09: Patterns of Recursion

Simple vs. general recursion

All of the recursion we have done to date has followed a pattern we call **simple recursion**.

The templates we have been using have been derived from a data definition and specify the form of the recursive application.

We will now learn to use a new pattern of recursion, **accumulative recursion**, and learn to recognize **mutual recursion** and **generative recursion**.

For the next several lecture modules we will use simple recursion and accumulative recursion. We will avoid mutual recursion and generative recursion until later in the course.

Simple recursion

Recall from Module 06:

In **simple recursion**, every argument in a recursive function application (or applications, if there are more than one) are either:

- unchanged, or
- *one step* closer to a base case, using the inverse of the function in the data definition.

```
;; (max-list-v1 lon) produces the maximum element of lon
(check-expect (max-list-v1 (list 6 2 3 7 1)) 7)

;; max-list-v1: (listof Num) → Num
;; Requires: lon is nonempty
(define (max-list-v1 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [else (max (first lon) (max-list-v1 (rest lon)))]))
```

"In-lining" max:

```
(define (max-list-v2 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else  (max-list-v2 (rest lon))]))
```

There may be two recursive applications of max-list-v2.

The code for max-list-v2 is correct.

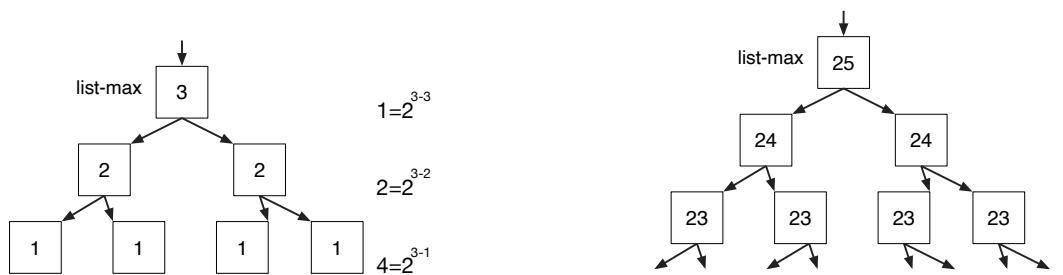But computing (max-list-v2 (countup-to 1 25)) is very slow.

Why?

The initial application is on a list of length 25.

There are two recursive applications on the rest of this list, which is of length 24.

Each of those makes two recursive applications.

max-list can make up to $2^n - 1$ recursive applications on a list of length n.

We informally call this **exponential blowup**.

We can take the number of recursive applications as a rough measure of a function's efficiency. `max-list-v2` can take up to $2^n - 1$ recursive applications.

`length` makes n recursive applications on a list of length n.

`length` is clearly more efficient than `max-list-v2`.

We say that `length`'s efficiency is proportional to n and `max-list-v2`'s efficiency is proportional to 2^n. We express the former as $O(n)$ and the later as $O(2^n)$.

There are "families" of algorithms with similar efficiencies. Examples, from most efficient to least:

"Big-O"	Example
$O(1)$	no recursive calls; `tax-payable` [M04]
$O(\log_2 n)$	divide in half, work on one half; `binary-search` on a *balanced* tree [M10]
$O(n)$	one recursive application for each item; `length`, `max-list-v1` [M06,09]
$O(n \log_2 n)$	divide in half, work on both halves; `mergesort` [M08]
$O(n^2)$	an $O(n)$ application for each item; `insertion-sort` [M06]
$O(2^n)$	two recursive applications for each item; `max-list-v2` [M09]

Much more about "Big-O" notation and efficiency in later courses.

Fast $O(n)$

```
(define (max-list-v1 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [else (max (first lon) (max-list-v1 (rest lon)))]))
```

Slow $O(2^n)$

```
(define (max-list-v2 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else  (max-list-v2 (rest lon))]))
```

A human approach

Humans don't seem to use either of the two versions of `max-list` shown earlier.

Instead, we tend to find the maximum of a list of numbers by scanning it, remembering the largest value seen so far. When we see a value that's larger than the largest seen so far, we remember the new value – until we see one that is still larger. When we get to the end of the list, the largest value seen so far is the largest value in the list.

Accumulative recursion

Computationally, we can pass down that largest value seen so far as a parameter called an **accumulator**.

This parameter accumulates the result of prior computation, and is used to compute the final answer that is produced in the base case.

This approach results in the code on the next slide.

> max-list/acc

```
(define (max-list-v3 lon)
  (max-list/acc (rest lon) (first lon)))

;; (max-list/acc lon max-so-far) produces the largest
;;     of the maximum element of lon and max-so-far

;; max-list/acc: (listof Num) Num → Num
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))
```

Now even `(max-list-v3 (countup-to 1 200000))` is fast.

```
(max-list-v3 (list 1 2 3 9 5))
⇒ (max-list/acc (list 2 3 9 5) 1)
⇒ (max-list/acc (list 3 9 5) 2)
⇒ (max-list/acc (list 9 5) 3)
⇒ (max-list/acc (list 5) 9)
⇒ (max-list/acc (list ) 9)
⇒ 9
```

This technique is known as **accumulative recursion**.

It is more difficult to develop and reason about such code, which is why simple recursion is preferable if it is appropriate.

- All arguments to recursive function applications are:
    - unchanged, or
    - one step closer to a base case in the data definition, or
    - a partial answer (passed in an *accumulator*).
- The value(s) in the accumulator(s) are used in one or more base cases.
- The accumulatively recursive function usually has a wrapper function that sets the initial value of the accumulator(s).

Using simple recursion:

```
;; (my-reverse lst) reverses lst using simple recursion
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst)
  (cond
    [(empty? lst) empty]
    [else (append (my-reverse (rest lst))
                  (list (first lst)))]]))
```

Intuitively, append does too much work in repeatedly moving over the produced list to add one element at the end.

This has the same worst-case behaviour as insertion sort, $O(n^2)$.

```
;; (my-reverse lst) reverses lst
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst)
  (my-rev/acc lst empty))

;; (my-rev/acc lst acc) produces the reverse of [lst] followed by [acc].
(check-expect (my-rev/acc (list 3 4) (list 2 1)) (list 4 3 2 1))
;; my-rev/acc: (listof X) (listof X) -> (listof X)
(define (my-rev/acc lst acc)
  (cond [(empty? lst) acc]
        [else (my-rev/acc (rest lst)  ; transfer first element to accumulator
                          (cons (first lst) acc))]]))
```

This is $O(n)$.

```
(my-reverse (list 1 2 3 4 5))
⇒ (my-rev/acc (list 1 2 3 4 5) empty)
⇒ (my-rev/acc (list 2 3 4 5) (cons 1 empty))
⇒ (my-rev/acc (list 3 4 5) (cons 2 (list 1)))
⇒ (my-rev/acc (list 4 5) (cons 3 (list 2 1)))
⇒ (my-rev/acc (list 5) (cons 4 (list 3 2 1)))
⇒ (my-rev/acc (list ) (cons 5 (list 4 3 2 1)))
⇒ (list 5 4 3 2 1)
```

The *n*th Fibonacci number is the sum of the two previous Fibonacci numbers:

$$f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}, n \geq 2$$

This can be implemented directly using simple recursion, as follows:

```
(define (fib n)
  (cond [(< n 2) n]
        [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

This works: `(fib 6)` $\Rightarrow$ 8, `(fib 25)` $\Rightarrow$ 75025. But `(fib 50)` takes days!

It suffers from exponential blowup.

As it turns out, not 2^n, but ϕ^n, where $\phi = \frac{1+\sqrt{5}}{2}$ is the Golden Ratio.

Using an accumulator avoids the exponential blowup. How can we do that?

Ex. 1

Write a function `(extend-fib n lst)` that consumes a `Nat` and a `(listof Nat)`. Given `lst`, a list containing at least 2 Fibonacci values in non-increasing order, it returns a list containing `n` more Fibonacci values.

Ex. 2

Write a function `(fiba n)` that is a wrapper for `extend-fib`, and produces the *n*th Fibonacci number.

```
(check-expect (fib-a 2) (fib-b 2))
(check-expect (fib-a 5) (fib-b 5))
(check-expect (fib-a 20) (fib-b 20))
```

Ex. 3

Given a `(listof Num)`, use accumulative recursion to write `mean`, which produces the average (mean) of the list.

Hint

- `mean` will be a wrapper function.
- How many accumulators do you need?

Mutual recursion

Mutual recursion occurs when two or more functions apply each other: `f` applies `g` and `g` applies `f`.

```
;; A two-player game
(define (game state)
  (a-turn state))

(define (a-turn state)
  (cond [(a-won? state) 'A-WON]
        [else (b-turn (strategy-a state))]]))

(define (b-turn state)
  (cond [(b-won? state) 'B-WON]
        [else (a-turn (strategy-b state))]]))
```

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

Generative Recursion: GCD

In Math 135, you learn that the Euclidean algorithm for Greatest Common Divisor (GCD) can be derived from the following identity for $m > 0$:

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

We also have $gcd(n, 0) = n$.

We can turn this reasoning directly into a Racket function.

> euclid-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))]]))
```

This function does not use simple, mutual or accumulative recursion.

> Generative recursion

The arguments in the recursive application were **generated** by doing a computation on `m` and `n`.

The function `euclid-gcd` uses **generative recursion**.

Once again, functions using generative recursion are easier to get wrong, harder to debug, and harder to reason about.

We will return to generative recursion in a later lecture module. Avoid generative recursion until then.

Simple vs. accumulative vs. mutual vs. generative recursion

In **simple recursion**, all arguments to the recursive function application (or applications, if there are more than one) are either unchanged, or *one step* closer to a base case in the data definition.

In **accumulative recursion**, parameters are as above, plus parameters containing partial answers used in the base case.

In **mutual recursion**, two or more functions call each other. Parameters usually behave as in simple recursion, but that is not required.

In **generative recursion**, parameters are freely calculated at each step.

Goals of this module

- You should be able to recognize uses of simple recursion, accumulative recursion, mutual recursion and generative recursion.
- You should be able to write functions using simple and accumulative recursion.
- You should know that some functions are much more efficient than others, that efficiency is expressed with "Big-O" notation, and that you'll learn more about this in future courses.
- You should be able to identify and avoid "exponential blowup".

The following functions and special forms have been introduced in this module:

```
reverse
```

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - ... / < <= = > >= abs add1 and append boolean? ceiling char-alphabetic?
char-downcase char-lower-case? char-numeric? char-upcase char-upper-case?
char-whitespace? char<=? char<? char=? char>=? char>? char? check-error check-expect
check-within cond cons cons? cos define define-struct define/trace e eighth else
empty? equal? error even? exp expt fifth first floor fourth integer? length list
list->string list? log max min modulo negative? not number->string number? odd? or pi
positive? quotient remainder rest reverse round second seventh sgn sin sixth sqr sqrt
string->list string-append string-downcase string-length string-lower-case?
string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=?
string>? string? sub1 substring symbol=? symbol? tan third zero?
```