# CMSC330 Spring 2017 Midterm 2

**Name (PRINT YOUR NAME as it appears on gradescope ):**

______________________________________________

**Discussion Time (circle one)** **10am 11am 12pm 1pm 2pm 3pm**

**Discussion TA (circle one)** **Aaron Alex Austin Ayman Daniel Eric**
**Greg Jake JT Sam Tal Tim Vitung**

**Instructions**

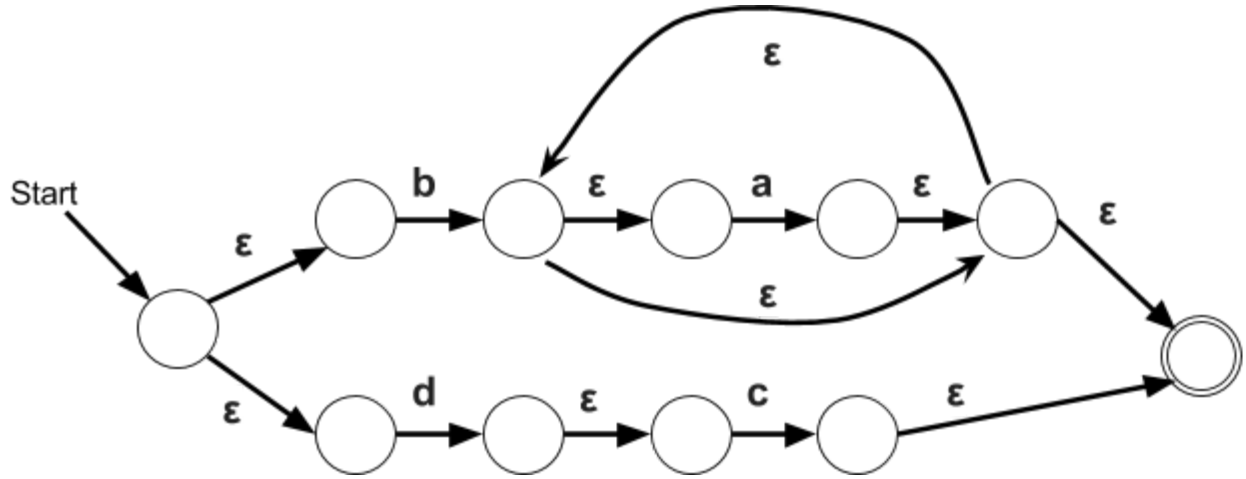
- Do not start this test until you are told to do so!
- You have 75 minutes to take this midterm.
- This exam has a total of 100 points, so allocate 45 seconds for each point.
- This is a closed book exam. No notes or other aids are allowed.
- Answer essay questions concisely in 2-3 sentences. Longer 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.

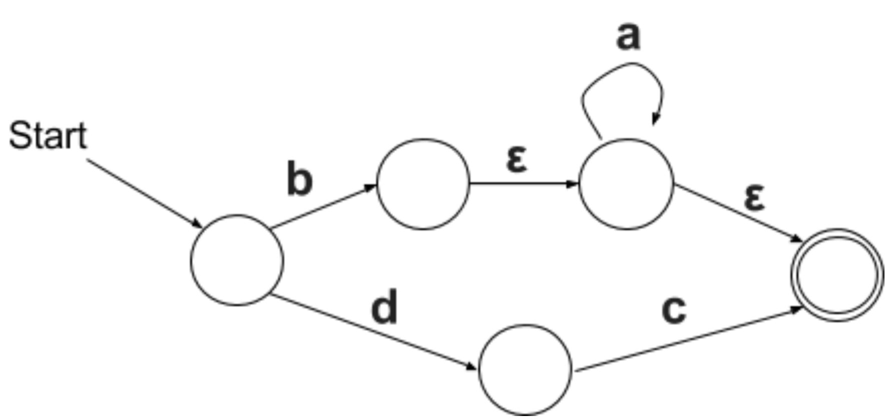
| | Problem | Score |
|---|---|---|
| 1 | Finite Automata | /20 |
| 2 | Context Free Grammars | /20 |
| 3 | Parsing | /13 |
| 4 | OCaml Programming | /10 |
| 5 | PL Concepts | /15 |
| 6 | Operational Semantics | /9 |
| 7 | Lambda Calculus | /13 |
| | Total | /100 |

# 1. Finite Automata (20 pts)

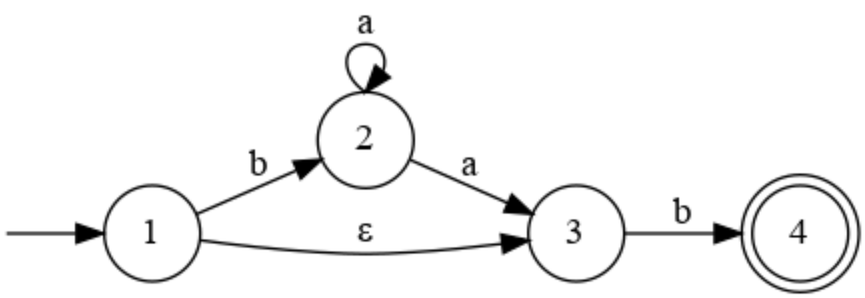
A. (5 pts) Construct an NFA that accepts the same language as the regular expression $ba^*|dc$ .

**Possible Answers:**

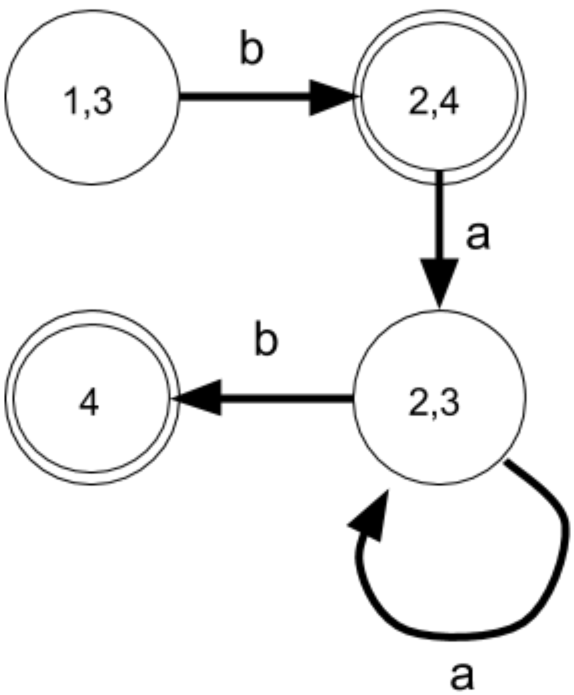


B. (5 pts) Reduce the following NFA to a DFA:




**Answer:**



C. (5 pts) Write a regular expression that accepts the same language as the NFA in problem B.

**Answer:** (ba+)?b

D. (3 pts) Can you write a regular expression for strings of length 5 or less that are palindromes (i.e., are mirror images of themselves)? Justify your answer.

**Answer:** Yes, you can write a regular expression for strings of length 5 or less that are palindromes. Because of the finiteness of the language, you would be able to make a regex that unions all possible palindromes of length 5 or less. You cannot, however, write a regular expression for strings of an unbound length.

E. (2 pts) True or **False**: there exist regular expressions that cannot be expressed as NFAs.

**Answer:** False.

## 2. Context Free Grammars (20 pts)

A. (4 pts) Consider the following CFG, where **a** and **b** are terminals, *S* and *T* are nonterminals.

```
S -> aT
T -> bbT | a
```

Consider the following strings; circle those that are accepted by the above CFG.

abb bba **aa** **abbbba**

**Answers:** aa and abbbba

B. (3 pts) Give a regular expression that accepts the same strings as the CFG as part A.

**Answer:** a(bb)*a

C. Consider the following CFGs (where **and**, **true**, and **false** are terminals, and *A* and *S* are nonterminals):

| CFG 1 | CFG 2 |
|---|---|
| `S -> S and A \| A`<br>`A -> true \| false` | `S -> A and S \| A`<br>`A -> true \| false` |

a. (2 pts) Which CFG treats **and** as a left associative operator?

   **Answer:** CFG 1

b. (2 pts) Which CFG *cannot* be used (as is) with a predictive parser?

   **Answer:** CFG 1

D. Given the CFG:

*S -> S***S* | *T*
*T* -> **a** | **b**

a) (3 pts) Give a leftmost derivation for the string **a*a*b**

**Answer:** S -> S*S -> **S*S*S -> T*S*S** -> a*S*S -> a*T*S -> a*a*S -> a*a*T -> a*a*b

b) (3 pts) Give a different leftmost derivation for the string **a*a*b**

**Answer:** S -> S*S -> **T*S -> a*S** -> a*S*S -> a*T*S -> a*a*S -> a*a*T -> a*a*b
Answers for a and b can be reversed.

c) (3 pts) Rewrite the grammar so it is unambiguous, treating * as a left associative operator.

**Answer:** S -> S*T | T
T -> a | b

## 3. Parsing (13 pts)

*S* → **cd** | **b***A* | *A***a**
*A* → **d***S* | ε

A. (5 pts) Calculate the first sets of the above grammar.

FIRST(*S*) = { **c, b, d, a** }

FIRST(*A*) = { **d, ε** }

B. (8 pts) Fill in the blanks for parse functions `parse_S` and `parse_A` for the CFG shown above. Both parse functions are of type `unit -> unit`. You may use the following helpers, described in class, which have their type signatures listed next to them:

- `lookahead: unit -> string`
- `match_tok: string -> unit`
- `raise_error: unit -> unit`

```
                                                    (* Answers in bold *)
let rec parse_S () =
    if lookahead () = “c” then
        (match_tok “c” ; match_tok “d”)

    else if (lookahead () = “b”) then
        match_tok “b”;
        parse_A ();
    else if (lookahead () = “d” || lookahead () = “a”) then
        parse_A ();
        match_tok “a”;
    else
        raise_error ();

;;

let rec parse_A () =

    if (lookahead () = “d”) then
        match_tok “d”;
        parse_S ();
    else
        (); (* epsilon *)
```

## 4. OCaml Programming (10 pts)

Recall the SmallC interpreter from project 3. Here are some snippets from its code:

```
type stmt =
  | NoOp
  | Seq of stmt * stmt
  | Declare of data_type * string
  | Assign of string * expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Print of expr

let eval_stmt (e:env) (s:stmt) = match s with
...
| While(guard_expr, body) -> begin
    let guard = eval_expr e guard_expr in
    match guard with
    | Val_Bool(true) -> eval_stmt (eval_stmt e body) s
    | Val_Bool(false) -> e
    | _ -> raise (TypeError("Can't use non-bool as while guard"))
  end
...
```

Imagine a new `stmt` variant to represent a `for` loop:

```
type stmt = ... (* as above *)
| For of stmt * expr * stmt * stmt
```

The tuple elements represent the initialization, the condition, the increment, and the body, respectively. Take for example, the smallC code:

```
for(i = 0; i < 10; i = i + 1) {printf(i)}
```

In this case, the fields line up as follows:

- `i = 0` is the initialization
- `i < 10` is the condition
- `i = i + 1` is the increment
- `printf(i)` is the body

After running (an updated version of) the lexer and parser, the example code above will be represented as:

```
For (Assign (“i”, Int 0),
     Less (Id “i”, Int 5),
     Assign (“i”, Plus (Id “i”, Int 1)),
     Print (Id “i”))
```

**Write the code for `eval_stmt` to handle `for` loops.** The semantics must satisfy the following:

- Before the first iteration, evaluate the initialization statement
- As long as the condition is true, evaluate the body followed by the increment statement
    - If the condition is non-boolean, raise an exception

(You might have a look at problem 6.C, below, before writing the code.)

You may assume a full, correct implementation of the whole project is accessible to you, including:

- `eval_expr: env -> expr -> value`
- `eval_stmt : env -> stmt -> env`
    - Excluding `for` itself

```
let eval_stmt (e:env) (s:stmt) = match s with
      ... (* all previous statement types are handled *)
      | For (init, cond, incr, body) -> (* your code below *)
```

**Long Answer:**

```
    let init_env = eval_stmt e init in
    let guard = eval_expr init_env cond in
    match guard with
      | Val_Bool(true) -> let body_env = eval_stmt init_env body in
                          let iter_env = eval_stmt body_env iter in
                          eval_stmt iter_env For(NoOp, cond, incr, body)
      | Val_Bool(false) -> init_env
      | _ -> raise (TypeError("Can't use non-bool as while guard"))
```

**Short Answer:**

```
    eval_stmt(e, Seq(init, While(cond, Seq(body, incr))))
```

## 5. PL concepts (15 pts)

A. (2 pts) In SmallC, which stage detects if some variable **x** is not declared before its first use? Circle the answer.

Lexer Parser **Interpreter**

B. (2 pts) True or **False**: An abstract syntax tree is the same as a parse tree.

C. (2 pts) An object is best encoded by one or more of which of the following? Circle the answer.

function **closure** module string

D. (3 pts) The Java class `Sequence` (on the left) is partially encoded as OCaml code on the right. What code should go in the gray portion?

| | |
|---|---|
| <pre>class Sequence {<br>  int s = 0;<br>  void start (int r) { s = r; }<br>  int next () { s++; return s; }<br>}<br><br>Sequence s = new Sequence();<br>s.start(10);<br>int t = s.next();<br>int u = s.next();</pre> | <pre>let make () =<br> let s = ref 0 in<br> ((fun r -> **s := r**),<br>  (fun ()-> s := !s + 1; !s))<br>;;<br><br>let (start, next) = make ();;<br>start 10;;<br>let t = next();;<br>let u = next();;</pre> |

**Answer in bold**

E. (6 pts) Rewrite the `smush` function to make it **tail recursive** (without changing its type). Here, the ^ operator is string concatenation (i.e., `“hello ” ^ “there” = “hello there”`). You are welcome to write helper functions.

```
let rec smush xs = match xs with
        [] -> “”
      | h::t -> h^(smush t);;

smush [] = “”;;
smush [“this ”; “is the ”; “word”] = “this is the word”;;
```

**Answer**:

```
let rec smush xs = match xs with
      [] -> []
      |[h] -> h
      | h1::(h2::t) -> smush ((h1 ^ h2)::t)
;;
```

## 6. Operational Semantics (9 pts)

A. (3 pts) Consider the operational semantics rules from the lecture notes for MicroOCaml, using an environment-based presentation.

| | |
|---|---|
| A(x) = v<br>A; x ⇒ v | A; n ⇒ n |
| A; e1 ⇒ v1    A,x:v1; e2 ⇒ v2<br>A; let x = e1 in e2 ⇒ v2 | A; e1 ⇒ n1    A; e2 ⇒ n2    n3 is n1+n2<br>A; e1 + e2 ⇒ n3 |

The following is a derivation of the program **let x = 3 in x+y** under an environment that initially maps y to 3. Fill in the three missing parts.

•,y:3,x:3; x ⇒ 3    •,y:3,x:3; [ **y** ] ⇒ 3
[ **•,y:3.x:3** ]; x+y ⇒ 6
•,y:3; let x = 3 in x+y ⇒ [ **6** ]

**Answers in bold**

B. (3 pts) The following rule is part of the operational semantics for SmallC:

```
A; e ⇒ true
A; s1 ⇒ A'
A; if e s1 s2 ⇒ A'
```

Explain this rule, in words. Your explanation should be something of the variety *if under environment A expression e evaluates to … then …* etc.

**Answer:**

If under environment A expression:

- e evaluates to true, and
- s1 evaluates to A'

Then under environment A *if e s1 s2* evaluates to A'

C. (3 pts) One of the operational semantics rules for while loops in SmallC is the following

```
A; e ⇒ true
A; s ⇒ A1
A1; while e s ⇒ A2
A; while e s ⇒ A2
```

Choose the rule below that is the equivalent one for `for` loops. Here we write `for s1 e s2 s` as corresponding to the SmallC syntax `for(s1; e; s2){s}`. The `skip` statement is equivalent to a no-op.

(A)

```
A; s1 ⇒ A1
A1; e ⇒ true
A2; s ⇒ A3
A3; s2 ⇒ A3
A; for s1 e s2 s ⇒ A3
```

**(B)**

```
A; s1 ⇒ A1
A1; e ⇒ true
A1; s ⇒ A2
A2; s2 ⇒ A3
A3; for skip e s2 s ⇒ A4
A; for s1 e s2 s ⇒ A4
```

(C)

```
A; s1 ⇒ A1
A1; e ⇒ true
A1; s2 ⇒ A2
A2; s ⇒ A3
A3; for skip e s2 s ⇒ A4
A; for s1 e s2 s ⇒ A4
```

(D)

```
A; s1 ⇒ A1
A1; e ⇒ true
A2; for skip e s2 s ⇒ A3
A3; s2 ⇒ A4
A; for s1 e s2 s ⇒ A4
```

## 7. Lambda Calculus (13 pts)

A. (1 pt) **True** or False: The lambda calculus can encode all computable functions.

B. (2 pts) Circle all occurrences of free variables in the following λ-term.

**x** (λx.x (λy.x y) **y**)

**Free variables bolded**

C. (2 pts) Determine whether the following λ-terms are α-equivalent (1 point each).

(λx.λy.y x) x     and     (λz.λy.z y) x     yes / **no**

λx.x λy.y z x     and     λv.v λy.y x v     yes / **no**

D. (2 pts) Perform one step of β-reduction on the following λ-term. (Perform alpha-conversion if necessary.)

(λx.λy.x y) (y λy.y)

**(λx.λz.x z) (y λy.y)**     **α-conversion (y -> z)**

**(λx.λz.x z) (y λa.a)**     **α-conversion (y -> a)**

**(λz.(y λa.a) z)**     **β-reduction (x -> (y λa.a))**

E. (5 pts) A programming language uses an evaluation strategy to determine when to evaluate the argument(s) of a function call. Reduce the following lambda expression using a call-by-value (aka *eager*) strategy and a call-by-name (aka *lazy*) strategy.

Call-by-Value
(λx.λy.x y z) (λc.c) ((λa.a) b)

**(λx.λy.x y z) (λc.c) b**     **β-reduction (a -> b)**

**(λy.(λc.c) y z) b**     **β-reduction (x -> (λc.c))**

*** First two reductions can be swapped**

**((λc.c) b z)**     **β-reduction (y -> b)**

**b z**     **β-reduction (c -> b)**

Call-by-name
(λx.λy.x y z) (λc.c) ((λa.a) b)

**(λy.(λc.c) y z) ((λa.a) b)**     **β-reduction (x -> (λc.c))**

**(λc.c) ((λa.a) b) z**     **β-reduction (y -> ((λa.a) b))**

**((λa.a) b) z**     **β-reduction (c -> ((λa.a) b))**

**b z**     **β-reduction (a -> b)**