Language Assignment #1: Scheme

Issued: Thursday, January 23
Due: Tuesday, February 4

Purpose

This assignment allows you to program in a language introduced in lecture: Scheme.

Scheme is a modern dialect of Lisp. Lisp was designed by John McCarthy, at MIT, in 1958. Scheme was developed by Guy Steele and Gerald Sussman, at MIT, in 1975.

Why Scheme?

Scheme is a garbage-collected functional language. As such, a program is simply a set of function definitions, followed by a function call. Scheme is a modern dialect of the venerable Lisp, the second high-level language, after Fortran. Lisp's syntax is breathtakingly simple, yet the language provides, or can be extended to provide, features found in any language.

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:

`submit jbuffenb` *class assignment*

For example:

```
submit jbuffenb cs354 hw1
```

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

When you submit a program, include: the source code, sample input data, and its corresponding results.

Scores are posted in our `pub/scores` directory, as they become available. You will receive a code, by email, indicating your row in the score sheet. You are encouraged to check your scores to ensure they are recorded properly. If you feel that a grading mistake has been made, contact me as soon as possible.

Translator

In our lab, `onyx` is the home-directory file server for its nodes (e.g., `onyxnode01`). There is also a shared directory for "apps" at `/usr/local/apps`. Nodes share a translator for Scheme, named `guile`, which is installed below `/usr/local/apps`, which is a non-standard location.

Due to network constraints, `onyx` can be reached from the public Internet, but a node can only be reached from `onyx`. So, you can SSH and login to `onyx`, then SSH and login to a node.

An easy way to use `guile`, from a node, is to permanently add a line to the end of your `.bashrc` file. To do so, login to a random node, from `onyx`, by executing the script:

```
pub/bin/sshnode
```

Then, execute the script:

```
pub/bin/bashrc
```

Don't change your `$PATH`; just execute the script. Then, logout from the node and login to a node.

There are a few ways to run a Scheme program, with `guile`. First, put the program in a file (e.g., `prog.scm`). Then, either:

- Execute the translator, with the filename as a command-line argument:

  ```
  $ guile prog.scm
  ```

- Execute the translator, without command-line arguments, and load the filename:

```
$ guile
guile> (load "prog.scm")
...
guile> (exit)
```

- Use the "pound-bang trick" to turn the file into a script, as demonstrated by the first two lines of:

```
pub/etc/append.scm
```

  Then, make the file executable, with `chmod`, and execute it directly:

```
$ chmod +x prog.scm
$ ./prog.scm
```

  Since `guile` may not be in `/usr/bin`, don't forget to adjust the pound-bang path, for the translator on *your* computer. Use `type` to find it:

```
$ type -a guile
```

Documentation

Scheme lecture slides are at:

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

Scheme is demonstrated by:

```
pub/sum/scheme
```

Scheme also is described in Section 11.3 of our textbook.

Links to programming-language documentation can be found at:

```
http://csweb.boisestate.edu/~buff/pl.html
```

The interactive interpreter also has online documentation, for some functions. For example:

```
$ guile
...
```

```
Enter ‘,help’ for help.
scheme@(guile-user)> ,d append
- Scheme Procedure: append .  args
     Return a list consisting of the
     elements the lists passed as
     arguments.
...
scheme@(guile-user)>
```

Assignment

Write and fully demonstrate a Scheme function that implements a sort-of duplication operation. Your function should be named `super-duper`, and have 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)
```

Other Requirements

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

You are required to use only a *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 builtin 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:

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

- It does not make a copy of its arguments, as required by parts of 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.