MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Problem Set 1

A. Paper Problems -

Problem 1: (PostFix + Pairs)

Do Exercise 3.24 on page 78 of the course notes.

Problem 2: (PostText)

Do Exercise 3.27 on page 80 of the course notes.

B. Programming Problems

In the next two problems, you will extend the PostFix interpreter described in Section 1.2.2 of the Scheme supplement to implement the extensions described in the first two problems. The PostFix interpreter will be presented in lecture. Before then, you should familiarize yourself with the Scheme+ programming language. You may want to read appendix A of the Scheme supplement as early as possible.

We especially urge people to work in groups on programming assignments. Remember that you must write up your own solutions and attribute any joint work. See the collaboration policy described in Handout 3 for more information.

First, you should play with the existing PostFix interpreter. Download `ps1/postfix.scm` from the course directory and start the interpreter by evaluating the following Scheme+ expressions:

```
(load "postfix.scm")
(pf-den-repl)
```

This will start a read-eval-print loop. (There is nothing to turn in for this part—just familiarize yourself with the interpreter.)

Problem 3: (PostFix + Pairs + {`dup`} Lab)

Extend the interpreter to handle the `pair`, `left`, and `right` commands of Problem 1 as well as the `dup` command discussed in Section 3.5 of the course notes.

As mentioned in Handout 3 (General Information), your solutions for this and all programming problems should clearly indicate what modifications you have made and include an English explanation of what those modifications do. Demonstrate that your program correctly solves the problem by testing it on a sufficient test suite.

Problem 4: (PostText Lab)

Extend the PostFix interpreter to implement the extensions described in Problem 2.

a. Implement a dictionary abstraction by writing Scheme+ procedures that have the following interface:

   - `(dict-empty)` creates an empty dictionary.
   - `(dict-bind` *`name value dict`*`)` returns a new dictionary that extends *dict* with a binding between *name* and *value*.
   - `(dict-lookup` *`name dict`*`)` returns the value associated with *name* in the given dictionary, *dict*.

   Based on the suggestion in Exercise 3.27(b), you should represent dictionaries as one-argument procedures that map symbols to values.

b. Define a new `state` datatype as follows:

   ```
   (define-datatype state (make-state stack dict))
   ```

   Modify the evaluation procedures in the interpreter so that they map states to states rather than mapping stacks to stacks.

c. Extend the interpreter to handle the I, `def`, and `get` commands described in Exercise 3.27. One of your test cases should be a non-terminating program.

Problem 5: (Factorial)

a. Do Exercise 3.25(a) on page 79 of the course notes. Test your command sequence by running it in the modified interpreter from Problem 3. (It may be easier to do the next part of this problem first.)

b. Reimplement factorial in PostText. Test it in the modified interpreter from Problem 4.

c. *Extra Credit.* Implement the Fibonacci function of Exercise 3.25(b) in both (PostFix + Pairs + {`dup`}) and PostText.