Midterm #1

CMSC 330: Organization of Programming Languages

March 5, 2013

Name _____

Instructions

Do not start until told to do so!

- This exam has 10 pages (including this one); make sure you have them all
- You have 75 minutes to complete the exam
- The exam is worth 100 points. Allocate your time wisely: some hard questions are worth only a few points, and some easy questions are worth a lot of points.
- If you have a question, please raise your hand and wait for the instructor.
- You may use the back of the exam sheets if you need extra space.
- 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.

	Question	Points	Score
1	PL concepts	6	
2	Ruby execution	12	
3	Regular expressions	12	
4	Finite automata (FA) concepts	8	
5	Non-deterministic FAs	12	
6	Deterministic FAs	12	
7	Context-free grammars	8	
8	Ruby programming	30	
	Total	100	

1. (Programming language concepts, 6 points total, 1 point each) Indicate whether the following are true ("T") or false ("F"):

   (a) ____ Ruby is object oriented

   (b) ____ Ruby uses dynamic type checking

   (c) ____ Ruby has syntax for method overloading

   (d) ____ Ruby supports both physical and structural equality

   (e) ____ A context-free grammar is ambiguous when it describes multiple languages

   (f) ____ A regular expression can be used to define the language $\{a^n b^n \mid 0 \leq n \leq 3\}$

   Answer:

   (a) T (b) T (c) F (d) T (e) F (f) T

2. (Ruby execution, 12 points total, each is worth 3 points) What is the output of the following Ruby programs? If the execution fails at run-time, write the output up until the failure happens, then write FAIL. If multiple outputs are possible, write them all.

(a)
```
h = { "yes" => [1,2] }
h["no"] = [3,4]
h.values.each { |v| print v[0] }
```

Answer:

Answer: *Multiple outputs:* `31` *(no newline) or* `13`

(b)
```
s = "abba is a band, is abba"
a = s.scan(/a[bn]|ba/)
puts a.length
```

Answer:

Answer: *5*

(c)
```
a = []
b = [1,2]
a[0] = b
a[1] = [1,2]
puts (a[0] == a[1])
```

Answer:

Answer: *true*

(d)
```
class D
  def initialize
    @n = 1
    @s = 3
  end
  def next
    @s = @s + @n
    @n = @n + 1
  end
  def state
    @s
  end
end
d = D.new
d.next
d.next
puts d.state
```

Answer:

Answer: *6*

3. (Regular expressions, 12 points)

(a) (4 points) Indicate whether or not the following strings are matched by the regular expression `aa*bb*|b+|(bb|aa)*`. (Write "A" if accepted, and "N" if not accepted.)

i. `aabb`

ii. `bbb`

iii. `aaa`

iv. `aaaa`

Answer:

(a) A; (b) A; (c) N; (d) A

(b) (4 points) Give a Ruby-compliant regular expression that defines the language containing all strings that have an even (but nonzero) number of `a` characters followed one, two, or three `b` characters. For example, `aab`, `aabb`, `aaaab`, and `aaaabbb` are all strings in this language.

Answer:

```
(aa)+b | (aa)+bb | (aa)+bbb
```

(c) (4 points) Give a Ruby-compliant regular expression that accepts the same strings as regular expression `a(b|c)*d+` but not strings `abd` or `acd`.

Answer:

```
(ad+)|a(b|c)dd+|a(b|c)(b|c)+d+
```

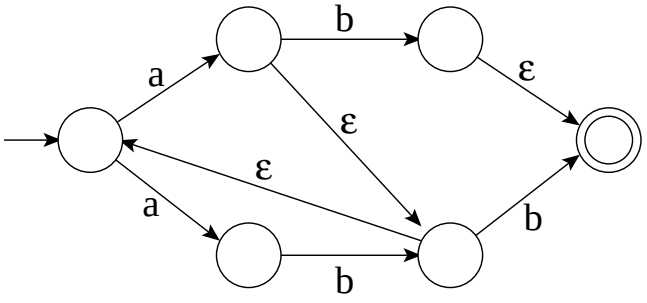
4. (Finite automata concepts, 8 points)

(a) (5 points total, 1 point each) Indicate whether the following are true ("T") or false ("F"):

i. ____ Finite automata require at least one final state.

ii. ____ A finite automaton with an ϵ-transition must be an NFA.

iii. ____ DFAs are always larger than their equivalent NFAs.



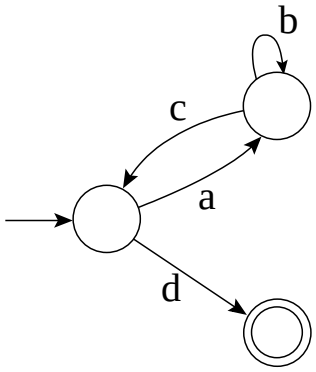
iv. ____ The finite automaton above accepts the string `abbb`.

v. ____ The finite automaton above accepts the string `ababab`.

Answer:

(a) F (b) T (c) F (d) F (e) T

(b) (3 points) Provide a regular expression for the following DFA:

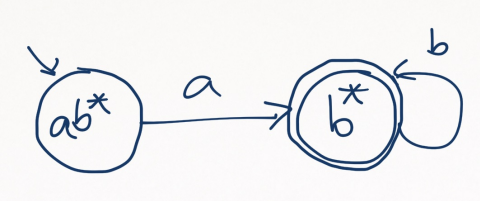


Answer:

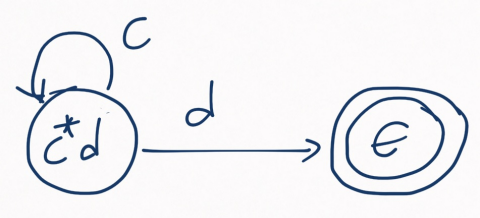
`(ab*c)*d`

5. (Nondeterministic finite automata, 12 points total, 3 points each)
   Reduce the following regular expressions to NFAs.

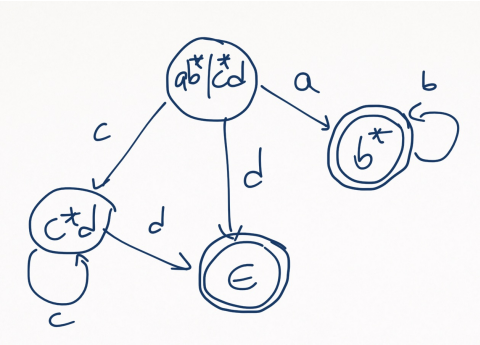
(a) `ab*`



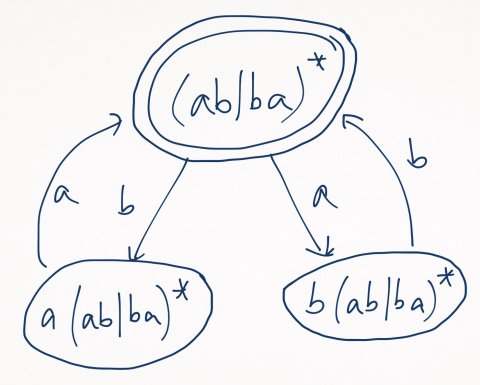
(b) `c*d`



(c) `ab*|c*d`

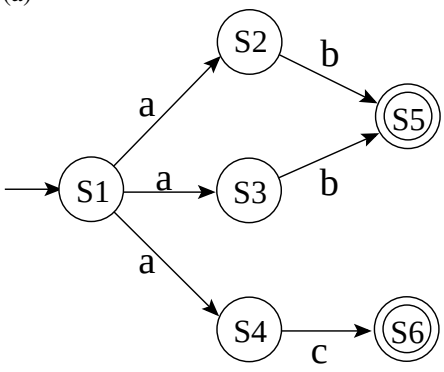


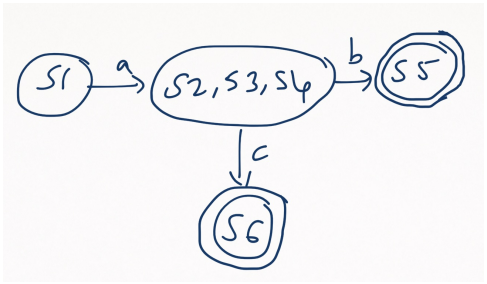
(d) `(ab|ba)*`



6. (Deterministic finite automata, 12 points total, 6 points each) Reduce the NFAs to DFAs.

(a)

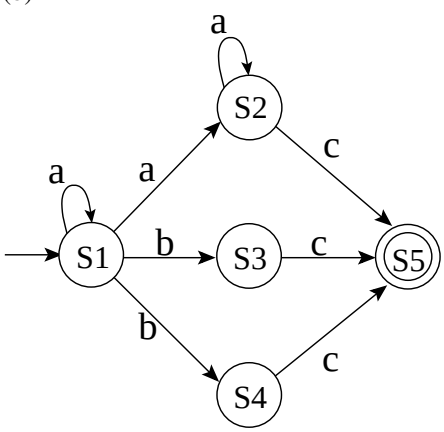


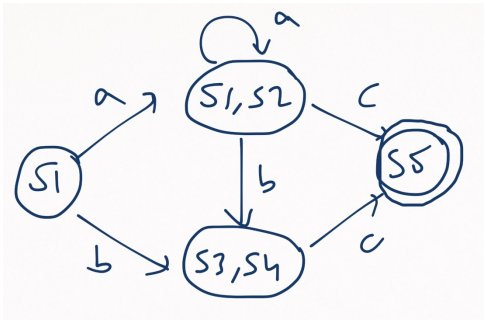


Answer:

$S_1, (S_2 \cup S_3 \cup S_4), (S_5), ((S_6))$

(b)





Answer:

$S_1, (S_1 \cup S_2), (S_3 \cup S_4), ((S_5))$

7. (Context-free grammars, 8 points)

(a) (5 points) Consider the following grammar (where S is the start symbol):

S → T | U
T → xSy | xy | ϵ
U → yT

It matches *one* of these strings. Circle the one it matches, and draw a parse tree for it.

- xxyx
- xyxxy
- yxxyy

Answer:

yxxyy. *The productions are S, U, yT, yxSy, yxTy, yxxyy (or instead of this last one, yxxTyy, yxxϵyy).*

(b) (3 points) The above grammar is ambiguous. Modify it to be unambiguous.

Answer:

Remove the xy *production rule from T.*

8. (Ruby programming, 30 points)

Write a Ruby class `Multiset` that implements a multiset. A *multiset* (a.k.a. a *bag*) is just a set in which the same element may appear multiple times. The skeleton code for the class is given on the next page. In brief, you will have to implement the following methods:

- `add(v)` adds an element `v` to the multiset.
- `union(m)` adds all the elements in multiset `m` to the current multiset.
- `==(m)` is structural equality method on multisets; returns `true` if the invoked multiset contains the same elements as argument `m`. (Note that the call `x == y` ends up invoking `x.==(y)`, so that in the definition of `==` the variable `self` plays the role of `x`.)
- `to_a` returns all the elements of this multiset as an array.
- `to_h` returns all the elements of this multiset as a hash. The returned hash will map each element to a non-zero count, where the count is the number of times the element appears in the multiset.

The code on the next page assumes you will store the contents of your multiset as an array, in the instance variable `@contents`. The constructor `initialize` takes a single argument `arr` and initializes the multiset to contain all the elements in the array `arr` by repeatedly invoking the `add` method that you will implement. Here are some "public tests" to help confirm the how a `Multiset` object behaves. (Note that the `inspect` method is like the `to_s` method in that it produces a string that represents the current object, but does so in a way that is more informative.)

```
m = Multiset.new [1,2,3]
m.add(3)
puts m.to_a.sort.inspect    # prints [1, 2, 3, 3]
puts m.to_h.inspect         # prints {1=>1, 2=>1, 3=>2}

x = m.to_a
x.pop
puts (m.to_a == [1,2,3,3])  # prints true

m.union(Multiset.new [1,2,3])
puts m.to_a.sort.inspect    # prints [1, 1, 2, 2, 3, 3, 3]
puts m.to_h.inspect         # prints {1=>2, 2=>2, 3=>3}

p = Multiset.new [1,1,2,2,3,3]
puts (p == m)               # prints false
p.add(3)
puts (p == m)               # prints true
puts (m == m)               # prints true
```

Other methods you may find useful:

- `<<` adds an element to an array, e.g., `x << 3` adds 3 to the (end of the) array `x`.
- `dup` copies an object, e.g., `x.dup` returns a copy of the array `x`. The `sort` method duplicates the original array and then sorts the duplicate before returning it.
- `keys` returns the keys in a hash; `values` returns the values.
- `each`, when invoked on an array, invokes the argument code block for each element in the array.

```
class Multiset
  def initialize(arr)
    @contents = [ ]
    arr.each { |v| add v }
  end

  def add(v)
    @contents.push v
    return self
  end

  def to_h
    x = { }
    @contents.each do |v|
      x[v] = 0 if not (x[v])
      c = x[v];
      x[v] = c+1
    end
    x
  end

  def ==(m)
    if m.class == Multiset then
      return r.to_a.sort == to_a.sort
    else return false
    end
  end
```

```
 def union(m)
    arr = m.to_a
    arr.each do |v| add v end
    return self
  end

  def to_a
    @contents.sort ## makes a copy of the array
  end

end ## class Multiset
```