CSE 341 Sample Final Exam Problem Set #2

(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 the difference between static and dynamic typing? Describe each in up to one sentence. What are some relative advantages and disadvantages of each approach?

2. **ML Functions**

   Define a ML function called `countE` that accepts a list of strings and returns the total count of the number of occurrences of the letter "e" (case-insensitively) in all strings. The list or some of the strings inside it might be empty. Here are some example calls:

```
- countE(["hello", "ru feeling", "good?", "", "ELEMENTARY my DEAR!"]);
val it = 7 : int
- countE(["deeelighted", "to SEE", "u!"]);
val it = 6 : int
- countE([])  +  countE([""])  +  countE(["", " ", "!!", "^_^"]);
val it = 0 : int
```

3. **ML Curried Functions / Composition**

   Define a ML function called `sumMin` that accepts a list of lists of integers and returns the sum of the minimum values of all the inner lists. 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. You may assume that the overall list of lists, and each list inside it, contain at least one element.

```
- sumMin [[3, 2, 9], [1, 4, 8, 7], [5], [9, 7, 6, 8]];          (* 2 + 1 + 5 + 6 *)
val it = 14 : int
```

CSE 341 Sample Final Exam Problem Set #2

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 5)
(define b 10)
(define c 20)
(let ((b (+ a c))
      (a (+ b c)))
  (list a b c))
```

b)
```
(define a 5)
(define b 10)
(define c 20)
(let* ((b (+ a c))
       (a (+ b c)))
  (list a b c))
```

c)
```
(define x 10)
(define (f n) (* x n))
(define a (f 3))
(define x 20)
(define b (f 4))
(define c a)
(set! a 42)
(list a b c)
```

d)
```
(define a '(x y))
(define b (cdr a))
(define c (append b a))
(define d (cons 'y '(a b)))
(list c (eq? (cdr a) b) (eq? (cdr d) a) (eq? (cdr c) a))
```

5. Scheme Procedures

Define a Scheme procedure called `sum` that takes two lists of numbers as arguments and that returns the list obtained by adding values in corresponding positions in the two lists. For example:

```
> (sum '(10 20 30) '(2 4 6))
(12 24 36)
```

If one list has more values than the other, then those values should appear at the end of the result, as in:

```
> (sum '(10 20 30 40 50 60) '(2 4 6))
(12 24 36 40 50 60)
> (sum '(10 20 30) '(2 4 6 8 10))
(12 24 36 8 10)
> (sum () '(1 2 3 4))
(1 2 3 4)
```

You may assume that each argument is a list and that it contains only numbers.

CSE 341 Sample Final Exam Problem Set #2

6. Scheme Procedures

Define a Scheme procedure called `repeat` that takes an integer *n* and a value *v* as arguments and that returns a list composed of *n* occurrences of *v*. For example:

```
> (repeat 8 5)
(5 5 5 5 5 5 5 5)
> (repeat 4 'a)
(a a a a)
> (repeat 5 '(a b))
((a b) (a b) (a b) (a b) (a b))
> (repeat 0 19)
()
```

You should call the error procedure with the message "negative argument" if passed a negative value:

```
> (repeat -1 'a)
. negative argument
```

Your procedure must be efficient in several respects. It should be tail-recursive, it should run in O(*n*) time, and it should check the error condition only once.

7. Scheme Procedures

Define a Scheme procedure called `inflate` that takes a list of count/value pairs and that returns the list obtained by replacing each pair with a sequence of the given number of occurrences of the given value. For example:

```
> (inflate '((3 . 4) (2 . a) (5 . (a b))))
(4 4 4 a a (a b) (a b) (a b) (a b) (a b))
```

Notice that each element of the list is a dotted pair. The first pair indicates 3 occurrences of the value 4. The second pair indicates 2 occurrences of the value `a`. The third pair indicates 5 occurrences of the value `(a b)`. Notice that the resulting list has these sequences of values in place of the pairs.

The value `(5 . (a b))` would normally be displayed as `(5 a b)`, but for the purposes of this problem, it is probably easier to think of it as just another dotted pair like the others. It is possible for a count to be 0, as in:

```
> (inflate '((0 . 5) (2 . 3) (0 . 14)))
(3 3)
```

You may assume that you are passed a legal sequence of dotted pairs where the first value in each pair is a nonnegative integer.

CSE 341 Sample Final Exam Problem Set #2

8. Scheme Procedures

Define a Scheme procedure `same-structure` that takes two values as arguments and that returns true (`#t`) if and only if the two values have the same list structure. Two values are said to have the same structure if one of the following is true:

- Neither value is a list
- Both values are lists of the same length; values in corresponding positions in the two lists have the same structure

For example:

```
> (same-structure 3 'a)
#t
> (same-structure 3 '(a))
#f
> (same-structure '(1 2 3) '(a b c))
#t
> (same-structure '(1 (2 3) 4 (5 6)) '(x (a b) y (c d)))
#t
> (same-structure '(1 (2 3) 4) '(x (y z) (a)))
#f
```

Remember that Scheme has a predicate called `list?` that allows you to test whether a value is a list.

9. Scheme Procedures

Define a Scheme procedure called `deflate` that takes a list of values and that returns a list of count/value pairs where adjacent values that are equal have been collapsed. This operation is the opposite of the `inflate` operation of problem 4. For example:

```
> (deflate '(1 1 1 19 19 3 3 3 a a b))
((3 . 1) (2 . 19) (3 . 3) (2 . a) (1 . b))
> (deflate '(a b a b c c c))
((1 . a) (1 . b) (1 . a) (1 . b) (3 . c))
> (deflate ())
()
```

Notice that the values have to be adjacent to be collapsed. In the second example, the different occurrences of `a` and `b` are not collapsed because they are not adjacent, while the three occurrences of c are collapsed. You should use the `equal?` predicate to compare values for equality. You may assume that you are passed a list as an argument.

CSE 341 Sample Final Exam Problem Set #2

10. Scheme Procedures

Define a Scheme procedure called `depth-count` that takes a list as an argument and that returns a list with a count of how many non-list values occurred at each depth in the list. For each value, its depth is equal to the number of surrounding parentheses in the overall list. For example, given this list:

```
(8 (a b) (c (d e) f) x (y (z (w))))
```

The values `8` and `x` have a depth of 1 because they are surrounded by a single set of parentheses (they are values of the outermost list). The values `a`, `b`, `c`, `f`, and `y` all have a depth of 2 because they are surrounded by two sets of parentheses (they are values of lists embedded one level deep inside the outermost list). The values `d`, `e`, and `z` all have a depth of 3. The value `w` has a depth of 4. Thus, `depth-count` should indicate 2 values of depth 1, 5 values of depth 2, 3 values of depth 3, and 1 value of depth 4:

```
> (depth-count '(8 (a b) (c (d e) f) x (y (z (w)))))
(2 5 3 1)
```

For a simple list, all values are at a depth of 1:

```
> (depth-count '(1 2 3))
(3)
```

The result should include a count for each depth in the list, even if there are no actual values at that depth:

```
> (depth-count '(((8)) () (())))
(0 0 1)
```

This should be true even if the list has no non-list values at all:

```
> (depth-count ())
(0)
> (depth-count '(() ()))
(0 0)
```

Remember that Scheme has a predicate called `list?` that allows you to test whether a value is a list. Your solution should be reasonably efficient. You may assume that the argument passed to your procedure is a list.

11. Scheme Macros

Define a Scheme **macro** called `nand` that takes two boolean expressions as parameters and returns `#t` (or any value other than `#f`) if the logical NAND of the two boolean expressions is true, and `#f` otherwise. The logical NAND of two boolean expressions *a* and *b* is defined to be true if either *a* or *b* (or both) are false. It is the opposite of the logical AND operation. Your macro must perform *short-circuit* evaluation; if it already knows the result to return after evaluating the first expression, it should not evaluate the second expression. In your solution you may *not* use Scheme's `and` or `or`, but you may use other constructs such as `if` and `cond`. Here are some example calls:

```
> (list (nand #f #f) (nand #f #t) (nand #t #f) (nand #t #t))
(#t #t #t #f)
> (nand (< 2 10) (= 4 7))         ; only first test is true
#t
> (nand (< 2 10) (= 4 4))         ; both tests are true
#f
```

CSE 341 Sample Final Exam Problem Set #2

12. JavaScript Expressions

Assuming that the following variables have been defined:

```
var a = 1;
var b = [a, a];
var c = ["aa", "aa", "aa"];
```

What is the value returned by each of the following expressions?

(a) `b[0] + b[1]`

(b) `b.length === 1 + 1`

(c) `b[b.length] == 0`

(d) `a + b[a] + c[b[a]]`

(e) `c.map(function(x) { return x.substring(0, 1); })`

13. JavaScript Functions

Add a **variadic** ("var-args") JavaScript method called `count` to all arrays that takes any number of values as parameters and returns the sum of the number of occurrences of all of those values in the array. If the array is empty, if no values are passed to count, or the array contains no occurrences of any of the values, return 0. You may assume that each value to compare against is a primitive type such as a string or number. For full credit, your method should run in no worse than O(*m* * *n*) time, where *m* is the length of the array and *n* is the number of values passed to count. Here are some example calls:

```
> var a1 = [1, 3, 9, 3, 7, 7, 3, 3, 4, 8, 2, 3];
> a1.count(3)
5
> a1.count(9, 7, 8)
4
> ["x", "y", "z", "x", "x", "a", "z"].count("x", "q", 42, "z")
5
> [].count(42)
0
```

CSE 341 Sample Final Exam Problem Set #2

14. JavaScript Functions

Add a method `toUnique` to all strings that returns a new string with any consecutive occurrences of the same character collapsed into a single occurrence of that character. For full credit, your solution must use **regular expressions**. For example:

```
> var s = "aa bbb cccc dedeeee   ffFFF    g  !!!!!!";
> s.toUnique()
"a b c dede fF g !"
```

15. JavaScript Functions

Define a function `deepEquals` that accepts two objects as parameters and returns `true` (or any truthy value) if the objects are "deeply equal" to each other; that is, if they contain exactly the same properties with the same values and types; and `false` (or any falsy value) otherwise. If either object is `null` or `undefined`, the other must also be `null` / `undefined` for them to be equal. You may assume that neither object contains a property that is itself an object or array; assume that all properties are simple types or functions. Your code should ignore any properties that are functions. For example:

```
> var o1 = {a: "x", b: "y", c: 42};
> var o2 = {b: "y", c: 42, a: "x", f: function(){}};
> var o3 = {a: "x", b: "y", c: 42, d: 17};
> var o4 = {a: "m", b: 20, c: "q", d: 17};
> var o5 = {a: "x", b: "y", f: function() {}, d: 17, c: 42, g: function() {}};
> var o6 = {};
> var o7 = null;
> deepEquals(o1, o2)         // same properties/values  (f is ignored)
true
> deepEquals(o1, o3)         // o1 doesn't have a d property
false
> deepEquals(o3, o2)         // o2 doesn't have a d property
false
> deepEquals(o3, o4)         // o4 has different values for the properties
false
> deepEquals(o5, o3)         // same properties/values  (f/g are ignored)
false
> deepEquals(o6, o7) || deepEquals(o7, o6)    // o6 is not null
false
```

CSE 341 Sample Final Exam Problem Set #2
Solutions

1.

Static typing is where type-checking is performed at compile-time, while dynamic typing is where type-checking is performed at run-time. Type-checking is the process of verifying and enforcing rules and constraints of types upon a given program or piece of code. Static typing can be useful in that type errors are caught sooner, and some code can be better optimized if its type properties are known well at compile-time. Dynamic typing is useful because it allows functions to be more flexible and more easily accept arguments of varying types; new types can also be generated at runtime.

2.

```
fun countE([]) = 0
|   countE(lst) =
    let
        fun countOne([]) = 0
        |   countOne(c::rest) = (if Char.toLower(c) = #"e" then 1 else 0)
                                + countOne(rest)
    in
        reduce(op +, mapx(countOne o explode, lst))
    end;

(* alternative shorter solution *)
val countE =
    length o
    (List.filter (fn(c) => Char.toLower(c) = #"e")) o
    explode o
    (List.foldl op^ "");
```

3.

```
val sumMin = (List.foldl op+ 0) o (map (hd o (curry quicksort op<)));

val sumMin = (reducex op+) o (map (reducex Int.min));
```

4.

```
a)  (30 25 20)
b)  (45 25 20)
c)  (42 80 30)
d)  ((y x y) #t #f #t)
```

5.

```
(define (sum lst1 lst2)
  (cond ((null? lst1) lst2)
        ((null? lst2) lst1)
        (else (cons (+ (car lst1) (car lst2))
                    (sum (cdr lst1) (cdr lst2))))))
```

6.

```
(define (repeat n val)
  (define (explore m lst)
    (if (= m 0)
        lst
        (explore (- m 1) (cons val lst))))
  (if (< n 0)
      (error "negative argument")
      (explore n ())))
```

CSE 341 Sample Final Exam Problem Set #2

7.

```
(define (inflate lst)
  (if (null? lst)
      ()
      (append (repeat (caar lst) (cdar lst)) (inflate (cdr lst)))))o
(inflate '((2 . 3) (5 . 14) (7 . 21)))
(define (inflate lst)
  (foldr append () (map (lambda (x) (repeat (car x) (cdr x))) lst)))
```

8.

```
(define (same-structure x y)
  (cond ((or (null? x) (null? y)) (and (null? x) (null? y)))
        ((or (not (list? x)) (not (list? y)))
         (and (not (list? x)) (not (list? y))))
        (else (and (same-structure (car x) (car y))
                   (same-structure (cdr x) (cdr y))))))
```

9.

```
(define (deflate lst)
  (define (explore value n lst)
    (cond ((null? lst) (list (cons n value)))
          ((equal? (car lst) value) (explore value (+ n 1) (cdr lst)))
          (else (cons (cons n value)
                      (explore (car lst) 1 (cdr lst))))))
  (if (null? lst)
      ()
      (explore (car lst) 1 (cdr lst))))
```

10.

```
(define (depth-count lst)
  (foldl sum '(0)
         (map (lambda (x)
                (if (not (list? x))
                    '(1)
                    (cons 0 (depth-count x))))
              lst)))

(define (depth-count lst)
  (cond ((null? lst) '(0))
        ((not (list? (car lst))) (sum '(1) (depth-count (cdr lst))))
        (else (sum (cons 0 (depth-count (car lst)))
                   (depth-count (cdr lst))))))
```

11.

```
(define-syntax nand
  (syntax-rules ()
    ((nand a b)
     (if (not a) #t (not b)))))

(define-syntax nand
  (syntax-rules ()
    ((nand a b)
     (if a (not b) #f))))
```

12.

```
a) 2
b) true
c) false
d) "2aa"
e) ["a", "a", "a"]
```

13.

```
Array.prototype.count = function() {
    var count = 0;
    for (var i = 0; i < this.length; i++) {
        for (var j = 0; j < arguments.length; j++) {
            if (this[i] == arguments[j]) {
                count++;
                break;
            }
        }
    }
    return count;
};
```

14.

```
String.prototype.toUnique = function() {
    return this.replace(/(.)\1+/g, "$1");
};
```

15.

```
function deepEquals(o1, o2) {
    if (!o1) { return !o2; }    // check whether either object is null/undefined
    if (!o2) { return !o1; }

    // returns true if oB contains all properties from oA
    function containsAll(oA, oB) {
        for (var key in oA) {
            if (typeof(oA[key]) !== "function" && oB[key] !== oA[key]) {
                return false;
            }
        }
        return true;
    }

    return containsAll(o1, o2) && containsAll(o2, o1);
}
```