Language Assignment #1: Scheme

Issued: Tuesday, February 2
Due: Tuesday, February 16

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:

```
submit jbuffenb class assignment
```

For example:

```
submit jbuffenb cs101 hw1
```

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

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" mounted 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 shared-library constraints, `onyx` cannot execute `guile`. It can only be executed by a node.

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

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.

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 Tree Search And Replace operation. Your function should be named `tsar`, and have this interface:

```
(tsar subj srch repl)
```

A “tree” is simply a Scheme list (i.e., a symbolic expression, or s-expr). The function returns a *copy* of the list `subj`, with every sublist equal to `srch` replaced by a *copy* of `repl`. The function *returns* its result; it does not display anything. Any of the arguments might be a list or an atom.

For each replacement, instances of the atom `RANDOM` within `repl` should be replaced by a random number between 0 and 100, the result of: `(random 100)`. This will demonstrate that proper copies are made. You might want to add this “feature” last.

For example:

```
(tsar 'z 'x 'y)
    ⇒ z
(tsar 'x 'x 'y)
    ⇒ y
(tsar 'x 'x 'x)
    ⇒ x
(tsar '() 'x 'y)
    ⇒ ()
(tsar '(x x) 'x 'y)
```

```
    ⇒ (y y)
(tsar '(x (x) z) 'x 'y)
    ⇒ (y (y) z)
(tsar '(x (x) z) '(x) '(y y))
    ⇒ (x (y y) z)
(tsar '(x (x) ((x)) z) '(x) '(y y))
    ⇒ (x (y y) ((y y)) z)
(tsar '(x (x) ((x)) z) '(x) '())
    ⇒ (x () (()) z)
(tsar '(x (x) ((x)) z) '() '(y y))
    ⇒ (x (x y y) ((x y y) y y) z y y)
(tsar '(x (x) ((x)) z) '(x) '(RANDOM y))
    ⇒ (x (74 y) ((46 y)) z)
(tsar '(x (x) ((x)) z) '(x) '(RANDOM RANDOM))
    ⇒ (x (76 76) ((63 63)) z)
```

Other Requirements

You are encouraged to define other functions, and call them from `tsar`.

The only builtin comparison function that you are allowed to use is `eq?`, for comparing atoms. Do not use the builtin function `equal?`, which is too much like a function I want you to write. Likewise, do not use the builtin function `copy-tree`.

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.