CSE 341 Sample Final Exam Problem Set #3

(The real exam won't have this many problems. We are giving you longer practice problem sets so you will have lots of problems to practice when you study. The kinds of problems you see will match the ones shown on these practice tests on our web site.)

1. Big Picture (short answer)

What is a closure? Why are closures useful? Describe one difference between the way ML handles closures and the way closures behave in Scheme or JavaScript.

2. ML Functions

Define a ML function called `ratio` that accepts a list of `int * int` tuples and returns a real number representing the sum of all tuples' first elements divided by the sum of all tuples' second elements. If the list is empty, return `0.0`. For full credit, your solution must be **tail recursive** and must run in O(*n*) time for a list of length *n*, making a single pass over the list. For example:

```
- ratio([(1, 1), (1, 2), (1, 3)]);                    (* (1+1+1) / (1+2+3) *)
val it = 0.5 : real
- ratio([(3, 4)]);
val it = 0.75 : real
- ratio([(1, 2), (3, 4), (5, 6), (10, 20)]);     (* (1+3+5+10) / (2+4+6+20) *)
val it = 0.59375 : real
- ratio([]);
val it = 0.0 : real
```

3. ML Curried Functions / Composition

Define a ML function called `backWords` that accepts a list of one-word strings and returns a new list containing the same strings but in the opposite order, and with each string's characters in the opposite order and converted to uppercase. Define the function using a `val` declaration with curried functions and the function composition operator `o`. Do not define any helper functions using `fun` declarations or the `fn` anonymous function notation. The list or any of its strings might be empty. See the exam rules sheet if you need a reference of string/char/list methods.

```
- backWords ["the", "QUICK", "BroWn", "", "foX"];
val it = ["XOF", "", "NWORB", "KCIUQ", "EHT"] : string list
```

CSE 341 Sample Final Exam Problem Set #3

4. Scheme Expressions

For each sequence of Scheme expressions below, indicate what value the final expression evaluates to. If an expression produces an error when evaluated, write `error` as your answer. Be sure to write a value of the appropriate type (e.g., `7.0` rather than `7` for a `real`; `String`s in quotes, e.g. `"hello"`; symbols with a leading quote, e.g. `'hello`; `true` or `false` for a `bool`).

Expression — Value

a)
```
(define a 2)
(define b 4)
(define c 8)
(let ((a (+ a c))
      (b (+ a b)))
  (list a b c))
```

b)
```
(define a 4)
(define b 8)
(define c 12)
(let* ((a (+ a c))
       (c (+ a b)))
  (list a b c))
```

c)
```
(define x 3)
(define (f n) (+ x n))
(define a (f 2))
(define x 13)
(define b (f 2))
(let ((x 23))
  (list a b x))
```

d)
```
(define a '(1 2))
(define b (cons 8 a))
(define c (append b a))
(list (eq? a (cdr b)) (eq? a (cddr c)) (eq? a (cdddr c)))
```

5. Scheme Procedures

Define a Scheme procedure `cube-sum` that accepts an integer *n* as a parameter and that returns the sum of the cubes of the first *n* positive integers. For example:

```
> (cube-sum 4)
100
```

In this case the procedure is computing the sum of the cubes of the first 4 integers (1 + 8 + 27 + 64). It should be legal to ask for the sum of 0 integers:

```
> (cube-sum 0)
0
```

But `cube-sum` should call the `error` procedure with the message "negative argument" if passed a negative value:

```
> (cube-sum -3)
. negative argument
```

CSE 341 Sample Final Exam Problem Set #3

6. Scheme Procedures

Define a Scheme procedure called `consecutive` that accepts a list of integers as a parameter and that returns `#t` if it is a list of consecutive integers and that returns `#f` otherwise. A list of integers is considered consecutive if each successive element in the list is one more than the previous one. For example:

```
> (consecutive '(8 9 10 11 12))
#t
> (consecutive '(1 3 2))
#f
```

The empty list and the list of one integer are considered to be consecutive. You may assume that your procedure is passed a list and that it contains only integers.

7. Scheme Procedures

Define a Scheme procedure called `powers` that accepts an integer *n* and a number *m* as parameters and that returns a list of the first *n* powers of *m* starting with m^1. For example:

```
> (powers 8 2)
(2 4 8 16 32 64 128 256)
> (powers 6 3)
(3 9 27 81 243 729)
> (powers 5 2.4)
(2.4 5.76 13.824 33.1776 79.62624)
```

It should be possible to ask for 0 powers of a number or for powers of negative numbers:

```
> (powers 0 19)
()
> (powers 8 -22)
(-22 484 -10648 234256 -5153632 113379904 -2494357888 54875873536)
```

But powers should call the `error` procedure with the message "negative argument" if passed a negative value for *n*:

```
> (powers -8 2)
. negative argument
```

Your procedure must compute the powers through multiplication rather than calling standard procedures like `exp` or `expt`. Some of the points will be awarded based on your ability to be efficient about multiplications. Your procedure should not need to perform more than *n* multiplications in computing *n* powers of a number.

8. Scheme Procedures

Define a Scheme procedure called `vector-product` that returns the vector product of two numeric lists. If you have a list that represents the sequence (a1, a2, a3, ...) and a list that represents the sequence (b1, b2, b3, ...), their vector product is defined to be the list (a1 * b1, a2 * b2, a3 * b3, ...). If the lists differ in length, your procedure should call the error procedure with the message "lists of different length". For example:

```
> (vector-product '(2 8 4) '(7 19 23))
(14 152 92)
> (vector-product '(1 7) '(3 4 5))
. lists of different length
```

Your procedure should not change either of the lists passed as parameters.

CSE 341 Sample Final Exam Problem Set #3

9. Scheme Procedures

Define a Scheme procedure called `transpose` that returns the transpose of a matrix. The matrix will be specified as a list of lists (a list of rows of the matrix). The transpose is obtained by exchanging rows and columns. What was row 1 becomes column 1, what was row 2 becomes column 2, and so on. You may assume that the matrix is rectangular (each row has the same number of columns), but you may not assume it is square. For example, if you are passed a matrix that has 3 rows of 6 values, your function must return a matrix with 6 rows of 3 values:

```
> (transpose '((1 2 3 4 5 6) (7 8 9 10 11 12) (13 14 15 16 17 18)))
((1 7 13) (2 8 14) (3 9 15) (4 10 16) (5 11 17) (6 12 18))
```

The empty matrix is legal (represented by the empty list) but it is not legal to have a nonempty matrix with 0 rows or 0 columns. Your procedure should not change the matrix passed as a parameter.

10. Scheme Procedures

Define a Scheme procedure `matrix-product` that returns the product of two matrices of numbers. As in problem 6, a matrix is specified as a list of lists (a list of rows). The procedure takes an *n*-by-*k* matrix (*n* rows of *k* values) and a *k*-by-*m* matrix (*k* rows of *m* values) as parameters and returns the *n*-by-*m* matrix formed by taking the product of the two matrices (*n* rows of *m* values). The value in row *i* and column *j* in the result is defined to be the sum of the values in the vector product of row *i* from the first matrix and column *j* from the second matrix. For example:

```
> (matrix-product '((1 2 3) (4 5 6))
                  '((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15)))
((46 52 58 64 70) (100 115 130 145 160))
> (matrix-product '((1 2 3) (4 5 6)) '((1 2) (3 4) (5 6)))
((22 28) (49 64))
```

Your procedure should not change either matrix passed as parameters.

11. Scheme Streams

Define a Scheme **stream** (infinite list) called `factorials` that contains the factorials of all integers from 0 upward. In other words, `factorials` is a thunk that, when called, returns a pair containing 0! and another thunk that, when called, returns a pair containing 1! and another thunk that, when called, returns a pair containing 2! another thunk ... and so on. For full credit, each element of the stream must be computed in constant time as it is needed, not by looping or recursion all the way back from 1 to each *n*. For example, using the `stream-slice` procedure we wrote in class to grab the first several elements of the stream, we can make the following calls if your stream is defined properly:

```
> factorials
#<procedure:factorials>
> (factorials)
(1 . #<procedure:.../whateverfile.scm:169:16>)
> ((cdr (factorials)))
(1 . #<procedure:.../whateverfile.scm:169:16>)
> ((cdr ((cdr (factorials)))))
(2 . #<procedure:.../whateverfile.scm:169:16>)
> ((cdr ((cdr ((cdr (factorials)))))))
(6 . #<procedure:.../whateverfile.scm:169:16>)
> (stream-slice factorials 12)
(1 1 2 6 24 120 720 5040 40320 362880 3628800 39916800)
```

CSE 341 Sample Final Exam Problem Set #3

12. JavaScript Expressions

Assuming that the following variables have been defined:

```
var a = [1, 2, 3, 4];
var b = {"0": 1, "1": 2, "2": 3, "3": 4, length: 4};
```

What value is returned (or what output is produced) by each of the following expressions/statements?

(a) `a == b`

(b) `typeof(a) === typeof(b)`

(c) `for (var i = 0; i < a.length; i++) { print(a[i] == b[i]); }`

(d) `b[a.length]`

(e) `a.filter(function(n) { return n >= b["length"]; })`

13. JavaScript Functions

Define a function `sortWords` that accepts a string as a parameter and rearranges the words of that string to be in sorted alphabetical order as defined by the standard relational operators. Words are sequences of characters separated by one or more whitespace characters (which include spaces, tabs, and/or newlines). Your function should also collapse the gap between adjacent words to a single space. For example:

```
> sortWords("the   quick        brown  fox   jumps over  \n  zany   \t\t   dogs")
"brown dogs fox jumps over quick the zany"
```

Hint: Regular expressions are useful for solving this problem.

CSE 341 Sample Final Exam Problem Set #3

14. JavaScript Functions

Add a method `union` to all arrays that accepts another array *a* as a parameter and returns the union of this array and *a*. That is, your method should return a new array containing all elements of this array, followed by all elements of *a* that do not appear in this array. This array and/or *a* might be empty. The elements of the arrays might appear in any order; the arrays are not necessarily sorted. You may assume that the elements of each array are simple types that can be compared for equality using the standard relational operators. Here is an example call:

```
> var list = [1, 4, 9, 9, 2, 7, 1];
> list.union([2, 5, 1, 4, 8])          // 2, 1, and 4 are already in the list
[1, 4, 9, 9, 2, 7, 1, 5, 8]
```

15. JavaScript Functions

(a) Add a method `add` to all arrays that accepts an index and a value as parameters and inserts the value at that index, shifting elements right if necessary to make room for it. The method should return the element value that was added. If the index is negative, no add should occur and the method should return `false`. If the index is greater than the length of the list, the intervening elements should be `undefined`. For example:

```
> var list = [1, 4, 9, 9, 2, 7, 1];
> list.add(2, 888);
888
> list
[1, 4, 888, 9, 9, 2, 7, 1]
> list.length
8
> list.add(12, 555);
555
> list
[1, 4, 888, 9, 9, 2, 7, 1, , , , , , 555]
```

(b) Add a method `remove` to all arrays that accepts an index as a parameter and removes the value at that index, shifting elements left if necessary to fill the hole left behind. The method should return the element value that was removed. If the index is negative or past the end of the array, no removal should occur and the method should return `false`. For example:

```
> var list = [1, 4, 9, 9, 2, 7, 1];
> list.remove(4);
2
> list
[1, 4, 9, 9, 7, 1]
> list.length
6
> list.remove(6);
false
> list
[1, 4, 9, 9, 7, 1]
```

CSE 341 Sample Final Exam Problem Set #3
Solutions

1.

A closure is a first-class function that binds to variables that are defined outside of its local environment ("free variables"). Closures are useful because they allow functions to refer to important values later, without having to explicitly pass them in to the function as parameters or force the programmer to worry about them. They also help the function retain a "memory" of variables that have gone out of scope by the time the function is actually called. One difference between ML's closures and those of Scheme and JavaScript is that re-defining variables in ML actually creates a new variable, so if you re-declare a variable that a previous function has closed over, it will not change the value used by the function. In Scheme/JS, if you re-define a variable, it assigns it a new value, so any functions that have closed over that variable will also see the change (for better or worse).

2.

```
fun ratio(lst) =
    let
        fun helper([], n, d) = if d > 0 then real(n) / real(d) else 0.0
        |   helper((a, b)::rest, n, d) = helper(rest, n + a, d + b)
    in
        helper(lst, 0, 0)
    end;
```

3.

```
val backWords = List.rev o map (implode o List.rev o (map Char.toUpper) o explode);
```

4.

```
a)  (10 6 8)
b)  (16 8 24)
c)  (5 15 23)
d)  (#t #f #t)
```

5.

```
(define (cube-sum n)
  (cond ((< n 0) (error "negative argument"))
        ((= n 0) 0)
        (else (+ (* n n n) (cube-sum (- n 1))))))
```

6.

```
(define (consecutive lst)
  (cond ((null? lst) #t)
        ((null? (cdr lst)) #t)
        (else (and (= (cadr lst) (+ 1 (car lst)))
                   (consecutive (cdr lst))))))
```

CSE 341 Sample Final Exam Problem Set #3

7.

```
(define (powers n m)
  (define (helper count power lst)
    (if (= count 0)
        lst
        (helper (- count 1) (* power m) (append lst (list power)))))
  (if (< n 0)
      (error "negative argument")
      (helper n m ())))
```

8.

```
(define (vector-product lst1 lst2)
  (cond ((and (null? lst1) (null? lst2)) ())
        ((or (null? lst1) (null? lst2))
         (error "lists of different length"))
        (else (cons (* (car lst1) (car lst2))
            (vector-product (cdr lst1) (cdr lst2))))))
```

9.

```
(define (transpose matrix)
  (cond ((null? matrix) ())
        ((null? (car matrix)) ())
        (else (cons (map car matrix) (transpose (map cdr matrix))))))
```

10.

```
(define (matrix-product m1 m2)
  (map (lambda (lst1) (map (lambda (lst2)
                             (foldl + 0 (vector-product lst1 lst2)))
                           (transpose m2))) m1))
```

11.

```
(define (factorials)
  (define (helper n factn)
    (cons factn (lambda () (helper (+ n 1) (* n factn)))))
  (helper 1 1))
```

12.

a) false
b) true
c) true
   true
   true
   true
d) undefined
e) [4]

13.

```
function sortWords(s) {
    var words = s.split(/\s+/);
    words.sort();
    return words.join(" ");
}
```

CSE 341 Sample Final Exam Problem Set #3

14.

```
Array.prototype.union = function(a) {
    var toAdd = [];
    if (!a) { return; }
    for (var i = 0; i < a.length; i++) {
        if (this.indexOf(a[i]) < 0) {
            toAdd.push(a[i]);
        }
    }
    return this.concat(toAdd);
};

// alternate shorter solution using higher-order functions
Array.prototype.union = function(a) {
    var that = this;   // because the lambda below has its own 'this'  (closure)
    return this.concat(
        !a ? [] : a.filter(function(e) { return that.indexOf(e) < 0; })
    );
};
```

15.

```
a)
Array.prototype.add = function(index, value) {
    if (index < 0) { return false; }
    for (var i = this.length; i > index; i--) {
        this[i] = this[i - 1];
    }
    this[index] = value;
    return value;
};

b)
Array.prototype.remove = function(index) {
    if (index < 0 || index >= this.length) { return false; }
    var result = this[index];
    for (var i = index; i < this.length; i++) {
        this[i] = this[i + 1];
    }
    return result;
};
```