Chapter 15
Local Variables, Local Environments

Problem 15.1

This problem has two parts. The first part can be implemented without using let; the second part is best implemented using let.

(a) Define a function called print-n-tosses that satisfies the following contract:

;; PRINT-N-TOSSES
;; ----------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: None
;; SIDE EFFECT: Prints the results of N random tosses of a
;; six-sided die in the Interactions Window.

Here are some examples:

> (print-n-tosses 10)
5 2 6 4 5 6 3 2 3 1
> (print-n-tosses 10)
5 2 6 1 5 6 2 5 5 3
> (print-n-tosses 10)
6 2 6 4 5 3 1 2 5 3

(b) Define a function called print-and-sum-n-tosses that satisfies the following contract.

;; PRINT-AND-SUM-N-TOSSES
;; ----------------------------------------
;; INPUT: N, a non-negative integer
;; OUTPUT: The sum of N random tosses of a six-sided die.
;; SIDE EFFECT: Prints out the tosses along the way.

Here are some sample interactions:

> (print-and-sum-n-tosses 5)
2 2 6 6 2 : 18
> (print-and-sum-n-tosses 5)
3 3 5 3 5 : 19
> (print-and-sum-n-tosses 5)
3 1 5 1 2 : 12
Note that the numbers to the left of the “:” are side-effect printing, whereas the numbers to the right of the “:” are output values.

Hints: Define an accumulator-based, tail-recursive function called `print-and-sum-n-tosses-acc` that takes an accumulator `acc` as an extra input. (You may wish to review Section 14.3.) In the recursive case, store the current toss in a local variable before printing it and making the tail-recursive function call. Afterward, define `print-and-sum-n-tosses` as a wrapper function that calls `print-and-sum-n-tosses-acc` with appropriate inputs. (You may wish to review Section 14.4.)

Problem 15.2

Define a function, called `sum-the-even-tosses`, that satisfies the following contract:

```scheme
;; SUM-THE-EVEN-TOSSES
;; -------------------------------------------------------------
;; INPUTS: N, a non-negative integer
;; OUTPUT: The sum of the even tosses out of N random tosses
;; SIDE EFFECT: Print out the tosses along the way.
```

Here is an example of its use:

```
> (sum-the-even-tosses 5)
3 6 2 1 5
8
> (sum-the-even-tosses 7)
2 4 1 6 6 3 2
20
```

Problem 15.3

Define a function called `num-occurs-in-n-tosses` that satisfies the following contract.

```scheme
;; NUM-OCCURS-IN-N-TOSSES
;; ---------------------------------------------------------------
;; INPUTS: TARGET, an integer from 1 to 6, inclusive
;;         N, a non-negative integer
;; OUTPUT: Reports the number of times the TARGET number showed up when tossing a six-sided die N times.
;; SIDE EFFECT: Prints out the random tosses along the way.
```

Here are some examples of it in action:

```
> (num-occurs-in-n-tosses 3 20)
4 3 3 1 6 3 5 6 4 1 6 5 5 4 3 5 3 3 5 2 ... 6
> (num-occurs-in-n-tosses 3 20)
4 6 5 1 6 5 3 2 3 4 2 4 2 4 4 5 3 6 3 5 ... 4
> (num-occurs-in-n-tosses 3 20)
5 4 5 1 4 2 4 3 5 3 1 1 2 5 5 1 6 4 2 3 ... 3
```

Notice that the numbers to the left of the dot-dot-dots are side-effect printing, whereas the numbers to the right of the dot-dot-dots are output values.
**Hint:** In the recursive case, use a let special form to store the value of the toss of a die. Then print it out and decide whether you hit the target number or not.

**Note:** You may choose to implement this function using tail recursion or not, as you wish. If using tail recursion, you should name your tail-recursive helper function num-occurs-in-n-tosses-acc. It will need an extra input—an accumulator—that accumulates the number of occurrences of the target number over all the tosses. After your accumulator-based helper function is working properly, you should then define a wrapper function, called num-occurs-in-n-tosses, that simply calls the accumulator-based function with appropriate inputs. Of course, you may wish to implement both versions!

---

**Problem 15.4: Flipping coins**

When flipping coins, any occurrence of \(n\) consecutive coin flips that come out the same (i.e., all H or all T) may be called a streak of length \(n\). For this problem, you must define a function, called max-streak-in-n-flips, that satisfies the following contract:

\[
\text{MAX-STREAK-IN-N-FLIPS}
\]

INPUT: \(N\), a non-negative integer

OUTPUT: The length of the longest streak of consecutive coin flips (whether all H or all T) that occur in a sequence of \(N\) random coin flips.

SIDE EFFECT: Prints out the coin flips along the way.

Here are some examples of its use:

\[
(\text{max-streak-in-n-flips 10}) \implies T\ H\ H\ H\ T\ T\ T\ H\ H\ ...
\]
4

\[
(\text{max-streak-in-n-flips 10}) \implies T\ T\ H\ H\ T\ T\ T\ H\ ...
\]
3

\[
(\text{max-streak-in-n-flips 15}) \implies T\ T\ H\ T\ H\ T\ H\ H\ T\ T\ ...
\]
3

In the first sequence, the longest streak involves four consecutive Hs. In the second and third sequences, the longest streak involves three consecutive Hs.

1. Begin by defining a separate helper function, called max-streak-in-n-flips-acc that takes additional inputs to keep track of things such as: the value of the most recent coin flip, the number of consecutive coin flips that have just come out the same, and the maximum number of consecutive coin flips that have been seen since you started flipping coins. Once you get things working properly, you should then define your wrapper function, max-streak-in-n-flips.

Consider the following sequence of coin flips:

\[
H\ T\ H\ T\ T\ H\ H\ H\ H\ T\ T\ ...
\]

What information would you need to keep track of along the way to solve this problem?

2. Be sure to test your function in cases where \(n = 0\) and \(n = 1\).
Problem 15.5

Define a function, \texttt{num-tosses-until-repeat}, that satisfies the following contract:

\begin{verbatim}
;; NUM-TOSSES-UNTIL-REPEAT
;; -----------------------------
;; INPUTS: None
;; OUTPUT: An integer specifying the number of random tosses of a
;; 6-sided die until two CONSECUTIVE tosses come out the same.
;; SIDE EFFECTS: Prints out the tosses along the way.
\end{verbatim}

Here are some examples that illustrate the desired behavior:

\begin{verbatim}
> (num-tosses-until-repeat)
3 5 1 3 4 2 3 6 1 3 2 2 --> Got a repeat!
12
> (num-tosses-until-repeat)
2 4 4 --> Got a repeat!
3
> (num-tosses-until-repeat)
1 6 1 4 4 --> Got a repeat!
5
\end{verbatim}

Note that, in each case, the first line of text is side-effect printing, while the second line displays the output value.

Hint: Define a helper function, \texttt{num-tosses-until-repeat-helper}, that does most of the work. It should satisfy the following contract:

\begin{verbatim}
;; NUM-TOSSES-UNTIL-REPEAT-HELPER
;; ---------------------------------------
;; INPUTS: NUM-TOSSES-SO-FAR, a non-negative integer
;; MOST-RECENT-TOSS, a non-negative integer
;; OUTPUT & SIDE-EFFECTS similar to NUM-TOSSES-UNTIL-REPEAT
\end{verbatim}

Note that \texttt{num-tosses-so-far} keeps track of how many tosses have been made so far. (It accumulates the number of tosses so far.) In addition, note that \texttt{most-recent-toss} keeps track of the value of the most recently tossed die so that a new toss can be compared to the most recent toss.

In the body of \texttt{num-tosses-until-repeat-helper}, you should toss one die and store its value in a local variable. Then print it out. Then compare this new toss to the most recent toss. If they are the same, then stop; otherwise, keep going—with adjusted inputs.

The \texttt{num-tosses-until-repeat} “wrapper” function should let the helper function do most (or all) of the work.

Problem 15.6

Define a function, called \texttt{num-tosses-until-three}, that satisfies the following contract:

\begin{verbatim}
;; NUM-TOSSES-UNTIL-THREE
;; -----------------------------
;; INPUTS: None
\end{verbatim}
;; OUTPUT: An integer representing the number of random tosses of a 6-sided die that were required before three CONSECUTIVE tosses came out the same. ;; SIDE EFFECTS: Prints out the tosses along the way

Here are some examples of it in action:

> (num-tosses-until-three)
5 2 1 4 4 4 --> We got three in a row! ← side-effect printing
6
> (num-tosses-until-three)
1 1 4 6 6 3 6 5 3 4 5 1 5 2 4 6 2 1 3 1 3 5 2 1 2 5 6 1 6 3 4 3
1 4 1 4 6 5 3 4 4 6 2 4 2 2 5 3 5 5 5 --> We got three in a row!
51

Hint: Define a helper function, called num-tosses-until-three-helper, that does most of the work:

;; NUM-TOSSES-UNTIL-THREE-HELPER
;; --------------------------------------------------------------
;; INPUTS: NUM-TOSSES-SO-FAR, an integer
;; PREV-TOSS-1, PREV-TOSS-2, the most recent tosses
;; (or #f if just getting started)
;; OUTPUT: When called with NUM-TOSSES-SO-FAR = 0, and PREV-TOSS-1 = PREV-TOSS-2 = #f, outputs the number of random tosses of a 6-sided die before three consecutive tosses came out the same.
;; SIDE EFFECTS: Prints out the tosses along the way.

Problem 15.7

Define a function, called toss-until-doubles, that satisfies the following contract:

;; TOSS-UNTIL-Doubles
;; ________________________________
;; INPUTS: None
;; OUTPUT: The sum of the first occurrence of "doubles"
;; SIDE EFFECT: Tosses a pair of dice, printing out the results (and their sum), until doubles are encountered!

Here are some examples of its use:

> (toss-until-doubles) ===> TOSSES: 5, 2; sum = 7
TOSSES: 1, 2; sum = 3
TOSSES: 3, 6; sum = 9
TOSSES: 2, 2; sum = 4
HEY! We got doubles!!
4
> (toss-until-doubles) ===> TOSSES: 5, 6; sum = 11
TOSSES: 2, 6; sum = 8
TOSSES: 1, 1; sum = 2
In the first example, each pair of tosses is printed out, along with their sum, until doubles are found. (The 2 and 2 count as doubles.) Then, the message “HEY! We got doubles!” is printed out. Finally, 4 (i.e., the sum of the recently tossed doubles) is returned as the output value; it is not displayed as side-effect printing. Similarly, in the second example, each pair of tosses is printed out until the 1 and 1 occurrence of doubles is found. In that case, 2 is the output value, not side-effect printing.

**Problem 15.8**

Define a function, called `toss-three-dice-until-beat-target`, that satisfies the following contract:

```scheme
;; ; TOSS-THREE-DICE-UNTIL-BEAT-TARGET
;; ; --------------------------------------------------------------
;; ; INPUT: TARGET, an integer LESS THAN 18
;; ; SIDE EFFECT: Simulates the repeated tossing of three dice, printing out the tosses and their sum, until the sum is GREATER than TARGET
;; ; OUTPUT: The sum of the three dice that beat the TARGET.
```

Here are some examples of its behavior:

```scheme
> (toss-three-dice-until-beat-target 12)
6 + 2 + 3 = 11
3 + 1 + 2 = 6
5 + 6 + 3 = 14
14
> (toss-three-dice-until-beat-target 12)
5 + 3 + 6 = 14
14
> (toss-three-dice-until-beat-target 14)
5 + 4 + 5 = 14
6 + 3 + 6 = 15
15
> (toss-three-dice-until-beat-target 14)
1 + 1 + 5 = 7
5 + 5 + 1 = 11
4 + 1 + 3 = 8
3 + 5 + 4 = 12
1 + 3 + 1 = 5
1 + 2 + 4 = 7
2 + 2 + 2 = 6
5 + 2 + 3 = 10
5 + 5 + 3 = 13
5 + 1 + 1 = 7
4 + 6 + 6 = 16
16
```
**Problem 15.9**

Define a function, called `toss-until-total-beats-target`, that satisfies the following contract:

```scheme
;;; TOSS-UNTIL-TOTAL-BEATS-TARGET
;;; -------------------------------------------------------------
;;; INPUT: TARGET, an integer
;;; SIDE EFFECT: Simulates the tossing of a die, printing out all tosses along the way, until the sum of all dice tossed is *greater* than TARGET.
;;; OUTPUT: The total of the dice that finally beat the target.
```

Here are some examples of its behavior:

```scheme
> (toss-until-total-beats-target 10)
5 4 4 ... 13
> (toss-until-total-beats-target 10)
4 5 6 ... 15
> (toss-until-total-beats-target 20)
1 3 4 1 4 5 ... 22
> (toss-until-total-beats-target 20)
5 1 6 3 4 3 ... 22
```

**Problem 15.10: Using `let*` to create a fuel report**

Define a function, called `fuel-report`, that satisfies the following contract:

```scheme
;;; FUEL-REPORT
;;; -----------------------------------
;;; INPUTS: STARTING-MILES, non-negative number representing the starting reading of the odometer of a car
;;; ENDING-MILES, non-negative number representing the ending reading of the odometer of a car
;;; COST-PER-GALLON, cost of gas purchased
;;; NUM-GALLONS, number of gallons purchased
;;; OUTPUT: none
;;; SIDE EFFECT: Prints out a fuel report including the number of miles traveled, the miles per gallon, the amount of money spent (in dollars), and the cost per mile (in dollars per mile).
;;; NOTE: miles-per-gallon = num-miles-traveled / num-gallons
;;; dollars-spent = cost-per-gallon * num-gallons
;;; cost-per-mile = num-dollars-spent / num-miles-traveled
```

Here are some examples of its desired behavior:

```scheme
> (fuel-report 0 100 5.0 10)
Miles traveled: 100, miles-per-gallon: 10
Dollars spent: 50.0, cost-per-mile: 0.5
> (fuel-report 25 75 4.0 3.0)
Miles traveled: 50, miles-per-gallon: 16.666666666666668
Dollars spent: 12.0, cost-per-mile: 0.24
```
Note that it does not generate any output; all of the text is side-effect printing.
The purpose of this problem is to practice using the let* special form to simplify a sequence of computations. Thus, you should use a single let* to create a sequence of local variables with the following names: miles-traveled, miles-per-gallon, dollars-spent and cost-per-mile. Note that the value of each variable depends only on the values of variables defined before it. For example, the value of miles-per-gallon depends only on miles-traveled and num-gallons. Similarly, the value of miles-traveled depends only on the inputs starting-miles and ending-miles.

Problem 15.11

Mimicking the structure of facty and facty-acc from Example 15.6.3, define a function called sum-cubes that uses letrec to define an accumulator-based, tail-recursive local helper function called sum-cubes-acc. The sum-cubes function should satisfy the following contract:

;; SUM-CUBES
;; -----------------------------------------------
;; INPUTS: N, a positive integer
;; OUTPUT: The sum: 1*1*1 + 2*2*2 + ... + N*N*N

Here are some examples of its desired behavior:

> (sum-cubes 3)
36
> (sum-cubes 4)
100

Problem 15.12

Same as Problem 15.1b, except that you should use letrec to define the recursive helper function as a local function.

Problem 15.13

Same as Problem 15.3, except that you should use letrec to define the recursive helper function as a local function.

Problem 15.14

Same as Problem 15.5, except that you should use letrec to define the recursive helper function as a local function.
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:

```scheme
;; 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:

```scheme
> (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:

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

Here are some examples:

```scheme
> (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:

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

Here are some examples:

```scheme
> (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) ⇒ #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:

```scheme
;; 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 8) >)
#f
> (string<=? "beard" "bread")   ← string<=? is built-in; it outputs \#t if its two inputs are in alphabetic order.
#t
> (sorted? "beard" "bread" "broom") string<=?)
#t
> (sorted? "bead" "breader" "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:

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (first-pair '(1 2 3 4 4 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:

```scheme
;; 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:

```scheme
> (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.

```scheme
> (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 \texttt{list-equal?} uses \texttt{eq?} to test the equality of corresponding elements, but \texttt{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 \texttt{remove-if}, that satisfies the following contract:

\begin{verbatim}
;; 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.
\end{verbatim}

Here are some examples:

\begin{verbatim}
> (remove-if even? '(1 2 3 4 5 6))
(1 3 5)
> (remove-if odd? '(1 2 3 4 5 6))
(2 4 6)
\end{verbatim}

\textit{Hint:} There can be two recursive cases, one where \texttt{(first listy)} "satisfies" \texttt{pred}, the other where \texttt{(first listy)} does not. In the first case, you do not want to include \texttt{(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 \texttt{replace}, that satisfies the following contract:

\begin{verbatim}
;; 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 \texttt{EQ?} predicate.)
\end{verbatim}

Here are some examples:

\begin{verbatim}
> (replace 'x 'y '(a x b x c x))
(a y b y c y)
> (replace 0 1 '(0 1 1 0 0 0 1))
(1 1 1 1 1 1 1)
\end{verbatim}
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)
```

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
;; LISTS, a list that contains at least N elements
;; OUTPUT:  A list containing the first N elements of LISTS,
;; in the same order as their order in LISTS.
```

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:

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (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)
> (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
Here are some examples:

```scheme
> (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.

```scheme
;; 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.

```scheme
;; 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 1 (2)) 1))
(1 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.

```scheme
;; 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:

```scheme
;; 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.* 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.

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:

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