11: 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.

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

We'll look at three examples:

- mutual recursion on a natural number.
- mutual recursion on a list.
- mutual recursion on a tree.

Mutual recursion on Nats and Lists is rare. We're showing examples to ease into the pattern. Mutual recursion on general trees is common and a very useful tool.

Mutual recursion on a `Nat`

Consider the following observations about even and odd natural numbers:

- a number is even if it is one more than an odd number
- a number is odd if it is one more than an even number
- 0 is even (and therefore is not odd)

We can use these observations to implement `(is-even? n)`, which produces `true` if *n* is even and `false` otherwise.

```
;; (is-even? n) produces true if n is even and false otherwise
;; Examples:
(check-expect (is-even? 4) true)
(check-expect (is-even? 5) false)

;; is-even?: Nat -> Bool
(define (is-even? n)
  (cond [(= 0 n) true]
        [else (is-odd? (sub1 n))]))

;; is-odd?: Nat -> Bool
(define (is-odd? n)
  (cond [(= 0 n) false]
        [else (is-even? (sub1 n))]))
```

Ex. 1

Alice and Bob are playing the game of "pebbles". There are *n* pebbles on a pile. Beginning with Alice, they alternate turns taking either one or two pebbles from the pile. The person who takes the last pebble wins.

Alice's strategy is to always take a single pebble. Bob's strategy is to take 1 pebble if there are an odd number and 2 pebbles if there is an even number.

Write `(pebbles n)` where `n` is the number of pebbles in play. The easiest approach is to use mutual recursion with one function implementing Alice's strategy and another function implementing Bob's strategy.

```
(check-expect (pebbles 1) "Alice wins")
(check-expect (pebbles 2) "Bob wins")
(check-expect (pebbles 3) "Bob wins")
```

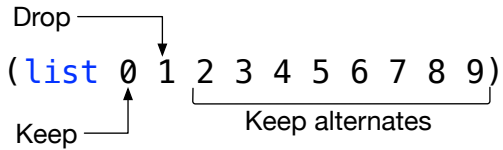
Mutual recursion on a list

We want to divide a list into two equal or nearly equally long lists.

If we had a function, `keep-alternates`, that keeps every other element of a list, we could apply it to the entire list and the rest of the list to divide it in two:

```
(check-expect (keep-alternates (list 0 1 2 3 4 5 6)) (list 0 2 4 6))
(check-expect (keep-alternates (rest (list 0 1 2 3 4 5 6))) (list 1 3 5))
```

Thinking about `keep-alternates`:

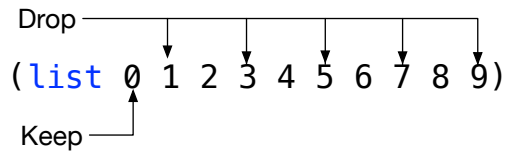


This suggests processing the list in two-element chunks. But that gets slightly messy with multiple base cases.

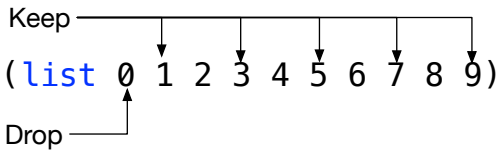
Another approach is that `keep-alternates` should keep the first element of the list and **drop** the alternating elements of the rest of the list.

How do we drop alternates? We keep the alternates from the rest of the list.

`keep-alternates`



`drop-alternates`



```
;; (keep-alternates lst) keeps the first element of lst and drops
;; alternating elements from the rest.
;; Examples:
(check-expect (keep-alternates (list 0 1 2 3 4 5 6)) (list 0 2 4 6))
(check-expect (keep-alternates (rest (list 0 1 2 3 4 5 6))) (list 1 3 5))
(check-expect (keep-alternates (list 'a 'b 'c 'd)) (list 'a 'c))

;; keep-alternates: (listof Any) -> (listof Any)
(define (keep-alternates lst)
  (cond [(empty? lst) empty]
        [else (cons (first lst) (drop-alternates (rest lst)))]))

;; (drop-alternates lst) drops the first element of lst and keeps
;; alternating elements from the rest.
(define (drop-alternates lst)
  (cond [(empty? lst) empty]
        [else (keep-alternates (rest lst))]))
```

Mutual recursion on a tree

Take a close look at our function for evaluating binary expressions:

```
;; eval: BinExp → Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))


(define (eval-binode node)
  (cond [(symbol=? '* (binode-op node))
         (* (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '/ (binode-op node))
         (/ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '+ (binode-op node))
         (+ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '- (binode-op node))
         (- (eval (binode-left node)) (eval (binode-right node)))]))
```

Self-referential data definitions

Mutual recursion arises naturally when the data definition references itself. Notice how a `BinExp` references a `BINode` and a `BINode` references a `BinExp`:

```
(define-struct binode (op left right))
;; A Binary arithmetic expression Internal Node (BINode)
;;     is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

;; A binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode
```

The mutual recursion will appear naturally in the templates, provided each part of the data definition is turned into a template.

```
;; binode-template: BINode → Any
(define (binode-template node)
  (... (binode-op node)
       (binexp-template (binode-left node))
       (binexp-template (binode-right node))))

;; binexp-template: BinExp → Any
(define (binexp-template ex)
  (cond [(number? ex) (... ex)]
        [(binode? ex) (binode-template ex)]))
```

In our Nat and List examples, the two functions were very similar to each other. Here, we see one handling a `BinExp` and the other handling a `BINode`. In the next module a common pattern will be for one function to handle a tree and the other to handle a list of (sub)trees.

Goals of this module

- You should understand the idea of mutual recursion for examples given in lecture.
- You should be able to develop templates from mutually recursive data definitions, and to write functions using the templates.

Summary: built-in functions

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

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?
```