Simple and accumulative recursion, which we have been using so far, is a way of deriving code whose form parallels a data definition.

**Generative recursion** is more general: the recursive cases are generated based on the problem to be solved.

The non-recursive cases also do not follow from a data definition.

It is much harder to come up with such solutions to problems.

It often requires deeper analysis and domain-specific knowledge.
> Example revisited: GCD

;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm
;; euclid-gcd: Nat Nat -> Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))])))
> Why does this work?

**Correctness:** Follows from Math 135 proof of the identity.

**Termination:** An application **terminates** if it can be reduced to a value in finite time.

All of our functions so far have terminated. But why?

For a non-recursive function, it is easy to argue that it terminates, assuming all applications inside it do.

It is not clear what to do for recursive functions.
Why did our functions using simple recursion terminate?

A simple recursive function always makes recursive applications on smaller instances, whose size is bounded below by the base case (e.g. the empty list).

We can thus bound the **depth of recursion** (the number of applications of the function before arriving at a base case).

As a result, the evaluation cannot go on forever.
(define (sum-list lst)
  (cond [[(empty? lst) 0]
            [else (+ (first lst) (sum-list (rest lst)))]]))

(sum-list (list 3 6 5 4)) ;; 1
⇒ (+ 3 (sum-list (list 6 5 4))) ;; 2
⇒ (+ 3 (+ 6 (sum-list (list 5 4)))) ;; 3
⇒ (+ 3 (+ 6 (+ 5 (sum-list (list 4))))) ;; 4
⇒ (+ 3 (+ 6 (+ 5 (+ 4 (sum-list (list )))))) ;; arrived at base case
⇒ (+ 3 (+ 6 (+ 5 (+ 4 0)))) ⇒ ... ⇒ 18

The depth of recursion of any application of sum-list is equal to the length of the list to which it is applied.

For generatively recursive functions, we need to make a similar argument.
In the case of `euclid-gcd`, our measure of progress is the size of the second argument.

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

After that, the second argument always gets smaller in the recursive application (since $m > n \mod m$), but it is bounded below by 0.

Thus any application of `euclid-gcd` has a depth of recursion bounded by the second argument.

In fact, it is always much faster than this.
> Termination is sometimes hard

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

The Collatz Conjecture is a decades-old open research problem to discover whether or not (collatz n) terminates for all values of n.
We can see better what collatz is doing by producing a list.

;; (collatz-list n) produces the list of the intermediate
;; results calculated by the collatz function.
;; collatz-list: Nat -> (listof Nat)
;; requires: n >= 1
(check-expect (collatz-list 1) (list 1))
(check-expect (collatz-list 5) (list 5 16 8 4 2 1))

(define (collatz-list n)
  (cons n (cond
             [ (= n 1) empty]
             [(even? n) (collatz-list (/ n 2))]
             [else (collatz-list (+ 1 (* 3 n)))])))
From Calculus you may know that an important way to calculate logarithms is to use Taylor series. In particular, the log base $e$ of a number $0 < x \leq 2$ is given by:

$$\ln x = \sum_{k=1}^{\infty} -\frac{(1 - x)^k}{k}$$

We can approximate this sum with $k$ steps. $k = 20$ gives reasonable accuracy.

\[
\text{(define (ln-small x k) (cond [ (= k 0) #i0] [else (- (ln-small x (- k 1)) (/ (expt (- 1 x) k) k))]))}
\]

Using \text{(ln-small x 20)} in a base case, write a function \text{(ln x)} that calculates the log base $e$ of any positive Num.

What kind of recursion does \text{ln-small} use? Why? What about \text{ln}?
The Quicksort algorithm is an example of **divide and conquer**:

- divide a problem into smaller subproblems;
- recursively solve each one;
- combine the solutions to solve the original problem.

Quicksort sorts a list of numbers into non-decreasing order by first choosing a **pivot** element from the list.

The subproblems consist of the elements less than the pivot, and those greater than the pivot.
If the list is \((\text{list } 9 \ 4 \ 15 \ 2 \ 12 \ 20)\), and the pivot is 9, then the subproblems are \((\text{list } 4 \ 2)\) and \((\text{list } 15 \ 12 \ 20)\).

Recursively sorting the two subproblem lists gives \((\text{list } 2 \ 4)\) and \((\text{list } 12 \ 15 \ 20)\).

It is now simple to combine them with the pivot to give the answer.

\[(\text{append } (\text{list } 2 \ 4) (\text{list } 9) (\text{list } 12 \ 15 \ 20)) \Rightarrow (\text{list } 2 \ 4 \ 9 \ 12 \ 15 \ 20)\]
The easiest pivot to select from a list \texttt{lon} is \texttt{(first lon)}.

A function which tests whether another item is less than the pivot is

\texttt{(lambda (x) (< x (first lon)))}.

The first subproblem is then \texttt{(filter (lambda (x) (< x (first lon))) lon)}

A similar expression will find the second subproblem
(items greater than the pivot).
(my-quicksort lon) sorts lon in non-decreasing order

(check-expect (my-quicksort (list 5 3 9)) (list 3 5 9))

;; my-quicksort: (listof Num) -> (listof Num)
(define (my-quicksort lon)
  (cond [(empty? lon) empty]
        [else (local [(define pivot (first lon))]
                    (define less (filter (lambda (x) (< x pivot)) (rest lon)))
                    (define greater (filter (lambda (x) (>= x pivot)) (rest lon)))
                    (append (my-quicksort less) (list pivot) (my-quicksort greater))))))
Termination of quicksort follows from the fact that both subproblems have fewer elements than the original list (since neither contains the pivot).

Thus the depth of recursion of an application of my-quicksort is bounded above by the number of elements in the argument list.

This would not have been true if we had mistakenly written

\[
\text{filter (lambda (x) (>= x pivot)) lon)
\]

instead of the correct

\[
\text{filter (lambda (x) (>= x pivot)) (rest lon)).}
\]
In the teaching languages, the built-in function `quicksort` consumes two arguments, a list and a comparison function.

`(quicksort (list 1 5 2 4 3) <) ⇒ (list 1 2 3 4 5)`
`(quicksort (list 1 5 2 4 3) >) ⇒ (list 5 4 3 2 1)`

`(quicksort (list "chili powder" "anise" "basil") string<?) ⇒ (list "anise" "basil" "chili powder")`
Intuitively, quicksort works best when the two recursive function applications are on arguments about the same size.

When one recursive function application is always on an empty list (as is the case when quicksort is applied to an already-sorted list), the pattern of recursion is similar to the worst case of insertion sort, and the number of steps is roughly proportional to the square of the length of the list.

We will go into more detail on efficiency considerations in CS 136.
The design recipe becomes much more vague when we move away from data-directed design.

The purpose statement remains unchanged, but additional documentation is often required to describe how the function works.

Examples need to illustrate the workings of the algorithm.

We cannot apply a template, since there is no data definition.

For divide and conquer algorithms, there are typically tests for the easy cases that don’t require recursion, followed by the formulation and recursive solution of subproblems, and then combination of the solutions.
Traditionally, the character set used in computers has included not only alphanumerics characters and punctuation, but “control” characters as well.

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

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" on one line and "cd" on the next line.
Consider converting a string such as "one\ntwo\nthree" into a list of strings, (list "one" "two" "three"), one for each line.

The solution will start with an application of string->list. That’s the only way we’ve studied of working with individual characters in a string.

This problem can be solved using simple recursion on the resulting list of characters – but it’s hard. The “simple” recursion gets bogged down in a lot of little details.

In this case a generative solution is easier.
The generative idea

Instead of thinking of the list of characters as a list of characters, think of it as a list of lines:

```
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.
;; (first-line loc) produces longest newline-free prefix of loc
;; first-line: (listof Char) -> (listof Char)

(define (first-line loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) empty]
        [else (cons (first loc) (first-line (rest loc)))]))

(check-expect (first-line empty) empty)
(check-expect (first-line (list #\a #\newline)) (list #\a))
(check-expect (first-line (string->list "abc\n\ndef")) (list #\a #\b #\c))
;; (rest-of-lines loc) produces loc with everything up to
;; and including the first newline removed
;; rest-of-lines: (listof Char) -> (listof Char)

(define (rest-of-lines loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) (rest loc)]
        [else (rest-of-lines (rest loc))])))

(check-expect (rest-of-lines empty) empty)
(check-expect (rest-of-lines (list #\a #\newline)) empty)
(check-expect (rest-of-lines (list #\a #\newline #\b)) (list #\b))
We can create a “list of lines template” using these helpers.

(define (loc->lol loc)
  (local [(define fline (first-line loc))
           (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) empty]
          [else (... fline ... (loc->lol rlines) ...)]))
This looks a lot like the template for functions consuming a \((\text{listof } X)\). That was simple recursion. Is this also simple recursion?

No, this is \textbf{not} simple recursion. Why not?
> loc->lol

;; loc->lol: (listof Char) -> (listof Str)
(check-expect (loc->lol (string->list "abc\ndef")) (list "abc" "def"))
(check-expect (loc->lol (string->list "")) empty)
(check-expect (loc->lol (string->list "\ndef")) (list "" "def"))

(define (loc->lol loc)
  (local [[(define fline (first-line loc))
            (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) empty]
      [else (cons (list->string fline)
                 (loc->lol rlines))]]))
Why is this generative recursion?

loc->lol can be rewritten as

```
(define (loc->lol loc)
  (cond [(empty? loc) empty]
        [else (cons (list->string (first-line loc))
                     (loc->lol (rest-of-lines loc)))]))
```

The recursive call to loc->lol is not using the data definition for a list of characters. 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 \texttt{(listof Char)}, our actual argument.

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

With generative recursion we needed that “aha” that transformed the problem into a list of lines. That “aha” is often difficult to see.

Was it worth it? Consider the solution using “simple” recursion two slides from now.
Generative recursion 3/3

When we use generative recursion, we need to be careful about termination.

Why should we believe that **string->lines** always terminates?

All of the helper functions use simple recursion except for **loc->lol**, so we focus on it.

Each recursive call is applied to (rest-of-lines loc) where loc is non-empty. But rest-of-lines produces either empty (which leads directly to termination in loc->lol) or a list of characters that is *at least* one character shorter. Therefore, the length of the argument to loc->lol is always decreasing until it becomes empty and the function terminates.
Using simple recursion

```
(define (list->lines loc)
  (cond
    [(empty? loc) (list empty)]
    [(and (empty? (rest loc)) (char=? #\newline (first loc)))
     (list empty)]
    [else
     (local [(define r (list->lines (rest loc)))]
       (cond
         [(char=? #\newline (first loc)) (cons empty r)]
         [else (cons (cons (first loc) (first r)) (rest r)))))])))
```
Goals of this module

- You should understand the idea of generative recursion, why termination is not assured, and how a quantitative notion of a measure of progress towards a solution can be used to justify that such a function will return a result.
- You should understand the examples given.
In this module we added the following to our toolbox:

These are the functions and special forms currently in our toolbox:

Exercise

The **McCarthy 91 function** is a function classically used in the study of formal verification. It is defined as follows:

\[
M(n) = \begin{cases} 
  n - 10, & \text{if } n > 100 \\
  M(M(n + 11)), & \text{if } n \leq 100
\end{cases}
\]

*McCarthy 91* seems to be similar to Collatz in that sometimes the argument to the next application is larger and sometimes it is smaller. Does it always terminate?

Write a implementation \((\text{mc91 } n)\) of this function in Racket.

Manually evaluate \((\text{mc91 } n)\) for various values of \(n\). What happens?

Can you predict what would happen for any value of \(n\) without actually running the function? Are you confident that *McCarthy 91* always terminates (or not)?
A positive natural number can be written uniquely as a product of prime factors. We call this *prime factor decomposition*. E.g. $24 = 2^3 \cdot 3$, and $42 = 2 \cdot 3 \cdot 7$. Instead of storing exponents, we will just list the prime factors, with repetition. So $24 = 2^3 \cdot 3 = 2 \cdot 2 \cdot 2 \cdot 3$ will be represented as \((\text{list } 2 \ 2 \ 2 \ 3)\), and $42 = 2 \cdot 3 \cdot 7$ will be represented as \((\text{list } 2 \ 3 \ 7)\).

Write a function \(\text{pfd } n\) that consumes a positive \text{Nat} and returns the prime factor decomposition of \(n\).

**Hint:** make \(\text{pfd}\) be a wrapper around a function with a parameter that counts up.

\[
\begin{align*}
\text{(pfd-from 2 42)} & \Rightarrow (\text{cons } 2 (\text{pfd-from 2 21})) \\
& \Rightarrow (\text{cons } 2 (\text{pfd-from 3 21})) \\
& \Rightarrow (\text{cons } 2 (\text{cons } 3 (\text{pfd 3 7}))) \\
& \Rightarrow \ldots \\
& \Rightarrow (\text{cons } 2 (\text{cons } 3 (\text{pfd 7 7})))
\end{align*}
\]