17: Generative Recursion

What is generative recursion?

Simple, accumulative, and mutual recursion, which we have been using so far, are ways 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
(check-expect (euclid-gcd 10 15) 5)
(check-expect (euclid-gcd 13 15) 1)

;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

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.

Termination of 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.

> Depth of recursion example

```
(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 $m > n \bmod m$), but it is bounded below by 0.

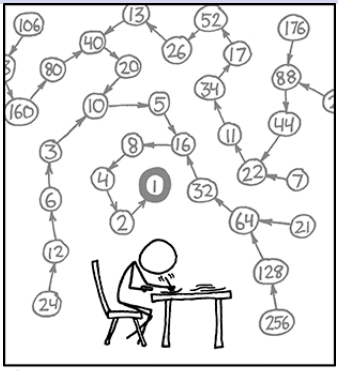
Thus any application of `euclid-gcd` has a depth of recursion bounded by the second argument.

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`.



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.

https://xkcd.com/710/

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.
(check-expect (collatz-list 1) (list 1))
(check-expect (collatz-list 5) (list 5 16 8 4 2 1))

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

Ex. 1

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.

```
(define (ln-small x k) (cond [(= k 0) #i0]
                             [else (- (ln-small x (- k 1))
                                      (/ (expt (- 1 x) k) k))]))
```

Using `(ln-small x 20)` in a base case, write a function `(ln x)` that calculates the log base e of any positive `Num`.
What kind of recursion does `ln-small` use? Why? What about `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 `(list 9 4 15 2 12 20)`, and the pivot is `9`, then the subproblems are `(list 4 2)` and `(list 15 12 20)`.

Recursively sorting the two subproblem lists gives `(list 2 4)` and `(list 12 15 20)`.

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

`(append (list 2 4) (list 9) (list 12 15 20))` $\Rightarrow$ `(list 2 4 9 12 15 20)`

The easiest pivot to select from a list `lon` is `(first lon)`.

A function which tests whether another item is less than the pivot is `(lambda (x) (< x (first lon)))`.

The first subproblem is then `(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

```
(filter (lambda (x) (>= x pivot)) lon)
```

instead of the correct

```
(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.

Modifying the design recipe

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.

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.

Ex. 2

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 `(mc91 n)` of this function in Racket.

Manually evaluate `(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)?

Ex. 3

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 `(list 2 2 2 3)`, and $42 = 2 \cdot 3 \cdot 7$ will be represented as `(list 2 3 7)`.
Write a function `(pfd n)` that consumes a positive `Nat` and returns the prime factor decomposition of `n`.
Hint: make `pfd` be a wrapper around a function with a parameter that counts up.

```
(pfd-from 2 42)
⇒ (cons 2 (pfd-from 2 21))
⇒ (cons 2 (pfd-from 3 21))
⇒ (cons 2 (cons 3 (pfd 3 7))) ⇒ ... ⇒ (cons 2 (cons 3 (pfd 7 7)))
```

Summary: built-in functions

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

```
quicksort
```

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? build-list 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 filter first floor foldl foldr fourth
    integer? lambda length list list->string list? local log map max min modulo negative?
    not number->string number? odd? or pi positive? quicksort 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?
```