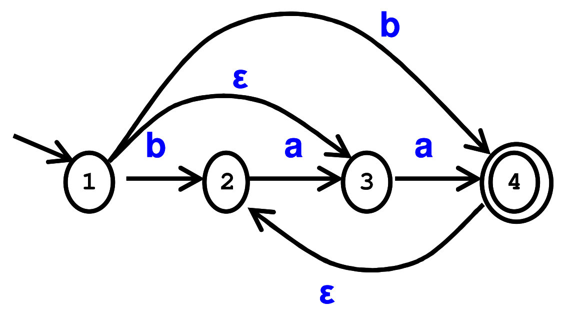
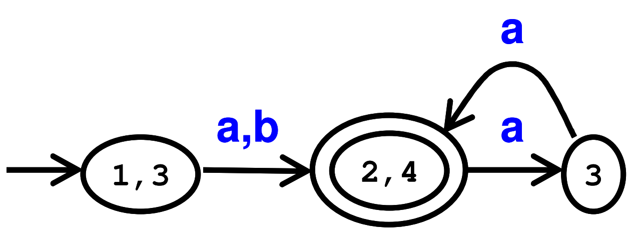
CMSC330 Fall 2009 Final Exam Solutions

1. (9 pts) Programming languages
   a. (3 pts) Briefly describe the difference between syntax and semantics of programming languages.
      Syntax refers to the program text, semantics refers to the program meaning.
   b. (3 pts) Explain briefly what type inference is.
      A compile-time algorithm for determining the type of a variable through its use.
   c. (3 pts) List one of two possible *scoping* rules we discussed in class, and briefly describe a *disadvantage* for this approach.
      Dynamic scoping – confusing semantics
      Static scoping – implementation complexity

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 = CMSC**
      puts $1
   b. a = { 1 => 2 } **# Output = 2**
      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.

Original NFA	Resulting DFA
	

   b. (3 pts) Write a regular expression for the NFA above
      (a|b)(aa)*

   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 $m \geq 0$. E,g,: 00, 0001, 00000111
      S → 00A **OR** **S → 0S1 | 00**
      A → 0A1 | epsilon

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 = int -> int -> int**
    b. (3 pts) Write an OCaml expression with the following type
        ‘a -> 'a **Code = fun x -> x OR let f x = x**

5. (6 pts) Lambda calculus
    (3 pts each) Evaluate the following λ-expressions as much as possible
    a. (λx.x) (λy.y) (λz.z)
        (λx.x) (λy.y) (λz.z) → (λy.y) (λz.z) → λz.z
    b. (λx.λy.x) y
        (λx.λy.x) y → (λx.λz.x) y → λz.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.
        3, since the x in guess is bound to formal parameter in incr x

        The sequences of calls & resulting values bound to the formal parameters is as follows.

        i. **incr (x=1) calls app (f=guess, y=1) binds (x=5, y=4, a=3) calls guess (a=4, since when guess is called the argument y is bound to 4)**
        ii. **In the body of guess x is free and refers to the x in incr x (x=1), leading to x+2=1+2=3**

    b. (3 pts) What value is returned by (incr 1) with dynamic scoping? Explain.
        7, since x in guess is bound to let x=5 (i.e., 5) in app f y

        Note “let z=5 in …” is really “(fun z-> …) 5” and adds a dynamic scope.

        The sequences of calls & resulting values bound to the formal parameters is as follows.

        i. **incr (x=1) calls app (f=guess, y=1) binds (x=5, y=4, a=3) calls guess (a=4, since when guess is called the argument y is bound to 4)**
        ii. **In the body of guess x is free and refers to the x in let x = 5 (x=5) in the body of app, leading to x+2=5+2=7**

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

      0 1 2 3   // unchanged

   b. (3 pts) Give the output if C uses call-by-reference

      2 2 2 3   // i=i+2, a[0] = i

   c. (3 pts) Give the output if C uses call-by-name

      2 1 2 2   // i=i+2, a[i] = i

8. (6 pts) Lazy evaluation

   a. (2 pts) Briefly describe lazy evaluation.

      Evaluating arguments only when they're used in the body of the function.

   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) ;;
let incr x = (x ( )) +1 ;;                 // 2 pts
incr (fun x -> (foo 2)) ;;                 // 2 pts
```

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.

      Object 1 is garbage, since both Anakin & Darth refer to object 2.

   b. (3 pts) Briefly describe why stop-and-copy reduces memory fragmentation.

      Because it copies live objects to a separate semispace, all free memory is contiguous in the new semispace.

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.

   (t1, 1-3), (t3, 4-6), (t2, 1-3), (t4, 4-6) OR
   (t2, 1-3), (t4, 4-6), (t1, 1-3), (t3, 4-6) OR …
   Key: (t1, 3) < (t3,6) & (t2, 3) < (t4,6)

b. (3 pts) Give a schedule under which x = 2 and y = 1.
(t1, 1-3), (t4, 4-6), (t2, 1-3), (t3, 4-6) OR
(t2, 1-3), (t3, 4-6), (t1, 1-3), (t4, 4-6) OR …
Key: (t1, 3) < (t4,6) & (t2, 3) < (t3,6)

c. (4 pts) Give a schedule under which x = 1 and y = 1, or argue that no such schedule is possible.
(t1, 1-3), (t3, 4), (t4, 4), (t3, 5-6), (t4, 5-6), (t2, 1-3) OR …
Key: (t3, 4), (t4, 4) < (t3,5), (t4,5) -> both t3, t4 consume same item

d. (4 pts) Give a schedule under which x = 2 and thread 4 blocks.
(t1, 1), (t2, 1), (t1, 2-3), (t2, 2-3), (t3, 4-6), (t4, 4) OR …
Key: (t1, 1), (t2,1) < (t1, 2), (t2,2) -> both t1, t2 produce at same time

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.

```
class CountDownLatch

        def initialize cnt
          @count = cnt
          @myLock = Monitor.new
          @myCondition = @myLock.new_cond
        end

        def countDown
          @myLock.synchronize {
             @count = @count - 1 if @count > 0
             @myCondition.broadcast if @count == 0
          }
        end

        def await
          @myLock.synchronize {
             @myCondition.wait_until { @count == 0 }
          }
        end
end
```

12. (22 pts) OCaml programming / garbage collection

a. (4 pts) First, we need to look up values in memory.

```
let rec lookup_multiple id_lst lst = match id_lst with
    [] -> []
    | (h::t) -> (lookup h lst)::(lookup_multiple t lst)
```

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

```
let rec addresses vlst = match vlst with
    [] -> []
    | (h::t) -> match h with
        Val_num n -> addresses t
        | Val_ptr n -> n::(addresses t)
```

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

```
let rec reachable addrs m =
    let x = addrs @ (next_reachable addrs m) in
    if (equal_states addrs x) then x else (reachable x m)
```