CS 257: Non-Imperative Programming: Scheme!
Homework 4 (Spring '05)

1. Write a function, *cond->if*, which takes a *cond* expression, and transforms it into a set of nested *if* expressions. For example,

```
> (cond->if '(cond ((> x y) (- x y)) ((< x y) (- y x)) (else 0)))
(if (> x y) (- x y) (if (< x y) (- y x) 0))
>
```

2. Get the code for polynomial arithmetic from the class homepage. Re-write the function, *p**, which multiplies two polynomials, so that it is completely tail-recursive. Hint: This should be done in two parts. First, re-write the subroutine, *t**, which multiplies a polynomial by a term so that it is tail-recursive. After you have tested *p** and verified that it works correctly with the tail-recursive *t**, then re-write *p** itself, so that the entire function is tail-recursive. You will need to add tail-recursive subroutines to the *letrec* (e.g., *t*-it* and *p*-it*) to do this.

3. Get the code for symbolic differentiation from the class homepage. Extend the *deriv* function so that it differentiates expressions of the form, $u-v$. The relevant differentiation rule is $d(u-v)/dx = du/dx - dv/dx$. In order to do this, you will need to add a new constructor function, *make-diff*, which takes expressions, u and v as arguments, and returns the expression for $u - v$. You will need to add two new selector functions, *minuend*, and *subtrahend*, which, when given an expression of the form, $u - v$, return u and v respectively.

4. Extend the *deriv* function so that it differentiates expressions of the form, u^n, where u is an arbitrary expression and n is a constant or number. The relevant differentiation rule is $du^n/dx = nu^{n-1}du/dx$. In order to do this, you will need to add a new constructor function, *make-expt*, which takes an expression, u, and a constant or number, n, as arguments and returns the expression for u^n. You should use *expt* to represent the exponentiation operator. You will also need to add two new selector functions, *base*, and *power*, which, when given an expression of the form, u^n, return u and n respectively.

5. The function *list?* takes an s-expression, s, as an argument and returns $\#t$ if s is a list and $\#f$ otherwise. Give a definition of the function, *list-all?*, which takes an s-expression, s, as an argument and returns $\#t$ if s is a list and all pairs in s are also lists. Otherwise, *list-all?* returns $\#f$. For example, *(list-all? '(1 ((2 3)) (4)))* should return $\#t$ and *(list-all? '(1 ((2 . 3)) (4)))* should return $\#f$. You may use *list?* to define *list-all?*.