CS 61A, Summer 2006 July 5, 2006
LAB ASSIGNMENT 2B

1. Experiment with incorrect Scheme expressions, and find at least five different kinds of error messages that STk can print. "Different kinds" means that you can't count both of

```
+: not a number: foo
*: not a number: baz
```

2. Define the recursive Fibonacci procedure as on page 37 of SICP. Trace it. How can you characterize the appearance in the trace result of base-case calls? How many base-case calls are there when you compute `(fib 5)`?

3. Try out the Replacement Modeler program. First load it:

```
(load "~cs61a/lib/modeler.scm")
```

Then enter the expressions below, one at a time. Each one will create a new window containing the argument expression to the `model` special form. Try each expression repeatedly; first, start the modeler and try just repeating the RETURN key (the one in the main alphabetic keyboard). Then, using the same expression again, try just repeating the ENTER key (in the numeric keypad). Then try selecting a subexpression by clicking the mouse, and use RETURN and ENTER on that subexpression.

```
(model (+ (* 2 3) (- 5 1)))
(model (every first '(tomorrow never knows)))
(model (keep even? '(76 909 1 110 8)))
(model (if (= 2 3) 'yes 'no))
(model (if (= 3 3) 'yes 'no))
(model (cond ((= 2 3) 'yes) ((= 2 2) 'no) (else 'maybe)))
```

Continued on next page ...

4. The following exercises will teach you how to use `stkdb`, a debugger for `stk` that is (intentionally) similar to the `gdb` and `gjdb` debuggers you will use in later CS courses. (The `stkdb` debugger is on the instructional CD, but does not work in Windows.)

There are quite a few exercises here; don't worry if you don't have time to work through them all. We won't require you to know anything about the `stkdb` debugger in this course, but it's good to know how to use it when you come across bugs in your projects!

4a. Load the program `~cs61a/lib/change.scm`. (It appears below.)

```
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

Select the command `Debug File` from the `Debugging` menu. This loads the program into `stk` and prepares it for debugging. (In the rest of the lab exercises, "select the command" means "select the command from the `Debugging` menu.")

4b. Set a *breakpoint* at the call to `or` by clicking there and selecting the command `Set Breakpoint`. This specifies that you want the debugger to regain control when the indicated expression is evaluated.

4c. In the Scheme buffer, evaluate the expression

```
(count-change 15)
```

Evaluation should be suspended at the expression at which you set the breakpoint, and the debugger should highlight this expression.

4d. Repeatedly give the `Step Over` command (abbreviated by the function key F6). `Step Over` evaluates the highlighted expression and returns to the debugger when that's done.

4e. Once the program finishes, start it again by re-evaluating `(count-change 15)` in the Scheme buffer. This time, go through the program with the `Step Into` command (abbreviated by the function key F5). When you reach the recursive call to `cc`, print the values of the arguments via the `See Local Variables` command.

4f. Continue stepping through the program. When you get to the third recursive call, print the argument values again. At this point, you can move up and down among the saved "copies" of the recursive procedure.

Do this using the `View Caller` command (abbreviated by the function key F3) and the `View Callee` command (abbreviated by the function key F4). Move to the initial call of `cc`, print the arguments, and move back to the most recently suspended call and print those arguments again.

4g. Now set `Auto-Display Backtrace/Locals` and continue stepping through the program until it finishes. Observe the extra information in the window.

• Now you'll use the debugger to find a bug. The decimal-value procedure in the file `~cs61a/lib/buggy.romannum.scm` is intended to return the decimal value of a given Roman numeral. Unfortunately, the call `(decimal-value '(I X))` produces an infinite recursion.

4h. Copy the program to your directory, then run emacs on it:

```
emacs buggy.romannum.scm
```

Select the command `Debug File`.

4i. In the Scheme buffer, evaluate the expression `(decimal-value '(I X))`. Then select the command `Interrupt`. Determine the arguments to the interrupted call either by evaluating the argument in the Scheme buffer or by selecting the command `See Local Variables`.

4j. Repeat the `Step Over` command (F6) until the procedure is about to make a recursive call. Then repeat the `Step In` command (F5) a few times to make the call. Display the arguments in order to verify the conditions for an infinite recursion.

4k. Fix the problem by adding a `butfirst`. Save the file. Then, with the cursor somewhere in the procedure you just fixed, select the command `Debug File`, which reloads the program into `stk`.

4l. Set a breakpoint at the spot of the recursive call. Re-evaluate the expression `(decimal-value '(I X))` in the Scheme buffer. The debugger should display the expression at which you set the breakpoint. `Step Into` the recursive call, then verify that the argument to the recursive call isn't the same as the original argument.

4m. Finally, select the command `Finish Function` as many times as necessary to carry out the rest of the evaluation and produce the value 9.

• More `stkdb` information

A breakpoint has to be set at a procedure call, not at a use of a variable or at a definition.

When the debugger is accepting your commands, its prompt is `stk[...]`. If by accident you lose this prompt and see `stk`'s `STk>` prompt again, you can restart the debugger by evaluating `(stkdb)`.

Some runtime errors (e.g. the error that results from an infinite recursion) kill the Scheme process. Going to the `.scm` file and typing `C-c d`, or typing `M-x run-scheme` should make Scheme mode accessible again.

If `stk` and `emacs` get even more confused, you can fix this by exiting `emacs` and starting it up again.