CMPU 101 § 53 · Computer Science I

Generative Recursion

22 February 2024

Where are we?

```
data List:
  | empty
  | link(first :: Any, rest :: List)
end
```

Self-reference

Recursive data

```
data List:
  | empty
  | link(first :: Any, rest :: List)
end
```

Self-reference

Recursive data

```
fun list-fun(lst :: List) -> ...:
 cases (List) lst:
    | empty => ...
    | link(f, r) =>
      ... f ...
      ... list-fun(r) ...
  end
end
```

Recursive call

Recursive functions

The same idea holds for lists, binary trees, trinary trees, *n*-ary trees, and all kinds of other recursive data types: *The structure of the function follows the structure of the data.*

The recursive functions we've written have used *structural* (or *natural*) *recursion*.

In structural recursion, each recursive call takes some sub-piece of the data.

Going through a list, we keep taking the **`rest`** of the list.

Going through a tree, we keep looking at the sub-trees.

Generative recursion

In ***generative recursion***, the recursive cases are generated based on the problem to be solved.

Generative recursion can be harder because neither the base nor recursive cases follow from a data definition.

Template for generative recursion

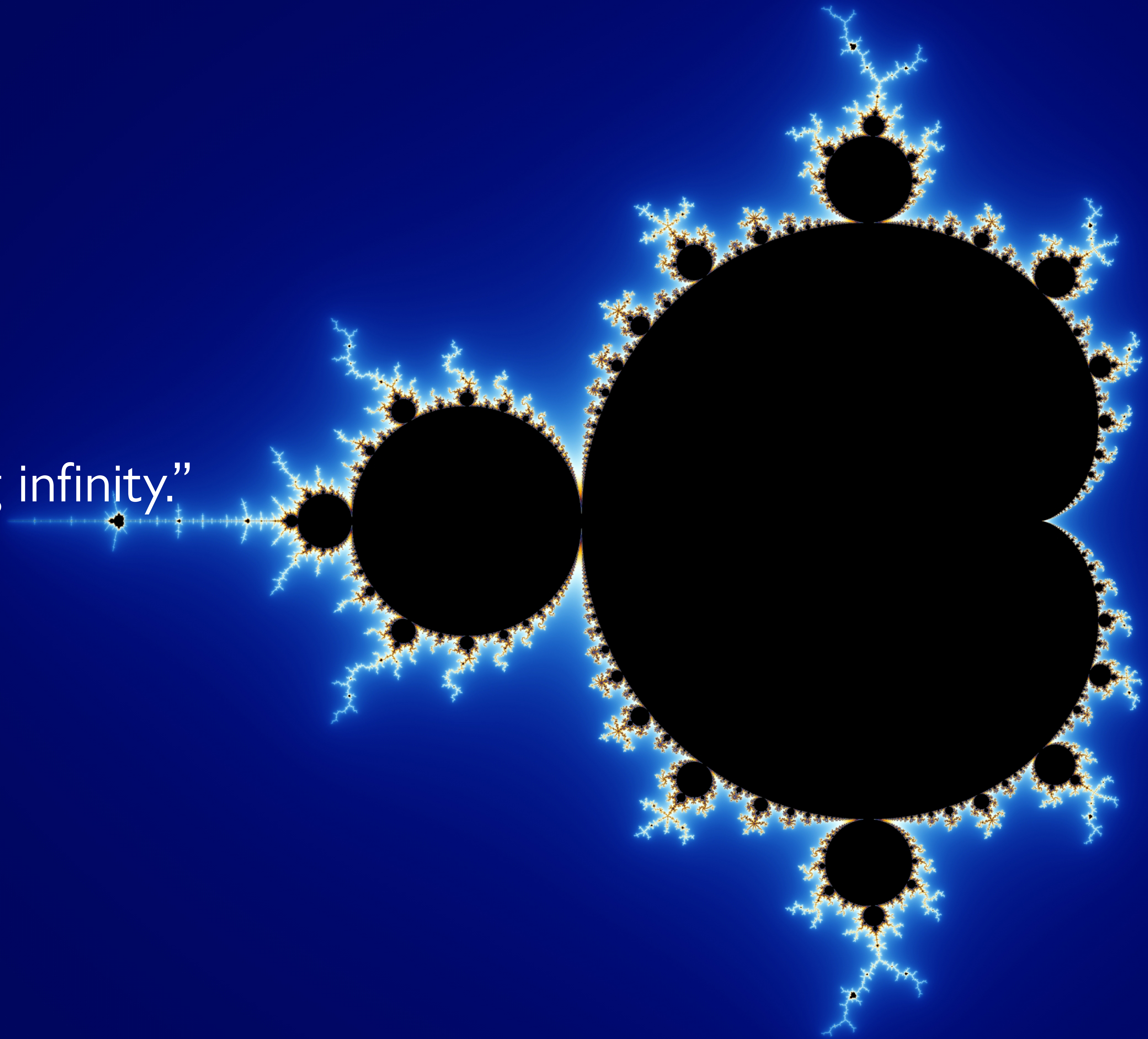
```
fun problem-solver(d) -> ...:
  if is-trivial(d):
    # Base case: The computation is in some way
    #   trivial.
    ... d ...
  else:
    # Recursive case: Transform the data d to generate
    #   new problems.
    combiner(
      ...d...,
      problem-solver(transform(d)),
      ...)
  end
end
```

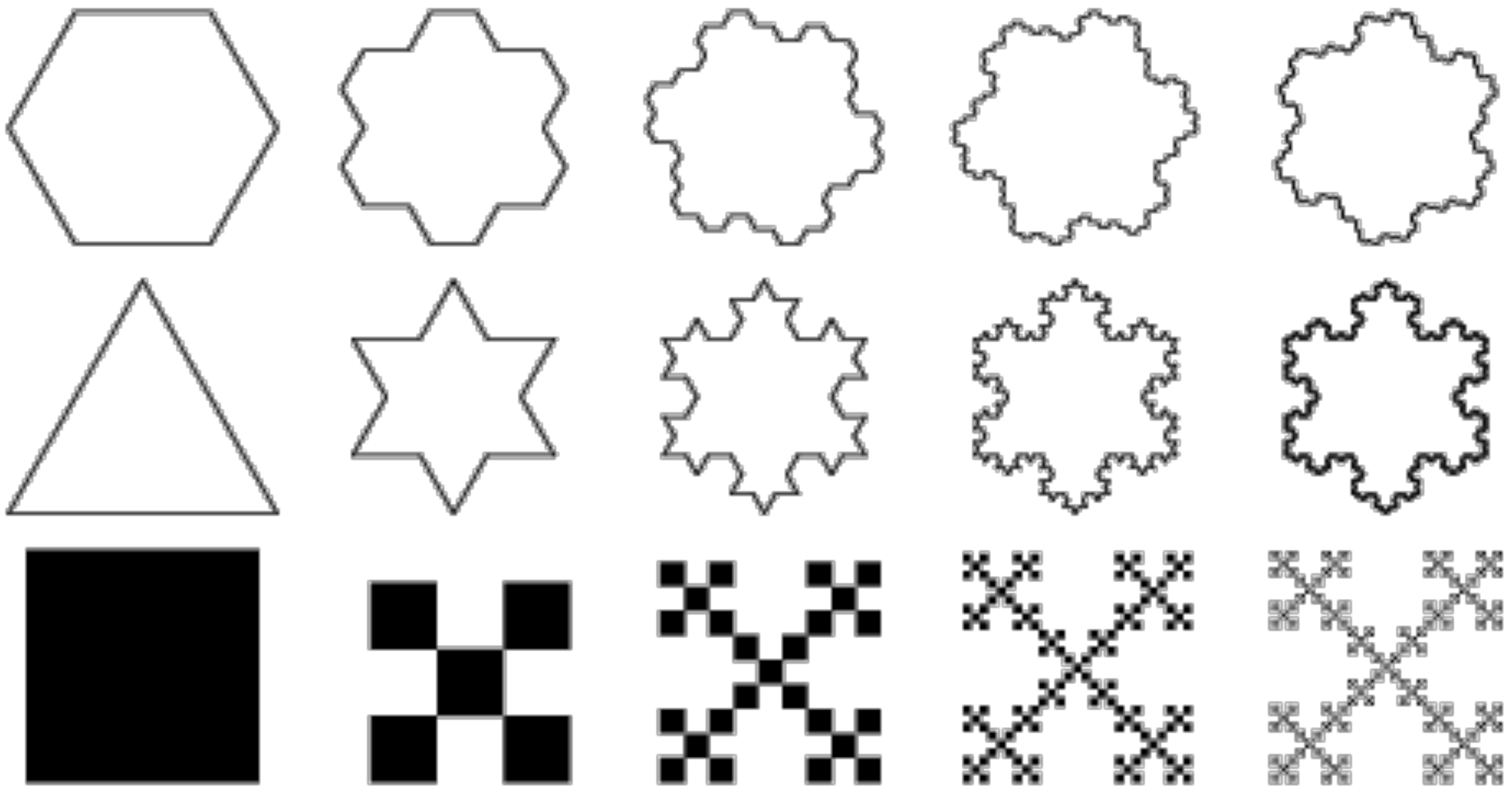
When you write a function with generative recursion you need to be careful about ***termination*** – how do you know you'll ever reach the base case?

Fractals

“A fractal is a way of seeing infinity.”

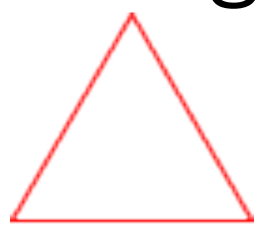
Benoit Mandelbrot



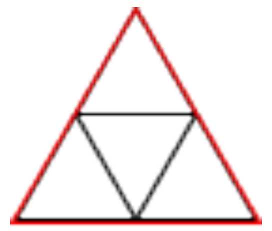


Let’s design a function that consumes a number and produces a *Sierpiński triangle* of that size:

Start with an equilateral triangle with side length *s*:

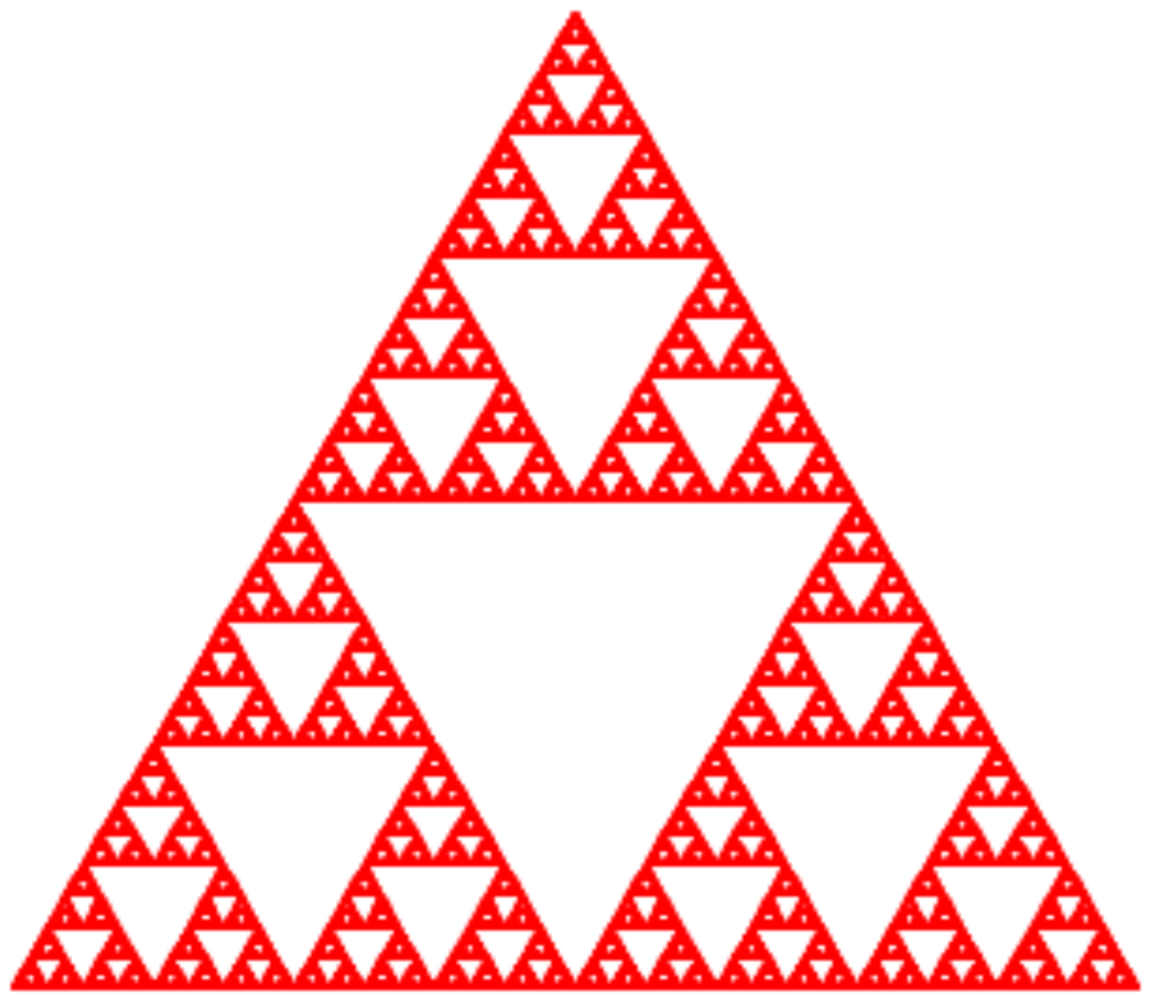


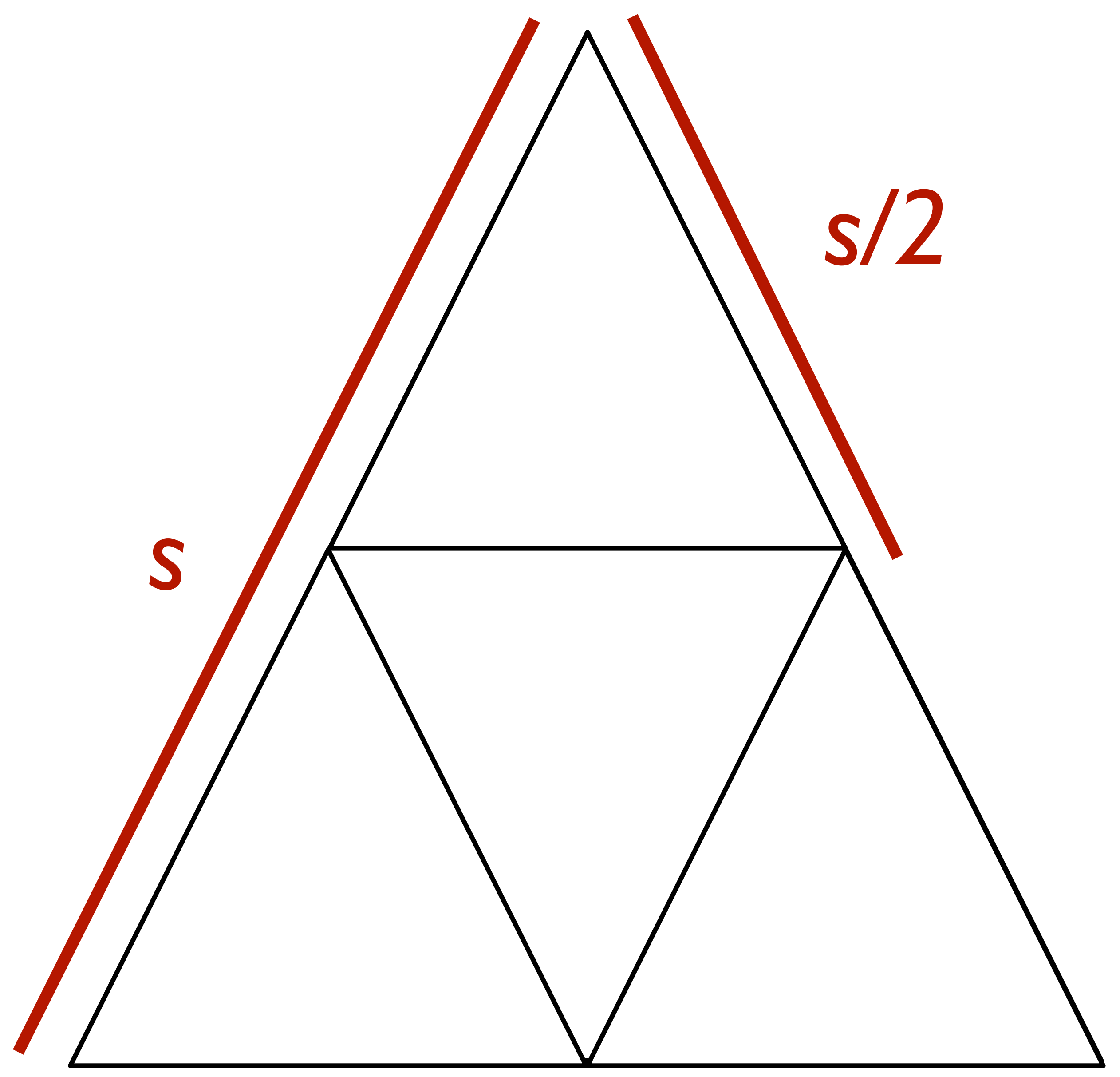
Inside that triangle are three more Sierpiński triangles:



And inside of each of those … and so on.

Producing something that looks like this:




s/2
s

```
# How small a shape can get before we stop drawing
smaller ones
CUTOFF = 10

fun s-tri(s :: Number) -> Image:
  doc: "Produce a Sierpiński triangle of the given size
by generating one for s/2 and placing one copy above
two copies"
  if s <= CUTOFF:
    triangle(size, "outline", "red")
  else:
    sub = s-tri(s / 2)
    above(sub,
      beside(sub, sub))
  end
end
```

How do we know that this function won't run forever?

Three-part termination argument:

Base case: `s <= CUTOFF`

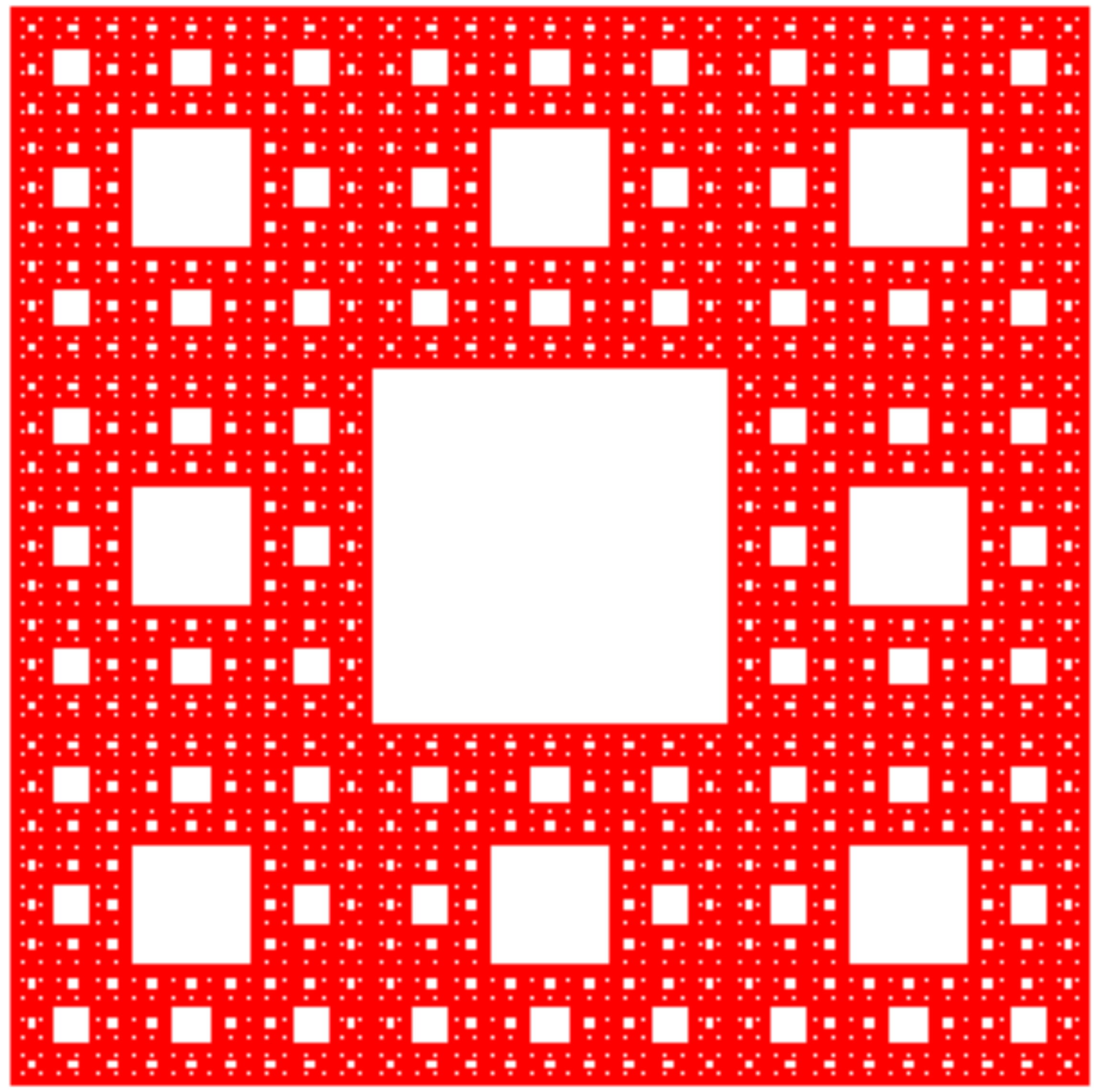
Reduction step: `s / 2`

Argument that repeated application of reduction step will eventually reach the base case:

As long as the cutoff is > 0 and **s** starts ≥ 0, repeated division by 2 will eventually be less than the cutoff.

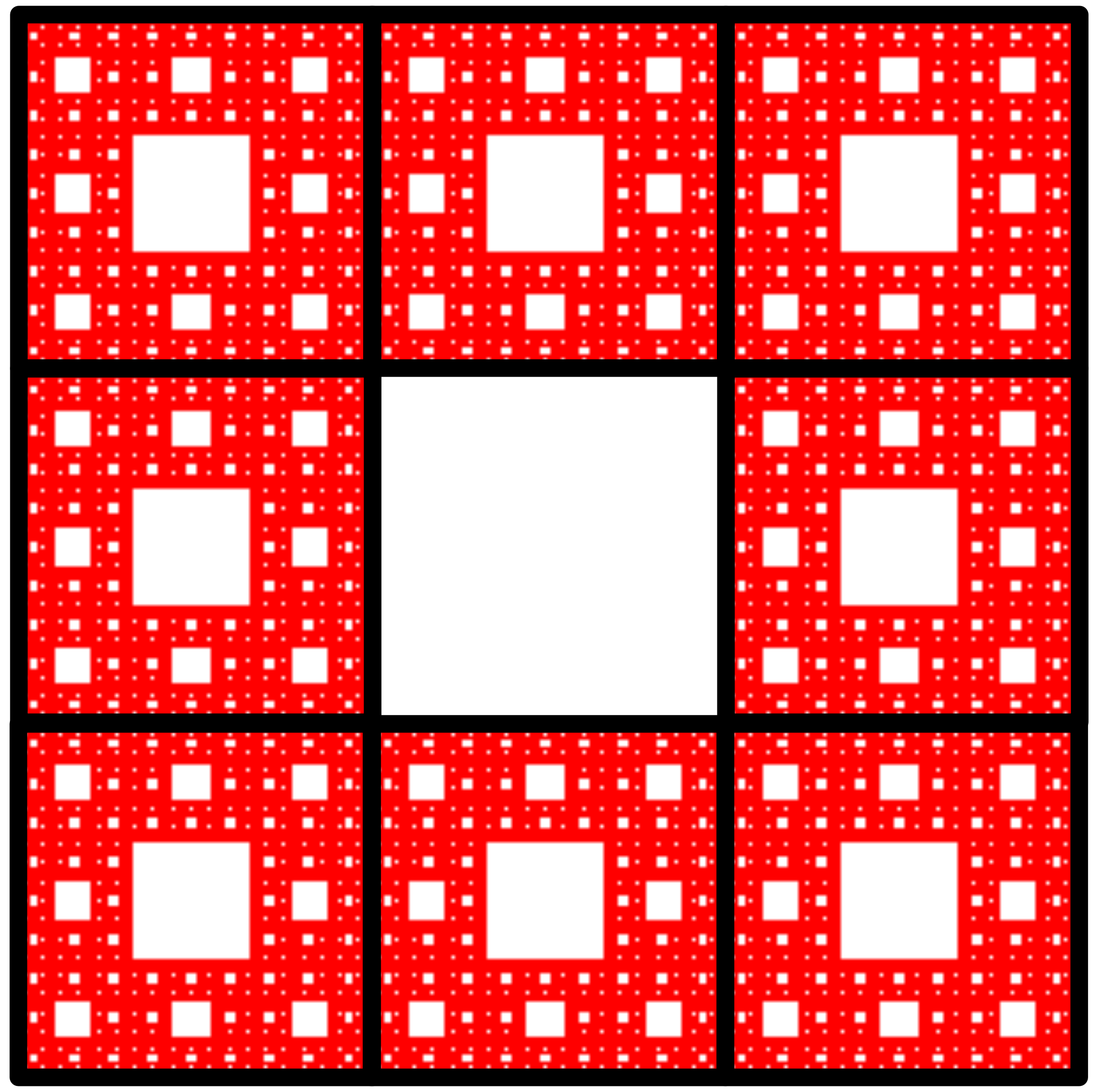
Exercise

Design a function **`s-carpet`** to produce a Sierpiński carpet of size *s*:



Exercise

Design a function **`s-carpet`** to produce a Sierpiński carpet of size *s*:



*There are **eight** copies of the recursive call positioned around a blank square*

```
fun s-carpet(s :: Number) -> Image:
  doc: "Draw a Sierpiński carpet of size s-by-s by
generating an s/3 carpet and positioning it on every
side of an empty s/3 square"
  if s <= CUTOFF:
    square(s, "outline", "red")
  else:
    sub = s-carpet(s / 3)
    blk = square(s / 3, "solid", "white")
    above3(
      beside3(sub, sub, sub),
      beside3(sub, blk, sub),
      beside3(sub, sub, sub))
  end
end
```

How do we know that this function won't run forever?

Three-part termination argument:

Base case: `s <= CUTOFF`

Reduction step: `s / 3`

Argument that repeated application of reduction step will eventually reach the base case:

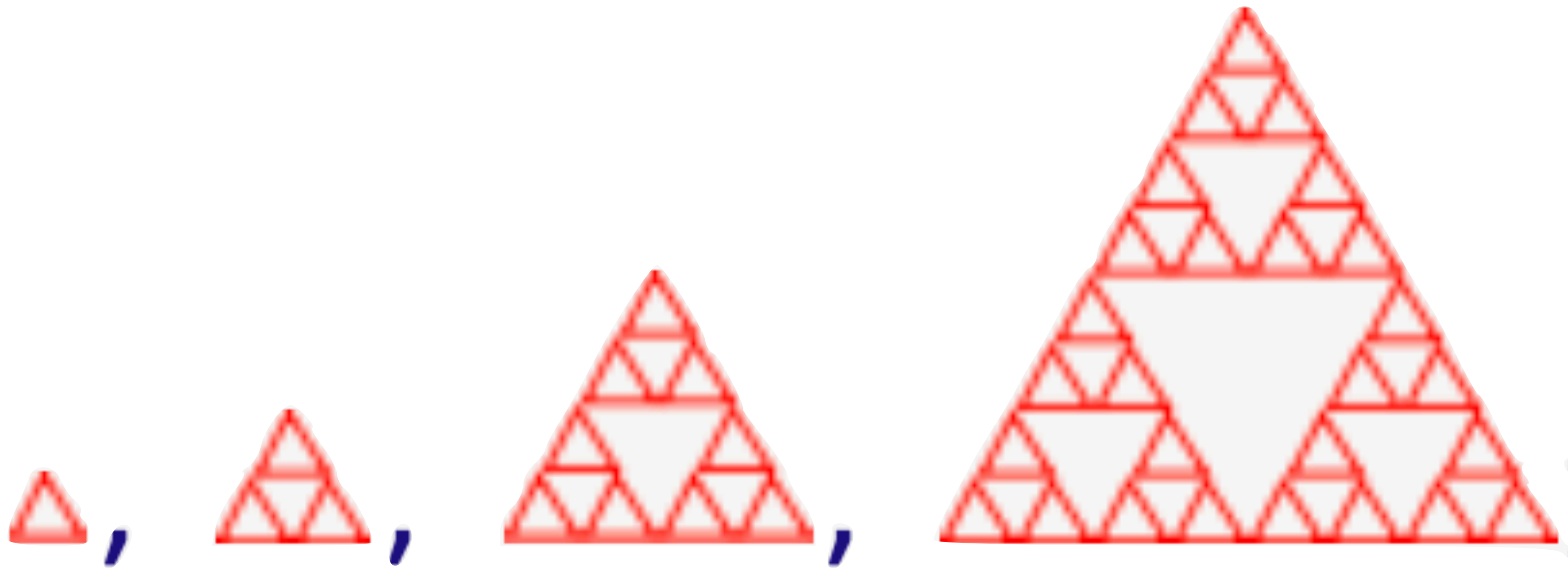
As long as the cutoff is > 0 and **s** starts ≥ 0, repeated division by 3 will eventually be less than the cutoff.

Animation

What if we want to see the progression of the fractal becoming more complex?

Exciting! Dynamic!

```
››› map(s-tri, [list: 10, 20, 40, 80])
```

[list: ]

It might be more fun to see this change over time rather than flattened into a list.

Pyret has a mechanism for supporting interactive visual programs, called a **`reactor`**.

To use it, first write

```
include reactors
```

```
reactor:
  init: initial-state,
  to-draw: draw-function,
  event-type: event-function,
end
```

Class code:

`tinyurl.com/101-2024-02-22`

Acknowledgments

This lecture incorporates material from:

Gregor Kiczales, University of British Columbia

Marc Smith, Vassar College