Lab 3: Value/Reference Semantics and Scheme

Learning objectives:

- Understand the difference between reference semantics and value semantics
- Gain experience writing Scheme code
- Gain experience with recursion

Project 2 is written entirely in Scheme, and as such this worksheet aims to provide you with some Scheme practice, among other things.

For all coding questions, you must use recursion and you may not use mutation (`set!`, `set-car!`, `set-cdr!`, etc.).

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab03/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Value and reference semantics.* Consider two fragments of code in C++ and Python:

C++	**Python**

`class Foo { ... };`
 	
`int main() {`
`  Foo x = Foo();`
`  Foo y = x;`
`  Foo &z = y;`
`  x = Foo();`
`}`
	`class Foo:`
	`  ...`
	
	`x = Foo()`
	`y = x`
	`z = y`
	`x = Foo()`

   For each program, draw a memory diagram that represents the state of the program just before it terminates.

2. *List manipulation.* Implement the following procedures in `lists.scm`. Simple tests for each procedure are included in the `lists.scm` starter file. To run the tests:

   ```
   $ plt-r5rs lists.scm
   ```

   You may only use the following special forms and primitive procedures: `define` at global scope, `let`, `let*`, `lambda`, `if`, `cond`, `and`, `or`, `quote`, `null?`, `list?`, `not`, `cons`, `car`, `cdr`, `list`, and `equal?`. Your functions do not have to be tail recursive.

   a) Implement the `list-append` procedure, which takes two lists as parameters and returns a new list that is the second list appended to the first list.

   ```
   (list-append '(1 2) '(2 3))  ; returns (1 2 2 3)
   (list-append '() '(2 3))     ; returns (2 3)
   ```

   b) Now implement the `deep-reverse` procedure, which takes a list as a parameter and returns a new list that is the reverse of the input list. If the input list contains a list, that list must be reversed as well. You may use your implementation of `list-append`, if necessary.

   ```
   (deep-reverse '(1 2 3 4 5))    ; returns (5 4 3 2 1)
   (deep-reverse '((2 4) (1 3)))  ; returns ((3 1) (4 2))
   ```

   c) Implement the `contains` procedure, which takes a list and a value and returns true if the value is found in the list and false otherwise.

```
(contains '(1 2 4) 4)  ; returns #t
(contains '(1 2 4) 3)  ; returns #f
```

3. *Sorting.* In this question, you will implement selection sort, which sorts a list of numbers by repeatedly finding the smallest element in a sublist and moving it to the beginning of that sublist. To sort `(6 4 5 3 9 3)`, for instance, the smallest element `3` is placed at the beginning of the list, and then the rest of the list is sorted. The full sorting process on the list is as follows, where the portion of the list being examined is underlined, and the smallest element is bolded (e.g. **3**).

(6 4 5 **3** 9 3)
(3 6 4 5 9 **3**)
(3 3 6 **4** 5 9)
(3 3 4 6 **5** 9)
(3 3 4 5 **6** 9)
(3 3 4 5 6 **9**)
(3 3 4 5 6 9)

You may only use the following special forms and primitive procedures: `define` at global scope, `let`, `let*`, `lambda`, `if`, `cond`, `and`, `or`, `quote`, `null?`, `list?`, `not`, `cons`, `car`, `cdr`, `list`, `append`, and number comparisons (=, <=, etc.). Your functions do not have to be tail recursive.

Write your implementations in `sort.scm`. Simple tests are included in the `sort.scm` starter file. To run the tests:

```
$ plt-r5rs sort.scm
```

a) Write the `smallest` procedure, which given a list of numbers and a value as arguments, returns the smallest number in either the list or the given value. The following are examples of using `smallest`:

```
(smallest '(4 3 7) -1)  ; returns -1
(smallest '(4 3 7) 5)   ; returns 3
```

b) Write the `remove` procedure that, given a list and a value, returns a new list that is the same as the original but with the first occurrence of the value removed. If the value does not occur in the list, the new list is the same as the original. The following are examples of using `remove`:

```
(remove '(4 3 7 3) -1)  ; returns (4 3 7 3)
(remove '(4 3 7 3) 3)   ; returns (4 7 3)
```

c) Write the `sort` procedure, which given a list of numbers, returns a copy of the list in sorted, increasing order. Implement selection sort using the `smallest` and `remove` procedures from the previous parts.

The following is an example of using `sort`:

```
(sort '6 4 5 3 9 3))  ; returns (3 3 4 5 6 9)
```