Chapter 16

Lists and List-Based Recursion

Problem 16.1

Define a function, called `all-numbers?`*, that satisfies the following contract:*

```
;;  ALL-NUMBERS?
;; --------------------------------------------------------------
;;  INPUT:   LISTY, a list
;;  OUTPUT:  #t (or something that counts as true) if all the
;;           items in LISTY are numbers; #f otherwise
```

Here are some examples:

```
> (all-numbers? '(1 2 3 4))
#t
> (all-numbers? '(1 2 a b #t c 4))
#f
```

Problem 16.2

Define a function, called `index-of`*, that satisfies the following contract:*

```
;;  INDEX-OF
;; -------------------------------------------------------------------
;;  INPUTS:  ITEM, anything
;;           LISTY, a list of stuff
;;  OUTPUT:  The index of the *first* occurrence of ITEM in LISTY
;;           or #f if ITEM doesn't appear in LISTY.
;;  NOTE:    Indices start at 0.
```

Here are some examples:

```
> (index-of 'a '(a b c d e a a b))
0
> (index-of 'c '(a b c d e c e f))
2
> (index-of 'g '(a b c d e f))
#f
```

Hint: Use the built-in `eq?` *function to test the equality of two pieces of data.*

Problem 16.3

Define a function, called `first-symbol`*, that satisfies the following contract:*

```
;;  FIRST-SYMBOL
;; ---------------------------------------------------
;;  INPUT:   LISTY, any list
;;  OUTPUT:  The first symbol that appears in LISTY;
;;           or #f, if no symbols appear in LISTY.
```

Here are some examples:

```
> (first-symbol '(3 #t x y #f))
x
> (first-symbol '(1 2 3))
#f
```

Hint: Use the built-in type-checker predicate, `symbol?`*.*

Problem 16.4

Define a function, called `has-symbol?`*, that satisfies the following contract:*

```
;;  HAS-SYMBOL?
;; ----------------------------------------------------
;;  INPUT:   LISTY, any list
;;  OUTPUT:  #t if LISTY contains at least one symbol
```

Here are some examples:

```
> (has-symbol? '(1 2 3))
#f
> (has-symbol? '(1 2 3 4 x 5 6))
#t
```

(Optional) Define a version of the `has-symbol?` *function that uses some combination of* `and`*,* `or` *and* `not`*, instead of* `if` *or* `cond`*. In that case, the body of the predicate should specify the condition under which this function should return true.*

Problem 16.5

Define a function, called `max-elt`*, that satisfies the following contract:*

```
;;  MAX-ELT
;; ----------------------------
;;  INPUT:   LISTY, a non-empty list of numbers
;;  OUTPUT:  The MAXIMUM number in LISTY
```

Here are some examples:

```
> (max-elt '(6 7 71 3 4))
71
```

```
    > (max-elt '(8))
    8
```

Hint: Notice that the base case should be a list that contains exactly one element. What is the easiest way to test for that? (Warning! Do not *use the* `length` *function for that purpose! Think about why!)*

Problem 16.6

Recall the built-in predicate, `even?`*. It takes a number as its only input and returns* `#t` *if that number is even; otherwise it returns* `#f`*. Now, if some number N is even (i.e., if* `(even? N)` $\Rightarrow$ `#t`*), then we say that N "satisfies" the* `even?` *predicate (i.e., makes it return* `#t` *as its output). So, for example, the number 6 satisfies the* `even?` *predicate, but does* not *satisfy the* `odd?` *predicate. Similarly, the number 7 satisfies the* `odd?` *predicate, but not the* `even?` *predicate.*
For this problem, define a function, called `contains-a-satisfier?`*, that satisfies the following contract:*

```
    ;;  CONTAINS-A-SATISFIER?
    ;; -------------------------------------------------------------
    ;;  INPUTS:  PRED, a predicate (e.g., EVEN?) that takes a single
    ;;            input
    ;;           LISTY, a list of suitable inputs for PRED
    ;;  OUTPUT:  #t if LISTY contains at least one element that
    ;;           "satisfies" PRED; #f otherwise.
```

Here are some examples:

```
    > (contains-a-satisfier? even? '(1 2 3 4 5))
    #t
    > (contains-a-satisfier? even? '(1 3 5 7 9))
    #f
```

The first example evaluates to `#t` *because the input list contains the number 2, which is even. The second example evaluates to* `#f` *because the input list does not contain any even numbers.*

- ⋆ *Note that you can make lots of* `tester` *expressions using any of the type-checker predicates that we have seen in class (e.g.,* `number?`*,* `symbol?`*,* `null?`*, etc.), as well as:* `even?` *and* `odd?`*. However, predicates such as <, >, <=, =, etc., which expect two inputs, would not work here.*
- ⋆ *If the input list is non-empty, check what happens when* `PRED` *is applied to* `(FIRST LISTY)`*, and react accordingly.*

Problem 16.7

Define a function, called `n-elt-list?`*, that satisfies the following contract:*

```
    ;;  N-ELT-LIST?
    ;; ------------------------------------------------------
    ;;  INPUTS:  N, a non-negative integer
    ;;           LISTY, a list
    ;;  OUTPUT:  #t if LISTY contains exactly N elements;
    ;;           #f otherwise.
```

Here are some examples of its use:

```
> (n-elt-list? 5 '(a b c d e))
#t
> (n-elt-list? 5 '(a b c))
#f
> (n-elt-list? 5 '(a b c d e f g))
#f
```

Implement two versions of this function: one that uses `cond`*, and one that uses only* `and`*,* `or` *and* `not`*.*

- ⋆ *Do not use the* `length` *function! If a list contains a billion elements, we don't want the* `length` *function to walk all the way through its one billion elements just to find out whether or not it is a 4-element list!*
- ⋆ *Consider the following four cases:*
  - $n = 0$ *and* `listy` *is empty*
  - $n = 0$ *and* `listy` *is non-empty*
  - $n > 0$ *and* `listy` *is empty*
  - $n > 0$ *and* `listy` *is non-empty*

  For which of these cases can you tell the answer immediately (i.e., which are base cases)?

Problem 16.8: Testing whether a list is sorted

Recall that the `incr?` *predicate, from In-Class Problem 16.2.4, returned* `#t` *if its input list was sorted into non-decreasing order. For this problem, you will define a more general predicate, called* `sorted?`*, that takes an extra input, called* `comparer`*. The* `sorted?` *predicate returns* `#t` *if its input list is sorted according to the* `comparer` *predicate. For example, if the* `comparer` *predicate is the* less-than *function, then the behavior of* `sorted?` *is the same as that of* `incr?`*. However, other choices of the* `comparer` *predicate lead different behavior. Here is the contract for* `sorted?`*, followed by some examples of its desired behavior.*

```
;;  SORTED?
;; -----------------------------------------------------------
;;  INPUTS:  LISTY, a non-empty list of stuff
;;           COMPARER, a predicate that returns #t
;;             if its two inputs are in some desired order
;;  OUTPUT:  #t, if the elements of LISTY are sorted into the
;;             order determined by COMPARER; #f, otherwise.

> (sorted? '(1 2 3 5 8) <)        ⟵ Equivalent to using incr?
#t
> (sorted? '(1 2 3 5 5 8) >)
#f
> (string<=? "beard" "bread")     ⟵ string<=? is built-in; it outputs #t if its
#t                                    two inputs are in alphabetic order.
> (sorted? '("bead" "beard" "bread" "broom") string<=?)
#t
> (sorted? '("bead" "bread" "beard" "broom") string<=?)
#f
```

The examples involving strings and the built-in `string<=?` *predicate illustrate the flexibility of the* `sorted?` *predicate.*

Problem 16.9: Computing *dot products*

Define a function, called `dotty`*, that satisfies the following contract:*

```
;;  DOTTY
;; ------------------------------------------------------------------
;;  INPUT:  LISTY, LISTZ, two lists of numbers, having
;;            the same length
;;  OUTPUT:  The "dot product" of LISTY and LISTZ.  In other
;;    words, if LISTY = (y1 y2 ... yn) and LISTZ = (z1 z2 ... zn),
;;    then the output is the sum:  y1*z1 + y2*z2 + ... + yn*zn.
```

Here are some examples:

```
> (dotty '(5 4 3) '(100 10 1))
543                              ⟵  (5 · 100) + (4 · 10) + (3 · 1)
> (dotty '(2 4) '(9 7))
46                               ⟵  (2 · 9) + (4 · 7)
> (dotty '(1 -2 1) '(2 3 4))
0                                ⟵  (1 · 2) + ((−2) · 3) + (1 · 4)
```

Hint: Even though there are two lists as input, the recursive processing is very similar to other examples we have done, especially since this function assumes that the input lists have the same number of elements.

Problem 16.10

Define a predicate, called `dominates?`*, that satisfies the following contract:*

```
;;  DOMINATES?
;; ----------------------------------------------------------
;;  INPUTS:  LISTY, LISTZ, two lists of numbers having
;;             the same length
;;  OUTPUT:  #t if each element of LISTY is greater than or
;;             equal to the corresponding element of LISTZ
```

Here are some examples:

```
> (dominates? '(10 10 12 15) '(2 5 3 1))
#t
> (dominates? '(10 10 12 15) '(2 5 18 6))
#f
```

Problem 16.11

Define a function, called `first-pair`*, that satisfies the following contract:*

```
;;  FIRST-PAIR
;; -------------------------------------------------------------
;;  INPUT:   LISTY, a list of stuff
;;  OUTPUT:  The first item that appears twice consecutively in
;;           LISTY (as judged by EQ?); otherwise, #f.
```

Here are some examples:

```
> (first-pair '(1 2 3 4 4 5 5 3 3 3))
4
> (first-pair '(a b f d d r c c c a))
d
> (first-pair '(a b c a b c))
#f
```

Hints: Note that a list containing zero or one elements cannot have any consecutive elements. Define a helper function that is the same as `first-pair`*, except that it takes an extra input, called* `prev`*, that keeps track of the most recently seen item.*

Problem 16.12: Checking whether two lists are "equal"

Define a function, called `list-equal?`*, that satisfies the following contract:*

```
;;  LIST-EQUAL?
;; -----------------------------------------------------
;;  INPUTS:  LISTY, LISTZ, any two lists
;;  OUTPUT:  #t if LISTY and LISTZ contain the same
;;    elements, in the same order, where equality of
;;    elements is judged by the EQ? predicate;
;;    #f, otherwise
```

Here are some examples:

```
> (list-equal? '(a b c) '(a b c))
#t
> (list-equal? '(1 2 3 4) (cons 1 (cons 2 (cons 3 (cons 4 ())))))
#t
> (list-equal? '(1 2 3) '(1 2 3 4 5))
#f
```

Hint: Walk through the lists in parallel, checking equality of their corresponding elements, until one or both lists run out of elements.

After you've implemented this function, you may wish to know that there is a built-in function, called `equal?`*, that does* almost *the same thing. As demonstrated below, the* `equal?` *predicate also works on hierarchical lists, whereas* `list-equal?` *does not.*

```
> (equal? '(a b c) '(a b c))
#t
> (equal? '(a (b (c) d)) '(a (b (c) d)))
#t
> (list-equal? '(a (b (c) d)) '(a (b (c) d)))
#f
```

The reason for the difference is that `list-equal?` *uses* `eq?` *to test the equality of corresponding elements, but* `eq?` *is not sophisticated enough to judge equality of* lists. *(Try it.) Hierarchical lists are covered in Section 16.7.*

Problem 16.13: Removing from a list items that have some property

Define a function, called `remove-if`*, that satisfies the following contract:*

```
;;  REMOVE-IF
;; --------------------------------------------------------------
;;  INPUTS:  PRED, a predicate that expects one input
;;           LISTY, a list of elements, each of which is a
;;             suitable input for PRED
;;  OUTPUT:  A list that contains all of the elements of LISTY,
;;             except those for which PRED returns #t.
```

Here are some examples:

```
> (remove-if even? '(1 2 3 4 5 6))
(1 3 5)
> (remove-if odd? '(1 2 3 4 5 6))
(2 4 6)
```

Hint: There can be two recursive cases, one where `(first listy)` *"satisfies"* `pred`*, the other where* `(first listy)` *does not. In the first case, you do not want to include* `(first listy)` *in the answer list; in the second case, you* do *want to include it.*

Problem 16.14: Replacing items in a list

Define a function, called `replace`*, that satisfies the following contract:*

```
;;  REPLACE
;; ----------------------------------
;;  INPUTS:  OLD, anything
;;           NEW, anything
;;           LISTY, a list of stuff
;;  OUTPUT:  A list just like LISTY except that each occurrence of
;;           OLD in LISTY has been replaced by an occurrence of
;;           NEW in the output.  (Equality of two items should
;;           be determined by the EQ? predicate.)
```

Here are some examples:

```
> (replace 'x 'y '(a x b x c x))
(a y b y c y)
> (replace 0 1 '(0 1 1 1 0 0 0 1))
(1 1 1 1 1 1 1 1)
```

Problem 16.15

Define a function, called `every-other-one`*, that satisfies the following contract.*

```
;;  EVERY-OTHER-ONE
;; ---------------------------------------------------------------
;;  INPUT:   LISTY, a list
;;  OUTPUT:  A list containing every other element of LISTY.
;;  Note:  The output list should contain roughly half the
;;    elements of LISTY; and their occurrences in the output list
;;    should be in the same order as their occurrences in LISTY.
```

Here are some examples of its behavior:

```
> (every-other-one '(a b c d e f g))
(a c e g)
> (every-other-one '(a b c d e f))
(a c e)
> (every-other-one '(0 1 0 1 0 1 0 1 0 1))
(0 0 0 0 0)
```

Problem 16.16: Fetching the first N elements of a list

Define a function, called `first-n-elts`*, that satisfies the following contract:*

```
;;  FIRST-N-ELTS
;; ------------------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;           LISTY, a list that contains at least N elements
;;  OUTPUT:  A list containing the first N elements of LISTY,
;;           in the same order as their order in LISTY.
```

Here are some examples of its use:

```
> (first-n-elts 3 '(a b c d e f g))
(a b c)
> (first-n-elts 5 '(a b c d e f g))
(a b c d e)
```

Note: You need not deal with the case where `listy` *has fewer than* `n` *elements.*

Problem 16.17

Define a function called `repeater` *that satisfies the following contract:*

```
;;  REPEATER
;; -----------------------------------------------------------
;;  INPUT:  LISTY, any list
;;  OUTPUT:  A list that contains twice as many elements as
;;           LISTY.  In particular, each element of LISTY
;;           should appear twice consecutively in the output.
```

Here are some more examples:

```
> (repeater '(life is fun))
(life life is is fun fun)
> (repeater '(i went home yesterday))
(i i went went home home yesterday yesterday)
```

Hint: In the recursive case, use `cons` *twice!*

Problem 16.18

Define a function, called `consec-sums`*, that satisfies the following contract:*

```
;;  CONSEC-SUMS
;; ----------------------------------------------------
;;  INPUT:   LISTY, a list of at least two numbers
;;  OUTPUT:  A list containing the sums of consecutive
;;           items from LISTY
```

Here are some examples:

```
> (consec-sums '(1 20 300 4000))
(21 320 4300)
> (consec-sums '(50 40 30 20 10))
(90 70 50 30)
```

Notice that the output list contains one fewer element than the input list.

Problem 16.19: Generating a list of random tosses of a die

Define a function, called `random-tosses`*, that satisfies the following contract:*

```
;;  RANDOM-TOSSES
;; -------------------------------------------------------
;;  INPUTS:  NUM, a non-negative integer
;;  OUTPUT:  A list containing NUM elements, each element
;;           obtained by randomly tossing a 6-sided die.
```

Here are some examples:

```
> (random-tosses 10)
(4 1 4 2 1 6 1 5 5 6)
> (random-tosses 10)
(6 2 2 3 3 6 5 6 3 2)
> (random-tosses 10)
(5 2 1 4 6 3 3 1 2 5)
```

Problem 16.20

Define a version of the `list-down-to-zero-acc` *function from Example 16.4.2 that accumulates the desired list in the wrong order, but then uses the built-in* `reverse` *function to reverse the accumulated list in the base case. Here's the contract:*

```
;;  LIST-DOWN-TO-ZERO-ACC-V2
;; ---------------------------------------------------------
;;  INPUTS:  N, a non-negative integer
;;           ACC, a list accumulator
;;  OUTPUT:  When called with ACC=(), the output
;;    is the list (N N-1 N-2 ... 2 1 0).  More generally,
;;    the output is the "concatenation" of the lists
;;    (N N-1 N-2 ... CURR) and ACC.
```

Here are some examples of the desired behavior:

```
> (list-down-to-zero-acc-v2 5 ())
(5 4 3 2 1 0)
> (list-down-to-zero-acc-v2 3 ())
(3 2 1 0)
```

Then define a wrapper function, `list-down-to-zero-wr-v2`*, that only takes a single input,* `n`*.*

Problem 16.21

Define a function, called `list-from-zero-to-n`*, that satisfies the following contract:*

```
;;  LIST-FROM-ZERO-TO-N
;; ---------------------------------------------
;;  INPUT:   N, a non-negative integer
;;  OUTPUT:  A list of the form (0 1 2 ... N)
```

Here are some examples:

```
> (list-from-zero-to-n 5)
(0 1 2 3 4 5)
> (list-from-zero-to-n 15)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

Use a helper function that accumulates the desired list.

Hint: Recall Example 16.4.2.

Problem 16.22: Concatenating lists using `transfer-all` and `reverse`

Define an alternative implementation of the `conc` *function from In-Class Problem 16.3.2 that lets the* `transfer-all` *and* `reverse` *functions (from Example 16.4.3) do all of the work. For example, one way to concatenate the lists* `(1 2 3)` *and* `(a b c d e)` *is to first reverse* `(1 2 3)` *and then transfer all of its elements onto the front of* `(a b c d e)`.

Problem 16.23: Removing duplicate elements from a list

Define a function that satisfies the following contract:

```
;;  REM-DUPES
;; -----------------------------------------------------
;;  INPUTS:  LISTY, any list
;;  OUTPUT:  A list that contains the same elements as
;;           LISTY, but without any duplicates.
```

The order of the elements in the output list does not really matter, but try to preserve as much of the order of elements in `listy` *as possible. Here are some examples:*

```
> (rem-dupes '(1 1 1 1 1))
(1)
> (rem-dupes '(a b r a c a d a b r a))
(a b r c d)
> (rem-dupes '(1 2 3 1 2 3 1 2 3 4 3 2 1))
(1 2 3 4)
```

Use an accumulator-based helper function, `rem-dupes-acc`*, that satisfies the following contract.*

```
;;  REM-DUPES-ACC
;; -----------------------------------------------------
;;  INPUTS:  LISTY, any list
;;           ACC, a list accumulator
;;  OUTPUT:  When called with ACC=(), the output is a
;;           list that contains the same elements as
;;           LISTY, but without any duplicates.
```

Hint: In the recursive case, use the built-in `member` *function to decide whether or not* `(first listy)` *already appears in the accumulator. Use that information to decide whether or not to accumulate it now.*

Problem 16.24

Suppose you have a list of dice, such as `(6 3 6 2 6)`*. And suppose that in the game you are playing, you are allowed to re-roll some of the dice. If you are trying to get as many sixes as possible, you might want to re-roll the three and the two, but not the sixes. For this problem, you will define a function called* `roll-some` *that would allow you to do this. For the above example, the function call would look like this:*

```
(roll-some '(6 3 6 2 6) '(#f #t #f #t #f))
```

where each `#f` *means "Don't roll that die!"; and each* `#t` *means "Do roll that die!" The contract is given below, followed by some examples.*

```
;;  ROLL-SOME
;; ----------------------------------------------------------
;;  INPUTS:  LIST-O-DICE, a list of numbers, each number
;;             in the range {1,2,3,4,5,6}
;;           LIST-O-BOOLEANS, a list of the same length
;;             as LIST-O-DICE, but containing only #t or #f
;;  OUTPUT:  A list that is the same as LIST-O-DICE, except that
;;    whenever an element of LIST-O-BOOLEANS is #t, then the
;;    corresponding entry of the output list is generated by a
;;    random toss of a 6-sided die.
```

Here are some examples:

```
> (roll-some '(1 2 3 4) '(#f #f #f #f))
(1 2 3 4)
> (roll-some '(0 0 0 0) '(#t #t #t #t))
(6 4 3 5)
> (roll-some '(6 3 6 2 6) '(#f #t #f #t #f))
(6 5 6 1 6)
> (roll-some '(6 3 6 2 6) '(#f #t #f #t #f))
(6 3 6 6 6)
> (roll-some '(6 3 6 2 6) '(#f #t #f #t #f))
(6 2 6 2 6)
```

Problem 16.25: Computing the depth of a hierarchical list

Define a function, called `depth*`*, that computes the maximum* depth *of any item in the given (possibly hierarchical) list. Here is its contract, followed by some examples illustrating the desired behavior.*

```
;;  DEPTH*
;; ---------------------------------------------------------------
;;  INPUT:   HLISTY, a (possibly hierarchical) list
;;  OUTPUT:  The maximum depth of any item in HLISTY

> (depth* '(a b c))
1
> (depth* '(1 (2 (3) 2) 1))
3
```

The first example involves a flat list, each of whose elements is considered to be at depth one. Thus, the maximum depth for that list is one. In the second example, each `1` *occurs at depth one, each* `2` *occurs at depth two, and the* `3` *occurs at depth three. Thus, the maximum depth for that list is three. (Notice that the* `3` *is nested within three sets of matching parentheses.) By convention, the depth of the empty list is taken to be zero.*

Problem 16.26: Replacing items in a hierarchical list

Define a function that satisfies the following contract.

```
;;  REPLACE*
;; ------------------------------------------------------------
;;  INPUT:   OLD, anything
;;           NEW, anything
;;           HLISTY, a (possibly hierarchical) list
;;  OUTPUT:  A list that is the same as HLISTY, except that
;;           each occurrence of OLD in HLISTY (as judged by
;;           EQ?) has been replaced by an occurrence of NEW.
```

Here are some examples:

```
> (replace* 1 'one '(1 2 (1 2 (1 1 (2)) 1)))
(one 2 (one 2 (one one (2)) one))
> (replace* 'x 'ecks '(a ((x) x) b (x (s) x)))
(a ((ecks) ecks) b (ecks (s) ecks))
```

Problem 16.27: A hierarchical version of `is-elt-of`

Define a function that satisfies the following contract.

```
;;  IS-ELT-OF*
;; ------------------------------------------------------------
;;  INPUTS:  ITEM, anything
;;           HLISTY, a (possibly hierarchical) list
;;  OUTPUT:  #t if ITEM appears somewhere within HLISTY (as
;;             judged by EQ?); #f otherwise.

> (is-elt-of* 3 '(1 2 (4 (8 (2 3 9) 6 5))))
#t
> (is-elt-of* 3 '(1 2 (4 (8 (2 0 9) 6 5))))
#f
```

Note: Since this is a predicate, you may wish to define it using some combination of `and`*,* `or` *and* `not`*, instead of using* `cond` *or* `if`*.*

Problem 16.28

Define a function, called `equal?*`*, that satisfies the following contract:*

```
;;  EQUAL?*
;; ------------------------------------------------------------
;;  INPUTS:  HLISTY, HLISTZ, two possibly hierarchical lists
;;  OUTPUT:  #t if HLISTY and HLISTZ contain the same items,
;;    in the same order, at every level of their hierarchies.
```

Here are some examples of its use:

```
> (equal?* '(a b c) '(a b c))
#t
> (equal?* '(a (b (c) d)) '(a (b (c) d)))
#t
> (equal?* '(a (b (c c) e)) '(a (b (c x) d)))
>#f
```

Note that, unlike the `list-equal?` *function seen in Problem 16.12, the* `equal?*` *function does not use the built-in* `eq?` *predicate to test the equality of corresponding items, because the* `eq?` *predicate is unable to confirm the equality of lists in general.*[a] *Instead, the* `equal?*` *function should use a recursive function call to deal with lists that appear as items anywhere within the hierarchy.*

- ⋆ *Be careful! It may be that* `(first hlisty)` *is a list, but* `(first hlistz)` *is not.*

Incidentally, the built-in `equal?` *function is able to correctly determine whether two hierarchical lists have the same contents, at every level of their hierarchies.*

[a] When comparing non-empty lists, the `eq?` predicate only checks whether they start with the *same* cons cell; it doesn't even look at the contents of that cons cell. The differences are illustrated by the following interactions:

```
> (eq? '(a b) '(a b))
#f
> (let ((listy '(a b))) (eq? listy listy))
#t
```

Problem 16.29

Define a function, called `replace-by-depth*`*, that satisfies the following contract:*

```
;;  REPLACE-BY-DEPTH*
;; -------------------------------------------------------
;;  INPUT:   HLISTY, a (possibly hierarchical) list
;;  OUTPUT:  A list that is just like DLISTY, except that
;;   each item in the list has been replaced by a NUMBER
;;   that is equal to the depth of that item in the list.
```

Here are some examples:

```
> (replace-by-depth* '(a (b ((c) d) e) (f)))
(1 (2 ((4) 3) 2) (2))
> (replace-by-depth* '(((x))))
(((3)))
```

Hint: Define a helper function that includes an extra input, `curr-depth`*, that keeps track of the current depth. That way, when you come across an item that needs to be replaced, you can just replace it by* `curr-depth`*. When should the value of* `curr-depth` *be increased?*