CMSC330 Fall 2015 Midterm #2 Solution
12:30pm/2:00pm/5:00pm

Name:

_____

Discussion Time:	**10am**	**11am**	**12pm**	**1pm**	**2pm**	**3pm**
TA Name (Circle):	**Adam**	**Maria**	**Chris**	**Chris**	**Michael**	**Candice**
	Amelia	**Amelia**	**Samuel**	**Josh**	**Max**	

Instructions

- Do not start this test until you are told to do so!
- You have 75 minutes to take this midterm.
- This exam has a total of 100 points, so allocate 45 seconds for each point.
- This is a closed book exam. No notes or other aids are allowed.
- Answer essay questions concisely in 2-3 sentences. Longer answers are not needed.
- For partial credit, show all of your work and clearly indicate your answers.
- Write neatly. Credit cannot be given for illegible answers.

	Problem	Score
1	Finite Automata	/20
2	Context Free Grammars	/15
3	Parsing	/10
4	OCaml	/20
5	Programming Language Concepts	/15
6	Operational Semantics	/10
7	Lambda Calculus	/10
	Total	/100

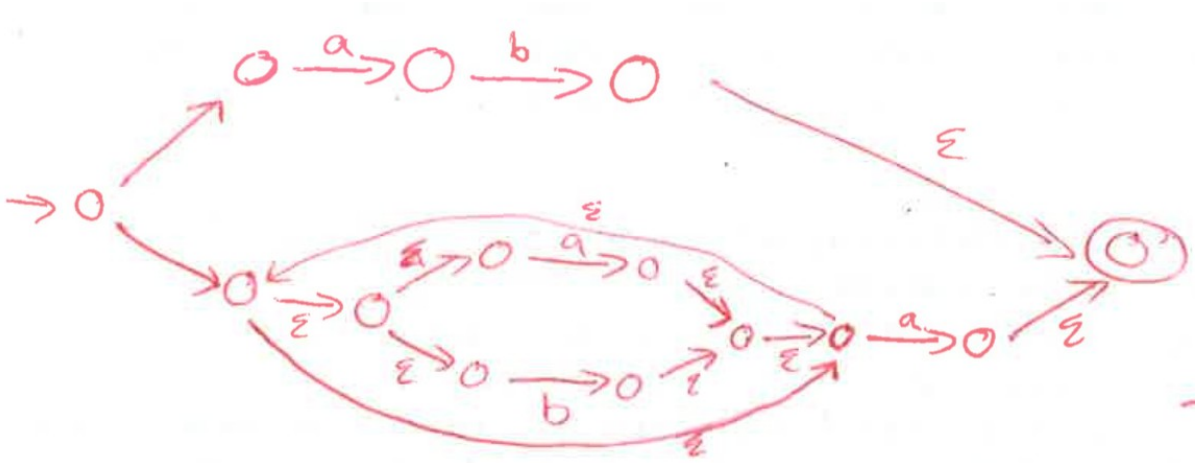
1. Finite Automata (20 pts)

a) (5 pts) Let L be the language accepted by the following regular expression

ab|((a|b)*a)

Give an NFA that accepts L. (Note that you can use page 4 for scratch paper)

Solution:

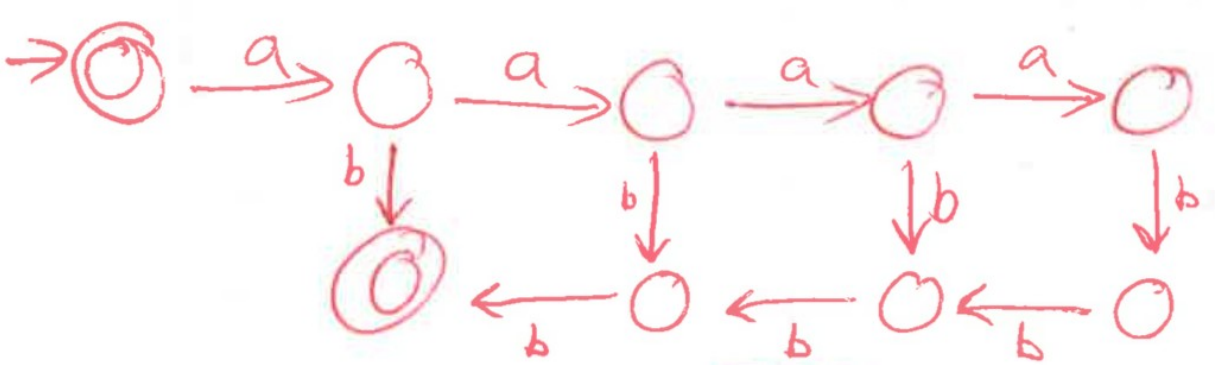


b) (5 pts) True or false: It is possible to design a **DFA that can accept strings in the language**

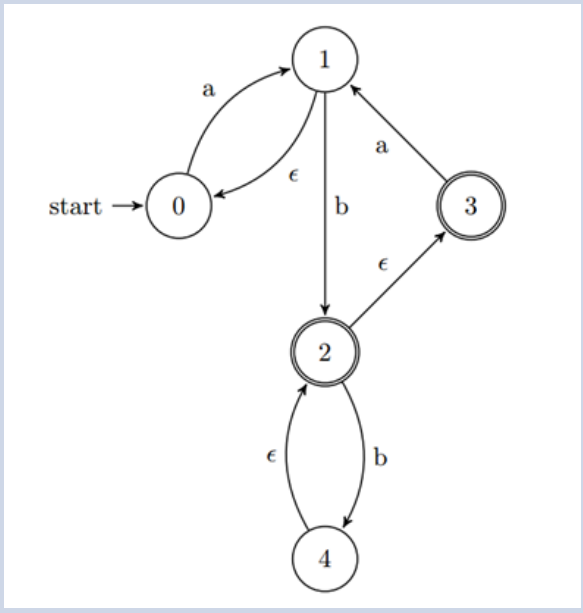
$L = \{ a^n b^n \mid 0 \leq n \leq 4 \}$,

i.e., all strings with up to four a's followed by an equal number of b's. **If this statement is true, show the DFA below. If false, explain why it is not possible.**

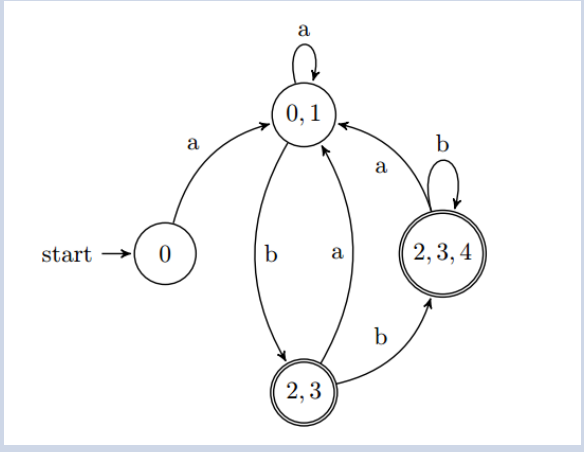
Answer: The statement is true:



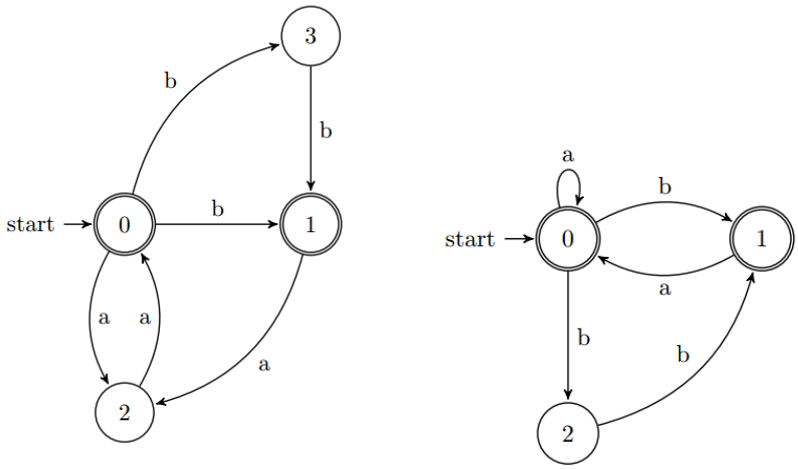
c) (5 pts) **Give a DFA that is equivalent to the following NFA.**



Solution:



d) (5 pts) **Are the following two DFA's equivalent? If not, give an example string that is accepted by one, but not the other.**



Answer: No, because in the original the automata does not accept the string “a”, but in the second automata, it does.

2. Context Free Grammars (15 pts)

Consider the following context free grammar (where uppercase letters are non-terminals, lowercase letters are terminals, and S is the start symbol).

S → Sa | TbS | cT | ScT
T → a | b

a) (6 pts) Indicate **whether either or both of the following two strings are accepted** by the grammar. For **each accepted string, give a derivation (aka a *parse*)** from the start symbol that shows the sequence of rewritings that accepts the string. You can alternatively provide a parse tree.

String	Accepted?	Derivation (if Accepted)
abacb	Y / N	

Solution: No. S → TbS → abS → abS → ? (S cannot parse to "acb")

ababcaca	Y / N	

Solution: Yes. S → ScT → TbScT → TbTbScT → TbTbcTcT → ababcaca

b) (5 pts) **Is the above grammar ambiguous? If so, give an example string that would be parsed ambiguously.**

Yes. If making both conversions S → TbS and S → ScT during a parse, they are interchangeable.
S → TbS → abS → abScT → abcTcT → abcacT → abcaca
S → ScT → TbScT → abScT → abcTcT → abcacT → abcaca

c) (4 pts)

Grammar 1:	Grammar 2:
E → E+T | E*T | T	E → E+T | T
T → a | b	T → T*P | P
	P -> a | b

Which grammar has the *incorrect* precedence for + and * operations? Recall that multiplication is higher precedence, so that a+b*a is equivalent to a+(b*a). **Give a parse tree for for a+b*a using the problematic grammar, and identify where the problem is.**

Solution: Grammar 2 has the correct precedence for + and * operations. Grammar 1 can generate parse three shown below, in which "+" has higher precedence than "*".

3. Parsing (10 pts)

This question will consider the following context-free grammar.

S → A | B
A → aSA | bB
B → c | d

a) (6 pts) **Compute the first sets of each non-terminal**

FIRST(S) = { }

FIRST(A) = { }

FIRST(B) = { }

Solution:

First(S) = { a, b, c, d }, First(A) = { a, b }, First(B) = { c, d }

For the next part you are given the following utilities for parsing.

`lookahead`	Variable holding next terminal
`match(x)`	Function to match next terminal to x
`error()`	Signals a parse error

b) (4 pts) Consider the following pseudocode for the `parse_S()` function:

```
parse_S() {
  if(lookahead == 'a' || lookahead == 'b') {
    parse_A();
  }
  else {
    parse_B();
  }
}
```

Is this function correct? If not, indicate where the problem is and how to fix it.

Solution: ~~It is correct. Another Another answer is to say that the else case should be `else if (lookahead == b) then { match(b); parse_B(); } else error();` But this is actually equivalent to what's there.~~If we assume parse_B will do error handling then yes, this function is correct. However, the most correct way to write the parser would be as follows.

```
parse_S() {
   if (lookahead == 'a' | lookahead == 'b') {
      parse_A();
   } else if (lookahead == 'c' | lookahead == 'd') {
      parse_B();
   } else {
      error();
   }
}
```

4. OCaml (20 pts)

a) (10 pts**) Write a function `average`** that takes in a list of floats and returns the average value of the elements in that list. (Recall that you should use the functions `+.` and `/.` for addition and division on floating point values, respectively.) The function you write should be **non-recursive, and employ one call to `fold`** (given below). If you implement it with more than one fold, or use recursion, you will not receive full credit. **You may not use any OCaml library functions.**

```
let rec fold f a l =
 match l with
   [] -> a
 | h::t -> fold f (f a h) t
```

Solution:

```
let average ls =
 let (sum,count) = fold (fun (x,y) h -> (x +. h, y +. 1)) (0.0,0.0) ls
in
           sum /. count;;
```

b) (6 pts) **What will the variables `result1` and `result2` contain after executing this code?** If an exception is thrown before the result(s) is/are produced, indicate that.

```
let p l =
  let r = ref 0 in
  let rec helper ls a =
    match ls with
      [] -> a / !r
    | h::t -> r := !r + 1; helper t (a+h) in
  helper l 0;;

let result1 = p [];;
let result2 = p [1;2;3;6];;
```

Solution:
result1: **Exception, Divide by zero** result2: **3**

c) (4 pts) Consider the following module type signature:

```
module type M1 =
 sig
    val f : int -> int -> int
    val g : 'a -> 'a list -> 'a list
end
;;
```

Given this, does the following code compile? If not, indicate where the problem is and the type error that arises.

```
module M1impl : M1 =
 struct
    let f y z = y + z;;
    let g w l = (w + 1)::l;;
    let h x = x + 1;;
 end
;;
```

Solution: No, Signature mismatch. M1impl.y has type int -> int list -> int list which does not match M1.y's type of 'a -> 'a list -> 'a list

5. Programming Language Concepts (15 pts)

True/false (3 points each) – **circle T for true and F for false**. ***You only have to answer 5 out of 6 questions.*** If you wish, you may add explanation of your answer for partial credit, but note that if you get the explanation wrong you may get the question wrong.

Solutions are in blue

T / **F** OCaml's + operator is an example of *ad hoc polymorphism.*

T / F Variable names can be reused in different *scopes*.

T / **F** The *Y combinator* is used to encode numbers in the lambda calculus.

T / F An *untyped language* allows any operation to be performed on any data

T / **F** In Java, **`Integer`** extends **`Object`**, and **`ArrayList<T>`** extends **`Collection<T>`**. As such, **`ArrayList<Integer>`** is a *subtype* of **`Collection<Object>`**.

T / **F** In the following OCaml code, variable **`x`** is *free* within the body of the function **`g`**.

```
let f x y =
  let rec g x z = x + y in
  g
```

6. Operational Semantics (10 pts)

a) Use the operational semantics rules given in class (copied on the last page of the exam for your reference) to **complete the missing parts of the hypotheses at each step**.

i) (3 points)

$$\dfrac{\dfrac{\_\_\_\_\_ ; x \Rightarrow \_\_\_\_\_ \quad \_\_\_\_\_ ; \_\_\_\_\_ \Rightarrow 3}{\bullet ; 2 \Rightarrow 2 \quad \_\_\_\_\_ ; x + 3 \Rightarrow 5}}{\bullet ;\ \text{let } x = 2 \text{ in } x + 3 \Rightarrow 5}$$

Solution:

$$\dfrac{\dfrac{\bullet , x{:}2 \ ; x \Rightarrow 2 \quad \bullet , x{:}2 \ ; 3 \Rightarrow 3}{\bullet ; 2 \Rightarrow 2 \quad \bullet , x{:}2 \ ; x + 3 \Rightarrow 5}}{\bullet ;\ \text{let } x = 2 \text{ in } x + 3 \Rightarrow 5}$$

ii) (4 points)

$$\dfrac{\dfrac{\bullet, y{:}5; (\text{fun } x \rightarrow x + 2) \Rightarrow \_\_\_\_\_ \quad \bullet, y{:}5; \_\_\_ \Rightarrow 5 \quad \_\_\_\_\_ ; x + 2 \Rightarrow \_\_\_\_\_}{\bullet, y{:}5; (\text{fun } x \rightarrow x + 2)\ y \Rightarrow \_\_\_\_\_}$$

Solution:

$$\dfrac{\dfrac{\bullet, y{:}5; (\text{fun } x \rightarrow x + 2) \Rightarrow (y{:}5; \lambda x.(x + 2)) \quad \bullet, y{:}5; y \Rightarrow 5 \quad \bullet, y{:}5, x{:}5; x + 2 \Rightarrow 7}{\bullet, y{:}5; (\text{fun } x \rightarrow x + 2)\ y \Rightarrow 7}}{\bullet, y{:}5; (\text{fun } x \rightarrow x + 2)\ y \Rightarrow 7}$$

b) (3 points) Consider the operational semantics derivation given for part a(ii). **Would the derivation change if we were using *dynamic scoping*, rather than *static scoping*? If so, describe (or mark) the change. If not, explain why it stays the same.**

Solution: It stays the same. The reason is that both the environment stored in the closure and the environment in which the function application is taking place are the same: •, y:5. As such, the rule for dynamically scoped application and statically scoped application nets the same result.

7. Lambda Calculus (10 pts)

a) (2 pts) Insert parentheses for the following λ-expression to clarify how it is parsed.

x y λx.x y

Solution:
((x y) (λx.(x y)))

b) (2 pts each) Reduce the following lambda terms as far as possible, and provide the final result. If the term reduces infinitely, say so. Show the alpha conversions and beta reductions you performed for partial credit.

i) (λw.w) (((λx.x) (λy.y)) (λz.z))

Solution: sequence of reduces is
((λx.x) (λy.y)) (λz.z)
(λy.y) (λz.z)
(λz.z)

ii) (λx.x x) (λx.x x) (λx.x)

Solution: Reduces Infinitely

iii) (λx.x (λx.y x)) (λz.z)

Solution: sequence of reduces is
(λz.z) (λx.y x)
λx.y x

iv) (λx.x λy.y x) y

Solution: sequence of reduces is
(λx.x λ**z.z** x) y
y (λz.z y)

Operational semantics rules reference