This lab exercise concerns the change counting program on pages 40–41 of Abelson and Sussman. Read it before you continue!

1. Identify two ways to change the program to reverse the order in which coins are tried, that is, to change the program so that pennies are tried first, then nickels, then dimes, and so on.

2. Abelson and Sussman claim that this change would not affect the correctness of the computation. However, it does affect the efficiency of the computation. Implement one of the ways you devised in exercise 1 for reversing the order in which coins are tried, and determine the extent to which the number of calls to cc is affected by the revision. Verify your answer on the computer, and provide an explanation. Hint: limit yourself to nickels and pennies, and compare the trees resulting from (cc 5 2) for each order.

3. Modify the cc procedure so that its kinds-of-coins parameter, instead of being an integer, is a sentence that contains the values of the coins to be used in making change. The coins should be tried in the sequence they appear in the sentence. For the count-change procedure to work the same in the revised program as in the original, it should call cc as follows:

   (define (count-change amount)
   (cc amount '(50 25 10 5 1)) )

4. Many Scheme procedures require a certain type of argument. For example, the arithmetic procedures only work if given numeric arguments. If given a non-number, an error results.

   Suppose we want to write safe versions of procedures, that can check if the argument is okay, and either call the underlying procedure or return #f for a bad argument instead of giving an error. (We’ll restrict our attention to procedures that take a single argument.)

   > (sqrt 'hello)
   ERROR: sqrt: bad number: hello
   > (type-check sqrt number? 'hello)
   #f
   > (type-check sqrt number? 4)
   2

   Write type-check. Its arguments are a function, a type-checking predicate that returns #t if and only if the datum is a legal argument to the function, and the datum.

5. We really don’t want to have to use type-check explicitly every time. Instead, we’d like to be able to use a safe-sqrt procedure:

   > (safe-sqrt 'hello)
   #f
   > (safe-sqrt 4)
   2

   Don’t write safe-sqrt! Instead, write a procedure make-safe that you can use this way:

   > (define safe-sqrt (make-safe sqrt number?))

   It should take two arguments, a function and a type-checking predicate, and return a new function that returns #f if its argument doesn’t satisfy the predicate.