Homework 4: Data Driven Recursion

Due:

All the directions from homework 2 apply to this homework. In particular, remember to hand in a printout for procedures, and either a calculation or a transcript showing your testing.

We highly recommend that you put each problem in a separate file, and when making a transcript use the Scheme procedure `load` to read that file in to the Scheme interpreter.

1 Section 4.2, Flat Recursion

1. (0 points; suggested practice) Do problem 4.1. Check your answer on the computer.
2. (6 points; basic) Do exercise 4.2.
3. (0 points; suggested practice) Do exercises 4.3 and 4.4. Check your answer on the computer.
4. (6 points) Define a procedure `list-abs` that takes a list of numbers, `lst` and returns a list of the corresponding absolute values. For example,

```
(list-abs '()) ==> ()
(list-abs '(3 -2 -1 -5)) ==> (3 2 1 5)
(list-abs '(-1 5 7 9 0 -1 -94)) ==> (1 5 7 9 0 1 94)
```

5. (5 points; extra credit) Define a procedure `deepen-n` takes a list, `ls`, a natural number, `n`, and returns the list with `n` pairs of parentheses wrapped around each top-level item of `ls`. You are responsible for making up good test cases.

2 Sections 4.3 and 4.4, Deep Recursion and Tree Rep of Lists

Recall that we call a list with nested sublists a *tree*.

6. (8 points; basic) Write a procedure, `prod-all`, that takes tree of numbers, `tr`, and returns the product of all the numbers in `tr`. When given the empty tree, `prod-all`, should return 1. For example,

```
(prod-all '()) ==> 1
(prod-all '(() ())) ==> 1
(prod-all '(1 2 3)) ==> 6
(prod-all '((1) ((2) ()) (((3))))) ==> 6
(prod-all '(9 (-5) (6 (1 1 78) 8) (1 ( 2 (3 () 4) 2 ) 0))) ==> 0
(prod-all '((1 1 3)
            (2 3 4)
            (3 4 5))) ==> 4320
```

7. (8 points) Write a procedure, `tree-abs`, that takes a tree of numbers, `tr`, and returns a tree with the same shape but with each number replaced by its absolute value. For example,

```
(tree-abs '()) ==> ()
(tree-abs '(() ())) ==> (() ())
(tree-abs '(1 -22 3)) ==> (1 -22 3)
(tree-abs '((-1) ((2) -4 ()) (((-33))))) ==> ((1) ((2) 4 ()) (((33))))
```

```
(tree-abs '((1 -1 -3)
            (2 3 -4)
            (3 4 5))) ==> ((1 1 3) (2 3 4) (3 4 5))
```

8. (10 points) Write a procedure, `insert-right-all`, that takes a tree of symbols, `ts`, a symbol `old`, and a symbol, `new`, and which returns a tree that is the same as `ts`, but which has `new` inserted to the right of each occurrence of the symbol `old` in `ts`. For example,

```
(insert-right-all 'bach 'rock '()) ==> ()
(insert-right-all 'bach 'rock '(() ())) ==> (() ())
(insert-right-all 'truth? 'love '(what is truth?)) ==> (what is truth? love)
(insert-right-all 'o 'n '((o x) (o (x) o))) ==> ((o n x) (o n (x) o n))
(insert-right-all 'n 'o '((o x) (o (x) () o))) ==> ((o x) (o (x) () o))
```

9. (0 points; suggested practice) Do exercise 4.8. Check your answer on the computer. For this problem, you can consider the argument to `count-parens-all` to be a tree of atomic items, where an atomic item is a Scheme datum that is not a pair and not the empty list.

10. (10 points; basic) Write a procedure `occurrences` that takes as its argument an item, `item`, and a tree, `tr`, and returns the number of items in `tr` that are the same as `item`. Use the appropriate sameness predicate for the data shown in the examples. For example,

```
(occurrences 'a '()) ==> 0
(occurrences 'a '(() ())) ==> 0
(occurrences 'b '((a) b ((b) c a) () d)) ==> 2
(occurrences 'c '((b) (((c) c) (c c) a) () d)) ==> 4
```

Be sure to write a TYPE comment for this procedure, since we are not giving the exact type to you.

11. (10 points) Write a procedure, `rightmost-symbol`, that takes a tree of symbols and returns the rightmost symbol in the tree. You may assume that the tree contains at least one symbol, and that no subtree is empty. For example,

```
(rightmost-symbol '(a)) ==> a
(rightmost-symbol '(a b c d)) ==> d
(rightmost-symbol '(((b) a) ((c) d) (e) (f) (g (h h) i) j)) ==> j
```

12. (5 points; extra credit) Generalize `rightmost-symbol` so that it handles the case where the subtrees may be empty. That is, in addition to the above examples,

```
(rightmost-symbol '(a b c () d)) ==> d
(rightmost-symbol '(a b c d ())) ==> d
(rightmost-symbol '((() a) ())) ==> a
(rightmost-symbol '((() a) (()))) ==> a
(rightmost-symbol '((() a) (()) () () (()))) ==> a
```

13. (5 points; extra credit) Define a procedure `middle-most` that takes tree with an odd number of atomic items and returns the middle atomic item in tree. We do not specify what your code should do for a list with an even number of items.

14. (8 points; extra credit) Write a procedure `arith` that takes a tree of symbols and numbers, and returns the number that is the result of evaluating the tree as an arithmetic expression. You may assume that the arithmetic expression is well formed, and that the only symbols in the tree are `+`, `-`, `*`, and `/`. For example,

```
(arith '(+ 1 2)) ==> 3
(arith '(+ (* 3 4) (/ 12 2))) ==> 18
```

Note that the input is quoted. You should make up additional examples.

15. (10 points; extra credit) Write a procedure `cond->if` that takes a Scheme expression involving numbers, variables, `quote`, procedure calls, `cond`, and `if`, and translates the expression into an equivalent one that doesn't use `cond`. That is, you are to translate each use of `cond` into `if`. You may assume that each `cond` comes with an `else`, and that there are no syntax errors. For example,

```
(cond->if '(cons (car y) (quote ()))) ==> (cons (car y) (quote ()))
(cond->if '(cond ((null? x) (quote ())) (else (car x))))
      ==> (if (null? x) (quote ()) (car x))
(cond->if '(+ 3 (cond ((< x 0) (- x)) ((zero? x) 0) (else x))))
      ==> (+ 3 (if (< x 0) (- x) (if (zero? x) 0 x)))
```

You should make up additional examples.

Note that `(quote x)` prints as `'x` in Scheme.

3 Section 4.5, Numerical Recursion and Iteration

16. (0 points; suggested practice) Do exercise 4.12. (This is a bit of easy fun.)

17. (0 points; suggested practice) Do exercise 4.13. Try it on the computer!

18. (6 points) Do exercise 4.14. Use inexact (floating point) arithmetic, otherwise your program will take too long to run. Show in a transcript that the inequality given in the problem holds.

19. (5 points) An iterative procedure is *not* a procedure with an accumulator! For example the procedure `last-item` of program 2.2 is iterative. Explain why `last-item` is iterative. (You can handwrite your answer to this.)

20. (7 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.

4 Section 4.6, Analyzing the Fibonacci Algorithm

21. (0 points; suggested practice) Do exercise 4.15. This is to help you understand the recursive version.

22. (0 points; suggested practice) Do exercise 4.16. This is to help you understand the iterative version.

23. (10 points; basic) Do exercise 4.17. Write down your best guess as to their rates of growth.

24. (6 points; basic) Write an iterative procedure, `sum-it`, that takes a list of numbers and returns the sum of the numbers in the list. For example,

```
(sum-it '()) ==> 0
(sum-it '(1 2 3 4)) ==> 10
```

25. (0 points; suggested practice) Do problem 4.19. Be sure your procedures are iterative.

26. (10 points; basic) For this problem you will write two iterative procedures. The first, `evens-up`, takes a natural number, `k`, and returns a list of the first `k` even natural numbers, in ascending order. The second, `evens-down`, produces the list in descending order. For example,

    ```
    (evens-up 0) ==> ()
    (evens-up 1) ==> (0)
    (evens-up 3) ==> (0 2 4)
    (evens-up 10) ==> (0 2 4 6 8 10 12 14 16 18)
    (evens-down 0) ==> ()
    (evens-down 1) ==> (0)
    (evens-down 3) ==> (4 2 0)
    (evens-down 10) ==> (18 16 14 12 10 8 6 4 2 0)
    ```

    You will get no points for procedures that are not iterative. (And do not use `reverse` either!)

27. (8 points) Write both an iterative *and* a recursive version of the procedure `sum-the-top-numbers`. The procedure `sum-the-top-numbers` takes a list of datum, `ls`, and returns the sum of the all all the top-level numbers in `ls` that are numbers. For example,

    ```
    (sum-the-top-numbers '()) ==> 0
    (sum-the-top-numbers '(3 bears 1 goldilocks)) ==> 4
    (sum-the-top-numbers '(10 miles 80 yards 3 inches)) ==> 93
    ```

28. (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.

29. (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.

30. (7 points; extra credit) Can you write an iterative version of the procedure `append`? If not, say why not. If so, do it.