Homework 2: Functional Programming

Due Thursday, Feb. 19 at 9:50 am

1. Warm up

(10 points)
Evaluate the following expressions **by hand** (you may check your work on a computer AFTER you are done with this part):

```
1. (first '(tom jeannie andrea duane))
2. (cons 'brent '(steve morgan))
3. (first (rest (rest '(inky (blinky pinky) clyde))))
4. (cons (+ 1 2) (rest '(+ 1 2)))
5. ((lambda (x) (* x x)) 7)
6. (let ([x '(1 2 3)]) (if (null? x) x (rest x)))
7. (let ([x 3] [y 4])
        (let ([x 2] [z (+ x 2)]) z))
8. (map (lambda (x) (+ x 1)) '(1 2 3 4))
```

2. Programming

(25 points)
For the following problems you may use any element of Scheme presented in lecture or the Scheme Review handout. You may use any of DEFINE, LET, and LETREC to create bindings unless otherwise specified. You may not use mutation. Ask if you feel you need to use a feature not presented in lecture or the handout. You may check your work using a computer.

To run Dr. Scheme in the Unix lab, you have to first configure the IDE. At an XTERM, in your home directory, type:

mkdir ~/.plt-scheme
ln –s /usr/local/334/4.1.4 ~/.plt-scheme/4.1.4

From then on, you can run Dr.Scheme by typing "/usr/local/plt/bin/drscheme &" at an xterm command prompt. (You can also download it from http://download.plt-scheme.org/ for most operating systems.) When Dr. Scheme comes up, you will see a window split into two panes. Be sure that the drop-down box on the lower-left of the window says "PLAI Scheme". If it does not, press the down arrow on that box, select "Choose Language..." and select PLAI Scheme.

In the Dr. Scheme window, the upper pane is the "definitions" pane. That contains your program. The lower pane is the "interactions" window. You can type short expressions here to have them evaluated immediately. This is a good way to launch your procedures and test aspects of the language. Pressing "run" evaluates the contents of the definitions pane. Pressing "debug" lets you single-step through the contents of the definitions pane in a graphical debugger. This takes some experimentation to use effectively, but is a valuable tool when programs grow large.

Write separate Scheme procedures to:

9. Return the largest of three numbers specified as separate arguments named *a*, *b* and *c* (do not use the built-in max procedure).

10. Compute the sum of the numbers in a linked list of integers named *L*.

11. Compute the length of a list (without using the built-in length function!)

12. Efficiently compute exponentiation by a non-negative integer exponent by the following recursive algorithm:

$$b^x = \begin{cases} 1 & x = 0 \\ \left(b^{(x/2)}\right)^2 & x \text{ is even} \\ b * \left(b^{(x-1)}\right) & x \text{ is odd} \end{cases}$$

13. Implement an efficient (contains? tree target) that returns #t if a value eq? to *target* appears anywhere within *tree*, which is a tree constructed from nested linked lists, and #f otherwise. E.g., (contains? '(a b (c d) (e (f g h))) 'g) returns #t and (contains? '(a b (c d) (e (f g h))) 'z) returns #f.

14. Let *f* and *g* be two one-argument functions. The composition *f* after *g* is defined to be the function that maps *x* to *f*(*g*(*x*)). Define a procedure composer that implements composition. For example, if add1 is a procedure that adds 1 to its argument, then `((composer square add1) 6)` returns `49`.

15. Write an expression to reduce (using the foldl function) a list of symbols to true if one or more of them are eq? to 'yes and false otherwise.

3. Short Answer

(15 points)

16. In addition to integer and decimal numeric literals, Scheme also supports complex numbers and rational numbers literals. An example of a complex literal is: 3+2i. An example of a rational literal is 7/5. Write a BNF grammar for number literals in Scheme that includes integers (you do not have to support hexadecimal), numbers with decimal places (you do not have to support exponential notation), complex numbers, and rational numbers. Start by copying the <int> and <decimal> definitions from class. Some examples of numbers that your grammar should be able to express include 7, -23, 0.14, 7.1+5i, 3/2+4i and 14/7.

17. The expression domain of a language describes the set of legal programs. The value domain describes the types of values that exist at runtime. For our purposes, to be in the value domain, something must be "first-class": it can be stored in a data structure, returned from a function, passed as an argument, and variables can be bound to it. Since the value

domain is generally large, we generally refer to sets of values, like "int", instead of individual values, like "4". We call these sets "types".

a. What types form the value domain of Java?
b. What types have literals in Java?
c. What types do not have literals in Java?
d. What type (that we know of from the Scheme Overview) does not have corresponding literals in Scheme? There is only one.

4. Challenges

(glory and experience, but no points awarded for these)

18. Write a BNF for Scheme list literals (as presented in class) that does *not* use the regular expression shorthand. Assume that <id>, <num>, <symbol>, and <bool> have already been defined. It should handle recursive lists, for example '(1 2 (a b (c)) 3 4 5).

19. Write a procedure to reverse a list in linear time. Do not use the reverse or list->vector procedures, but you can use reverse to check your work. Note that the built-in append function takes linear time by itself.

20. Rewrite your solution to part 13 so that it does not use IF, CASE, or COND.

21. Rewrite your solution to part 11 using only literals, LAMBDA, IF, NULL?, REST, and + (specifically, no DEFINE, COND, LENGTH, LET, or LETREC.)

22. Although not all of the value domain is supported by literals (see 17cd), all literals in Scheme and Java are in the value domain. Why?