# CSE 341 Section Handout #7
# Cheat Sheet

## Higher-Order Functions

```
(map procedure list)
(filter procedure list)
(foldl procedure initialValue list)
(foldr procedure initialValue list)
```

## Improper Lists (Pairs)

```
(cons expr expr)                                ; if second expr is not a pair/list
'(expr . expr)

(list '(expr . expr) '(expr . expr) ... '(expr . expr))  ; association list (map)
(assoc key associationList)           ; returns the key/value pair for the given key
```

## Symbols

```
(quote name)
'name
(symbol? expr)
(symbol=? expr expr)                  ; or use eq? or equal?
'(code)                               ; list of symbols; code as data
(eval code)
```

## Testing and Converting Between Types

```
(type? expr)
  (number? expr)     (integer? expr)     (rational? expr)
  (real? expr)       (exact? expr)       (inexact? expr)
  (symbol? expr)     (string? expr)      (list? expr)
  (pair? expr)       (procedure? expr)
(exact->inexact expr)
(inexact->exact expr)
(string->symbol expr)
(symbol->string expr)
```

## Raising Exceptions

```
(error string)
```

## String Procedures

```
(string-length str)
(substring str start end)
(string-append str str2 str3 ... strN)
(string->list str)
(list->string charList)
(string<? str1 str2)      ; also <=, =, >=, >
(string-ci<? str1 str2)   ; case-insensitive
(string-upcase str)
(string-downcase str)
(string-titlecase str)
(string-upcase str)
```

# CSE 341 Section Handout #7
# Questions

## General Scheme Programming

**1.** Define a procedure `count` that takes a value and a list as parameters and that returns the number of occurrences of the value in the list. You should use a deep equality comparison. For example:

- `(count 3 '(7 9 2 4 a (3 2) 3 "hello" 3))` should return `2`
- `(count 'a '(3 a b a 19 (a b) c a))` should return `3`
- `(count '(a b) '(a b c (a b) d 3 a b))` should return `1`

**2.** Define a procedure `zip` that takes two lists as parameters and that returns the list obtained by combining pairs of values in corresponding positions into lists of 2 elements. The first element should be a list containing the first values from each list. The second element should be a list containing the second values from each list. And so on. If one list is shorter than the other, then you should return a list of that shorter length. For example:

- `(zip '(1 2 3) '(a b c))` should return `((1 a) (2 b) (3 c))`
- `(zip '(1 2 3 4 5) '(a b c d))` should return `((1 a) (2 b) (3 c) (4 d))`

## Derivative Example

For the rest of the problems, begin with the definition of the `deriv` procedure shown in lecture:

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((symbol? exp)
         (if (eq? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (cadr exp) var)
                   (deriv (caddr exp) var)))
        (else (error "illegal expression"))))

(define (make-sum exp1 exp2)
  (list '+ exp1 exp2))

(define (sum? exp)
  (and (pair? exp) (eq? (car exp) '+)))
```

**3.** Extend the code to handle products as well as sums. Recall the product rule of derivatives:

$(f \cdot g)' = (f')g + f(g')$

In writing this code, introduce new helper procedures named `product?` and `make-product?`.

# CSE 341 Section Handout #7
# Problems (continued)

## Derivative Example (continued)

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((symbol? exp)
         (if (eq? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (cadr exp) var)
                   (deriv (caddr exp) var)))
        (else (error "illegal expression"))))

(define (make-sum exp1 exp2)
  (list '+ exp1 exp2))

(define (sum? exp)
  (and (pair? exp) (eq? (car exp) '+)))
```

**4.** Write new versions of `make-sum` and `make-product` that perform some simplification. For example, there is no reason to produce results like:

```
(+ x 0)
(+ 1 1)
```

The first is simply `x` and the second is `2`. Try to think of as many cases as you can to simplify.

**5.** Modify the derivative code to allow expressions with more than 2 values being added or multiplied:

```
> (derivative '(+ x x x x x x) 'x)
6
```

It is best to do this while minimally modifying the `deriv` function itself. One way to make this work is to modify the code so that everywhere the code asks for the second argument to a derivative, currently using `caddr`, it will instead call a procedure you'll write named `arg2` that sometimes returns an expression list with a `+` or `*` as the first element. In the example above, it would behave this way:

```
> (arg2 '(+ x y z a b c))
(+ y z a b c)
```

Of course, `arg2` should still return a single result in the simple case:

```
> (arg2 '(+ x y))
y
```

**6.** Extend the derivative function so that it handles expressions that involve a variable carried to a numeric exponent. For example, instead of saying:

```
> (derivative '(* x x x x x x) 'x)
```

We want to be able to say:

```
> (derivative '(^ x 6) 'x)
```

The clauses always begin with an `^` for exponentiation followed by a single variable and a number.

# CSE 341 Section Handout #7
# Solutions

1.

```
(define (count target lst)
  (cond ((null? lst) 0)
        ((equal? (car lst) target) (+ 1 (count target (cdr lst))))
        (else (count target (cdr lst)))))
```

2.

```
(define (count target lst)
  (cond ((null? lst) 0)
        ((equal? (car lst) target) (+ 1 (count target (cdr lst))))
        (else (count target (cdr lst)))))
```

3.

```
(define (derivative exp var)
  (cond ((number? exp) 0)
        ((symbol? exp)
         (if (eq? exp var) 1 0))
        ((sum? exp)
         (make-sum (derivative (cadr exp) var)
                   (derivative (caddr exp) var)))
        ((product? exp)
         (make-sum
          (make-product (cadr exp)
                        (derivative (caddr exp) var))
          (make-product (derivative (cadr exp) var)
                        (caddr exp))))
        (else (error "illegal expression"))))

(define (product? exp)
  (and (pair? exp) (eq? (car exp) '*)))

(define (make-product exp1 exp2)
  (list '* exp1 exp2))
```

4.

```
(define (make-sum exp1 exp2)
  (cond ((eq? exp1 0) exp2)
        ((eq? exp2 0) exp1)
        ((and (number? exp1) (number? exp2)) (+ exp1 exp2))
        (else (list '+ exp1 exp2))))

(define (make-product exp1 exp2)
  (cond ((or (eq? exp1 0) (eq? exp2 0)) 0)
        ((eq? exp1 1) exp2)
        ((eq? exp2 1) exp1)
        ((and (number? exp1) (number? exp2)) (* exp1 exp2))
        (else (list '* exp1 exp2))))
```

# CSE 341 Section Handout #7
# Solutions

5.

```
(define (derivative exp var)
  (cond ((number? exp) 0)
        ((symbol? exp)
         (if (eq? exp var) 1 0))
        ((sum? exp)
         (make-sum (derivative (cadr exp) var)
                   (derivative (arg2 exp) var)))
        ((product? exp)
         (make-sum
          (make-product (cadr exp)
                        (derivative (arg2 exp) var))
          (make-product (derivative (cadr exp) var)
                        (arg2 exp))))
        (else (error "illegal expression"))))

(define (arg2 exp)
  (if (pair? (cdddr exp))
      (cons (car exp) (cddr exp))
      (caddr exp)))
```

6.

```
(define (derivative exp var)
  (cond ((number? exp) 0)
        ((symbol? exp)
         (if (eq? exp var) 1 0))
        ((sum? exp)
         (make-sum (derivative (cadr exp) var)
                   (derivative (caddr exp) var)))
        ((product? exp)
         (make-sum
          (make-product (cadr exp)
                        (derivative (caddr exp) var))
          (make-product (derivative (cadr exp) var)
                        (caddr exp))))
        ((pow? exp)
         (if (eq? (cadr exp) var)
             (make-product (caddr exp)
                           (make-pow (cadr exp) (- (caddr exp) 1)))
             0))
        (else (error "illegal expression"))))

...
(define (pow? exp)
  (and (pair? exp) (eq? (car exp) '^) (symbol? (cadr exp))
       (number? (caddr exp))))

(define (make-pow exp1 exp2)
  (cond ((= exp2 0) 1)
        ((= exp2 1) exp1)
        (else (list '^ exp1 exp2))))
```