This lab gives you supervised practice with the idea that “templates plus functional examples equals code.” There are five problems, and I would like you to do two or three of them. Two of the problems are repeats from an earlier lab; do one of these problems only if neither of you has done it already.

For each problem, carry out these steps of the design recipe:

- If they are not provided, write a signature, purpose statement, and header.
- If they are not provided, write functional examples.
- Write a template based on the input data.
- For each non-trivial case in each template, use the functional examples to fill in a table of the examples. If, after putting all the examples in the table, you are not yet confident that you know how to complete the function, create more functional examples (or summon the lab staff).
- Show your table to the lab staff.
- Write a simple explanation of how you would finish the problem. If a helper function seems called for, write a signature, purpose statement, and header.
- In any table, put a row of XXXXX’s below any column that you expect not to use in your solution.
- Move onto another problem without finishing the code.

If you complete tables for three problems, you may then choose to finish one.

What tables should look like

Here is the structure of a table of examples:

- The first column should be labeled WANTED and should show the answer that the function is trying to return. This column should come straight from the result of your check-expect in your functional example.
- You may find it useful to put a column for each input.
- There should be a column for each application of a selector function, even if you think the selector function won’t be used in the final code. Each of these columns can be filled in by looking at the check-expect and applying the column’s selector function to the input from the check-expect.
- There should be a column for each natural recursion, even if you think the recursion won’t be used in the final code. This column is the most difficult to fill in, because you must consider the input and the purpose statement.
- If you feel you need a helper function, you may find it useful to add a column for the results you expect it to return. You can then use that column to help you create functional examples for the helper.
- There should be a row for each functional example.
Example problem I: Summing natural numbers

Data definition:

A natural is one of:

- \( 0 \)
- \((\text{add1 } n)\) where \( n \) is a natural

Here is a header, purpose statement, functional examples, and template:

; sum-to : natural -> number
; to return the sum of the natural numbers from zero to \( n \) inclusive
(define (sum-to n)
  (cond [[(zero? n)] ...
        [(positive? n) (... (sub1 n) (sum-to (sub1 n)))]])

(check-expect (sum-to 0) 0)
(check-expect (sum-to 1) 1)
(check-expect (sum-to 4) 10)
(check-expect (sum-to 5) 15)

It is not worth making a table for the zero? case.
The table for the positive? case has one selector function and one natural recursion. It looks like this:

<table>
<thead>
<tr>
<th>WANTED</th>
<th>n</th>
<th>(sub1 n)</th>
<th>(sum-to (sub1 n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 1: summing to positive natural numbers

Explanation of combination: The case should combine \( n \) and \((\text{sum-to (sub1 n)})\) by adding them. The selection \((\text{sub1 n})\) need not be used except in the natural recursion.

Example problem II: Reversing a list

Here is a header, purpose statement, functional examples, and template:

; my-reverse : (listof number) -> (listof number)
; to reverse the given list
(define (my-reverse ns)
  (cond [[(empty? ns)] ...
        [(cons? ns) (... (first ns) (rest ns) (my-reverse (rest ns)))]])

(check-expect (my-reverse '()) '())
(check-expect (my-reverse '(1 2)) '(2 1))
(check-expect (my-reverse '(1 2 3 4 5)) '(5 4 3 2 1))

2
It is not worth making a table for the empty? case.

The table for the cons? case has two selector functions and one natural recursion. It looks like this:

<table>
<thead>
<tr>
<th>WANTED</th>
<th>ns</th>
<th>(first ns)</th>
<th>(rest ns)</th>
<th>(my-reverse (rest ns))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'(2 1)</td>
<td>'(1 2)</td>
<td>1</td>
<td>'(2)</td>
<td>'(2)</td>
</tr>
<tr>
<td>'(5 4 3 2 1)</td>
<td>'(1 2 3 4 5)</td>
<td>1</td>
<td>'(2 3 4 5)</td>
<td>'(5 4 3 2)</td>
</tr>
</tbody>
</table>

Table 2: reversing nonempty lists

Explanation of combination: The case should combine (first ns) and (my-reverse (rest ns)) by appending the two lists (my-reverse (rest ns)) and (list (first ns)) in that order.

The real problems

1. **Symmetric substitution cipher.**
   A substitution cipher (weakly) hides a message by substituting one letter for another. The cipher is symmetric if doing the same substitution twice reproduces the original message. Such ciphers are easy to break, but if all you want to do is keep people from stumbling over information accidentally, they are useful.

   Here is a diagram of a popular substitution cipher:

   Decryption Key
   A|B|C|D|E|F|G|H|I|J|K|L|M
   -------------------------
   N|O|P|Q|R|S|T|U|V|W|X|Y|Z
   (letter above equals below, and vice versa)

   Define a function substitute-char that takes as arguments a 1-character string c and two lists of 1-character strings top and bottom. The lists top and bottom together may be assumed to contain no duplicates.

   • If the string c appears in the list top, the function should return the corresponding string from the list bottom.
   • If the string c appears in the list bottom, the function should return the corresponding string from the list top.
   • If the string c appears in neither top nor bottom, the function should return c.

   Here are a signature, purpose statement, and header:

   ```scheme
   ;; substitute-char : 1-char (listof 1-char) (listof 1-char) -> 1-char
   ;; to return the character paired with c in the two rows,
   ;; or if c does not appear, to return c itself
   (define (substitute-char c top bottom)
      ...)
   ```

   Here are some functional examples:

   ```scheme
   (check-expect (substitute-char "H" (explode "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"))
    "U")
   (check-expect (substitute-char "H" (explode "NOPQRSTUVWXYZnopqrstuvwxyz"))
    "U")
   ```
Lists are all very well, but for people it is easier to work with strings. To finish this problem, define a function `two-row-cipher` that takes three arguments: a cleartext string, a top-row string, and a bottom-row string. The function converts the cleartext to a corresponding ciphertext by applying the substitution to each character; it returns the ciphertext, also as a string. I recommend that you use `substitute-char` to substitute for each character of the cleartext string, and that you use BSL functions `explode` and `implode`.

Example:

```scheme
(check-expect (two-row-cipher "Fishy" "ABCDEFGHIJKLMabcdefghijklm" "NOPQRSTUVWXYZnopqrstuvwxyz") "Svful")
```

Use your function to decrypt the following message, which contains an observation made by a keen student of human nature:

```
Vg vf n gehgu havirefnyyyl npxabjyrqtrq, gung n fvatyr zna va cbffrvba bs n tbbq sbeghar zhfg or va jnag bs n jvsr.
```

2. **Anagrams.**

Two words are anagrams if they are made from exactly the same letters. Here are functional examples:

- "fowl" and "wolf" are anagrams
- "fowl" and "foul" are not anagrams
- "read" and "dare" are anagrams
- "read" and "dared" are not anagrams
- "read" and "red" are not anagrams
- "stropped" and "posterated" are not anagrams

Define a function that takes as arguments two lists of 1strings and returns a Boolean saying if they are anagrams.

```scheme
;; anagrams-lists? : (listof 1string) (listof 1string) -> boolean
;; tells if the two given lists are made from exactly the same letters
(define (anagrams-lists? word1 word2) ...)
```

3. **Bowling pins.**

Bowling pins are laid out in rows with one pin in the first row, two in the second, and so on. The standard layout has four rows, roughly as follows:

```
*   *   *   *
*   *   *
*   *
*   *
```

4
The number of rows in a layout is a natural number:

A natural is one of:

• 0
• (add1 n) where n is a natural

When you are working with natural numbers you check conditions using BSL functions zero? and positive?, and when you get a positive natural, you use sub1 as you would use a selector function.

Using the 2htdp/image teachpack, design a function that takes as input a number of rows and returns an image of a bowling-pin layout with the given number of rows. You’ll find use for the functions beside, above, circle, and overlay.

Domain knowledge: you can get a pretty good layout by representing a single pin as a black circle laid on top of a somewhat larger white circle.

4. Removing duplicates.
Define a function nub which removes duplicates from a list of numbers. Your function may remove any duplicate numbers it likes, but the order of non-duplicate numbers must not be disturbed.

Here are some functional examples:

(check-expect (nub '(1 2 3 4 5)) '(1 2 3 4 5))
(check-expect (nub '(1 2 3 4 5 5 5)) '(1 2 3 4 5))
(check-expect (nub '(2 1 2 4 5 2)) '(2 1 4 5)); chose to remove last duplicates

5. Scrabble words.
If you have played Scrabble or Words With Friends, you know that you are given tiles with letters on them and you try to make words. To teach a computer to play Scrabble, one problem we have to solve is to figure out what words it can make. Here is a potentially useful function:

;; can-make-with? : string string -> boolean
;; to tell if the given word can be made with the given tiles
(define (can-make-with? word tiles)
  (can-make-with-lists? (explode word) (explode tiles)))

(check-expect (can-make-with? "frog" "asdlflor") false)
(check-expect (can-make-with? "frog" "gsdlflor") true)

Design the function can-make-with-lists? all the way through a template and a table of examples.

Submitting the lab

Ten minutes before the end of the lab, put the following text at the beginning of a DrRacket file. You may use an empty file or the source code you have been working on:

#| What I did during this lab:
  (you fill in this part)

What I learned during this lab:
  (you fill in this part)

| #
The lab staff will help you articulate what you learned.

Finally, submit this file through the handin server as `lab-templates`. You will need to submit it using two usernames connected by a + sign, as in

```
Jane.Doe+Richard.Roe
```

You submit using Jane Doe’s password.