Homework 4: Data Driven Recursion

Due:

To ease our grading and your testing, please do the following. (If you are working at home, please follow similar conventions.)

- (Optional step) Make a subdirectory for this homework by executing the command: `mkdir hw4`.
- (Optional step) Change your working directory to that directory each time that you start to work on problems for this homework by executing the command `cd hw4`.
- Make a file to hold a comment template. Edit a file `code-heading.ss` that contains the lines in the same format as the following, but with your information filled in.

```
;;; NAME: Your Name Here       GRADING TA: Your T. A. s. Name here
;;;
;;; PROBLEM:
;;;
```

  Fill in your name and your TAs name as you enter `code-heading.ss`.

- Edit each problem's procedure into a file named after the problem number that is created by making a copy of `code-heading.ss`. For example, for problem 4.1 you should make a file `hw4-01.ss`. You would do this by executing the commands in Vincent:

```
cp code-heading.ss hw4-01.ss
emacs hw4-01.ss
```

  Fill in the problem number and then the code following the code heading.

- Put your transcript of testing in a file with the same name as the code, but with suffix `.out`. For example, put tests of `hw4-01.ss` in the file `hw4-01.out`.
- To make a printout of problems 1-3, say, execute the following command, replacing `du89` with your favorite printer.

```
rm -f TOPRINT
touch TOPRINT
foreach n (01 02 03)
   cat hw4-${n}.ss hw4-${n}.out >> TOPRINT
   echo '***********************************' >> TOPRINT
end
lpr -Pdu89 TOPRINT
```

  Don't be surprised to see the prompt `foreach?` when you are entering the lines between `foreach` and `lpr`; until you type the `end` this is the shell's way of reminding you that you are in a foreach. You need to type `01` instead of `1`, as file names such as `hw4-01.ss` are made of characters not numbers. The exact number of `*`'s doesn't matter, but the single quotes are important.

We are suggesting some problems for additional practice. These have 0 points. You are not to hand these in to your TA. You should check your own work on them, and if you have questions, discuss them in recitation or with the staff, or with others in class.

1. (6 points; basic) Do exercise 4.1.

2. (0 points; suggested practice) Do exercise 4.2.

3. (6 points) Do exercise 4.3.

4. (6 points) Do exercise 4.4.

5. (4 points; extra credit) Define a procedure `lex-<=` that takes two lists of numbers, `lst1` and `lst2` and returns a boolean. It returns `#t` if `lst1` is empty, otherwise it returns `#t` if the head of `lst1` is strictly less than the head of `lst2`, `#f` if the head of `lst1` is strictly greater than the head of `lst2`, and otherwise returns the result of `(lex-<= (cdr lst1) (cdr lst2))`. You are responsible for making up good test cases.

6. (8 points; basic) Do exercise 4.5.

7. (10 points) Do exercise 4.6.

8. (0 points; suggested practice) Write `insert-right-all` that is like `insert-left-all` but which inserts the item `new` to the right of each occurrence of the item `old` in `ls`.

9. (8 points; basic) Do exercise 4.7.

10. (10 points; basic) Do exercise 4.8.

11. (10 points; basic) Do exercise 4.9.

12. (8 points) Do exercise 4.10. The remark "use the appropriate sameness predicate for the data shown in the examples" which may appear in your book should be ignored. (This is an example of computer-induced typing mistakes.)

13. (0 points; suggested practice) Do exercise 4.11.

14. (5 points; extra credit) Define a procedure `middle-most` that takes a list with an odd number of atomic items and returns the middle atomic item in the list. We do not specify what your code should do for a list with an even number of items.

15. (0 points; suggested practice) Do exercise 4.12.

16. (3 points) Do exercise 4.13. Say why it does what it does. You may handwrite your answer to this.

17. (6 points) Do exercise 4.14.

18. (5 points; extra credit) Do exercise 1.17 in *Structure and Interpretation of Computer Programs*, by Abelson and Sussman. This is on reserve at the library.

    In Chez Scheme you can use the procedure `cpu-time` to measure the time used by your program in milliseconds instead of `runtime`. You can also use `writeln` instead of `print`. Or try invoking `(support-sicp)` to get a mode that is like the on in Abelson and Sussman's book. If necessary use MIT Scheme as described in the *How to Run Scheme* handout.

19. (5 points; extra credit) Define a procedure `factors-of` that takes a positive number greater than zero and returns the prime factors of that number as a list. Note that 1 is *not* a prime. You may assume that the argument is 2 or greater. For example,

```
(factors-of 15) ==> (3 5)
(factors-of 16) ==> (2 2 2 2)
(factors-of 17) ==> (17)
```

See Chapter 1 of *Structure and Interpretation of Computer Programs*, by Abelson and Sussman for hints. This is on reserve at the library.

20. (4 points) Do exercise 4.15. Hand in a printout of your code, a transcript, and your comparison and observations (which may be handwritten).

21. (4 points) Do exercise 4.16. Hand in a printout of your code, a transcript, and your comparisons (which may be handwritten).

22. (10 points; basic) Do exercise 4.17. Write down your best guess as to their rates of growth.

23. (6 points; basic) Do exercise 4.18.

24. (10 points; basic) Do exercise 4.19. Note the instructions specify how to write each of the 2 procedures.

25. (8 points) Do exercise 4.20.

26. (5 points; extra credit) Do exercise 1.21 in *Structure and Interpretation of Computer Programs*, by Abelson and Sussman. This is on reserve at the library.

    You can handwrite this.

27. (8 points; extra credit) Do exercise 1.22 in *Structure and Interpretation of Computer Programs*, by Abelson and Sussman. This is on reserve at the library.