be drawn at each "tip" of the first fractal? From the standpoint of the control structure of the code, what would such semantics represent?).

These means of combination should produce some interesting, and perhaps unexpected fractal shapes when they are rendered – probably because neither fractal operator is commutative. For example, while it might be easy to mentally visualise the generator resulting from composing `tree` with `gosper` (i.e. `tree @ gosper`) it is less obvious what `gosper @ tree` means, let alone what its fractal will look like.

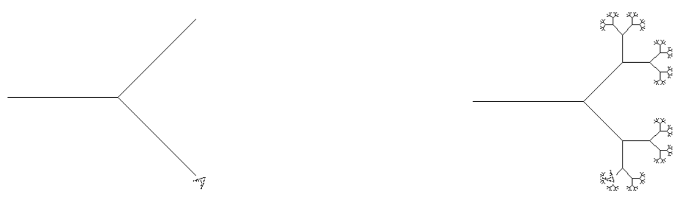

(a) The fractal generator represented by `tree`.
(b) The fractal represented by `tree` rendered at 200 steps.

Figure 8: The generator and fractal for the code presented for `tree`

Precedence

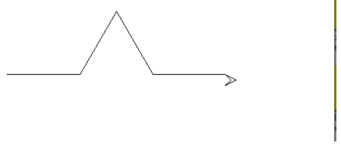
The composition operation has a higher precedence than the sequence operation. So the following two expressions would be evaluated in the order indicated by their parenthesized versions:

```
gosper @ koch !  gosper ≡ (gosper @ koch) !  gosper
gosper !  koch @ gosper ≡ gosper !  (koch @ gosper)
```

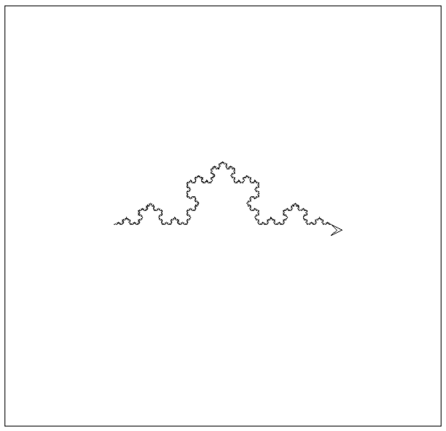
Examples

Below is the definition of the Koch curve, which also sometimes called the *snowflake curve* because when three Koch curves are used to replace the sides of an equilateral triangle, the resulting figure resembles a snowflake.

```
def koch fractal (0.333):
    self
    left 60 self
    right 120 self
    left 60 self
    end
```



(a) The Koch generator resulting from calling `render[1](200) koch` (after moving the turtle 100 steps back from its home position).



(b) The Koch curve resulting from calling `render(200) koch` (after moving the turtle 100 steps back from its home position).

Keyword	Abbrev	Meaning
`forward` r	`fd`	Move the turtle forward by the specified distance.
`back` r	`bk`	Move the turtle backward by the specified distance.
`left` r	`lt`	Turn the turtle left by the specified angle (in degrees), without changing its position.
`right` r	`rt`	Turn the turtle right by the specified angle, without changing its position.
`penup`	`pu`	Raise the pen. Now, when the turtle moves, it will not draw a line.
`pendown`	`pd`	Lower the pen. Now the turtle will draw a line as it moves.
`home`		Restore the turtle to its initial position and orientation, without drawing a line.
`clear`		Clear the turtle display.

Table 1: Table of FRACTALcommands that directly manipulate the turtle

Keyword	Abbrev	Meaning
`render`[n](r) f		Show fractal f approximated at level n, and scaled proportionately to a line segment of length r. Any fractal at level 0 is regarded as a line segment. So the level of approximations represents the number of levels of recursion to be executed in drawing the fractal. If no level is provided, the fractal will be rendered at the highest resolution available. Practically, this means that straight lines are drawn when the length of the line segment for rendering becomes less than 2 turtle steps.
`save`		Save the current state of the turtle (on a stack).
`restore`		Restore a previously saved turtle state to be current.
`define` *name* f	`def`	Bind *name* to the fractal value f.
`self`		Meaningful only within a fractal generator's body. Render this fractal at the scale factor built into the fractal's definition. Practically, `self` draws the next lower level approximation of the current fractal, appropriately scaled.
`repeat`[*id*] n: *body* `end`	`rep`	Execute *body* n times, setting *id* to the values from 1 to n inclusively, on each pass.

Table 2: Statements other than turtle commands in FRACTAL. Forms that have square brackets in their syntax should be interpreted as having the component in square brackets as optional. If the optional component is provided, the square brackets must be provided.

Keyword	Abbrev	Meaning
`fractal`		Declare the start of a fractal generator's body
`end`		Declare the end of a fractal generator's body
f_1 `@` f_2		Compose fractal f_1 with f_2
f_1 `!` f_2		Sequence fractal f_1 with f_2

Table 3: Commands for defining and combining fractals.