General recursion

CS135 Lecture 14

Three properties of functions we consider in CS135

1. **Termination.** Our templates and Rules for Recursion guarantee termination. Since some argument gets smaller on each recursive application, we eventually reach a base case.
2. **Correctness.** Ideally we would prove correctness. Instead, we use testing to help ensure correctness.
3. **Efficiency.** We measure efficiency by the number of substitution steps, as measured by the stepper.

No rules.

Up until now, we have applied strict Rules for Recursion.

Recursion on natural numbers, lists, and binary trees follow the same pattern:

1. We recur by making something smaller: **`sub1`**, **`rest`**, **`get-left`**, **`get-right`**
2. We test termination by checking for the smallest thing: **`0`** or **`empty`**

Following these rules both guarantees termination and makes solving problems easier by giving us templates to follow. Structural recursion allows us to generalize these rules to any recursive data definition.

But…

From now on it’s up to you to decide the rules to follow (unless we say otherwise).

L14.0 Famous names in recursion

Euclid’s Algorithm

In Math 135, you learn that Euclid’s algorithm for the Greatest Common Divisor (GCD) can be derived from the following identities for natural numbers *n* and *m*, with m > 0:

$$gcd(n, m) = gcd(m, n \bmod m)$$

$$gcd(n, 0) = n$$

The algorithm is described in *Euclid's Elements* (c. 300 BCE) making it perhaps the oldest non-trivial algorithm known.

We can implement these equations directly in Racket. They are already recursive.

https://en.wikipedia.org/wiki/Euclidean_algorithm

Euclid’s algorithm

```
;; Compute gcd(n,m) using Euclid's algorithm
;; euclid: Nat Nat -> Nat
(define (euclid n m)
  (cond [(zero? m) n]
        [else (euclid m (remainder n m))]))

(check-expect (euclid 100 85) 5)
(check-expect (euclid 12300369 9900297) 300009)
```

How do we know this algorithm will terminate?

Why does Euclid’s algorithm terminate?

Note that on each recursive call the second argument becomes smaller.

If the first argument is smaller than the second argument, the first recursive function application switches them, which makes the second argument smaller.

After that, the second argument is always smaller than the first argument in any recursive application, due to the application of the remainder modulo *m*.

```
(euclid 100 85)⇒(euclid 85 15)⇒(euclid 15 10)⇒(euclid 10 5)⇒ 5
```

The second argument decreases with each recursive call, but is bounded below by 0, so the recursion must eventually stop.

Fibonacci numbers

Fibonacci numbers can be defined as a function over the natural numbers:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n \geq 2 \end{cases}$$

The Fibonacci sequence is *F*(0), *F*(1), *F*(2),...

It starts: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

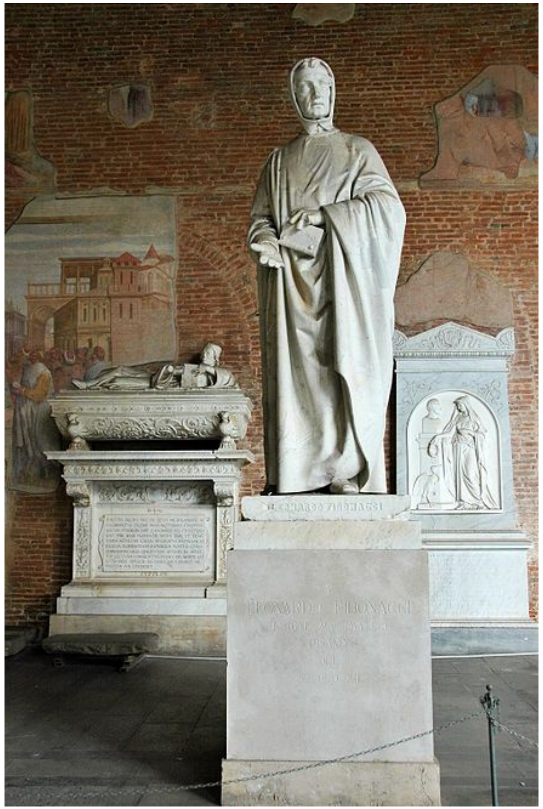
https://en.wikipedia.org/wiki/Fibonacci_sequence

Fibonacci numbers

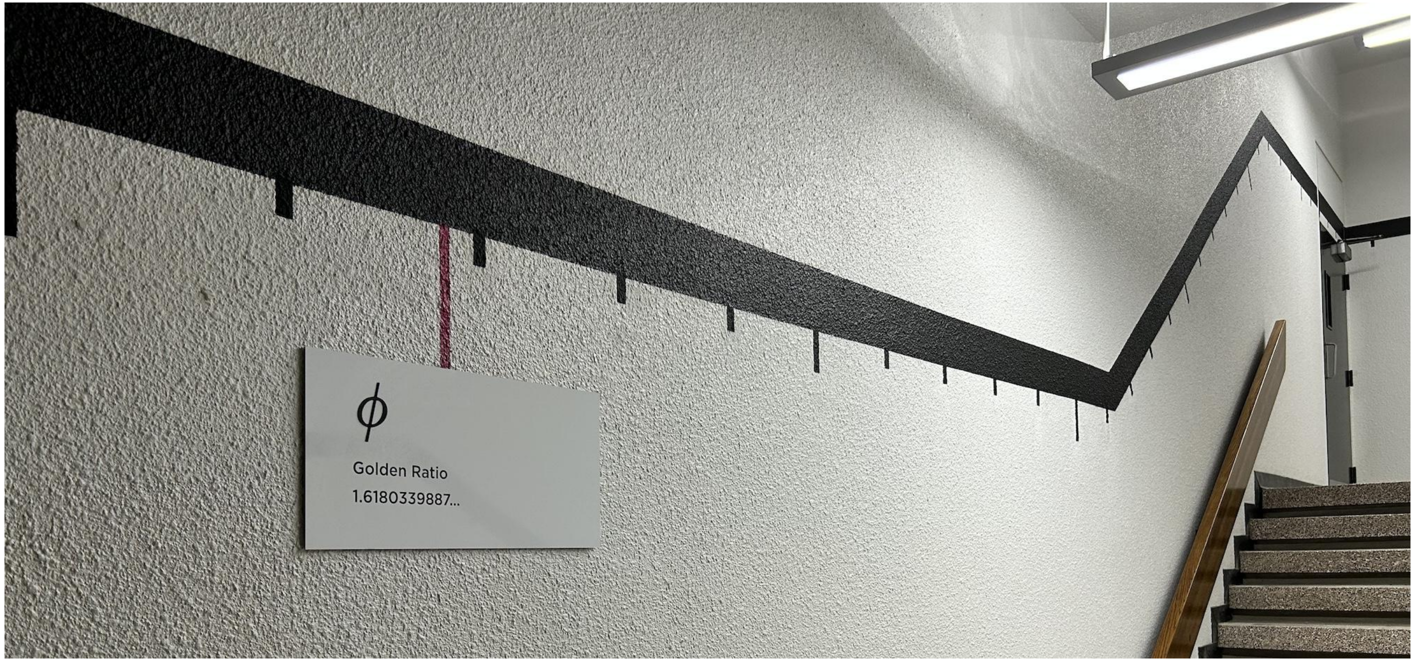
Born around 1170, Fibonacci was one of the most famous mathematicians of the European Middle Ages.

Along with other properties of these numbers, the ratio of successive Fibonacci numbers converges to the *Golden Ratio*:

$$\lim_{n\to\infty} \frac{F(n+1)}{F(n)} = \phi = \frac{1+\sqrt{5}}{2}$$



The Golden Ratio



Famous enough to be on the MC stairs

Computing Fibonacci numbers

The definition of Fibonacci numbers can be converted directly into Racket

```
;; Compute the nth Fibonacci number
;; fibonacci: Nat -> Nat
(define (fibonacci n)
  (cond [(zero? n) 0]
        [(= n 1) 1]
        [else (+ (fibonacci (sub1 n)) (fibonacci (- n 2)))]))
(check-expect (fibonacci 0) 0)
(check-expect (fibonacci 1) 1)
(check-expect (fibonacci 6) 8)
```

The efficiency is $O(\phi^n)$, i.e., exponential. Does this follow the Rules for Recursion?

Computing Fibonacci numbers

If we think about how a human would compute the *n*th Fibonacci number, we can see that *F*(*n*) can be computed in linear time, *O*(*n*).

```
;; fastfib: Nat Nat Nat -> Nat
(define (fastfib a b n)
  (cond [(zero? n) a]
        [else (fastfib b (+ a b) (sub1 n))]))

;; Compute the nth Fibonacci number
;; fibonacci2: Nat -> Nat
(define (fibonacci2 n)
   (fastfib 0 1 n))
```

Bonus: This version follows the Rules for Recursion.

Collatz: Termination is sometimes hard

The following function produces 1 if it terminates. Despite considerable attention from mathematicians, it is unknown if it will terminate for every *n* > 0.

```
;; collatz: Nat -> Nat
;; Requires: n > 0
(define (collatz n)
  (cond [(= n 1) 1]
        [(even? n) (collatz (/ n 2))]
        [else (collatz (+ 1 (* 3 n)))]))
```

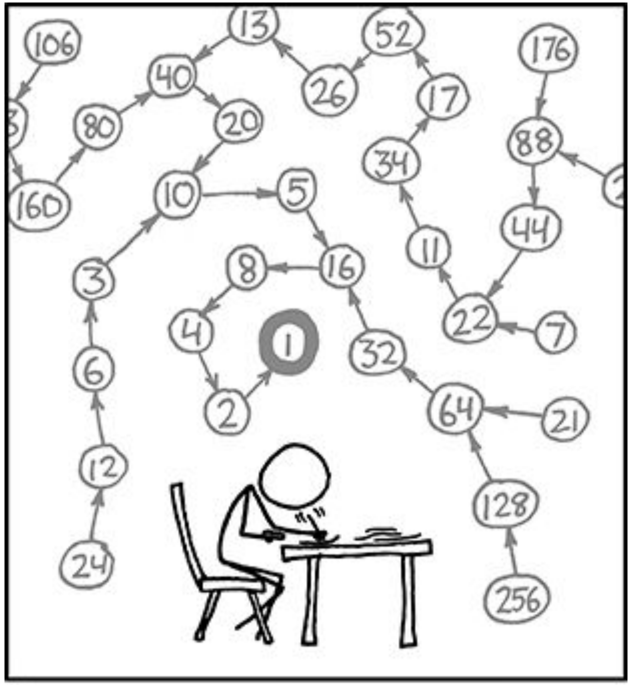
Named after the mathematician **Lothar Collatz**, who introduced it in 1937.

https://en.wikipedia.org/wiki/Collatz_conjecture

The Collatz conjecture

The Collatz conjecture (that the function will terminate for all *n* >0) has been verified for all positive integers at least up to 2.95×10^{20}

http://xkcd.com/710/



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

The Collatz conjecture

We can better see what **collatz** is doing by producing a list.

```
;; Produce the list of the intermediate
;; results calculated by the collatz function.
;; collatz-list: Nat -> (listof Nat)
;; requires: n > 0
(define (collatz-list n)
  (cons n (cond [(= n 1) empty]
                [(even? n) (collatz-list (/ n 2))]
                [else (collatz-list (+ 1 (* 3 n)))])))
(check-expect (collatz-list 1) (list 1))
(check-expect (collatz-list 4) (list 4 2 1))
```

L14.1 Setting limits

Counting down

The Rules for Recursion make it easy to generate a list of natural numbers in decreasing order.

```
(define (count-down n)
  (cond [(zero? n) (list 0)]
        [else (cons n (count-down (sub1 n)))]))
(check-expect (count-down 9) (list 9 8 7 6 5 4 3 2 1 0))
```

But to generate a list of numbers in increasing order, while following the rules, we either need to use **`reverse`**, do some arithmetic with **`n`**, or count up with an accumulator.

Counting up with an accumulator

```
;; Produce a strictly increasing list of natural numbers by
;; counting up from a natural number n times
;; count-up: Nat Nat -> (listof Nat)
(define (count-up from n)
  (cond [(zero? n) empty]
        [else (cons from (count-up (add1 from) (sub1 n)))]))

(check-expect (count-up 100 5)
              (list 100 101 102 103 104))
(check-expect (count-up 9999999 0) empty)
```

Counting up with a limit

In the case of counting up, breaking the rules makes things simpler.

Breaking the rules allows us to count up to a limit with the template:

```
(define (count-up-template n limit)
  [cond [(>= n limit) ...]
        [else (... (count-up-template (add1 n) limit))]])
```

We can test termination with > or >= depending on the problem at hand.

We sometimes refer to this template as the “count up pattern”.

List of the natural numbers up to *n*

```
(define (up-to/helper i limit)
  (cond [(> i limit) empty]
        [else (cons i (up-to/helper (add1 i) limit))]))

;; Produce the natural numbers from 0 up to n
;; up-to: Nat -> (listof Nat)
(define (up-to n)
  (up-to/helper 0 n))

(check-expect (up-to 9)
              (list 0 1 2 3 4 5 6 7 8 9))
```

The Sieve of Eratosthenes

The *sieve of Eratosthenes* is a way to find all prime numbers up to a certain limit, starting with a list of numbers from 2 up to that limit:

1. The **`first`** of the list is prime.
2. Filter the **`rest`** of the list to remove multiples of the **`first`**.
3. Recursively sieve the filtered list.

We only need to filter up to the square root of the limit, since any larger composite number will have already been filtered out as a multiple of a smaller prime.

The sieve is named after Eratosthenes of Cyrene an Ancient Greek mathematician who described the idea in the 3rd century BCE.

Filtering multiples

```
;; filter multiples of a natural number from a list
;; remove-multiples: Nat (listof Nat) -> (listof Nat)
(define (remove-multiples n lst)
  (cond [(empty? lst) empty]
        [(zero? (remainder (first lst) n))
         (remove-multiples n (rest lst))]
        [else (cons (first lst)
                    (remove-multiples n (rest lst)))]))
(check-expect
 (remove-multiples 3 (list 144 1 9 7 9 6 2 3 14))
 (list 1 7 2 14))
```

The Sieve of Eratosthenes (helper)

```
;; sieve/helper: Num (listof Nat) -> (listof Nat)
(define (sieve/helper limit remaining)
  (cond [(empty? remaining) empty]
        [(> (first remaining) limit) remaining]
        [else (cons (first remaining)
                    (sieve/helper limit
                                  (remove-multiples
                                   (first remaining)
                                   (rest remaining))))]))
```

The Sieve of Eratosthenes

```
;; Produce a list of primes up to n
;; sieve: Nat -> (listof Nat)
;; Requires: n > 1
(define (sieve n)
  (sieve/helper (sqrt n) (rest (rest (up-to n)))))

(check-expect (sieve 16) (list 2 3 5 7 11 13))
```

L14.2 Mutual Recursion

Mutual recursion

Mutual recursion occurs when two or more functions apply each other:

`f` applies `g` and `g` applies `f`

Mutually recursive functions can sometimes be merged into a single function, but using mutual recursion often leads to simpler code that is easier to understand.

In this module we will consider mutual recursion on lists.

As we will see in lecture L15, mutual recursion on general trees is a common and useful tool.

Drawing Hands
M.C. Escher
https://mcescher.com/

Keeping alternative elements of a list

```
(define (keep-next lst)(...)  ; keep the next element
(define (skip-next lst)(...)  ; skip the next element

(define eg0 (list 'a 'b 'c 'd 'e 'f 'g))
(check-expect (keep-next eg0) (list 'a 'c 'e 'g))
(check-expect (skip-next eg0) (list 'b 'd 'f))
```

To solve this problem, we note that there’s two ways to keep alternative elements:

1. Keep the first element, skip the second, keep the third, etc.; or
2. Skip the first element, keep the second, skip the third, etc.

Keeping alternative elements of a list

```
(define (keep-next lst)
  (cond [(empty? lst) empty]
        [else (cons (first lst) (skip-next (rest lst)))]))

(define (skip-next lst)
  (cond [(empty? lst) empty]
        [else (keep-next (rest lst))]))

(define eg0 (list 'a 'b 'c 'd 'e 'f 'g))
(check-expect (keep-next eg0) (list 'a 'c 'e 'g))
(check-expect (skip-next eg0) (list 'b 'd 'f))
```

Mergesort

We can use **`keep-next`** and **`skip-next`** to implement a sort that is more efficient than the insertion sort from back in lecture L09.

1. Split the list into two lists of equal length, or nearly equal:

   ```
   (list 8 4 3 9 1 6 2 5 0 7) ⇒ (list 8 3 1 2 0) (list 4 9 6 5 7)
   ```

2. Sort each of the two shorter lists:

   ```
   (list 0 1 2 3 8) (list 4 5 6 7 9)
   ```

3. Merge the sorted lists:

   ```
   (list 0 1 2 3 4 5 6 7 8 9)
   ```

What are the base cases?

Reminder: Merging two sorted lists (from lecture L10).

```
;; merge: (listof Num) (listof Num) -> (listof Num)
;; Requires: lists must be sorted in ascending order
(define (merge lst1 lst2)
  (cond [(empty? lst1) lst2]
        [(empty? lst2) lst1]
        [(< (first lst1) (first lst2))
         (cons (first lst1) (merge (rest lst1) lst2))]
        [(> (first lst1) (first lst2))
         (cons (first lst2) (merge lst1 (rest lst2)))]
        [else (cons (first lst1)
                    (cons (first lst2)
                          (merge (rest lst1) (rest lst2))))]))
```

Mergesort

```
;; sort a list of numbers in ascending order
;; mergesort: (listof Num) -> (listof Num)
(define (mergesort lst)
  (cond [(or (empty? lst) (empty? (rest lst))) lst]
        [else (merge (mergesort (keep-next lst))
                     (mergesort (skip-next lst)))]))

(define eg1 (list 8 4 3 9 1 6 2 5 0 7))
(define digits (list 0 1 2 3 4 5 6 7 8 9))
(check-expect (mergesort eg1) digits)
(check-expect (mergesort (list 2 1 1 2)) (list 1 1 2 2))
```

How do we know that **`mergesort`** will terminate?

Termination depends on the behaviour of **`keep-next`** and **`skip-next`**.

```
(define (mergesort lst)
  (cond [(or (empty? lst) (empty? (rest lst))) lst]
        [else (merge (mergesort (keep-next lst))
                     (mergesort (skip-next lst)))]))
```

Both functions make their arguments smaller (by roughly half) unless their argument is the empty list or a list with one element.

These are the base cases for **`mergesort`**.

The efficiency of **`mergesort`** is O(*n* log *n*)

We’ll leave a formal discussion and proof to a future course, but let’s use the stepper to consider the case where the list is in decreasing order:

`(mergesort empty)`	⇒ 5 steps
`(mergesort (count-down 2))`	⇒ 174 steps
`(mergesort (count-down 4))`	⇒ 390 steps
`(mergesort (count-down 8))`	⇒ 902 steps
`(mergesort (count-down 16))`	⇒ 2110 steps

The number of steps slightly more than doubles as the size of the list doubles.

We can guess that a list of 32 elements might take around 5000 steps.

In a later CS course you may see a proof that **`mergesort`** is O(*n* log *n*)

Merge sort vs. Insertion sort

Merge sort

```
    (mergesort (count-down 16))     ⇒ 2110 steps
    (mergesort (count-down 10000))  ⇒ approximately 1 second
```

Insertion sort

```
    (sort (count-down 16))          ⇒ 1633 steps
    (sort (count-down 10000))       ⇒ approximately 40 seconds
```

In general, insertion sort is faster on shorter lists, but becomes noticeably slower than merge sort on longer lists.

L14 Summary

Breaking the rules

Unless we explicitly require it in a problem or question, we will no longer strictly enforce the rules for recursion.

Nonetheless, the problems we give you will almost never require anything other than structural recursion. Most problems will not require an accumulator.

If you are struggling with a problem, start with rules and the templates you know. Follow the design recipe. Keep it simple.

“Generative recursion”

Sometimes you may see the term “generative recursion”, especially in older study materials or if you have a friend in CS115.

You can think of generative recursion as opposite of structural recursion, because the recursion does depend solely on structure of the data consumed. Recursive applications are “generated” by computation.

We use the term “general recursion” to cover both types. We will not ask questions on exams or assignments that explicitly require “generative recursion”.

L14: You should know

- Euclid’s algorithm, Fibonacci numbers, and the Collatz conjecture.
- The “count-up pattern” and the Sieve of Eratosthenes.
- Mutual recursion.
- Merge sort.
- Why none of these follow our Rules for Recursion.

In exam or assignment questions, we might still explicitly require a certain type of recursion (structural, accumulative, or mutual). You should know what we mean.

If we ask you to “follow the rules for recursion” we mean the rules at the end of lecture L11.

Otherwise, it’s up to you.

L14: Allowed constructs

Newly allowed constructs:
none

Previously allowed constructs:
`( ) [ ] + - * / = < > <= >= ;`
`abs acos add1 and append asin atan boolean? char? char=? char<? check-expect check-within cond cons cons? cos define e else empty empty? even? exp expt false first inexact? integer? length list list? log max min not number? odd? or pi quotient rational? remainder rest reverse second sin sqr sqrt string? string=? string<? string->list list->string sub1 symbol? symbol=? tan third true zero?`
`listof Any anyof Bool Char Int Nat Num Rat Str Sym`