CMSC330 Fall 2009 Final Exam

Name _____

Do not start this exam until you are told to do so!

Instructions

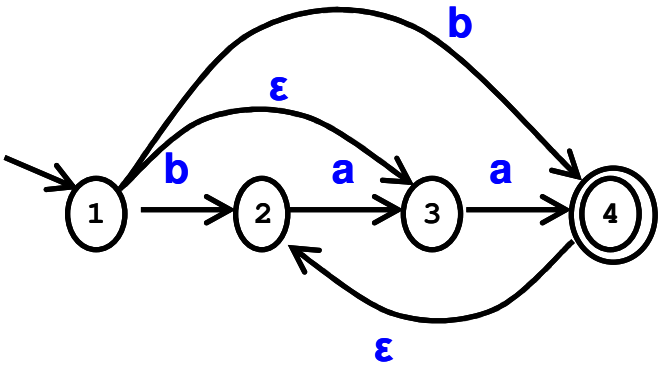
- You have 120 minutes for to take this midterm.
- This exam has a total of 120 points. An average of 1 minute per point.
- This is a closed book exam. No notes or other aids are allowed.
- If you have a question, please raise your hand and wait for the instructor.
- Answer essay questions concisely using 1-2 sentences. Longer answers are not necessary and a penalty may be applied.
- In order to be eligible for partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Problem	Score	Max Score
1	Programming languages		9
2	Ruby		6
3	RE, CFL, automata		12
4	OCaml types & type inference		6
5	Lambda calculus		6
6	Scoping		6
7	Parameter passing		8
8	Lazy evaluation		6
9	Garbage collection		5
10	Multithreading		14
11	Ruby Multithreading		20
12	OCaml programming / GC		22
	Total		120

1. (9 pts) Programming languages
   a. (3 pts) Briefly describe the difference between syntax and semantics of programming languages.
   b. (3 pts) Explain briefly what type inference is.
   c. (3 pts) List one of two possible *scoping* rules we discussed in class, and briefly describe a *disadvantage* for this approach.

2. (6 pts) Ruby
   What is the output (if any) of the following Ruby programs? Write FAIL if code does not execute.
   a. "CMSC 330" =~ /([a-zA-Z]+)/ # **Output =**
      puts $1
   b. a = { 1 => 2 } # **Output =**
      a.keys.each{ |x| puts a[x] }

3. (12 pts) Regular expressions, context-free grammars, and finite automata
   a. (6 pts) Convert the following NFA to a DFA using the subset construction algorithm. Be sure to label each state in the DFA with the corresponding state(s) in the NFA.



   b. (3 pts) Write a regular expression for the NFA above
   c. (3 pts) Write a context-free grammar for all binary numbers (strings consisting of 0s and 1s) of the form 0^n1^m, where n = m+2 and m ≥ 0. E,g,: 00, 0001, 00000111

4. (6 pts) OCaml types and type inference
   a. (3 pts) Give the type of **f** in the following OCaml expression
      let f x y = x + y in f 3 4 **Type =**
   b. (3 pts) Write an OCaml expression with the following type
      'a -> 'a **Code =**

5. (6 pts) Lambda calculus
   (3 pts each) Evaluate the following λ-expressions as much as possible
   a. (λx.x) (λy.y) (λz.z)
   b. (λx.λy.x) y

6. (6 pts) Scoping

Consider the following OCaml code.

```
let app f y = let x = 5 in let y = 4 in let a = 3 in f y ;;
let incr x  = let guess a = x+2 in app guess x ;;
(incr 1) ;;
```

a. (3 pts) What value is returned by (incr 1) with static scoping? Explain.
b. (3 pts) What value is returned by (incr 1) with dynamic scoping? Explain.

7. (8 pts) Parameter passing

Consider the following C code.

```
int i = 0;
void foo(int f, int g) {
  f  = f+2;
  g = f;
}
int main( ) {
  int a[] = {1, 2, 3};
  foo(i, a[i]);
  printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

a. (2 pts) Give the output if C uses call-by-value
b. (3 pts) Give the output if C uses call-by-reference
c. (3 pts) Give the output if C uses call-by-name

8. (6 pts) Lazy evaluation
   a. (2 pts) Briefly describe lazy evaluation.
   b. (4 pts) Rewrite the following code (using thunks) so that *incr* evaluates its argument only when it is used, even though OCaml uses call-by-value.

```
let incr x = x+1 ;;
incr (foo 2) ;;
```

9. (5 pts) Garbage collection

Consider the following Java code.

```
Jedi Darth, Anakin;
private void plotTwist( ) {
        Anakin = new Jedi( );  // object 1
        Darth = new Jedi( );     // object 2
        Anakin = Darth;
        Darth = Anakin;
}
```

a. (2 pts) What object(s) are garbage when plotTwist ( ) returns? Explain.
b. (3 pts) Briefly describe why stop-and-copy reduces memory fragmentation.

10. (14 pts) Multithreading

Consider the following attempt to implement the producer/consumer pattern in Ruby.

```
  class Buffer
    def initialize
      @lock = Monitor.new
      @cond = @lock.new_cond
      @buf = nil
      @empty = true
    end
    def produce(o)
      @lock.synchronize {
1.      @cond.wait_until { @empty }
      }
      @lock.synchronize {
2.      @empty = false
        @cond.broadcast
3.      @buf = o
      }
    end

    def consume
      @lock.synchronize {
4.      @cond.wait_while { @empty }
      }
      @lock.synchronize {
5.      @empty = true
        @cond.broadcast
6.      return @buf     # returns @buf and also releases the lock
      }
    end
  end

  t1 = Thread.new { produce 1 }
  t2 = Thread.new { produce 2 }
  t3 = Thread.new { x = consume }
  t4 = Thread.new { y = consume }
```

For the following problems, give schedules as a list of thread name/line number pairs, e.g., (t1, 1), (t3, 4)…

a. (3 pts) Give a schedule under which x = 1 and y = 2.
b. (3 pts) Give a schedule under which x = 2 and y = 1.
c. (4 pts) Give a schedule under which x = 1 and y = 1, or argue that no such schedule is possible.
d. (4 pts) Give a schedule under which x = 2 and thread 4 blocks.

11. (20 pts) Ruby multithreading

Using Ruby monitors and condition variables, write a Ruby class CountDownLatch that implements the following behavior:

- c = CountDownLatch.new(n) creates a new latch with count n.
- c.countDown() decrements the latch's count by one. If the count is already zero, it is not decremented further. This function may be called from different threads, so it must use locking to prevent data races.
- c.await() blocks the current thread until the count reaches 0, at which point the current thread is woken up and can continue. If the count is already 0, c.await does not block.

For example, suppose c = CountDownLatch.new(3). Here is a possible sequence of calls from some threads to this classes methods, and the effect:

initially - latch has count 3
thread 1 - c.countDown() - latch is now 2
thread 1 - c.await() - thread 1 suspended
thread 2 - c.countDown() - latch is now 1
thread 2 - c.await() - thread 2 suspended
thread 3 - c.countDown() - latch is now 0, thread 1 and 2 woken up
thread 3 - c.await() - nothing happens, since latch is 0
thread 3 - c.countDown() - nothing happen; latch cannot go below 0.

Hint: You'll likely want to use wait_while/wait_until and broadcast to sleep and let other threads know when the count is zero, respectively.

Helpful functions:

```
m = Monitor.new  // returns monitor
    m.synchronize { … }  // only 1 thread can execute code block at a time
    c = m.new_cond       // returns conditional variable for monitor
    c.wait_while { … }   // sleeps while code in condition block is true
    c.wait_until { … }   // sleeps until code in condition block is true
    c.broadcast          // wakes up all threads sleeping on condition var
    t = Thread.new {… }  // creates thread, executes code block in new thread
    t.join               // waits until thread t exits
```

12. (22 pts) OCaml programming / garbage collection

In the last problem of midterm 2, you implemented an interpreter for a small language with pointers. In this question, you will implement part of a very simple garbage collector for that language. Recall that values in this language are defined as

```
type value =
  Val_num of int        (* numbers *)
| Val_ptr of int        (* pointers *)
```

and that the program state included a memory of type

```
(int * value) list
```

Here the memory is an associative list, where the keys are addresses (integers from Val_ptrs) and the values are the contents of the address.

a. (4 pts) First, we need to look up values in memory. Assume you are given a function *lookup* : 'a -> ('a * 'b) list -> 'b such that "lookup k assoc_list" returns the element k is mapped to in assoc_list, or raises an exception if no such mapping exists. (This is the lookup function you implemented in midterm 2, and is also called assoc in the OCaml standard library). Using *lookup*, write a function

```
lookup_multiple : 'a list -> ('a * 'b) list -> 'b list
```

such that "lookup_multiple ks assoc_list" returns a list of the elements corresponding to the ks in assoc_list, in the same order as in ks, or raises an exception (the same exception as lookup) if there is some element in ks with no such mapping. Do not worry about efficiency. For example:

```
lookup 1 [(1,'a');(2,'b');(3,'c')]                  (* returns ‘a’ *)
lookup 3 [(1,'a');(2,'b');(3,'c')]                  (* returns ‘c’ *)
lookup_multiple [1] [(1,'a');(2,'b');(3,'c')]       (* returns  ['a'] *)
lookup_multiple [1; 3] [(1,'a');(2,'b');(3,'c')]    (* returns  ['a';'c'] *)
```

b. (6 pts) Second, we need to find all addresses in memory. Write a function

```
addresses : value list -> int list
```

that returns all the addresses in Val_ptrs in the input list. For example:

```
addresses [Val_num 1]                                       (* returns [] *)
addresses [Val_ptr 2]                                       (* returns [2] *)
addresses [Val_num 1; Val_ptr 2; Val_ptr 3; Val_num 4]      (* returns [2;3] *)
```

Do **not** use map, fold, or any library functions in your implementation.

c. (12 pts) Third, we need to find all reachable memory locations. Write a function

reachable : int list -> (int * value) list -> int list

such that "reachable addrs m" returns **all** addresses that are reachable in 0 or more steps starting from addrs. (Thus, the return value of this function must always include addrs itself). The order of addresses returned does not matter. For example:

```
let mem = [ (1, Val_num 42); (2, Val_ptr 43);
            (43, Val_ptr 44); (44, Val_num 45)] ;;
reachable [1] mem                    (* returns [1] *)
reachable [2] mem                    (* returns [2; 43; 44] *)
reachable [1; 2] mem                 (* returns [1; 2; 43; 44] *)
```

The problem is simplified by using the function *next_reachable*, which we can write using the functions *lookup_multiple* & *addresses*:

```
let next_reachable addrs mem = addresses (lookup_multiple addrs mem)
```

This function essentially "dereferences" all the addresses in *addrs* to find their contents in *mem*, and then returns only the addresses found. For example:

```
next_reachable [1] mem               (* returns [] *)
next_reachable [2] mem               (* returns [43] *)
next_reachable [43] mem              (* returns [44] *)
next_reachable [44] mem              (* returns [] *)
next_reachable [1; 2] mem            (* returns [43] *)
```

Essentially, next_reachable lets us compute one step of reachability: given a list *addrs* and a memory *mem*, if *addrs* are reachable, then so is everything in "next_reachable addrs mem".

When implementing reachable, you may use the function next_reachable. You may also assume you are given a function equal_sets : 'a list -> 'a list -> bool that returns true if and only if its two inputs contain the same elements in any order. For example:

```
equal_sets [1;2] [1;2]               (* returns true *)
equal_sets [1;2] [2;1]               (* returns true *)
equal_sets [1;2] [1;3]               (* returns false *)
```