Generating Fractals

27 February 2023
Where are we?
data List:
  | empty
  | link(first :: Any, rest :: List)
end

fun list-fun(lst :: List) -> ...
cases (List) lst:
  | empty => ...
  | link(f, r) =>
    ... f ...
    ... list-fun(r) ...
end
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
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:
[See class code]
How do we know that this function won’t run forever?

Three-part termination argument:

*Base case:* \( s \leq \text{CUTOFF} \)

*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 \( \geq 0 \), repeated division by 2 will eventually be less than the cutoff.
Design a function \texttt{s-carpet} to produce a Sierpiński carpet of size $s$: 

\begin{center}
\includegraphics[width=0.5\textwidth]{s-carpet_diagram.png}
\end{center}
Design a function \texttt{s-carpet} to produce a Sierpiński carpet of size $s$:

There are eight copies of the recursive call positioned around a blank square.
[See class code]
How do we know that this function won’t run forever?

Three-part termination argument:

\[ \text{Base case: } s \leq \text{CUTOFF} \]

\[ \text{Reduction step: } s / 3 \]

\[ \text{Argument that repeated application of reduction step will eventually reach the base case:} \]

As long as the cutoff is > 0 and s starts \(\geq 0\), repeated division by 3 will eventually be less than the cutoff.
Animation
Class code:

https://tinyurl.com/101-52-2023-02-27
Acknowledgments

This lecture incorporates material from:

Gregor Kiczales, University of British Columbia
Marc Smith, Vassar College