Language Assignment #1: Scheme

Issued: Tuesday, January 29
Due: Thursday, February 14

Purpose

This assignment asks you to begin using a functional programming language named Scheme, which is a modern dialect of the venerable language Lisp. Lisp was designed by John McCarthy, at MIT, in 1958. Scheme was designed by Guy Steele and Gerald Sussman, at MIT, in 1975.

Submission

Homework is due at 11:59PM, Mountain Time, on the day it is due. Late work is not accepted. To submit your solution to an assignment, login to a lab computer, change to the directory containing the files you want to submit, and execute:

```
1  submit jbuffenb class assignment
```

For example:

```
1  submit jbuffenb cs101 hw1
```

The `submit` program has a nice `man` page.

Documentation

Scheme lecture slides are at:

```
1  pub/slides/slides-scheme.pdf
```

Scheme is described in Section 11.3 of our textbook.

The onyx cluster has a Scheme interpreter, named Guile, the documentation of which can be viewed by:

```
info Guile Reference
info R5RS
```

and demonstrated by:

```
pub/sum/scheme
```

This documentation, in HTML, is also at:

```
https://www.gnu.org/software/guile/docs/docs.html
```

The interactive interpreter also has online documentation. For example:

```
$ guile
GNU Guile 2.0.14
⋮
Enter ‘,help’ for help.
scheme@(guile-user)> ,d cons
- Scheme Procedure: cons x y
    Return a newly allocated pair whose car is X
    and whose cdr is Y.  The pair is guaranteed
    to be different (in the sense of ‘eq?’)
    from every previously existing object.
scheme@(guile-user)>
```

Assignment

Write and fully demonstrate a function named `super-duper`, with this interface:

```
(super-duper source count)
```

The function returns a *copy* of the list `source`, with every element duplicated `count` times. If `source` is an atom, it is immediately returned, without duplication.

For example:

```
(super-duper 123 1)
⇒ 123

(super-duper 123 2)
⇒ 123

(super-duper '() 1)
⇒ ()

(super-duper '() 2)
⇒ ()

(super-duper '(x) 1)
⇒ (x)

(super-duper '(x) 2)
⇒ (x x)

(super-duper '(x y) 1)
⇒ (x y)

(super-duper '(x y) 2)
⇒ (x x y y)

(super-duper '((a b) y) 3)
⇒ ((a a a b b b) (a a a b b b) (a a a b b b)
    y y y)
```

Of course, you can define other functions and call them from `super-duper`.

You are required to use only the *pure* subset of Scheme:

- no side-effecting functions, with an exclamation mark in their names (e.g., `set-car!` and `set-cdr!`)
- no loops (e.g., `do`, `foreach`, and `map`)

Historically, students often want to use the built-in function `append`. There are several reasons why you should not use `append`:

- It doesn't really do what you want. Use `cons`.

- It is just a function. See:

```
1 pub/etc/append.scm
2 pub/etc/append1.scm
```

- It does not make a copy of its arguments, as required by the assignment. Paraphrasing the reference manual: `append` doesn't modify the given arguments, but the return value may share structure with the final argument.

Test your solution thoroughly. The quality of your test suite will influence your grade.

Finally, do not try to find a solution on the Internet. You'll possibly be asked to solve a similar problem on an exam, and if you have not developed a solution on your own, you will not be able to do so on the exam.