CMSC330 Spring 2008 Midterm #1 Solutions

1. (12 pts) Programming languages
   a. Explain why goals for programming languages have changed since the 1960's
      Because computers have become powerful (cheap) and programmers (relatively) expensive. // 2 pts
   b. Do programs run faster when they are interpreted or compiled? Explain why.
      Compiled, because interpreted programs have an additional layer of software (the interpreter). // 2 pts
   c. Describe the syntax and semantics of a Ruby **while** loop. Make clear which is which.
      Syntax = "while" followed by guard, followed by body, followed by "end".
      Semantics = if guard does not evaluate to nil, execute body, repeat // 2 pts
   d. Name a disadvantage of implicit variable declarations. Give a Ruby code example.
      Incorrect variable usage (e.g., misspelling) is not caught at compile time. "cat = 1" vs "ca = 1", where ca is not a real variable // 2 pts
   e. (4 pts) Explain why implicit variable declarations and static types are not considered orthogonal language features.
      Because it is difficult to statically check the type of a variable that is implicitly declared
      // knowing what orthogonal means // 2 pts
      // variable declarations interfere with type checking // 2 pts

2. (18 pts) Ruby features
   What is the output (if any) of the following Ruby programs? Write FAIL if code does not execute.
   a. 3.times {|i| puts i } **// 0 1 2 // 2 pts**
   b.
   ```
   x = 0
   puts "x" if x
   puts "not x" if !x
   ```
   // x // 2 pts
   c.
   ```
   a = [1,2] ; a[3] = "b" ; a.push("a")
   puts a
   ```
   // 1 2 nil b a // 2 pts
   d.
   ```
   x = "car"; y = x; x = "cat"
   puts y
   puts "==" if x=="cat"
   puts "equal" if !x.equal?("cat")
   ```
   // car == equal // 2 pts
   e.
   ```
   x = "CMSC 330"
   x =~ /([0-9])([0-9])([0-9])/
   puts "#{$1} #{$3}"
   ```
   // 3 0 // 2 pts
   f.
   ```
   h = {"guys" => 1, "gals" => 0}
   h["dudes"] = h["guys"] + 1
   h.values.each { |x| puts x }
   ```
   // 1 0 2 (any order) // 2 pts
   g.
   ```
   def down x
       until x < 0
           yield x
           x = x-1
       end
   ```

```
end
down(3) { |i| puts i }          // 3 2 1 0          // 2 pts
```

h. Describe the set of strings accepted by the Ruby regular expression /[^0-9]+/
1 or more consecutive non-digit characters.

i. Describe the set of strings accepted by the Ruby regular expression /^[0-9]+/
1 or more consecutive digits at the *beginning* of a line.

3. (23 pts) Ruby programming

Consider the following programming problem. We need to read the transitions of an NFA from a text file and count how many transitions exist for each label, then print the labels in alphabetical order. A transition is a line of the form **fromState.label.toState**, where states and labels are separated by a period (.). State names are numbers (sequences of digits), and labels are either lowercase letters between a and z, or the letter E (for epsilon). Write a complete Ruby program that finds the name of the text file containing the transitions from the command line, opens the file, reads the transitions, counts the number of transitions for each label, then prints each label (in alphabetical order) and its # of transitions (separated by a space) on a separate line. Any line that does not contain a valid transition should be ignored.

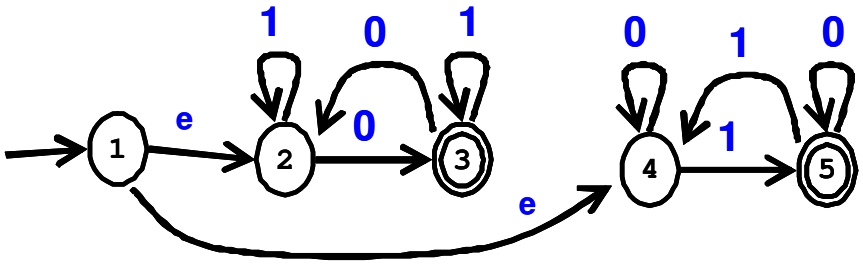
Example Input	11.c.2 1.a.22 21.b.3	11.A.2 21.b.9 21a99 3.E.45 22.a.2 22.b.2
Example Output	a 1 b 1 c 1	E 1 a 1 b 2

```
trans = { }                                  // init hash                    2 pts
file = File.new(ARGV[0], "r")                // ARGV[0]   open file          2 pts each
until file.eof? do
        line = file.readline                 // read all lines from file     3 pts
        if line =~ /(\d+)\.([a-zE])\.(\d+)/  // ensure valid format          4 pts
                if !trans[$2]                // look up label                2 pts
                        trans[$2] = 0        // init hash entry              2 pts
                end
                trans[$2] += 1               // increment count for label    2 pts
        end
end
labels = trans.keys                          // get labels                   2 pts
labels.sort!                                 // sort labels                  2 pts
labels.each { |x| puts("#{x} #{trans[x]}") } // print labels in order        2 pts
```

4. (10 pts) Regular expressions and languages
   a. Given the language A = {“aa”, ”c”}, what is the language A^1?
      {“aa”, “c”} **// 2 pts**
   b. Describe in English the language accepted by the regular expression 1(1|0)*1.
      Binary numbers beginning and ending in 1 **// 2 pts**
   c. Give a regular expression for all binary numbers that include more than two 0’s.
      1*01*01*0(0|1)* **// 3 pts**
   d. Give a regular expression for all binary numbers that don’t include exactly two 0’s.
      1* | 1*01* | 1*01*01*0(0|1)* **// 3 pts**

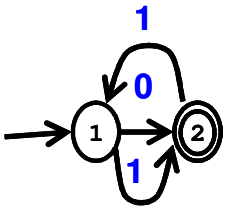
5. (10 pts) Finite automata properties
   a. (2 pts) How long could it take a DFA with n states and t transitions to accept a string with s symbols?
      S steps
   b. (4 pts) For a given language, can the minimal size DFA required to accept the language be smaller than the minimal size NFA required? Explain why.
      No, because all DFA are also NFA.
   c. (4 pts) The complement of a set S is the set of all elements not in S. Explain why for every regular expression R, we know there exists a regular expression ^R that accepts only strings not accepted by R.
      Because we can reduce R to a DFA, take its complement, and reduce the DFA back to a regular expression.

6. (27 pts) Finite automata
   a. (3 pts) Give a NFA for binary numbers with either an odd number of 0s or an odd number of 1s.



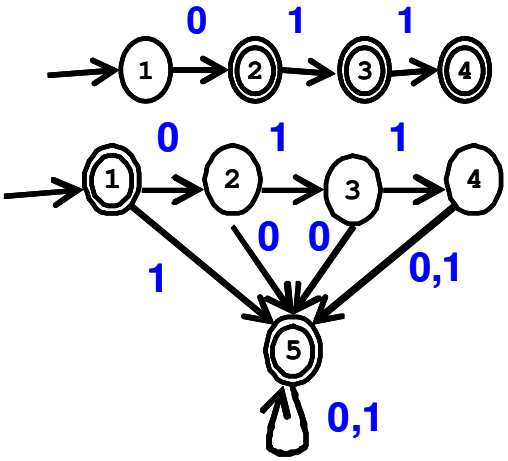
// odd # 0’s **1 pt**
// odd # 1’s **1 pt**
// union **1 pt**

   b. (3 pts) Give a regular expression for the language accepted by the following DFA



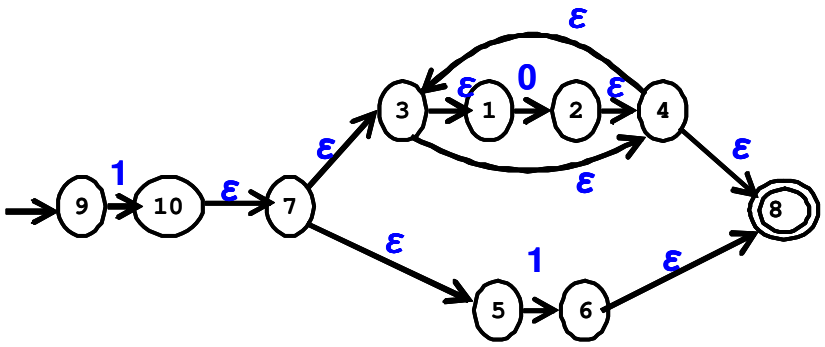
(0|1)(10|11)*

c. (3 pts) Give the DFA that is the complement of the following DFA.



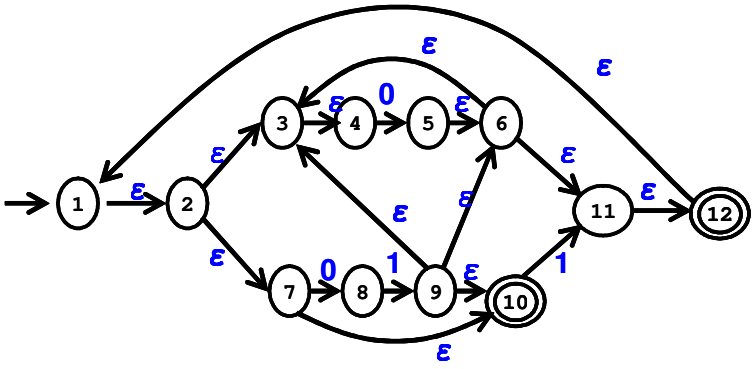
// swap states **1 pts**
// former dead state **2 pts**
// brand new DFA **(-1 pt even if correct)**

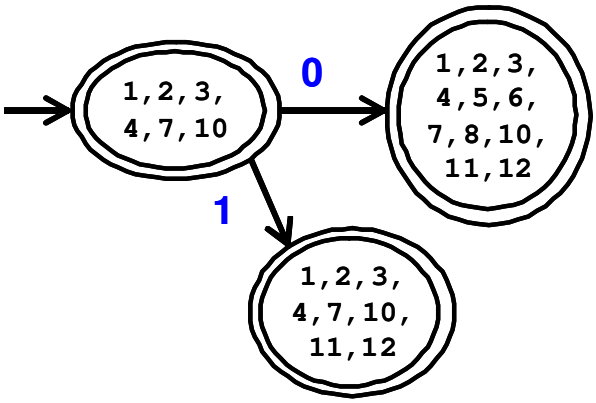
d. (4 pts) Reduce the regular expression 1(0* | 1) to an NFA, using the algorithm shown in class.



// correct result **2 pts**
// used construction method **2 pts**

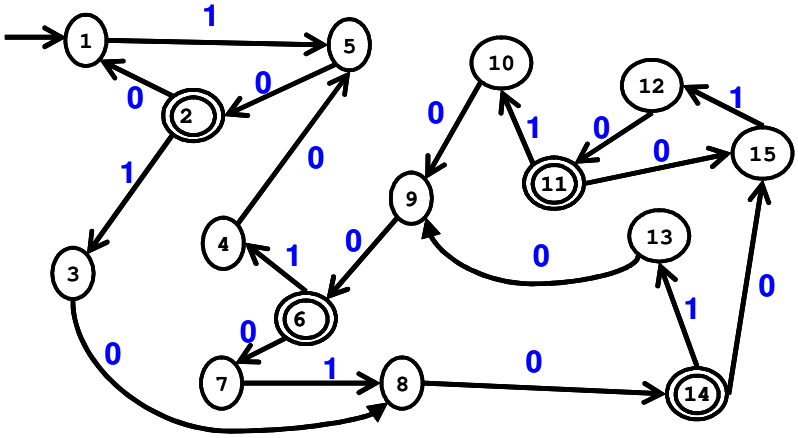
e. (8 pts) Start reducing the following NFA to a DFA using the subset algorithm. Draw the start state of the DFA, and the states created upon transitions on 0 and 1. List the NFA states represented by each of your DFA states. Which of the DFA states are final states? (Note you only need to create 3 DFA states, not the full DFA)

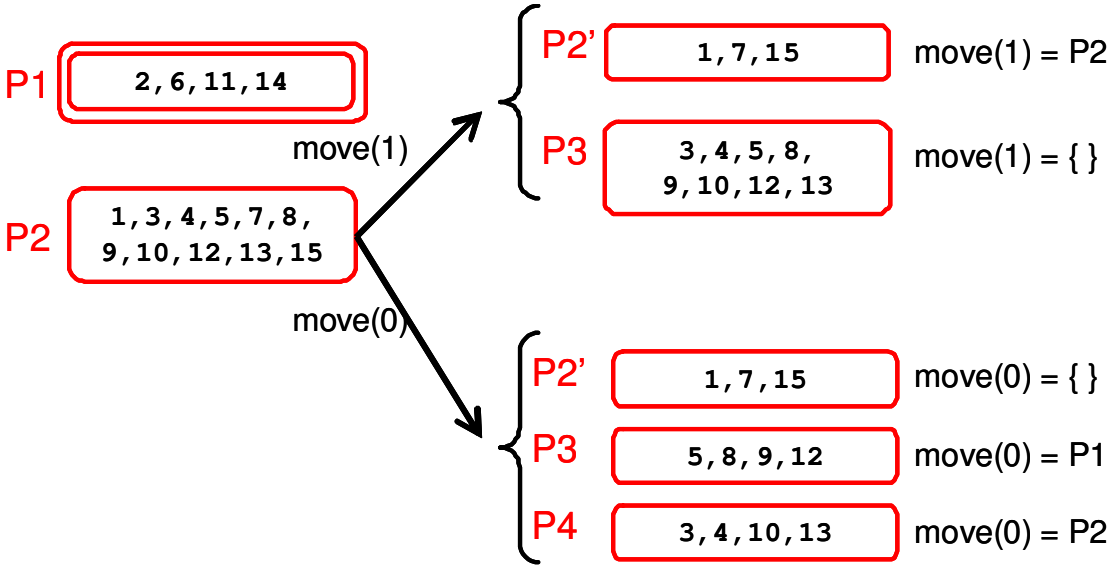




// start state	**2 pts**
// new states	**2 pts each**
// transitions	**1 pt**
// final states	**1 pt**

f. (6pts) Start applying Hopcroft reduction to minimize the following DFA. Show your initial partitions. Show the partitions resulting from your first split, and describe your reason for the split. (Show = list the DFA states in each partition)





// initial partitions 2 pts
// new partitions 2 pts
// explanation 2 pts