# CS116 Winter 2011 Assignment 5
# Due: Tuesday, February 15 at 10:00AM

The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.

**Important Information:**

1. Read the course Style Guide for Mutation in Scheme (Appendix B in the course notes) in addition to the Style Guide (Appendix A) for information on assignment policies and how to organize and submit your work. In particular, your solutions should be placed in files `a5qY.rkt`, where `Y` is a value from 1 to 4.
2. Be sure to download the interface file to get started.
3. Do not copy the purpose directly from the assignment description – it should be in your own words.
4. You should use abstract list functions (`map, filter, foldr, build-list`) where appropriate. Using explicit recursion is also allowed.
5. All helper functions and constants should be contained within the main function definition using `local` or `lambda.` When a **short and simple** function is used only once as a helper to an abstract list function, you must use lambda. For more complex helper functions, you may define them separately for readability (recall that locally defined functions require a contract and purpose).
6. You may assume that all consumed data will satisfy any stated assumptions, unless indicated otherwise.

**Language level:** Advanced Student.
**Coverage**: Module 5

**Note:** While the length of this assignment is significantly longer than the length of the previous assignments seen so far in CS116, the actual amount of coding is not. Do not be discouraged by the length of the questions.

## Question 1: Assignment and Mutation

This question uses the following structure from Assignment 1:

```
(define-struct tweet (sender message))
;; A tweet is a structure (make-tweet s m) where
;; s is a string for the sender's name,
;; m is a string for the sender's message
;;   (with maximum length 140 characters).
```

**(Part A)** Complete the Scheme function `first-mutate`, which assigns values to all the variables as shown in the memory model of Figure 1A given below:

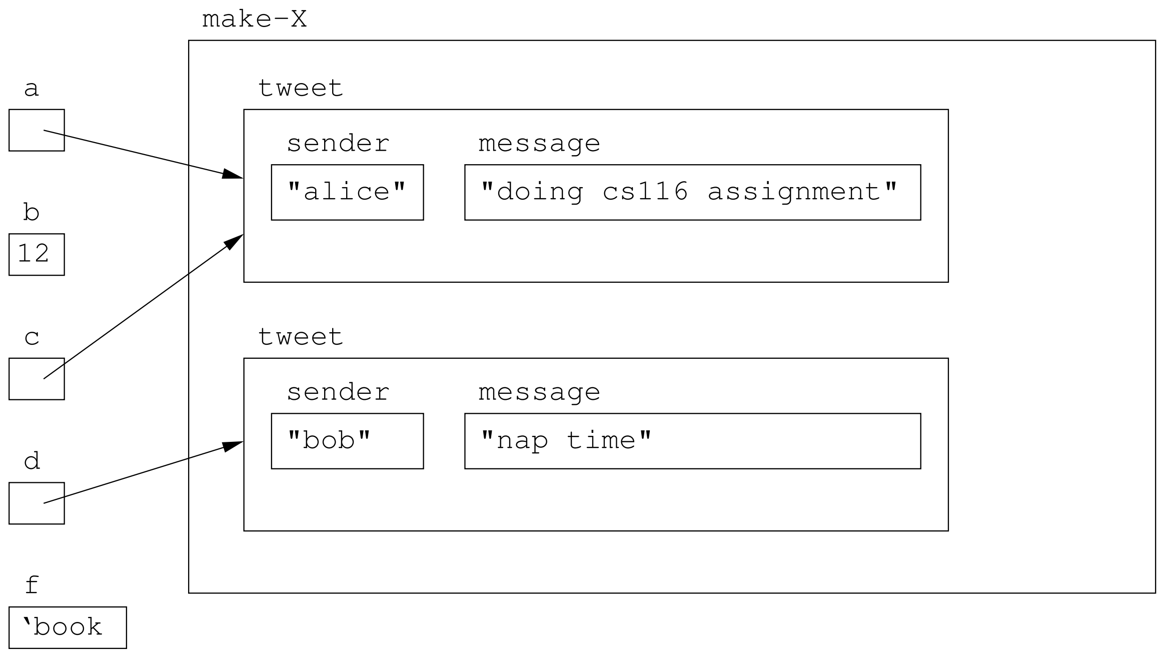

Figure 1A.

All of the variables are defined in the interface file. You must use not re-define any of them in your solution. You may use `make-tweet` as many times as you find necessary. The function `first-mutate` will consume no arguments and will produce `(void)`.

**(Part B)** Complete the Scheme function `second-mutate`, which modifies the state variables as defined in part (A) to the values shown in the memory model of Figure 1B given below:

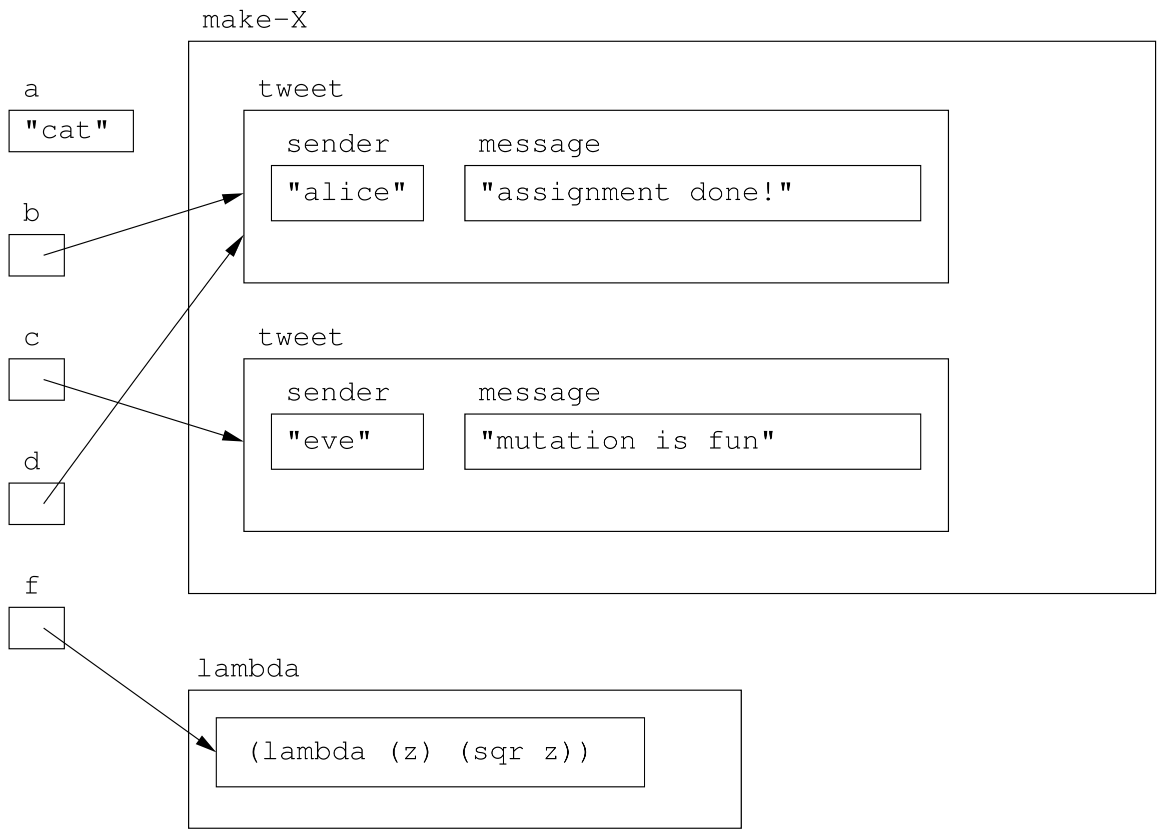

Figure 1B.

You are **not** allowed to use `make-tweet` in your solution and you are not allowed to define any new variables. Assume that part (A) works correctly. We will test your function `second-mutate` using a correct version of `first-mutate`.

Note: for both part (A) and part (B), when following the design recipe you only need to provide one example and one test for each part. (That is, one example and test for part (A) and one example and test for part (B).)

## Question 2: A Taste of Linked Lists

In this problem we will use the following structure:

```
(define-struct node (value next))
;; A node is a structure(make-node v n)
;; where v is an integer
;; and n is either another node or the
;; symbol 'end
```

Consider the state variables `a-linked-list` and `a-node` defined in the memory map given in Figure 2 below:

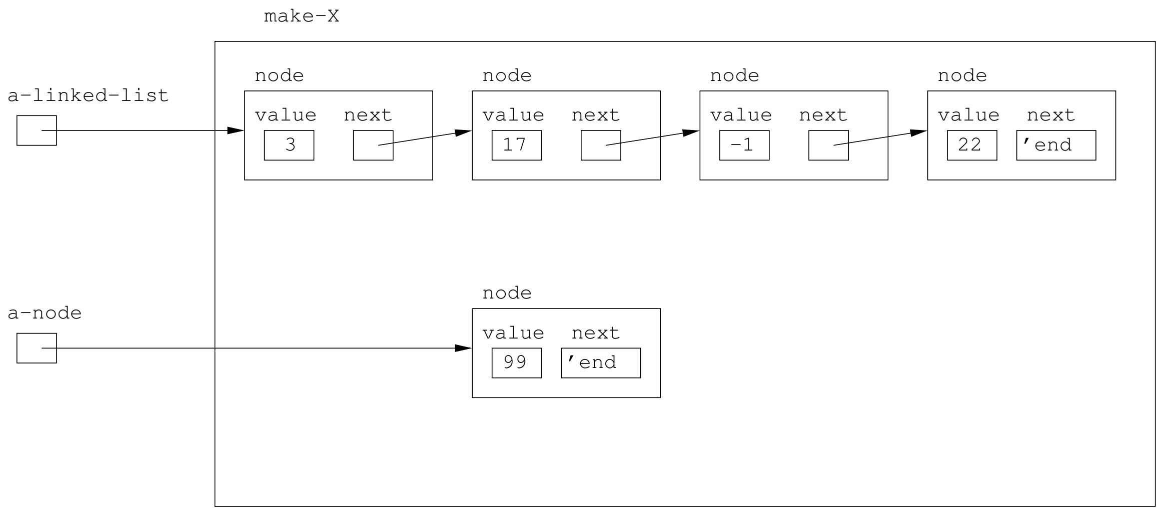

Figure 2.

We say that `a-linked-list` and `a-node` are linked lists (`a-node` is a linked list with a single node.) A linked list can be used to implement Scheme's `list` data structure. For example, `(node-value some-linked-list)` simulates Scheme's `(first)` function and `(node-next some-linked-list)` simulates Schemes `(rest)` function.

In this problem, you will complete three Scheme functions: two that add a node to a linked list and one that removes a node from a linked list.

**(Part A)** Assume that `a-linked-list` and `a-node` are initially defined as in Figure 2. Complete the Scheme function `add-a-node`, which adds the node `a-node` to the linked list `a-linked-list` in the 3rd position and produces a list containing all the `node-value` values of the nodes in `a-linked-list` after `a-node` has been added, in the same order that they appear when you visit the nodes in order.

For example, with `a-linked-list` and `a-node` defined as above in Figure 2,

```
(add-a-node) ⇒ (list 3 17 99 -1 22)
```

and the state variables should correspond to the memory map in Figure 2A.

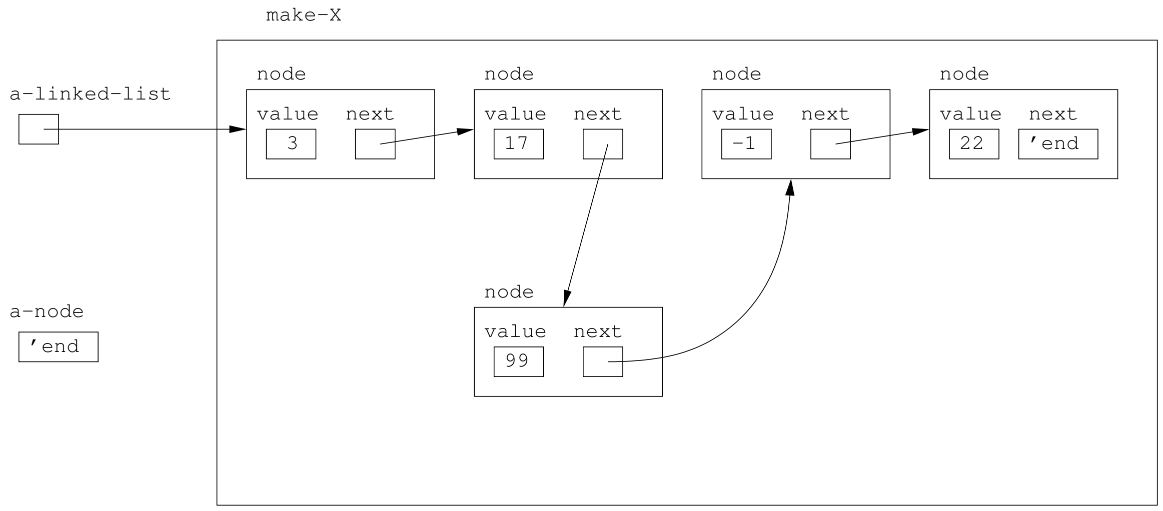

Figure 2A

The function `add-a-node` consumes no parameters and produces the list of integer values as described above. The function has two effects: the first effect is that the node `a-node` is added to the linked list `a-linked-list` in the 3rd position and the second effect is that the state variable `a-node` is set to the symbol `'end`.

You are **not** allowed to call `make-node` in your solution to part (A). You are allowed to define one local variable to help in your solution.

Note A1: You may assume that `a-linked-list` has at least 3 nodes in it (for example, it has 4 nodes in Figure A) and that `a-node` has exactly one single node in it (for example, as in Figure A).

Note A2: When testing your function the `node-value` values in `a-linked-list` and `a-node` will be different from the values shown in the memory map. Your function should not explicitly include the values shown in the map.

**(Part B)** Complete the function `remove-second`, which removes the second node in the state variable `a-linked-list`. The function consumes no parameters and produces a list of integers that correspond to the `node-value` integers in `a-linked-list` after the 2nd node has been removed. If `a-linked-list` only had one node in it before `remove-second` is called, it simply produces the list of integers in `a-linked-list`. The effect of `remove-second` is to remove the second node from `a-linked-list` if it initially had two or more nodes in it.

For example, with `a-linked-list` defined in Figure 2A,

```
(remove-second) ⇒ (list 3 99 -1 22)
```

and the state variables after the function executes should correspond to the memory map in Figure 2B:

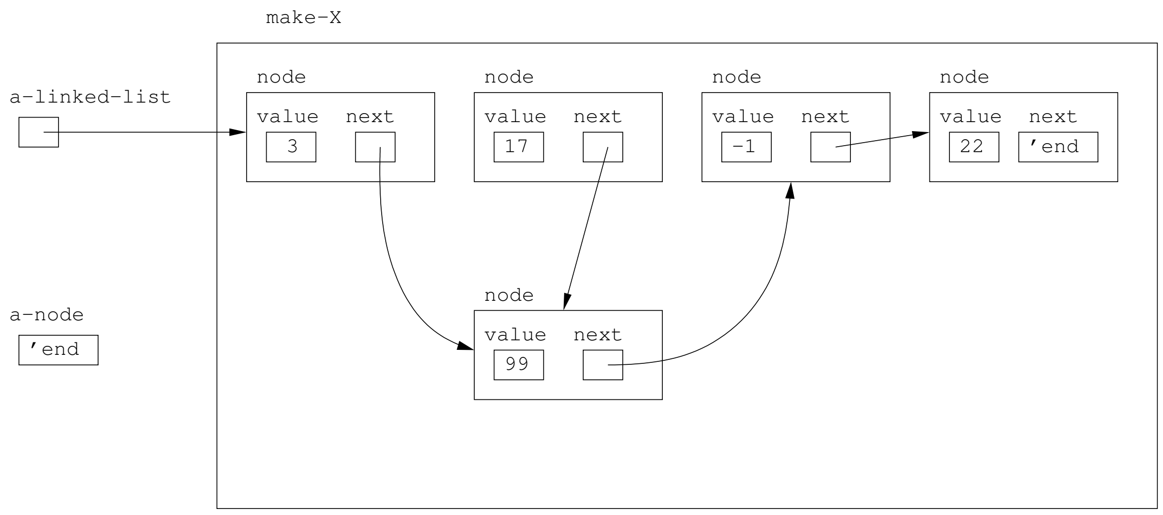

Figure 2B.

You are **not** allowed to define any new variables in this problem.

Note B: You may assume that `a-linked-list` will have at least one node in it.

**(Part C)** Complete the Scheme function `add-second`, which consumes an integer `n` and produces a list of integers. The function adds a new node with `node-value` equal to `n` in the 2nd position of `a-linked-list` and then produces a list of integers that corresponds to the integers in the nodes in `a-linked-list` after the new node has been added.

For example, with `a-linked-list` defined in Figure 2B,

```
(add-second 12) ⇒ (list 3 12 99 -1 22)
```

and the state variables should correspond to the memory map shown in Figure 2C.

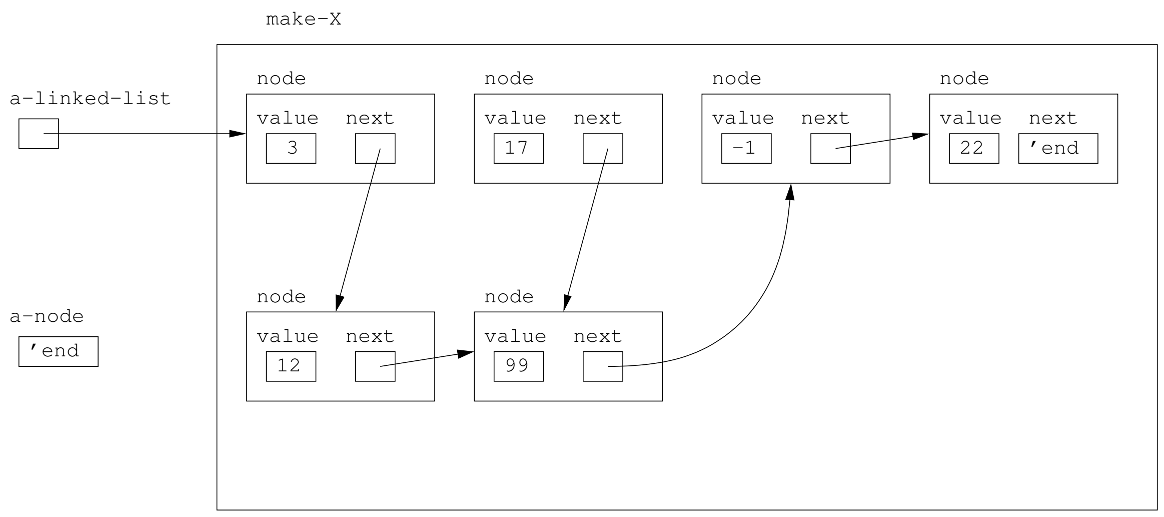

Figure 2C.

You are allowed to call `make-node` once in your solution to part (C). You are allowed to define one local variable within your function to help in your solution.

Note C: You may assume that `a-linked-list` will have at least one node in it.

## Question 3: Lions and tigers and bears, oh my!

In this problem, a state variable `words`, containing `n` strings defined by

```
 (define words (list word1 word2 ... wordn))
```

will be provided in the interface, where `word1, word2, ..., wordn` are strings. Without being allowed to create any new state variables, you will complete the Scheme function `next-word`, which consumes no parameters and produces a string as follows:

The first time `next-word` is called it must produce the string `word1`. After the first time the function is called, what `next-word` produces is defined by the following rules:

- if the previous call to `next-word` produced the string `worda`, where $1 \leq a < n$, then the next call to `next-word` must produce the string `wordb`, with `b=a+1`.
- if the previous call to `next-word` produced the string `wordn`, then the next call to `next-word` must produce the string `word1`.

You may use `set!` to change the state variable `words` in any way that you wish in `next-word`. (So the effect of `next-word` will perhaps be to modify `words`.) However, the order of the strings produced by consecutive calls to `next-word` should follow the same order of the **initial** definition of `words`. You may create any local variables that you might need inside `next-word`, but you are not allowed to create any new state variables.

For example, suppose the state variable `words` is defined in the interface file as

```
(define words (list "Lions" "Tigers" "Bears" "Oh, my!"))
```

The first time `next-word` is called it must produce the string `"Lions"`. The next four consecutive calls to the `next-word` function must produce `"Tigers", "Bears", "Oh, my!", "Lions"`, in that order. Thus,

```
(define words (list "Lions" "Tigers" "Bears" "Oh, my!"))
(define (next-word) ... )   ;;[your implementation here]
(next-word) ⇒ "Lions"
(next-word) ⇒ "Tigers"
(next-word) ⇒ "Bears"
(next-word) ⇒ "Oh, my!"
(next-word) ⇒ "Lions"
(next-word) ⇒ "Tigers"
```

Note: you can assume that there is at least one string in the state variable `words` in the interface.

## Question 4: The Caesar Cipher

The Caesar cipher is an example of a simple encryption scheme. The encryption process takes each character in the plaintext (which is the message to be encrypted) and replaces it with the character in the alphabet that appears 3 places to the **right** of that character. When "3 places to the right" goes past the end of the alphabet, we wrap around to the start of the alphabet. In this problem, our alphabet consists of the underscore character "_", the period "." and the 26 lower-case Latin characters, in that order. We will refer to this alphabet as $\mathcal{A}$. The encryption process for each character in $\mathcal{A}$ is described as follows:

| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| _ | . | 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 |
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 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 | _ | . | a |

For example, "b" is encrypted to "e", "m" is encrypted to "p" and "x" is encrypted "_".

The resulting message after encryption is called the ciphertext (of the original plaintext). For example, the plaintext "wax." when encrypted with the Caesar cipher, with our alphabet $\mathcal{A}$, yields the ciphertext "zd_c".

To decrypt (or undo) the ciphertext, we replace each character in the ciphertext with the character in the alphabet that appears 3 places to the **left** of it. In this case, we wrap around to the end of the alphabet when the ciphertext character is one of the first three characters in the alphabet. The decryption rule for each character in $\mathcal{A}$ is also given in the above table, where we replace each ↓ with ↑ .

Using the same examples as above, we see that "e" is decrypted to "b", "p" is decrypted to "m", "_" is decrypted "x", and the ciphertext "zd_c" is decrypted to "wax.".

For this question we will store both plaintext and ciphertext messages in a `string-wrapper` structure (a simple wrapper for a string so that we can do structural mutation). In particular, we will use the following structure:

```
(define-struct string-wrapper (text))
;; A string-wrapper is a structure
;;     (make-string-wrapper s)
;; where s is a string whose elements (characters) can
;; only be one of
;; (a) the underscore character "_"
;; (b) the period ".", or
;; (c) the 26 lower-case Latin alphabet letters
;;     "a", "b", "c", ..., "x", "y", "z"
```

**(Part A)** In the first part of this problem you will complete the Scheme function `caesar-encrypt`, which consumes a `string-wrapper` and produces (`void`). The effect of the function is to replace the text in the consumed `string-wrapper` (the plaintext) with the ciphertext obtained by encrypting the plaintext with the Caesar cipher using the alphabet $\mathcal{A}$.

**(Part B)** In the second part of this problem you will complete the Scheme function `caesar-decrypt`, which consumes a `string-wrapper` and produces (`void` ). The effect of the function will be to replace the text of the consumed string (which is the ciphertext) with the plaintext obtained by decrypting the ciphertext with the decryption method described above (using the alphabet $\mathcal{A}$). That is, we decrypt the ciphertext with the decryption algorithm of the Caesar cipher.

For example,

```
(define message (make-string-wrapper "hello._goodbye."))

(string-wrapper-text message)⇒ "hello._goodbye."
(caesar-encrypt message)
(string-wrapper-text message) ⇒ "khoorcbjrrge.hc"
(caesar-decrypt message)
(string-wrapper-text message) ⇒ "hello._goodbye."
```

Note 1: The encryption/decryption of an empty string is just the empty string.

Note 2: No marks will be given for a solution in which all 28 individual encryption rules ("_" → "b", "."→"c", "a" → "d", "b" → "e", … "z" → "a") are explicitly included in the function (or in a helper function). The same holds for the decryption algorithm.

Note 3: There are several ways of approaching this problem. Scheme provides several built-in functions for working with strings that may be useful.