Homework 2: Procedures and Recursion

Due:

General Directions

Your solutions to these and all other problems should only use the features of Scheme introduced up to point in the text where the problem occurs. (In particular, you may not use `define` except at the top level.)

For a problem that asks you to write a procedure `foo`, type the Scheme code for `foo` into a file named `foo.ss`, and hand in a printout of your code and either a Scheme transcript showing how you tested the procedure, or a calculation showing that your code is correct. (The form of calculations will be discussed in class.) For other problems, you may handwrite a solution if you wish.

Each Scheme procedure should be labeled with a comment of the following form.

```
;; NAME: <your name here> TA: <grading TA's name here> SECTION: <your section>
```

This comment should be on the first line (closest to the top). Arrange your printouts so that each procedure is followed by either the calculation or the transcript. If your transcript or calculation is not attached to your procedure, put the same information on it, and indicate what problem it is for. If you make a transcript, be sure that you test all relevant cases.

We suggest the following comment conventions to help you summarize the problem to be solved, and understand your solution should you wish to come back to it later.

- Use `TYPE:` comments to give the types of procedures, as in the *Make Your Own Scheme 'Reference Card'* handout. Such a comment should follow the name of the procedure being defined, unless you have it elsewhere (e.g., in a description of the problem).
- Use `ENSURES:` comments to describe the result returned by the procedure. Such a comment should follow the list of formal parameters in a `lambda`, so that `result` can be described using the formal parameters.
- If a procedure is written under assumptions about the values of its arguments, and if these assumptions are not recorded in the `TYPE:` comment, then describe the assumptions in a `REQUIRES:` comment, which should follow the list of formal parameters in a `lambda`, and precede the `ENSURES:` comment. The `REQUIRES:` comment is omitted if no such assumptions are made.

For example, a fully commented procedure is the following.

```
(define second  ; TYPE: all T . list of T -> T
  (lambda (lst)
    ; REQUIRES: lst has at least 2 elements
    ; ENSURES: result is the second element of lst
    (cadr lst)))
```

The above example is a bit overdone, and one might leave out the `ENSURES:` clause in a real program, because the procedure is so simple. Once you become familiar with such comments, you may omit any of them, if you believe that the code is clear enough without them. However, by omitting such comments, you are making a judgement that your code is clear enough without them. If, in our judgement, you are wrong, then you will lose points. Consider these comments as required for all but the simpliest helping procedures.

1 Section 2.2, Procedures

1. (0 points; suggested practice) Do exercises 2.2 and 2.3. Test your answer to 2.2 on the computer, and check your answer to 2.3 on the computer.

2. (4 points; basic) Define a procedure `rotate` that rotates a 3-item list. The procedure `rotate` returns a list that is a rearrangement of the input so that the first item becomes last, the second item becomes first, and the third item becomes second. Test your procedure on the following:

```
(rotate '(1 2 3)) ==> (2 3 1)
(rotate '((a b) (c d) (e f))) ==> ((c d) (e f) (a b))
```

3. (5 points) Define a procedure, `transpose-alist`, that takes a 3-item list of 2-item lists, `alist`, and returns a 2-item list of 3-item lists. For example,

```
(transpose-alist '((one uno) (two dos) (three tres)))
  ==> ((one two three) (uno dos tres))
(transpose-alist '((iowa des-moines) (michigan lansing) (new-york albany)))
  ==> ((iowa michigan new-york) (des-moines lansing new-york albany))
```

4. (5 points; extra credit only) Write a procedure, `transpose-alist-bad`, that is the same as `transpose-alist`, except that your code should be as hard to read (bad) as you can make it. However, it must actually work! Instead of handing a printout, post your solution and a transcript showing how it was tested to the newsgroup `isu.coms.227`. (See the file `/home/cs227/doc/reading-news.txt` for details on how.) However, and this is important, do *not* post your solution to this problem until after the above problem is due. Make the subject of your post:

```
A really, really bad program by <your name>, section <your section>
```

2 Section 2.3, Conditional Expressions

5. (0 points; suggested practice) Do as many parts of exercises 2.6, 2.7, 2.8, and 2.9 as you need to feel confident about the special forms `and` and `or`. Check your answers on the computer

6. (6 points; basic) Find all the syntax errors and type errors in the following expressions, circle the mistakes, and write out a corrected version. Assume that the code is indented reasonably. You should handwrite your answer to this, and try to find the errors without using the computer. (You can check your answers on the computer if you wish.)

   (a)
   ```
   (car (cons (3 '())))
   ```
   (b)
   ```
   (cond
     ((null? ls) '()
     (zero? n) (cons n '())
     (else (cons d (sub1 n) ls))
   ```
   (c)
   ```
   (if null? lst
      '()
      (else (foo (car lst) (cdr lst))))
   ```

3 Section 2.4, Recursion

7. (6 points) Do exercise 2.10 (translate `cond` to `if`).

8. (0 points; suggested practice) Do exercise 2.12.

9. (10 points; basic) Do exercise 2.13. Note that you are to define 3 procedures for this problem (see the end of the problem).

10. (12 points; basic) Define a procedure `insert-left-add1` that takes a number, `num`, and a list of numbers, `lst`, and returns a list that is the same as `lst` except that if `num` is an item in `lst`, the result has $1 +$ `num` inserted to the left of the first occurrence of `num` in `lst`. For example,

```
(insert-left-add1 4 '(3 7 6 4)) => (3 7 6 5 4)
(insert-left-add1 6 '(1 2 3)) => (1 2 3)
(insert-left-add1 2 '(2)) => (3 2)
(insert-left-add1 7 '()) => ()
```

11. (8 points) Define a procedure, `list-of-second-items`, that takes a list of lists, and returns a list of the second item in each sublist. You may assume that each sublist has at least two items. For example,

```
(list-of-second-items '()) ==> ()
(list-of-second-items '((george bush) (bill clinton)))
   ==> (bush clinton)
(list-of-second-items '((stop go) (slow fast faster) (no yes maybe)))
   ==> (go fast yes)
```

12. (8 points; basic) Do exercise 2.17.

13. (10 points) Do exercise 2.18.

14. (0 points; suggested practice) Do exercise 2.20. How does this relate to type checking?

15. (15 points) There are intentionally no examples for this problem. The problem is not very well specified. (This is just as in real life.) If you need to make additional assumptions, write them down. Part of your grade will depend on how well you design the input and the output.

    Suppose you have a list representing your private collection of record albums (see problem 12 in homework 1). Write a procedure `titles` that takes the list representing your collection as an argument, and returns the titles of all the albums in your collection as a list.

16. (10 points; extra credit) Define a procedure, `big-mac-1st`, that takes two items, `b` and `m`, and a list `ls` as arguments. It replaces the first occurrence of 3 items that are `equal?` to `b` in a row, that is `b b b`, with the items `b m b m b`. For example,

```
(big-mac-1st 'red 'green '()) ==> ()
(big-mac-1st 'bread 'meat '(bread cheese bread bread bread))
  ==> (bread cheese bread meat bread meat bread)
```

4 Section 2.5, Tracing and Debugging

17. (0 points; suggested practice) Do problem 2.22 on page 64. Check your answer on the computer.

18. (0 points; suggested practice) Do problems 2.23 and 2.24. Check your answers on the computer.

    To use the code for the procedures `entering` and `leaving`, you will need to type, at the Scheme prompt, the command `(load "/home/cs227/lib/ch2.ss")`.

19. (8 points; basic) Write a trace similar to the one used in `remove-1st-trace` (p. 62 of the text) to trace the following procedure, `insert-right-1st`.

```
(define insert-right-1st
   (lambda (new old ls)
      (cond
         ((null? ls) '())
         ((equal? (car ls) old) (cons old (cons new (cdr ls))))
         (else (cons (car ls) (insert-right-1st new old (cdr ls)))))))
```

Your traced procedure, `insert-right-1st-trace`, should show the binding of the parameter `ls` each time the `cond` expression is entered and whenever the `cond` expression is exited. Turn in a transcript of the following application.

```
(insert-right-1st-trace 'not 'does '(my dog does have fleas))
```

(Having trouble with `entering` and `leaving`? Look at the previous problem.)

20. (3 points; extra credit) Do problem 2.26, except instead to building the tables for the applications given in the problem, do them for `(last-item '(grocho harpo zepho))` and `(member? 'ringo '(grocho harpo zepho))`.

21. (10 points; extra credit) Consider classifying the procedures that you have seen in this chapter by their types (e.g., all T . list of T $\rightarrow$ T, etc.). Is there any general form that procedures of a given type take? If so, describe the general form of the procedures of that type, and give the names of the examples. If not, is there any reason for the lack of a general form?

22. (15 points; extra credit) Categorize the kinds of mistakes that you have made in solving the problems in this chapter. For each kind of mistake, describe it and give an example along with a fix. Post your findings to the newsgroup `isu.coms.227`, and be sure to include your name so you can get credit for this. (See the file `/home/cs227/doc/reading-news.txt` for details on how.) When posting be sure not to give away any solutions to homework problems.

    Can you suggest ways to help avoid these mistakes?