Exam 2

Still grading

Assignment 5

Out tonight (hopefully)
The recursive functions we’ve been writing so far have used *structural recursion*.

Each recursive call takes some sub-piece of the data, e.g.,

- going through a list, we keep taking the *rest* of the list
- processing a natural number \( n \), we take \( n-1 \)

The *structure* of the data determines the structure of the recursive function.

Today we’ll look at *generative recursion*.

This is more general – 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 the data definitions.
Template for generative recursion

(define (genrec-fun d)
    (if (trivial? d)
        ;; Base case: The computation is in some way trivial.
        (trivial-answer d)
        ;; Recursive case: Generate new problems.
        (... d ...)
        (genrec-fun (next-problem d))))
Termination

When we’ve written a function using structural recursion, the depth of recursion has been limited by the size of the input, e.g.,

\[
\text{(sum–list (list 3 6 5 4))}
\rightarrow (+ 3 \text{(sum–list (list 6 5 4)))}
\rightarrow (+ 3 (+ 6 \text{(sum–list (list 5 4)))})
\]

For generative recursion, *termination can’t be taken for granted*; we’ll need to argue why we know our computation will end.
Example: 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 $s$:

![Equilateral Triangle](image)

Inside that triangle are three more Sierpinski triangles:

![Sierpinski Triangle](image)

And inside of each of those … and so on.

Producing something that looks like this:
[sierpinski-triangle.rkt]
How do we know that this function won’t run forever?

Three-part termination argument:

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

* **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 `scarpet` to produce a Sierpinski carpet of size $s$: 

![Sierpinski Carpet](image)
Design a function \texttt{scarpet} to produce a Sierpinski carpet of size $s$:

There are \textbf{eight} copies of the recursive call positioned around a blank square.
[sierpinski-carpet.rkt]
How do we know that this function won’t run forever?

Three-part termination argument:

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

*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 \(\geq 0\), repeated division by 3 will eventually be less than the cutoff.
Termination arguments aren’t always this easy! Consider this function:

```scheme
;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
    (list 1)
    (cons n (if (even? n)
               (hailstones (/ n 2))
               (hailstones (+ n 1))))
```

Termination arguments aren’t always this easy! Consider this function:

```scheme
;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
      (list 1)
      (cons n (if (even? n)
                 (hailstones (/ n 2))
                 (hailstones (+ 1 (* n 3)))))))
```
Termination arguments aren’t always this easy! Consider this function:

```
;; Natural -> [List-of Natural]
(define (hailstones n)
 (if (= n 1)
  (list 1)
  (cons n (if (even? n)
           (hailstones (/ n 2))
           (hailstones (+ 1 (* n 3)))))))
```

Termination argument:
Termination arguments aren’t always this easy! Consider this function:

```scheme
;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
      (list 1)
      (cons n (if (even? n)
                 (hailstones (/ n 2))
                 (hailstones (+ 1 (* n 3)))))))
```

Termination argument:

*Base case:* $(= n 1)$
Termination arguments aren’t always this easy! Consider this function:

;;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
    (list 1)
    (cons n (if (even? n)
              (hailstones (/ n 2))
              (hailstones (+ 1 (* n 3)))))
))

Termination argument:

Base case: (= n 1)

Reduction step:
Termination arguments aren’t always this easy! Consider this function:

;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
      (list 1)
      (cons n (if (even? n)
                 (hailstones (/ n 2))
                 (hailstones (+ 1 (* n 3)))))))

Termination argument:

**Base case:** (= n 1)

**Reduction step:**
if n is even: (/ n 2)
Termination arguments aren’t always this easy! Consider this function:

```scheme
;;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
      (list 1)
      (cons n (if (even? n)
                (hailstones (/ n 2))
                (hailstones (+ 1 (* n 3)))))))
```

Termination argument:

**Base case:** (= n 1)

**Reduction step:**
- if n is even: (/ n 2)
- if n is odd: (+ 1 (* n 3))
Termination arguments aren’t always this easy! Consider this function:

```scheme
;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
      (list 1)
      (cons n (if (even? n)
                  (hailstones (/ n 2))
                  (hailstones (+ 1 (* n 3)))))))
```

Termination argument:

**Base case:** $(= n 1)$

**Reduction step:**

- if $n$ is even: $(/ n 2)$
- if $n$ is odd: $(+ 1 (* n 3))$

*Argument that repeated application of reduction step will eventually reach the base case:*
Termination arguments aren’t always this easy! Consider this function:

```scheme
;; Natural -> [List-of Natural]
(define (hailstones n)
  (if (= n 1)
    (list 1)
    (cons n (if (even? n)
      (hailstones (/ n 2))
      (hailstones (+ 1 (* n 3)))))))
```

Termination argument:

**Base case:** (= n 1)

**Reduction step:**

- if n is even: (/ n 2)
- if n is odd: (+ 1 (* n 3))

*Argument that repeated application of reduction step will eventually reach the base case:* 😐
The *Collatz conjecture* says that the *hailstones* function always terminates.

However, it’s not clear how to prove it (i.e., make a precise argument) – this is a decades-old open research problem.

Figure it out and mathematical fame is yours!
Example: Breaking strings into lines
Traditionally, the character set used in computers has included not only alphanumeric characters and punctuation, but also “control” characters.

An example is the newline character, which signals the start of a new line of text. The characters "\" and "n" appearing consecutively in a string are interpreted as a single newline characters.

For example, the string "ab\ncd" is a five-character string with a newline as the third character. It would typically be printed as ab one line and cd on the next.
Consider the problem of converting a string like "one\ntwo\nthree"
into a list of strings,
(list "one" "two" "three"),
one for each line.
We can start by converting a string into a list of all its characters using the built-in `explode` function.

Then the problem could be solved using simple structural recursion on that list of characters – but it’s hard! The recursion gets bogged down in a lot of little details.

The generative solution is easier!
Instead of thinking of the list of characters as a list of characters, think of it as a list of lines:

```
one
two
three
```

```
one
two
three
```

A list of lines is either empty or a line followed by a list of lines.

Start with helper functions that divide the list of characters into the first line and the rest of the lines.
Why is this generative recursion?

`split-lines` can be rewritten as:

```
(define (split-lines los)
  (cond [[(empty? los) '()]
    [else
      (cons (implode (first-line los))
          (split-lines (rest-of-lines los)))]])
```

The recursive call to `split-lines` is *not* using the data definition for `[List-of 1String]`. It often gets many steps closer to the base case in one recursive application!
It is using a data definition of a “list of lines”, but that’s a higher-level abstraction that we imposed on top of the \textit{[List-of 1String]}, our actual argument.

The key part of the generative recursion pattern is that the argument to \texttt{split-lines} is being generated by \texttt{rest-of-lines}.

With generative recursion, we needed that “aha” that transformed the problem into a list of lines.

Was it worth it? Consider the solution using structural recursion on the next slide. This still needs a wrapper function to do both pre- and post-processing.
(define (split-lines los)
  (local [(define (helper los)
            (cond [(empty? los)
                   (list '())
                   [(and (empty? (rest los))
                      (string=? "\n" (first los)))
                     (list '())]
                [else
                   (local [(define r (helper (rest los)))]
                     (cond [(string=? (first los) "\n")
                            (cons '() r)]
                           [else
                            (cons (cons (first los) (first r))
                                  (first r))]])]))])
  (map implode (helper los))))
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Gregor Kiczales
Shriram Krishnamurthi
Marc Smith
U. Waterloo CS135