Problems for Introduction to Computer Science via Scheme

Luke Hunsberger

© 2020 Luke Hunsberger
All rights reserved

Spring 2020
<table>
<thead>
<tr>
<th></th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Scheme Expressions vs. Scheme Data</td>
</tr>
<tr>
<td>3</td>
<td>Evaluating Scheme Data</td>
</tr>
<tr>
<td>4</td>
<td>Introduction to DrScheme</td>
</tr>
<tr>
<td>5</td>
<td>Built-In Functions</td>
</tr>
<tr>
<td>6</td>
<td>Non-Empty Lists</td>
</tr>
<tr>
<td>7</td>
<td>Special Forms</td>
</tr>
<tr>
<td>8</td>
<td>Predicates</td>
</tr>
<tr>
<td>9</td>
<td>Defining Functions</td>
</tr>
<tr>
<td>10</td>
<td>Some practicalities</td>
</tr>
<tr>
<td>11</td>
<td>Conditional Expressions I</td>
</tr>
<tr>
<td>12</td>
<td>Recursion I</td>
</tr>
<tr>
<td>13</td>
<td>Conditional Expressions II</td>
</tr>
<tr>
<td>14</td>
<td>Recursion II</td>
</tr>
<tr>
<td>15</td>
<td>Local Variables, Local Environments</td>
</tr>
<tr>
<td>16</td>
<td>Lists and List-Based Recursion</td>
</tr>
<tr>
<td>17</td>
<td>Iteration</td>
</tr>
<tr>
<td>18</td>
<td>Vectors</td>
</tr>
<tr>
<td>19</td>
<td>Data Structures</td>
</tr>
<tr>
<td>20</td>
<td>The Model-View-Controller Paradigm</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

No problems for this chapter.
Chapter 2

Scheme Expressions vs. Scheme Data

No problems for this chapter.
Chapter 3

Evaluating Scheme Data

No problems for this chapter.
Chapter 4

Introduction to DrScheme

No problems for this chapter.
Chapter 5

Built-In Functions

Problem 5.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type (e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it evaluates to. The first one is done for you as an illustration. Note that each answer should be one of the Scheme data types without any additional information (e.g., number, not the number 47).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents a datum of this type</th>
<th>Evaluates to a datum of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>#t</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6

Non-Empty Lists

Problem 6.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type (e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it evaluates to. The first one is done for you as an illustration. Note that each answer should be one of the Scheme data types without any additional information (e.g., function, not the addition function).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents a datum of this type</th>
<th>Evaluates to a datum of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>#t</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>(* 4 6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(eval 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(void)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problem 6.2

Describe in detail the steps that DrScheme goes through in evaluating the datum denoted by the expression (void). Beware of conflating the void datum, the void symbol, and the built-in void function.
Chapter 7

Special Forms

Problem 7.1

When the Default Rule is used to evaluate a non-empty list, the first step is to evaluate each element in the list. However, special forms are not evaluated by the Default Rule. As a result, it can happen that some of the elements in a special form are evaluated, while others are not. For this problem, summarize the following information about the define and quote special forms:

1. how many elements in the non-empty list;
2. which elements get evaluated and which do not;
3. whether there is an output value and, if so, what it is; and
4. whether there is a side effect and, if so, what it is.

Problem 7.2

For each statement below, decide which of the words in parentheses apply:

- Evaluation of a define special form (always, never, sometimes) causes a side effect.
- Evaluation of a quote special form (always, never, sometimes) causes a side effect.
Chapter 8

Predicates

Problem 8.1

Each of the following Scheme expressions denotes/represents some kind of Scheme datum. For each, state the data type (e.g., number, boolean, symbol, list, function, etc.) of the Scheme datum it represents. In addition, specify the data type of the Scheme datum it evaluates to. The first one is done for you as an illustration. Each answer should be one of the Scheme data types without any additional information (e.g., symbol, not the symbol xyz).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents a datum of this type</th>
<th>Evaluates to a datum of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(* 4 6)</td>
<td>list</td>
<td>number</td>
</tr>
<tr>
<td>’cs101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(&gt; 4 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(number? ’x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(symbol? ’x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(symbol? eval)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problem 8.2

Write down a contract for the built-in >= function.

Problem 8.3

Explain the steps taken by DrScheme in the following interactions:

```scheme
> +
#<procedure:+>
> (define addn +)
> (addn 4 5)
9
> (define function? procedure?)
> (function? +)
#t
```
Problem 8.4

*Explain the output values generated by the following sequence of interactions.*

```scheme
> (define myvar 'sunday)
> (define yourvar 'monday)
> (eq? myvar 'sunday)
  #t
> (eq? myvar myvar)
  #t
> (eq? myvar 'myvar)
  #f
> (eq? yourvar yourvar)
  #f
> (eq? yourvar 'monday)
  #t
> (eq? yourvar 'sunday)
  #f
```

Problem 8.5

*Describe in detail the steps that DrScheme goes through in evaluating the expression* `(void? (void))`. *Be sure to carefully distinguish the void datum, the built-in void function, and the built-in void? type-checker predicate.*
Chapter 9

Defining Functions

Problem 9.1

Consider the following Interactions Window session:

> (define pie 3.14159)
> (define funk (lambda (x) (* x pie)))
> (funk 10)
31.4159

Accurately describe the process that DrScheme goes through in evaluating these three expressions to generate the result, 31.4159. Strive to be complete, while also being concise.

Problem 9.2

Mathematicians tell us that sums of the form, $1 + 2 + \ldots + n$, are equal to $n(n + 1)/2$. For example, $1 + 2 + 3 + 4 = 10 = (4 \cdot 5)/2$. Define a Scheme function, called sum-from-one-to, that takes a positive integer $n$ as its only input, and returns the sum of all the integers from 1 to $n$ as its output. It should not generate any side effects. The desired behavior is illustrated below.

> (sum-from-one-to 4)
10
> (sum-from-one-to 5)
15
Chapter 10

Some practicalities

<table>
<thead>
<tr>
<th>Problem 10.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write down a contract for the tester function using the form below:</td>
</tr>
<tr>
<td>Name:</td>
</tr>
<tr>
<td>Input:</td>
</tr>
<tr>
<td>Output:</td>
</tr>
<tr>
<td>Side Effects:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem 10.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrScheme uses the Default Rule to evaluate the list denoted by <code>(tester '(+ 1 2))</code>. As indicated above, the result of evaluating this list is the number 3. Carefully describe the process DrScheme goes through to generate this result. (You may wish to review Examples 9.3.3 and 9.3.4 to recall how DrScheme applies a function to inputs.) In particular, what value is associated with the input parameter <code>datum</code> in the local environment? What value is passed to the <code>printf</code> function called in the body of the tester function? And what steps does DrScheme go through to evaluate the expression <code>(eval datum)</code> in the body of the tester function?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem 10.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>How would you change the definition of the tester function so that it printed out the result of evaluating the Scheme datum as a side effect, instead of returning it as the output value? In this case, the output of the tester function would be the <code>void</code> datum. As with the original function, the new tester function should still print out the input datum, too.</td>
</tr>
</tbody>
</table>
Chapter 11

Conditional Expressions I

Problem 11.1: The \texttt{maxx} function

Define a function, called \texttt{maxx}, that takes two numbers as its inputs. It should return the maximum of the two numbers, as illustrated below:

\begin{verbatim}
> (maxx 2 3)
3
> (maxx 5 1)
5
> (maxx 4 4)
4
\end{verbatim}

As you may have guessed, there is a built-in \texttt{max} function. However, you are not allowed to use it for this problem! Instead, use \texttt{if} to determine which input number is bigger. The operation of the \texttt{maxx} function could be described thusly: if \( x \) is bigger than \( y \), then the output should be \( x \); otherwise, it should be \( y \). Here’s the contract:

\begin{verbatim}
;; MAXX
;; -----------------------------
;; INPUTS: X, Y, two numbers
;; OUTPUT: The maximum of X and Y
\end{verbatim}

And some tester expressions:

\begin{verbatim}
(tester '(maxx 2 3))
(tester '(maxx 5 1))
(tester '(maxx 4 4))
\end{verbatim}

Problem 11.2: Printing out a message about bananas!

Here’s the contract for the \texttt{banana-msg} function. Note that it does not generate any output value; however, it does cause some side-effect printing to occur in the Interactions Window:

\begin{verbatim}
;; BANANA-MSG
;; -----------------------------
;; INPUT: NUM, an integer
;; OUTPUT: don’t care
\end{verbatim}
Here are some examples of its use (in the Interactions Window):

> (banana-msg 3)
I ate 3 bananas! ← these are not output values!!
> (banana-msg 1)
I ate 1 banana! ← they are side-effect printing!!
> (banana-msg 0)
I ate 0 bananas!

Here are some tester expressions to copy into your Definitions Window:

(tester '(banana-msg -2))
(tester '(banana-msg 0))
(tester '(banana-msg 1))
(tester '(banana-msg 3))

There are many ways to solve this problem. Recall the description of the printf function in Chapter 10.

---

**Problem 11.3**

Define a function, called quadrant, that satisfies the following contract:

```
;; QUADRANT
;; ---------------------------
;; INPUTS: X, Y, two numbers
;; OUTPUT: A number specifying the quadrant to which the point (X,Y) belongs in the XY-plane; or 0 if it lies on an axis.
```

Recall that the first quadrant is where both x and y are positive; the second quadrant is where x is negative and y is positive; the third quadrant is where both x and y are negative; and the fourth quadrant is where x is positive and y is negative. Be sure to test your function on a variety of inputs (for all four quadrants and various places on the axes, including the origin).

---

**Problem 11.4**

Define a function, called data-type-of, that satisfies the following contract. Note that it returns a symbol as its output.

```
;; DATA-TYPE-OF
;; ---------------------------
;; INPUT:  DATUM, anything
;; OUTPUT: A SYMBOL representing the data type of DATUM, one of: NUMBER, BOOLEAN, LIST, SYMBOL, STRING, etc.
```

Note: You don’t need to handle every possible data type. Returning the symbol unknown is okay if you get tired. Here are some examples of its behavior in the Interactions Window:
> (data-type-of #t)
  boolean
> (data-type-of ())
  list
> (data-type-of 45)
  number

And here are some tester expressions to copy into your Definitions Window:

(tester '(data-type-of 3))
(tester '(data-type-of #t))
(tester '(data-type-of '(+ 2 3)))
(tester '(data-type-of (+ 2 3)))
(tester '(data-type-of "abc"))

Hint: Use the built-in type-checker predicates from Chapter 8.

Problem 11.5

The implies function takes two boolean inputs, and generates a boolean output, as illustrated below:

(implies #t #t) ===> #t
(implies #f #t) ===> #t
(implies #t #f) ===> #f
(implies #f #f) ===> #t

Write a contract for the implies function, and then implement it in Scheme.

⋆ There are only four different combinations of inputs for this function; however, you can include more complicated input expressions, such as: (implies (> 3 2) (< 4 5)). And since anything other than #f counts as true, you can even do things like: (implies 'hi 'there).

Problem 11.6: Simplifying expressions

Replace each of the following expressions with a simpler, equivalent expression.

(if (eq? (> x y) #t) #t #f)
(eq? (if (> x y) #t #f) #t)
Chapter 12

Recursion I

**Problem 12.1**

Define a function, called `power`, that takes two inputs: `x`, any real number, and `p` any non-negative integer. It should return as its output the value of `x` raised to the `p`th power (i.e., \(x^p\)), as illustrated below.

\[
\begin{align*}
> (\text{power} \ 2 \ 3) & \quad \leftarrow \ 2^3 = 2 \cdot 2 \cdot 2 = 8 \\
8 & \\
> (\text{power} \ 3 \ 2) & \quad \leftarrow \ 3^2 = 3 \cdot 3 = 9 \\
9 & \\
> (\text{power} \ 2 \ 5) & \quad \leftarrow \ 2^5 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32 \\
32 & 
\end{align*}
\]

* Hint: Use recursion, similar to how it is used in `facty-v1` from Example 12.1.2. For example, note that \(2^9 = 2 \cdot (2^8)\).

* Be sure to include a contract for your function.
Chapter 13

Conditional Expressions II

Problem 13.1

For each statement below, decide which of the words in parentheses apply:

- Evaluation of an if special form (always, never, sometimes) causes a side effect.
- Evaluation of an and special form (always, never, sometimes) causes a side effect.
- Evaluation of an or special form (always, never, sometimes) causes a side effect.
- Evaluation of an if special form always requires evaluating exactly (one, two, all) of its inputs.
- Evaluation of an and special form always requires evaluating at least (one, two, all) of its inputs.
- Evaluation of an or special form always requires evaluating at least (one, two, all) of its inputs.

Problem 13.2

Recall that times in the 24-hour military clock involve hours that range from 0 to 23. For example, 00:00 corresponds to midnight; 08:23 is sometime in the morning; 12:00 corresponds to noon; and 15:39 is sometime in the afternoon.

(a) Define a function, called time-of-day, that takes two numerical inputs, mil-hours and minutes, where mil-hours represents the number of hours according to the 24-hour military clock, and minutes represents the number of minutes.

- You may assume that mil-hours and minutes are integers such that:
  0 ≤ mil-hours < 24 and 0 ≤ minutes < 60.
- time-of-day should return a symbol as its output. In particular, it should return one of the following: midnight, am, noon or pm, as appropriate. For example:

  > (time-of-day 15 39)
  pm
  > (time-of-day 12 0)
  noon

Note that the pm and noon are output values that are symbols; they are not side-effect printing.

- HINT: You may use if or cond in the body of your function. In either case, be sure to include comments that briefly describe each case that you’re handling.
NOTE: 12:00 midnight is neither a.m. nor p.m. Rather, it is a boundary between a.m. and p.m. Similar remarks apply to 12:00 noon. So (time-of-day 12 0) should return noon, not am or pm. Similarly (time-of-day 0 0) should return midnight, not am or pm. But (time-of-day 0 15) should return am, since 00:15 in military time corresponds to 12:15 a.m. in civilian time.

Be sure to write a contract for your function!

(b) Define a function, called \texttt{mil-to-civil-hrs}, that takes a single numerical input, \texttt{mil-hours}, where $0 \leq \text{mil-hours} < 24$. It should generate as its output, the corresponding number of hours according to the 12-hour civilian clock. For example:

\begin{verbatim}
(mil-to-civil-hrs 19) ===> 7
\end{verbatim}

Because 19:00 on the military clock corresponds to 7:00 p.m. on the civilian clock.

\begin{itemize}
  \item \textit{Hint}: Use a \texttt{cond} in the body of your function.
  \item \textit{Hint}: Be careful about 0.
  \item Be sure to include a contract for your function!
\end{itemize}

(c) Define a function, called \texttt{print-civil-time-from-mil}. It should take two numerical inputs, \texttt{mil-hours} and \texttt{minutes}, as in part (a). However, this function, unlike the above functions, should not generate any Scheme datum as output. Instead, it should have the side effect of displaying the time in the 12-hour civilian format, as the following interactions window session illustrates:

\begin{verbatim}
> (print-civil-time-from-mil 15 39)
3:39 pm
\end{verbatim}

\begin{itemize}
  \item In this example, 3:39 pm is side-effect printing, generated using the built-in \texttt{printf} function. There is no output value.
  \item Use the \texttt{time-of-day} and \texttt{mil-to-civil-hrs} functions as helpers. You shouldn’t need to re-implement the computations done by \texttt{time-of-day} or \texttt{mil-to-civil-hrs}.
  \item Be sure to include a contract for your function!
\end{itemize}

(Optional) If the number of minutes is small, you will have to work a little harder to make sure that a leading zero is displayed. Consider 3:6 pm versus 3:06 pm.

\begin{verbatim}
> (print-civil-time-from-mil 15 6)
3:06 pm
\end{verbatim}

There are many examples that demonstrate the use of the built-in \texttt{printf} function in the amst-helper.txt file available in each lab and assignment directory.

---

\textbf{Problem 13.3}

Define a function, \texttt{compute-tax}, that takes a single input, \texttt{income}. It should generate as its output the amount of tax owed on that income, as determined by the following tax brackets:

\begin{itemize}
  \item Income below $10,000 is taxed at 10%.
  \item Income between $10,000 and $30,000 (only the amount after the first $10,000) is taxed at 15%.
  \item Income above $30,000 (only the amount after the first $30,000) is taxed at 25%.
\end{itemize}

\implies If you make more than $10,000, the first $10,000 of your income is taxed at 10%; only the amount
above $10,000 is taxed at the higher rates. Similarly, if you make more than $30,000, the first $10,000 of income is taxed at 10%, the next $20,000 (i.e., the amount between $10,000 and $30,000) is taxed at 15%, and the rest is taxed at 25%. Thus, if you earn $100,000, your taxes will not be $25,000 (i.e., 25% of $100,000), instead, they will be:

\[ (10\% \text{ of } 10,000) + (15\% \text{ of } 20,000) + (25\% \text{ of } 70,000) \]

which equals: $1,000 + $3,000 + $17,500 = $21,500.

This function should not cause any side effects. Be sure to include tester expressions that test a representative set of cases. You may use if or cond for this problem. And, as always, be sure to include a contract for your function.

### Problem 13.4

The CMPU-101 bookstore is open on Saturdays from 11:45 a.m. to 12:15 p.m., inclusive, and all day Tuesday, except from 12:01 p.m. to 12:59 p.m., inclusive.

(a) Define a function, called bookstore-open?, that satisfies the following contract. Use a cond special form to structure the body of this function.

```
;;; BOOKSTORE-OPEN?
;;; ---------------------------------------
;;; INPUTS: DAY, a symbol, one of SUN, MON, TUE, ..., FRI, SAT
;;;          HOUR, an integer from 1 to 12, inclusive
;;;          MINUTES, an integer from 0 to 59, inclusive
;;;          AM-OR-PM, a symbol, either AM or PM
;;; OUTPUT: #t, if the inputs specify one of the following:
;;;          Saturday from 11:45 am to 12:15 pm, inclusive.
;;;          All day Tuesday, except from 12:01 pm to 12:59 pm,
;;;          inclusive; #f otherwise.
```

Here are some examples of its behavior:

```
> (bookstore-open? 'sat 11 30 'am)
#f
> (bookstore-open? 'sat 11 50 'am)
#t
> (bookstore-open? 'sat 12 12 'pm)
#t
> (bookstore-open? 'sat 12 49 'pm)
#f
```

(b) Same as above, except this time use the boolean operators, and, or and not, instead of conditional expressions, as described in Section 13.3.

### Problem 13.5

The following predicate is defined using a cond special form:

```
(define office-open? (lambda (day am-or-pm)
```


(cond
  ;; Case 1: Closed on Fridays
  ((eq? day 'fri)
   #f)
  ;; Case 2: Open Wed afternoons
  ((and (eq? day 'wed)
        (eq? am-or-pm 'pm))
   #t)
  ;; Case 3: Closed on Tuesday mornings
  ((and (eq? day 'tue)
        (eq? am-or-pm 'am))
   #f)
  ;; Case 4: Open all other times
  (else
   #t)))

Your job is to define a predicate, called office-open?-alt, that works just like office-open?, except that it is defined using and, or and not, instead of the conditional expressions, if or cond.

☆ Careful! Some of the cases above output #t, while others output #f.
Chapter 14

Recursion II

Problem 14.1

Define a function, called sum-recips, that computes sums of the following form:

\[ \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \]

where \( n \) is some positive integer. Here are some sample interactions:

> (sum-recips 1)
1
> (sum-recips 2)
3/2
> (sum-recips 3)
11/6

And the corresponding tester expressions:

(tester '(sum-recips 1))
(tester '(sum-recips 2))
(tester '(sum-recips 3))

Insert some more tester expressions of your own. If you want to encourage DrScheme to display numbers in “floating point” form (e.g., 1.5 instead of 3/2), just use 1.0 in your base case, instead of 1.

Consider the following:

> (/ 3 2)
3/2
> (/ 3.0 2)
1.5

Be sure to include a contract for your function!

Problem 14.2

Define a function, called alt-sum, that takes a positive integer \( n \) as its only input. It should return as its output the following sum:

\[ 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots \pm \frac{1}{n} \]
where the sign of each term is negative if the denominator is even, and positive if the denominator is odd. Here are some examples:

> (alt-sum 1)
1
> (alt-sum 2)
0.5
> (alt-sum 3)
0.8333333333333333

For example, (alt-sum 3) returns the sum, \(1 - \frac{1}{2} + \frac{1}{3} = \frac{5}{6}\). To ensure that you get the desired “floating point” notation, you can use expressions such as (/ 1.0 n) instead of (/ 1 n), as illustrated below:

> (/ 1 5)
1/5
> (/ 1.0 5)
0.2

This is especially relevant for cases where \(n\) is large. The value of (alt-sum 100) in fractional notation would be very cumbersome!

* There are built-in functions called even? and odd? that return #t if their input is an even (or odd) number, as illustrated below.

> (even? 4)
#t
> (even? 9)
#f
> (odd? 4)
#f
> (odd? 9)
#t

The following set of problems involve functions that do not generate any output value, but instead cause side-effect printing to occur. For such functions, the following alt-tester function will generate nicer looking test results in the Interactions Window.

```
;; ALT-TESTER
;; ---------------------
;; INPUT: DATUM, anything
;; OUTPUT: void
;; SIDE EFFECT: Displays DATUM, then prints a newline,
;; then evaluates DATUM, and finally prints another newline.

(define alt-tester
  (lambda (datum)
    (printf "A ==>" datum)
    (newline)
    (eval datum)
    (newline)))
```

It is the same as the tester function seen earlier, except that it makes sure that any side-effect printing caused by evaluating the expression (eval datum) starts on a new line, and it does not generate any output value! To
enable use of this function, copy-and-paste the above definition into your Definitions Window.

Problem 14.3

Why would it be difficult to implement the `print-n-dashes` function from Example 14.2 using `if` instead of `cond`?

Problem 14.4

Define a function, called `print-thing-n-times`, that takes two inputs: `thing` and `n`, where `thing` can be anything, and `n` is a non-negative integer. It should not generate an output value; instead, it should have the side effect of printing out `thing n` times in the Interactions Window, as illustrated below:

```
> (print-thing-n-times 'Hi 5)
HiHiHiHiHi
> (print-thing-n-times '*- 3)
*-*-*-*-*
```

Be sure to include a contract for your function.

Problem 14.5

Define a function, called `print-down-to-zero`, that takes a non-negative integer `n` as its only input. It should not generate any output value; instead, it should have the side effect of printing out the values from `n` down to zero in the Interactions Window, as illustrated below:

```
> (print-down-to-zero 5)
5 4 3 2 1 0
> (print-down-to-zero 22)
22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Be sure to include a contract for your function.

Problem 14.6: Printing rectangles and squares

Copy-and-paste the contract and function definition for the `print-n-dashes` function from Example 14.2 into your Definitions Window. Recall that `print-n-dashes` does not generate any Scheme output value; instead, it has the side effect of displaying a row of `n` dashes in the Interactions Window, as illustrated below.

```
> (print-n-dashes 5)
-----
> (print-n-dashes 12)
----------
```

(Printing Rectangles) For this part, you must define a function called `print-rectangle`, that takes two inputs, both of which are non-negative integers. This function should not generate any output value. Instead, it should have the following side effect: It should display a rectangular pattern of dashes whose number of rows and number of columns correspond to the two numerical inputs, as illustrated below.

```
  > (print-rectangle 3 5)
  -----
  -----
  -----
```

(Printing Squares) For this part, you must define a function called `print-square`, that takes one input, which is a non-negative integer. This function should not generate any output value. Instead, it should display a square of dashes whose side length corresponds to the numerical input, as illustrated below.

```
  > (print-square 3)  
  -----
  -----
  -----
```
> (print-rectangle 5 2)
--
--
--
--
--
> (print-rectangle 2 5)
-----
-----

Here are some hints:

- Pay attention to which input specifies the number of rows, and which specifies the number of columns.
- Use recursion.
- Use print-n-dashes as a helper function to print individual rows.

(Printing Squares) Define a function, called print-square, that takes a single non-negative integer as its only input. It should not generate any output value, but instead should print out a square pattern of dashes in the Interactions Window, whose number of rows and columns is specified by the single numerical input, as illustrated below.

> (print-square 3)
---
---
---
> (print-square 4)
----
----
----
----

*Hint:* Use the print-rectangle function as a helper. Your print-square function should not be complicated!

Problem 14.7: Printing upside-down triangles

Copy-and-paste the contract and definition for the print-n-dashes function from Example 14.2 into your Definitions Window. Then define a function, called print-upside-down-triangle, that satisfies the following contract:

```scheme
;; PRINT-UPSIDE-DOWN-TRIANGLE
;; ------------------------------------
;; INPUT: NUM-ROWS, a non-negative integer
;; OUTPUT: Nothing
;; SIDE EFFECT: Prints an upside-down triangle in the Interactions Window consisting of NUM-ROWS rows.
```

Here are some examples of its behavior:

> (print-upside-down-triangle 3)
---

---
Note that this is very similar to printing a rectangle (cf. Problem 14.6), except that the width of the row decreases with each recursive function call.

Problem 14.8: Printing rightside-up triangles

Copy-and-paste the contract and definition for the print-n-dashes function from Example 14.2 into your Definitions Window. Then define a function, called print-rightside-up-triangle, that satisfies the following contract:

```
;; PRINT-RIGHTSIDE-UP-TRIANGLE
;; ------------------------------------
;; INPUTS: NUM-ROWS, a non-negative integer
;; CURR-WIDTH, the width of the current row
;; OUTPUT: Nothing
;; SIDE EFFECT: Prints a rightside-up triangle in the
;; Interactions Window consisting of NUM-ROWS rows.
```

Here are some examples illustrating its behavior:

```
> (print-rightside-up-triangle 3 1)
--
--
--

> (print-rightside-up-triangle 5 1)
--
--
---
----
-----
```

Notice that this function is called with curr-width equal to 1, because that’s the width of the first row to be printed.

Problem 14.9

Suppose that func is a function whose output values are within some small non-negative range, say, from 0 to 50. For example, suppose that (func 3) evaluates to 25. That output value could be represented graphically by a horizontal line containing 25 asterisks. Similarly, if (func 4) evaluates to 16, then the next line of printing could show 16 asterisks. Your job is to define a function, called plotter, that plots the output values of a given function over a specified range of inputs. Here’s the contract:
;; PLOTTER
;; -----------------------------------------------
;; INPUTS: FUNC, a function that expects a numerical input
;; FROM, a starting input value (an integer)
;; TO, an ending input value (an integer)
;; OUTPUT: None
;; SIDE EFFECT: Displays the output values of FUNC for each
;; input in the range, FROM, FROM+1, FROM+2, ..., TO-2, TO-1, TO.
;; For each input value, the corresponding output value is
;; displayed by the appropriate number of asterisks printed on a
;; single line of the Interactions Window.

Here is an example that uses facty-v1 from Example 12.1.2:

> (plotter facty-v1 1 4)
* 
**
******
************************

And here is an example using abs, a built-in function that computes the absolute value of its input. (Notice what happens when abs is given an input of zero.)

> (plotter abs -3 3)
*** 
** 
* 
* 
** 
***

Finally, here’s an example where that uses lambda to create a squaring function on the spot, without bothering to give it a name!

> (plotter (lambda (x) (* x x)) 1 5)
* 
**** 
********
************
************************

Although you can run the above examples in the Interactions Window, you should also put the corresponding alt-tester expressions in your Definitions Window. Insert additional alt-tester expressions to demonstrate that your plotter function works as desired.

Problem 14.10

⇒ This problem assumes that you have already defined the print-thing-n-times function from Problem 14.4. That function can be used to print a line of + signs, or a line of − signs, in the Interactions Window.
Define a tail-recursive function called fancy-plotter that satisfies the following contract:

;; FANCY- PLOTTER
;; -----------------------------------------------
;; INPUTS: FUNC, a function that takes numerical input
;; FROM, a starting number (integer)
;; TO, a stopping number (integer)
;; OUTPUT: None
;; SIDE EFFECTS: This function plots the function values for FUNC
;; for each input in the range from FROM to TO. Each input
;; value will generate one line of printing in the Interactions
;; Window. For example, if (FUNC FROM) ==> 5, then this
;; function will display a line of five + signs; if (FUNC FROM)
;; ==> -5, then this function will display a line of five -
;; signs; if (FUNC FROM) ==> 0, then this function will simply
;; display a zero.

Here are some examples, one of which uses the built-in sin function:

> (fancy-plotter (lambda (x) (* x x x)) -3 3)
---------------------------
--------
-
0
+
+++++++                        
+++++++++++++++++++++++++++++++                        

> (fancy-plotter (lambda (x) (* 20 (sin (/ x 4)))) -20 20)
++++++++++++++++++++                        
++++++++++++++++++++                        
++++++++++++++++++++                        
++++++++++++++++++++                        
++++++++++++++++                        
++++++++++++++++                        
++++++++++++                        
++++++++++++                        
+++                        
---                        
--------                        
-----------------------
-----------------------
-----------------------
-----------------------
-----------------------
-----------------------
-----------------------
0                        
+++++                        
++++++++++++++++++++                        
++++++++++++++++++++                        

Problem 14.11

For this problem, you will implement a `print-checkerboard` function that displays a checkerboard pattern in the Interactions Window.

(a) Define a tail-recursive function called `print-checkerboard-acc` that takes four inputs: `num-rows, num-cols, curr-row` and `curr-col`. `num-rows` and `num-cols` specify the overall size of the checkerboard; `curr-row` and `curr-col` specify the location of the next square to be printed.

When called with appropriate initial values for `curr-row` and `curr-col`, this function should cause a `num-rows`-by-`num-cols` checkerboard pattern to be printed in the Interactions Window.

- The values of `num-rows` and `num-cols` should not change across the various recursive function calls, but the values of `curr-row` and `curr-col` will change.
- If the current square is somewhere in the middle of the board, this function should print just that one square. It should then let the recursive function call print the rest of the checkerboard. (How should the values of `curr-row` and `curr-col` be updated in this case?)
- If the sum of `curr-row` and `curr-col` is even, then print one kind of square (e.g., X); if their sum is odd, then print the other kind of square (e.g., _). (You may use the built-in functions, `even?` and `odd?`, to test whether a given number is even or odd.)
- How do you recognize that you have already finished printing out the entire checkerboard (i.e., you’ve hit the base case)?
- How do you recognize that you have finished printing out the current row? How should the values of `curr-row` and `curr-col` be updated in that case?

(b) Define a wrapper function called `print-checkerboard` that takes two inputs, `num-rows` and `num-cols`. It should cause a `num-rows`-by-`num-cols` checkerboard pattern to be displayed in the Interactions Window, as illustrated below:
> (print-checkerboard 3 6)
  X _ X _ X _
  _ X _ X _ X
  X _ X _ X _

Note that this function should just call the function from part (a) with appropriate inputs. You may wish to use the alt-tester function when writing test cases in the Definitions Window.

Problem 14.12

The following functions use the built-in quotient and remainder functions to access the individual digits in the base-ten representation of a number. For the purposes of this problem, the rightmost digit in a number will be considered to be in position zero, the next rightmost digit in position one, and so on. For example, the 3 in 9999399 will be considered to be in position 2.

(a) Define a function called nth-rightmost-digit that satisfies the following contract:

;; NTH-RIGHTMOST-DIGIT
;; -------------------------------------------------------------
;; INPUTS: NUM, a non-negative integer
;; N, a non-negative integer
;; OUTPUT: The Nth rightmost digit of NUM, where N=0 refers
;; to the rightmost digit.

Here are some examples of the desired behavior:

> (nth-rightmost-digit 92845 0)  5
> (nth-rightmost-digit 92845 2)  8

Hint: Consider how dividing a number by ten, using the built-in quotient and remainder functions, can effectively “peel off” the rightmost digit of the number.

> (quotient 345678 10)  34567
> (remainder 345678 10)  8

(b) Define a tail-recursive, accumulator-based function called num-occurs-acc that satisfies the following contract:

;; NUM-OCCURS-ACC
;; --------------------------------------------
;; INPUTS: DIGIT, an integer from 0 to 9, inclusive
;; NUM, a non-negative integer
;; ACC, an accumulator
;; OUTPUT: When called with ACC = 0, the output is the number
;; of occurrences of DIGIT in the decimal repr’n of NUM.
;; Example: (num-occurs-acc 3 3212334 0) => 4

After you have done so, then define the following “wrapper” function:
Here are some examples of the desired behavior:

> (num-occurs-wr 3 12312344444443)
3
> (num-occurs-wr 0 100)
2
> (num-occurs-wr 5 1234)
0

**Hint 1:** Use the built-in quotient and remainder functions. In the context of this problem, consider the following examples:

> (quotient 345678 10)
34567
> (remainder 345678 10)
8

So, dividing by ten each time allows you to effectively “peel off” the rightmost digit.

**Hint 2:** The base case should be when you have exactly one digit left (i.e., when num ≤ ten).

---

### Problem 14.13: Approximating the natural logarithm function

Mathematicians tell us that the natural logarithm function can be approximated using certain kinds of sums. In particular, for any real number \( x \in (-1, 1] \), and for any “sufficiently large” positive integer \( n \), the value \( \log(1 + x) \) is well approximated by the following sum:

\[
x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \pm \frac{x^n}{n}
\]

For example, the value \( \log(1.5) \), where \( x = 0.5 \), is well approximated by the sum:

\[
0.5 - \frac{(0.5)^2}{2} + \frac{(0.5)^3}{3} - \frac{(0.5)^4}{4}
\]

Admittedly, these facts are probably not obvious! But that’s okay, since this is not a math class! We can just accept what the mathematicians tell us and go about the business of computing these kinds of sums. So... for this problem, define an accumulator-based function, called approx-log-acc that satisfies the following contract. Your function should be tail-recursive. Also, you may want to use the built-in even? and odd? functions, seen previously. The power function, from Problem 12.1, should be helpful.

```scheme
;; APPROX-LOG-ACC
;; ---------------------------------------------
```
INPUTS: X, any number such that -1 < X <= 1
FROM, the index of the "current term" in the sum
TO, the index of the "last term" in the sum
ACC, an additive accumulator
OUTPUT: When called with FROM=1, ACC=0, and TO=N the output is
the sum: X - X*X/2 + X*X*X/3 - ... (+/-)X^N/N

After you have defined approx-log-acc, define the following wrapper function:

(define APPROX-LOG-WR
  (lambda (x to)
    ;; Call the acc-based helper function with FROM=1 and ACC=0
    (approx-log-acc x 1 to 0.0)))

Notice that (approx-log-wr 0.5 4) should compute the sum seen earlier that is supposed to be a good approximation of \( \log(1.5) \). To verify these sorts of examples, you can use Scheme’s built-in log function, but keep in mind that the above sums are good approximations for \( \log(1+x) \), not for \( \log(x) \). So, for example, the expression (approx-log-wr 0.5 4) will evaluate to a good approximation of \( \log(1.5) \), since \( 1 + 0.5 = 1.5 \).

> (approx-log-wr 0.5 10) ← x = 0.5
0.4054346478174603
> (log 1.5) ← 1 + x = 1.5
0.4054651081081644

Although approx-log-wr will be good at estimating values of \( \log(1+x) \) for small values of \( x \), it doesn’t do so well as the value of \( x \) approaches 1. Compare the values returned by (approx-log-wr 1 n) for various values of \( n \) against the value of (log 2). How big does \( n \) have to be before the answer is correct to within 3 decimal places? Be sure to include a variety of tester expressions in your definitions file to see how well expressions of the form (approx-log-wr x n) approximate \( \log(1+x) \) for various values of \( x \) and \( n \).

Problem 14.14: Computing geometric sums

This problem concerns the computation of sums such as those shown below:

\[
1 + 10 + 10^2 + 10^3 = 1 + 10 + 100 + 1000 = 1111 \\
1 + 2 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31 \\
1 + 3 + 3^2 + 3^3 = 1 + 3 + 9 + 27 = 40
\]

More generally, for any number \( x \) and any non-negative integer \( n \), the following expression is called a geometric sum:

\[
1 + x + x^2 + x^3 + \ldots + x^n
\]

where terms of the form \( x^k \) stand for “\( x \) raised to the \( k \)th power”. Your job is to define a function, called geom, that takes two inputs, \( x \) and \( n \), and whose output is the value of the corresponding geometric sum, as shown above.

Now, there are lots of ways to do this problem. Here, we are going to focus on a way that involves defining an accumulator-based tail-recursive helper function, called geom-helper-acc, that takes the following additional inputs:

- \( k \), a counter that goes from 0 up to \( n \)
Problems for Introduction to Computer Science via Scheme

42 Problems for Introduction to Computer Science via Scheme

© 2020 Luke Hunsberger

Spring 2020

• x-to-the-k, a variable that takes on the values, 1, x, x^2, x^3, etc.

• acc, a variable that accumulates the desired sum

(When I say that k is a counter “that goes from 0 up to n”, I really mean that the value of the input k on successive recursive function calls increases by one each time.)

Consider the case where x = 2 and n = 4. The sum we want to compute is: 1 + 2^2 + 2^3 + 2^4, which happens to be equal to 31. Here are the successive values we want the variables, k, x-to-the-k and acc to take on during successive recursive function calls:

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-to-the-k</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>acc</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>15</td>
<td>31</td>
</tr>
</tbody>
</table>

← We’ll stop here, since k > 4
← That’s the desired answer!!

As suggested earlier, the value of k increases by one for each successive recursive function call; x-to-the-k is multiplied by x each time; and acc accumulates the most recent value of x-to-the-k.

In particular, the value of acc in one column is the sum of the values of x-to-the-k and acc from the preceding column:

1 + 0 = 1; 2 + 1 = 3; 4 + 3 = 7; 8 + 7 = 15; 16 + 15 = 31.

Okay, you are now ready to define the accumulator-based, tail-recursive helper function, geom-helper-acc. It should take the following inputs: x, n, k, x-to-the-k and acc.

* For the base case... when should this function stop?

* For the recursive case... make a tail-recursive function call with appropriately adjusted inputs.

Afterward, you can define geom as a wrapper function that simply calls the above helper function with appropriate initial values for its five inputs. Here’s how it should work in the end:

> (geom-sum 10 3)
1111
> (geom-sum 2 4)
31

As always, be sure to include contracts for each function you define.

Problem 14.15: Approximating the arctangent function

Mathematicians tell us that the arctangent function can be approximated using sums of the following form:

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \pm \frac{x^n}{n}$$

where n is any odd positive integer. Define a function, called approx-arctan-acc, that uses the following inputs to keep track of relevant information over the course of the recursive function calls:
x  Fixed value
from Positive odd integer, the denominator of the current term
sign Alternates between 1 and −1
curr-power Current power of \( x \) (computed incrementally)
n Fixed value, indicates last term in the sum
acc An accumulator

What initial values should be given to the inputs from, sign, curr-power and acc?
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:

```scheme
> (sum-the-even-tosses 5) 3 6 2 1 5 ← side-effect printing: the 5 tosses 8 ← output: the sum of the even tosses
> (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:

```scheme
> (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:

```
;; 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:

```
(max-streak-in-n-flips 10) ===> T H H H T T H H ... 4
(max-streak-in-n-flips 10) ===> T H T H T T T H ... 3
(max-streak-in-n-flips 15) ===> T T T H T T T H H H T T H ... 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.

* 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 H T H T ...
```

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

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

Define a function, `num-tosses-until-repeat`, that satisfies the following contract:

```scheme
;; 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.
```

Here are some examples that illustrate the desired behavior:

```scheme
> (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
```

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, `num-tosses-until-repeat-helper`, that does most of the work. It should satisfy the following contract:

```scheme
;; 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
```

Note that `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 `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 `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 `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 `num-tosses-until-three`, that satisfies the following contract:

```scheme
;; NUM-TOSSES-UNTIL-THREE
;; -----------------------------
;; INPUTS: None
```
;; 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  ← output value
> (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
HEY! We got doubles!!
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:

;; 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:

> (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 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 = 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: 16.66666666666668
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.
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?.

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.

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) ⇒ #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-SATISIFIER?
;; ---------------------------------------------------------------------------
;; INPUTS: PRED, a predicate (e.g., EVEN?) that takes a single input
;; LİSTY, a list of suitable inputs for PRED
;; OUTPUT: #t if LİSTY 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-LİST?
;; ---------------------------------------------------------------------------
;; INPUTS: N, a non-negative integer
;; LİSTY, a list
;; OUTPUT: #t if LİSTY 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
> (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
```
Here are some examples:

> (first-pair '(1 2 3 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:

```
;; 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 0 0 0 1))
(1 1 1 1 1 1 1)
Problem 16.15

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

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (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:

```scheme
;; 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:

```scheme
> (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
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) '(#f #f #f #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 1 (2)) 1))
(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.

```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.
```

```scheme
> (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:

```scheme
> (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 \texttt{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 \texttt{eq?} predicate only checks whether they start with the \textit{same} cons cell; it doesn’t even look at the contents of that cons cell. The differences are illustrated by the following interactions:

\begin{verbatim}
> (eq? '(a b) '(a b))
#f
> (let ((listy '(a b))) (eq? listy listy))
#t
\end{verbatim}

Problem 16.29

Define a function, called \texttt{replace-by-depth*}, that satisfies the following contract:

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

Here are some examples:

\begin{verbatim}
> (replace-by-depth* '(a (b ((c) d) e) (f)))
(1 (2 ((4) 3) 2) (2))
> (replace-by-depth* '(((x))))
(((3)))
\end{verbatim}

\textit{Hint:} Define a helper function that includes an extra input, \texttt{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 \texttt{curr-depth}. \textbf{When should the value of curr-depth be increased?}
Chapter 17

Iteration

Problem 17.1: Iterative versions of \texttt{insert} and \texttt{isort}

Recall the \texttt{insert} and \texttt{insertion-sort} functions defined in Section 16.5.1. First, define an iterative version of the \texttt{insert} function, called \texttt{insert-iter}.

⇒ *Hint:* It may help to review the accumulator-based \texttt{insert-acc} function discussed in In-Class Problem 16.5.3.

⇒ *Hint:* Use \texttt{let} to create a local variable acc that is initially empty. Then use a while loop to accumulate all of the numbers of the sorted list that are smaller than item. After the while loop, transfer all of the accumulated numbers back onto the sorted list... but with item in its proper place!

Next, define an iterative version of the \texttt{insertion-sort} function, called \texttt{isort-iter}, that uses \texttt{insert-iter} as a helper function.

⇒ *Hint:* Use \texttt{let} to create a local variable called sorted that is initially empty, then use \texttt{dolist} to insert each element of the unsorted listy into the sorted accumulator.

Problem 17.2: Iterative versions of \texttt{merge} and \texttt{split}

Recall the \texttt{merge} and \texttt{split} functions described in Section 16.5.2. Define iterative versions of these functions, called \texttt{merge-iter} and \texttt{split-iter}, respectively.

⇒ *Hint:* For \texttt{merge-iter}, use \texttt{let} to create a local variable called acc, then use a while loop to accumulate numbers from the two input lists, as long as both lists remain non-empty. After the while loop, transfer all of the accumulated numbers onto whichever of the two input lists is non-empty. (Why?)

⇒ *Hint:* For \texttt{split-iter}, use \texttt{let} to create local variables called lefty and righty, that are both initially empty. Then use a while loop to process the numbers in the input list, listy. On each iteration, push one element of listy onto lefty, and another onto righty. Beware the case where listy only has one element!

Note: The Merge Sort algorithm is inherently recursive (why?); so we don’t attempt to implement an interactive version of it. However, you could define a version that uses \texttt{merge-iter} and \texttt{split-iter}.
Problem 17.3: More fun with iteration

(A) Recall the is-elt-of? predicate from Example 16.2.3, which is similar to the built-in member function from and the index-of function from Example 16.2.4. Use a while loop to define an iterative version of the member function, called member-iter.

(B) Recall the fetch-nth-element function from In-Class Problem 16.2.3, which is equivalent to the built-in list-ref function. Use a while loop to define an iterative version, called list-ref-incr.

(C) Recall the print-n-dashes function from Example 14.2.1. Use dotimes to define an iterative version, called print-n-dashes-iter.

(D) Recall the print-rectangle function from Problem 14.6. Define an iterative version, called print-rectangle-iter, that does not use the print-dashes-iter function, but instead uses two dotimes loops, one nested inside the other. The outer loop should control the number of rows, the inner loop should control the number of dashes printed in each row.

(E) Recall the print-histy function from Example 16.2.6. Define an iterative version that uses dolist to walk through the given input list and, for each number in that list, uses dotimes to print out the desired number of asterisks.

(F) Recall the list-of-n-random-numbers function from In-Class Problem 16.5.2. Use dotimes to define an iterative version, called list-of-n-random-numbers-iter. (Use let to define a local variable called acc, initially empty.)

(G) Recall the function, num-occurs-in-n-tosses, from Problem 15.3. Define an iterative version, called num-occurs-in-n-tosses-iter, that uses a dotimes special form and an accumulator.

(H) Recall the function, toss-until-doubles, from Problem 15.7. Define an iterative version using a while loop.

Problem 17.4: Iteratively approximating logarithms

Recall the approx-log-acc and approx-log-wr functions from Problem 14.13.

(A) Define an iterative function, approx-log-iter, that takes the same inputs as approx-log-wr. It should use let to create local variables named from and acc, with appropriate initial values. Then it should use a while loop to iteratively accumulate terms in the sum, $x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots \pm \frac{x^n}{n}$. Alternatively, your function could use let to create only the acc local variable, leaving the management of from to a dotimes special form—instead of a while loop.

(B) Define a more efficient version of approx-log-iter, called approx-log-better. Instead of computing each power of $x$ from scratch on each iteration, introduce an extra local variable called curr-power, whose value on successive iterations is $x$, then $x^2$, then $x^3$, etc.

To see the difference in performance, you may wish to compare the results of evaluating the following expressions:

(time (approx-log-iter 1 10000))
(time (approx-log-better 1 10000))
Problem 17.5: Iteratively approximating $\pi$

Recall the $\text{approx-pi-acc}$ and $\text{approx-pi-wr}$ functions from Examples 14.3.5 and 14.4.2, respectively. Define an iterative function called $\text{approx-pi-iter}$ that, like $\text{approx-pi-wr}$, takes a single input $n$. Your function should use `let` to create local variables called `from`, `sign` and `acc`, with suitable initial values. It should then use a `while` loop to accumulate the desired terms in the sum $4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} \ldots \pm \frac{4}{n}$.

Problem 17.6: Iteratively approximating $e$

(A) Recall the $\text{approx-e-acc}$ and $\text{approx-e-wr}$ functions from Examples 14.3.6 and 14.4.3, respectively. Define an iterative function called $\text{approx-e-iter}$ that, like $\text{approx-e-wr}$, takes a single input $n$. Your function should use `let` to create local variables called `indy`, `curr-denom` and `acc`, with suitable initial values. It should then use a `while` loop to accumulate the desired terms in the sum, $1 + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$. Alternatively, your function could use `let` to create only `curr-denom` and `acc`, leaving the management of `indy` to a `dotimes` loop (instead of using `while`).

(B) More generally, mathematicians tell us that the function $e^x$ is very well approximated by sums of the form, $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$. Define an iterative function called $\text{approx-e-x-iter}$, that computes such sums. It should be similar to $\text{approx-e-iter}$, except that:

- It takes an extra input $x$, that can be any real number; and
- It creates an extra local variable `curr-power`, whose values on successive iterations is 1, then $x$, then $x^2$, etc.

You may wish to compare expressions such as $(\text{approx-e-x-iter} \ x \ n)$ to $(\exp \ x)$. The built-in `exp` function approximates the value of $e^x$. 
Chapter 18

Vectors

Problem 18.1

Define a (destructive) function that satisfies the following contract.

;;; DOUBLE-ALL!
;;; -----------------------------------------------
;;; INPUT: VECKY, a vector of numbers
;;; OUTPUT: The same vector, modified as described below
;;; SIDE EFFECT: Doubles the contents of each slot (destructively)

> (define vecky (vector 10 20 30 40 50))
> vecky
#(10 20 30 40 50)
> (double-all! vecky)
#(20 40 60 80 100)
> vecky
#(20 40 60 80 100)

Problem 18.2

Define a (destructive) function, called roll-em!, that satisfies the following contract:

;;; ROLL-EM!
;;; -----------------------------------------------
;;; INPUT: DICE, a vector of numbers
;;; OUTPUT: The same vector, but with contents destructively
;;; modified, as follows
;;; SIDE EFFECT: Replaces each slot with a random toss of a
;;; six-sided die

> (define dice (make-vector 5))
> (roll-em! dice)
#(3 2 5 3 4)
> dice
#(3 2 5 3 4)
> (roll-em! dice)
#(1 6 1 5 5)
> dice
#(1 6 1 5 5)

*Hint: Use the *toss-die* function from the previous chapter.*

## Problem 18.3

**Define a (destructive) function called *roll-some!* that satisfies the following contract:**

```scheme
;; ROLL-SOME!
;; --------------------------------
;; INPUTS: DICE-VECK, a vector of dice values
;; ROLLER, a vector of the same length as DICE-VECK,
;; but consisting solely of 1s and 0s
;; OUTPUT: DICE-VECK, modified as described below
;; SIDE EFFECT: Walks through the two input vectors in parallel.
;; For each index I, if the Ith element of ROLLER is a 1, then
;; the Ith element of DICE-VECK is replaced by a random toss
;; of a 6-sided die; otherwise, it is unchanged.
```

*Here are some examples:*

```scheme
> my-dice
#(1 2 1 6 6 3 2 3)
> (roll-some! my-dice #(1 1 1 0 0 1 1 1))
#(5 4 2 6 6 3 4)
> (roll-some! my-dice #(1 1 1 0 0 0 1 1))
#(1 6 3 6 6 2 2)
> (roll-some! my-dice #(1 0 1 0 0 0 1 1))
#(4 6 5 6 6 1 6)
> my-dice
#(4 6 5 6 6 1 6)
```

## Problem 18.4

**Define a non-destructive function, called *vector-reverse*, that satisfies the following contract. Because it is non-destructive, it must create a new vector, instead of modifying the given vector.**

```scheme
;; VECTOR-REVERSE
;; --------------------------------
;; INPUT: VECK, a vector
;; OUTPUT: A *new* vector that is just like VECK, except
;; that its elements are in the reverse order.
;; SIDE EFFECTS: none
```

*Here are some examples:*

```scheme
> (vector-reverse #(a b c d))
#(d c b a)
> (vector-reverse #(1 2 3))
#(3 2 1)
```
Notice that the inputs in the above examples are immutable! So, if the function had tried to modify them, it would have caused an error!

Hints: Create a new vector of the appropriate length. Then use `do` times to walk thru the vector, setting its elements to appropriate values. Recall the `print-in-reverse` function from Example 18.4.3 for ideas. Don’t forget to return the new vector as output.

Problem 18.5

Define a (destructive) function that satisfies the following contract:

```scheme
;; VECTOR-REVERSE!
;; -------
;; INPUT: VECKY, a vector
;; OUTPUT: The same vector, modified as described below
;; SIDE EFFECT: Destructively reverses the order of the
;; elements in VECKY.

> (vector-reverse! (vector 1 2 3 4 5))
#(5 4 3 2 1)
> (define vecky (vector 'a 'b 'c 'd))
> vecky
#(a b c d)
> (vector-reverse! vecky)
#(d c b a)
> vecky
#(d c b a)
```

Problem 18.6

Define a non-destructive function that satisfies the following contract:

```scheme
;; VECTOR-MAP
;; -------
;; INPUTS: FUNK, a function that expects one input
;; VEKK, a vector of suitable inputs for FUNK
;; OUTPUT: A *new* vector of the same length as VEQUE
;; each of whose elements is obtained by applying FUNK to
;; the corresponding element of VEKK.
;; SIDE EFFECT: *NONE*

Here is an example that assumes that the `facty` function has already been defined.

> (define vek #(1 2 3 4 5)) ← #(1 2 3 4 5) is an immutable vector
> (vector-map facty vek)
#(1 2 6 24 120) ← New vector created by vector-map
> vek
#(1 2 3 4 5)
```

Notice that `vek` has been defined to be an immutable vector (i.e., its contents can’t be changed) and, thus, even if `vector-map` wanted to change its contents, it could not. (Attempting to do so would cause an error.) After `vector-map` is finished, `vek` remains the same (i.e., `vector-map` is non-destructive).
Hints: Use make-vector to create a new vector of the same length as vek. Then use dotimes to walk thru the new vector, setting each element to the result obtained by applying funk to the corresponding element of vek.

Problem 18.7

Define a destructive function that satisfies the following contract:

;;; VECTOR-MAP!
;;; ----------------------------------------------
;;; INPUTS: FUNK, a function that expects one input
;;; VEQUE, a vector of suitable inputs for FUNK
;;; OUTPUT: VEQUE, destructively modified...
;;; SIDE EFFECT: Destructively modifies VEQUE by replacing
;;; each of its elements by the result of applying FUNK to
;;; the corresponding element of VEQUE.

Here is an example:

> (define vec (vector 1 2 3 4 5)) ← VEC is a mutable vector
> (vector-map! facty vec)
#(1 2 6 24 120) ← VECTOR-MAP! does its thing
> vec
#(1 2 6 24 120) ← VEC has been changed!

Recall that the built-in vector function creates a mutable vector (i.e., one whose contents can be changed). vector-map! walks through the input vector, destructively modifying its contents. Afterward, vec is shown to be modified. Thus, vector-map! is destructive!

Hints: No need to create a new vector. Just use dotimes to walk through the given vector, destructively modifying its contents as you go.

Problem 18.8

Define a destructive function that satisfies the following contract: Define a function that satisfies the following contract:

;;; VECK-REPLACE!
;;; -----------------------------------------------
;;; INPUTS: OLD, anything
;;; NEW, anything
;;; VECKY, a vector
;;; OUTPUT: VECKY, destructively modified as follows.
;;; SIDE EFFECT: Replaces each occurrence of OLD in VECKY
;;; by NEW, where equality is as judged by EQ?.

Here is an example:

> (define vequi (vector 1 2 1 3 1 4)) ← VEQUI is mutable
> vequi
#(1 2 1 3 1 4)
> (veck-replace! 1 111 vequi)
Problem 18.9

Define a non-destructive function, called \texttt{veck-index-of}, that satisfies the following contract:

\begin{verbatim}
;; VECK-INDEX-OF
;; -----------------------------------------------
;; INPUTS: ITEM, anything
;; VECKY, a vector
;; OUTPUT: The index of the first slot of VECKY that contains
;; an first occurrence of ITEM; or #f if ITEM does
;; not appear in VECKY.
\end{verbatim}

Here are some examples of its use:

\begin{verbatim}
> (veck-index-of 'x #(a b c x y z x x))
3
> (veck-index-of 'z #(a b c x y z x x))
5
> (veck-index-of 'w #(a b c x y z x x))
#f
\end{verbatim}

Hint: Define a helper function, called \texttt{veck-index-of-helper}, that includes an extra input \texttt{I}, that serves as an index into the given vector. Make recursive function calls, incrementing \texttt{I}, until you find an occurrence of \texttt{ITEM}, or \texttt{I} gets too big. Here’s the contract for the helper function:

\begin{verbatim}
;; INDEX-OF-HELPER
;; -------------------------------
;; INPUTS: ITEM, anything
;; VECKY, a vector
;; I, a numerical index
;; OUTPUT: The index of the slot of VECKY that contains the
;; first occurrence of ITEM at or after index I;
;; or #f if ITEM does not appear in VECKY.
\end{verbatim}

Problem 18.10: Converting a vector into a list

The goal of this problem is to define a function, called \texttt{veck-to-list}, that takes a vector as its only input, and returns as its output a list containing the same elements, in the same order, as illustrated below:

\begin{verbatim}
> (veck-to-list #(a b c d))
(a b c d)
> (veck-to-list (make-vector 5))
(0 0 0 0 0)
> (veck-to-list (vector 1 #t ()))
(1 #t ())
\end{verbatim}

Here is the contract for \texttt{veck-to-list}:
Given the tools that we have seen so far, for this problem, it is probably easiest to use \texttt{do\texttt{times}} to walk through the vector, accumulating each element onto a list accumulator. Although the resulting accumulator would come out backwards, you could reverse it in the end. Alternatively, you could use an approach like that seen in Example 18.4.3 with the \texttt{print-in-reverse} function to walk through the vector in the reverse order so that the list accumulator would come out in the correct order, without needing to be reversed after the fact. Incidentally, after you have implemented this function, you can compare it to the built-in function, \texttt{vector->list}, that does the same thing!

Problem 18.11: Converting a list into a vector

The goal of this problem is to define a function, called \texttt{list-to-veck}, that takes a list as its only input, and returns as its output a vector containing the same elements, in the same order, as illustrated below.

\begin{verbatim}
> (list-to-veck '(a b c d))
(a b c d)
> (list-to-veck (list 1 #t ()))
(1 #t ())
\end{verbatim}

Here is the contract for \texttt{list-to-veck}:

\begin{verbatim}
;; LIST-TO-VECK
;; ---------------------------------
;; INPUT: LISTY, a list
;; OUTPUT: A vector containing the same elements as LISTY,
;; and in the same order
;; SIDE EFFECTS: none
\end{verbatim}

\textbf{Hint:}

\begin{verbatim}
* Start by creating a vector of the same length as the input list. Then use \texttt{dolist} to walk through the list. You’ll need a separate index (or counter) variable to keep track of which slot of the vector you’re working on, but that’s okay.
\end{verbatim}

Incidentally, after you have finished defining this function, you may want to compare it to the built-in \texttt{list->vector} function that does the same thing!

Problem 18.12

Define a non-destructive function, called \texttt{every-other-one-vector}, that satisfies the following contract.

\begin{verbatim}
;; EVERY-OTHER-ONE-VECTOR
;; ---------------------------------
;; INPUT: VECKY, a vector
;; OUTPUT: A vector containing every other element of VECKY.
;; Note: The output vector should contain roughly half the
\end{verbatim}
Here are some examples of its behavior:

> (every-other-one #(0 1 2 3 4 5 6))
#(0 2 4 6)
> (every-other-one #(1 2 3 4 5 6))
#(1 3 5)

Hints:

⋆ Create a new vector whose length is roughly half that of the input vector. Then use dotimes to walk through that new vector, copying relevant elements from the input vector to the new vector.

⋆ You may find the built-in even?, odd?, or quotient functions helpful. (even? and odd? were introduced in Problem 14.2; quotient was introduced in Section 5.3.)

⋆ You should not use lists for this function; use vectors!

---

Problem 18.13: Computing a histogram

Define a function, called compute-histogram, that satisfies the following contract:

;;; COMPUTE-HISTOGRAM
;;; -----------------------------------------------
;;; INPUT: VECK-O-DICE, a vector of dice values (each 1 thru 6)
;;; OUTPUT: A vector of length 7, where the ith slot contains
;;; the number of occurrences of i in VECK-O-DICE.
;;; (The 0th slot of the output vector is ignored.)

Here's an example:

> (define my-dice #(1 2 1 2 6 6 6 5 2))
> (compute-histogram my-dice)
#(0 2 3 0 0 1 3)

In this case, my-dice contains:

two 1s
three 2s
no 3s
no 4s
one 5
three 6s

These counts are reflected in the histogram computed by this function. Note that the histogram is a vector with seven slots, numbered 0 thru 6. The zeroeth slot is ignored. We only care about slots 1 thru 6. For each i > 0, the slot of the output vector at index i contains the number of occurrences of i in my-dice (or veck-o-dice). Here's another example:

> (define my-dice #(3 3 3 3 1 1 1 1))
> (compute-histogram my-dice)
#(0 4 0 5 0 0 0)
Problem 18.14

* You may wish to review Problem 16.6 before starting this problem.

Define a function, called veck-has-satisfier?, that satisfies the following contract:

```scheme
;; VECK-HAS-SATISIFIER?
;; -----------------------------------
;; INPUTS: FUNK, a predicate that expects one input
;; VECK, a vector of suitable inputs for FUNK
;; OUTPUT: #t, if VECK contains an element that satisfies
;; FUNK (i.e., that makes FUNK return #t)
;; #f, otherwise.
```

Here are some examples:

```scheme
> (veck-has-satisfier? number? #(a #t 3 x))
#t
> (veck-has-satisfier? symbol? #(1 2 3 4))
#f
```

Note that it would be inefficient to implement this function using dotimes because dotimes invariably walks through the entire vector. This function should stop as soon as it finds an element of veck that satisfies funk. Therefore, you should define a recursive helper function that takes an extra input, i, that serves as an index into the vector veck.

Problem 18.15

Define a function, called has-three-of-a-kind?, that satisfies the following contract:

```scheme
;; HAS-THREE-OF-A-KIND?
;; -----------------------------------
;; INPUT: VECK-O-DICE
;; OUTPUT: #t if the vector of dice contains *at least* three of one kind; #f otherwise.
```

Here are some examples:

```scheme
> (has-three-of-a-kind? #(1 2 1 2 1)) ← has three ones
#t
> (has-three-of-a-kind? #(4 2 4 4 4)) ← has four fours
#t
> (has-three-of-a-kind? #(6 6 6 6 6)) ← has five sixes
#t
> (has-three-of-a-kind? #(6 5 6 5 2)) ← does not have three of a kind
#f
```
Notice that having four or five of a kind also counts as having three of a kind.

Hint: One way to solve this problem: Compute a histogram vector, then check whether it has an element that is 3 or bigger. Can you think of a way to use \texttt{veck-has-satisfier} from Problem 18.14 to check whether the histogram vector contains an element that is 3 or bigger?

**Problem 18.16**

Define a function, called \texttt{has-large-straight?}, that satisfies the following contract:

\[
\begin{align*}
\text{;;;; HAS-LARGE-STRaight?} \\
\text{;;;; -----------------------------------------------} \\
\text{;;;; INPUT: VECK-O-DICE, a vector of five dice values} \\
\text{;;;; OUTPUT: #t if the vector of dice contains all of the} \\
\text{;;;; numbers in \{1,2,3,4,5\} or \{2,3,4,5,6\}, in any order;} \\
\text{;;;; #f otherwise.}
\end{align*}
\]

Here are some examples:

\[
\begin{align*}
\text{> (has-large-straight? #(2 4 5 3 1))} \\
\text{#t} \\
\text{> (has-large-straight? #(6 5 4 2 3))} \\
\text{#t} \\
\text{> (has-large-straight? #(6 4 1 2 3))} \\
\text{#f}
\end{align*}
\]

Hints: You may assume that the input vector has exactly five slots. Compute a histogram and go from there! What must the histogram look like for a large straight? (Use the built-in \texttt{equal?} predicate (cf. In-class problem 18.4.2) to make your life easier!) Alternatively, convert the vector of dice into a list, then use the built-in \texttt{sort} function (cf. Example 16.5.6) to sort its contents into non-decreasing order. What must it look like at that point?
Chapter 19

Data Structures

No problems for this chapter.
Chapter 20

The Model-View-Controller Paradigm

No problems for this chapter.