CSE 341, Autumn 2010
Final Exam
Thursday, December 16, 2010

Name: _____

Section: _____ **TA:** _____

Student ID #: _____

Please do not begin working until notified by the instructor.
Good luck!

Score summary: (for grader only)

Problem	Description	Earned	Max
1	Big Picture		8
2	ML Functions		8
3	ML Curry/Composition		8
4	Scheme Expressions		6
5	Scheme Procedures		8
6	Scheme Procedures		8
7	Scheme Procedures		10
8	Scheme Streams		8
9	JavaScript Expressions		10
10	JavaScript Functions		8
11	JavaScript Regex		8
12	JavaScript Functions		10
TOTAL	**Total Points**		**100**

Exam Rules

- You have **110 minutes** to complete this exam. You must stop working once the instructor calls for papers. You may receive a deduction if you keep working after the instructor calls for papers.
- The exam is open-book/notes. You must work alone and may not use any computing devices including calculators. Cell phones, music players, and other electronics may NOT be out during the exam for any reason.
- Please be quiet during the exam. If you have a question or need, please raise your hand.
- Corrections or clarifications to the exam will be written at the front of the room.
- Please obey the University Code of Conduct during the exam.
- When you have finished the exam, please turn in your exam quietly and leave the room.

Programming Guidelines:

Unless otherwise noted, you may call standard library functions available in the top-level environment, such as:

- **ML**: operators (`~`, `+`, `-`, `*`, `/`, `div`, `mod`, `::`, `@`, `^`, `o`, `not`, `andalso`, `orelse`, `<`, `>`, `<=`, `>=`, `=`, `<>`), numeric functions (e.g. `abs` and `Int.max`), list functions (e.g. `hd`, `tl`, `length`, `rev`, `foldl`, `foldr`), conversion functions (`real`, `trunc`, `floor`, `ceil`, `ord`, `chr`, `str`), string functions (`implode`, `explode`, `concat`, `size`), standard tuple functions (`#1`, `#2`, etc.), and any other functions from ML basic data type structures such as `Int`, `Real`, `String`, `Bool`, and `Char` (but not `List`);

```
fun quicksort(f, list), fun m -- n,          fun curry f x y,
fun mapx(f, list),      fun map f list,
fun filterx(f, list),   fun filter2 f list,  fun List.filter f list,
fun reduce(f, list),    fun reduce2 f list,  fun List.foldl/foldr f init list
```

- **Scheme**: (your code must run successfully in DrScheme's "Pretty Big" language level) standard math (`+`, `-`, `*`, `/`, `modulo`, `quotient`, `remainder`, `abs`, `sin`, `cos`, `max`, `min`, `expt`, `sqrt`, `floor`, `ceiling`, `truncate`, `round`, `exact->inexact`, `inexact->exact`), logic (`=`, `<`, `>`, `<=`, `>=`, `eq?`, `eqv?`, `equal?`, `and`, `or`, `not`), type predicates (`number?`, `integer?`, `real?`, `symbol?`, `boolean?`, `string?`, `list?`, `symbol?`, `null?`, `procedure?`), higher-order procedures (`map`, `filter`, `foldl`, `foldr`, `sort`, `andmap`, `ormap`), list procedures (`append`, `assoc`, `car`, `cadr`, `cdr`, `cddr`, `cons`, `length`, `list`, `member`, `remove`), symbols (`quote`, `symbol=?`, `string->symbol`, `symbol->string`), strings (`string-append`, `substring`, `string->list`, `string<?`, `string-ci<?`, `string-upcase`, `string-downcase`, `string-titlecase`), `error`, `delay`, `force`; macros, quasi-quotes
- **JavaScript**: standard operators, `print`, `typeof`, `parseInt`, `parseFloat`, strings (`charAt`, `charCodeAt`, `indexOf`, `join`, `length`, `match`, `replace`, `slice`, `split`, `toLowerCase`, `toUpperCase`), Math (`Math.abs`, `ceil`, `sin`, `round`, etc.), arrays (`concat`, `filter`, `indexOf`, `join`, `map`, `pop`, `push`, `reduce`, `reverse`, `shift`, `slice`, `sort`, `splice`, `unshift`), functions (`apply`, `call`, `bind`), `instanceof`; regular expressions, prototypes, variadic functions, anonymous functions (lambdas)

You also may call any function that is a problem on this exam, whether or not you correctly solve that problem.

The following are explicitly forbidden unless the problem specifically authorizes you to use them:

- **ML**: mutation; arrays; vectors; `while` loops
- **Scheme**: `eval`, mutation (`set!`, `set-car!`, `mcons`, `set-mcar!`, etc.)
- **JavaScript**: `eval`; `with`; the Underscore library or other JavaScript libraries; importing Java classes via Rhino

You don't need to write any `use`, `open`, `include`, `load`, or other "import" statements in your exam code. Do not abbreviate any code. You can write helper functions if they are defined locally and not at the top level.

Unless this page or the problem mentions otherwise, your code you write will be graded purely on external correctness (proper behavior and output) and not on internal correctness (style). Some functions do have specific style constraints.

1. **Big Picture (short answer)**
   Why do functional languages like ML and Scheme try to minimize mutable state? What is "bad" about mutable state, and what benefits can be achieved by avoiding the use of mutable state?

2. ML Functions

Define a ML function called `range` that accepts a list of integers and returns the range of values in the list. The range of a list is defined to be 1 more than the difference between the largest and smallest element of the list. An empty list is defined to have a range of 0, and a single-element list has a range of 1. For example:

```
- range([9, 8, 4, 7, 5]);
val it = 6 : int
- range([10, 10, 19, 3, 6, 21, 37, 11]);
val it = 35 : int
- range([]);
val it = 0 : int
- range([42]);
val it = 1 : int
```

For full credit, your solution must be **tail-recursive** and must make only **one pass** over the list.

3. ML Currying/Composition

Define a ML function called `acronym` that accepts a list of words (strings) and returns a string representing an acronym made from the strings in the list. An *acronym* is an abbreviation made from using the initial letters of each word of a given phrase. For example, the acronym for the Computer Science & Engineering department is "CSE".

```
- acronym ["Self", "Contained", "Underwater", "Breathing", "Apparatus"];
val it = "SCUBA" : string
```

The acronym you return should contain the first letters of all words in the list *that begin with an uppercase letter*. Any words in the list that do not begin with an uppercase letter are not represented in the acronym. For example:

```
- acronym ["National", "Association", "for", "the",
           "Advancement", "of", "Colored", "People"];
val it = "NAACP" : string
```

You may assume that the list passed in is non-empty and each string is a single word with at least one character.

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. (Hint: The `Char` structure contains many useful functions such as `toLower`, `toUpper`, `isLower`, `isUpper`, etc.)

4. Scheme Expressions

For each sequence of Scheme expressions below, indicate the value to which the final expression evaluates. 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`; `#t` or `#f` for a `bool`). Write the values the way they would appear in the DrScheme interaction pane.

Expression — Value

a)

```
(define x 1)
(define y 2)
(define z 7)
(let ((z (+ x y z)))
  (let* ((x (+ y z))
         (y (+ x z)))
    (list x y z)))
```

b)

```
(define x 1)
(define y 2)
(define (f n) (+ x y n))
(set! y 3)
(define a (f y))
(define c x)
(define x 4)
(define b (let ((a 5)) (f a)))
(list a b c x y)
```

c)

```
(define a 4)
(define b 5)
(define c 6)
(define L1 '(a b))
(define L2 (cons a b))
(define L3 (list a b))
(list (+ a b c)
      (eq? (length L1) (length L3))
      (equal? (car L1) (car L2))
      (eqv? (cdr L2) (cdr L3))
      (equal? (cdr L2) (cadr L3)))
```

5. Scheme Procedures

Define a Scheme procedure called `is-sorted` that takes a list of numbers as its parameter and returns `#t` (or any value other than `#f`) if the numbers in the list appear in non-decreasing order, and `#f` otherwise. You may assume that the parameter passed is a list and that each element of the list is a number. The empty list and any one-element list are considered to be sorted. For full credit, your solution should run in O(*n*) time for a list of size *n* and should make only a single pass over the elements of the list. Here are some example calls:

```
> (is-sorted '(1 4 4 5 7 8 12))
#t
> (is-sorted '(1 3 2 6 7 4 5))
#f
> (and (is-sorted '()) (is-sorted '(42)))
#t
```

6. Scheme Procedures

Define a Scheme procedure called `merge` that accepts two sorted lists of numbers as its parameters and returns a new list that contains all elements from both lists merged into a single list, in sorted order. If either list is empty, your procedure should return the complete contents of the other list. The classic way to perform this algorithm is to repeatedly compare the lists' front elements and choosing the smaller one each time until both lists are exhausted.

You may assume that both parameters are lists, that all elements of both lists are numbers, and that the numbers in both lists are arranged into sorted (non-decreasing) order. For full credit, your method should run in O($m + n$) time where m and n are the lengths of the two lists; you should make only a single pass over the elements of each list. Here are some example calls:

```
> (merge '(1 4 9 15 32 32) '(0 6 7 20))
(0 1 4 6 7 9 15 20 32 32)
> (merge '(15 29 47) '(15 17 20 22 99))
(15 15 17 20 22 29 47 99)
> (merge '(42) '(5 19 27 42 60))
(5 19 27 42 42 60)
> (merge '(4 9 11) '())
(4 9 11)
```

7. Scheme Procedures

Define a Scheme procedure called `level-sum` that computes the sum of all elements of the list that are at a given level of nesting. The procedure takes two parameters: a list *L*, where each element of *L* is either a number or a nested list; and an integer, representing the level of nesting to examine:

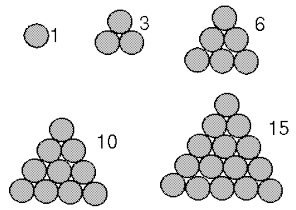
- Level 0 represents numbers directly contained in the list *L* itself (but not in any sub-lists inside *L*).
- Level 1 represents numbers inside of any sub-list *L′* directly nested inside of *L* (but not the numbers directly inside L, or any sublists of any *L′*, etc.).
- Level 2 represents all numbers inside of each list *L″* directly nested inside a list *L′* that is nested directly inside of *L*. ... And so on.

Your procedure should return the sum of all numbers in the list that are at the given level of nesting. If the list is empty or contains no values at the given level of nesting, or if the level passed is less than 0, your procedure should return 0. Here are some example calls:

```
> ; nesting   0 0  1 1 1  0  1  2 2  1  0
> (define L '(5 3 (6 4 1) 7 (2 (9 3) 5) 8))
> (level-sum L 0)                                    ; 5 + 3 + 7 + 8
23
> (level-sum L 1)                                    ; 6 + 4 + 1 + 2 + 5
18
> (level-sum L 2)                                    ; 9 + 3
12
> (level-sum L 3)                                    ; no elements at this level
0
```

8. Scheme Streams

Define a Scheme **stream** (infinite list) called `triangular` that contains the *triangular numbers*. The triangular numbers are an infinite sequence of integers where each number *i* represents the number of points occupied by an equilateral triangle of size *i*. A good way to visualize it is to think of triangles made of small circles as pictured in the diagram on this page at right. The first several triangular numbers are 1, 3, 6, 10, 15, 21, 28, 36, ... A useful observation is that the *i*th triangular number can be computed from the previous triangular number by adding *i* to it. For example, the 4th triangular number can be found by adding 4 to the third triangular number, 6, to make 10.



The First 5 Triangular Numbers

You are to define a stream that contains all of the triangular numbers. Your `triangular` variable stores a thunk that, when called, returns a pair containing the first triangular number (1) and another thunk that, when called, returns a pair containing the second triangular number (3) and another thunk that, when called, returns a pair containing the third triangular number (6) and 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:

```
> triangular
#<procedure:triangular>
> (triangular)
(1 . #<procedure:.../whateverfile.scm:169:16>)
> ((cdr (triangular)))
(3 . #<procedure:.../whateverfile.scm:169:16>)
> ((cdr ((cdr (triangular)))))
(6 . #<procedure:.../whateverfile.scm:169:16>)
> ((cdr ((cdr ((cdr (triangular)))))))
(10 . #<procedure:.../whateverfile.scm:169:16>)
> (stream-slice triangular 12)
(1 3 6 10 15 21 28 36 45 55 66 78)
```

9. JavaScript Expressions

Assuming that the following variables have been defined:

```
var a = [[1, 2], [3, 4, 5], [6]];
var b = 2;
var c = {a: b, b: "c", c: a};
```

What is the value returned by each of the following expressions? 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 number; `String`s in quotes, e.g. `"hello"`; ; `true` or `false` for a `boolean`).

(a) `a.length` _____

(b) `b === (parseInt("5 hi") - "3")` _____

(c) `typeof(a) + typeof(c)` _____

(d) `c.a + c.b` _____

(e) `a.map(function(x) {return x.length;})` _____

10. JavaScript Functions

Extend all JavaScript arrays to have a new **variadic** (var-args) method called `without` that accepts any number of values as parameters. The call `a.without(`*`arg1`*`, `*`arg2`*`, ..., `*`argN`*`)` should return a new array with the same elements as `a`, in the same order, except with any occurrences of the values *`arg1`* ... *`argN`* omitted. If no parameters are passed, the result should be the same as the original array. You may assume that the parameter values passed are of types that can be compared using the standard relational operators for equality.

```
> [2, 6, 4, 8, 4, 9, 7, 1, 2, 9, 4].without(4, 2, 7)
[6, 8, 9, 1, 9]
> [10, 10, 20, 20, 10, 20].without(10, 20)
[]
> ["a", "b", "c", "b", "e", "c", "d"].without("b", "x", "z", "y", "f", "e")
["a", "c", "c", "d"]
```

11. JavaScript Functions

Extend all JavaScript strings to have a new method called `isIpAddress` that returns `true` (or any truthy value) if the string is in the proper format of an internet protocol (IP) address, and `false` (or any falsy value) otherwise. An IP address contains four integers, each with up to three digits, and separated by periods (dots), such as `"123.65.8.214"`. (Real IP addresses demand that the numbers have values between 0 and 255, but we will ignore that requirement for this problem; you should accept any integer from 0 through 999.) Any extraneous text before or after the numbers is not allowed, including whitespace; strings containing such extraneous text would return a falsy result when your method is called on them. For full credit, your function must use **regular expressions** and its body can be no more than 3 lines long.

```
> "192.168.1.64".isIpAddress()
true
> "10.0.0.1".isIpAddress()
true
"128.208.3.88".isIpAddress()
true
> "eat at joe's".isIpAddress()
false
> "hi 1.2.3.4 bye".isIpAddress()
false
```

```
> ".1.2.3.4.".isIpAddress()
false
> "9999.9999.9999.9999".isIpAddress()
false
> "123 123 123 123".isIpAddress()
false
> "1.2.3".isIpAddress()
false
```

12. JavaScript Functions

Extend all JavaScript arrays to have a new method called `flatten` that returns a new array that contains the same values as the current array, but with the contents of any inner nested arrays "flattened" out to be part of the main overall array. That is, any nested array *a* containing elements [a_0, a_1, ..., a_n] is replaced by the elements a_0, a_1, ..., a_n in the overall array. Note that nested arrays may themselves contain other nested arrays, which would also need to be flattened. If a nested array is empty, it is discarded. Note that you are returning a new array; you should not modify the original array upon which the method is called.

JavaScript does not have a surefire way to detect whether a given value is an array; for this problem, assume that any element that is an object and has a `length` property is a nested array.

```
> var a = [1, 2, [3, 4], 5, [6, [7, 8, [9, 10], 11]], [], [[[[[12], 13]]]]];
> a.flatten()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

Your method should run in O(*n*) time where *n* is the combined length of the array and its nested arrays.