CMSC330 Spring 2018 Midterm 2
9:30am/ 11:00am/ 3:30pm

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

____

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

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	PL Concepts	/9
2	Finite Automata	/21
3	Context Free Grammars	/20
4	Parsing	/17
5	Operational Semantics	/10
6	Lambda Calculus	/13
7	FP & Objects, Tail Recursion	/10
	Total	/100

1. PL concepts [9 pts]

   A. [4 pts] Circle true or false for each of the following (1 point each):

   a) True / False   Any language accepted by an NFA can be accepted by a DFA

   b) True / False   There are some regexps that do *not* have a corresponding DFA

   c) True / False   Lambda calculus is Turing complete

   d) True / False   The Y combinator is used to encode numbers and addition

   B. [1 pt] In my SmallC interpreter, the token list for string “`1-1`” showed up as `[Tok_Num 1; Tok_Num -1]`. I was expecting `[Tok Num 1, Tok_Minus, Tok_Num 1]`. This problem is caused by an error in my (circle the right one):

      a) Interpreter
      b) Lexer
      c) Parser
      d) Type Checker

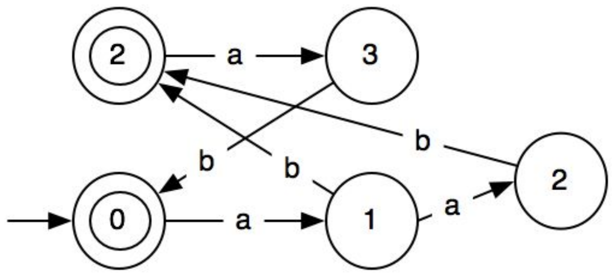
   C. [4 pts] What is **printed** when evaluating the following expression using the CBV (Call by value) and CBN (Call by name) evaluation strategy?

```
(fun x -> x; x) (print_string "hi")
```

CBV	CBN

2. Finite Automata [21 pts]

A. [4 pts] Consider the following automaton which operates over alphabet {a,b}.

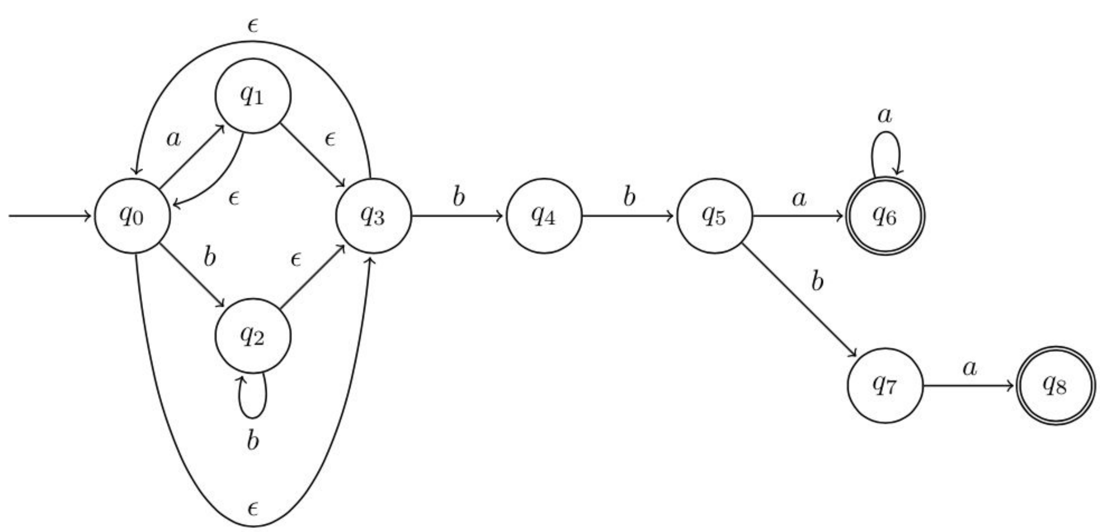


Which of the following are true about it (circle the letter of the statement)?

a. (2 pts) It is an DFA
b. (2 pts) It is minimal

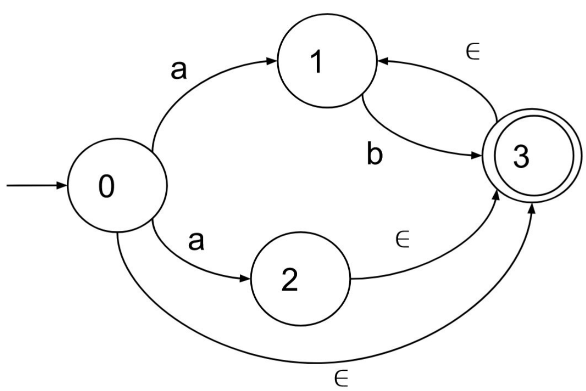
B. [5 pts] Which of the following strings are accepted by this automaton? Circle them.

aababaa   abbbaaa   bbba   bbabbaba   aaabbbaabba



C. [6 pts] Draw a finite automaton that accepts the same strings as the regular expression `(a|b)+|(ab*c)`

D. [6 pts] Convert following NFA to a DFA.



3. Context Free Grammars [20 pts]

A. [1 pt]   True / False   In the following grammar, the + operator is left-associative.

$$E \rightarrow E{+}T \mid E{-}T \mid T$$
$$T \rightarrow a \mid b \mid c \mid (E)$$

B. [11 pts] Consider the following CFG, in which **p** and **q** are terminals, and A and B are nonterminals.

   a. [4 pts] Which of the following strings are accepted? Circle them.

      A -> **p**A**q** | B
      B -> **p**B | B**q** | **pq**

Circle: **pppqqq**   **pqpq**   **pppq**   **p**

   b. [3 pts] Give a regular expression that accepts the same strings as the CFG. If this is not possible, explain why.

   c. [4 pts] Show that the CFG is ambiguous.

C. [4 pts] Change the following CFG to eliminate left recursion

S -> S **and** S | T
T -> **true** | **false**

D. [4 pts] Give a CFG that starts with one or more **y** followed by twice as many **x** or **z.** The grammar accepts the following strings (and many others): **yxx**, **yzz**, **yzx**, **yxx**, **yyxzzx**, **yyzxzx**, **yyyxxxxxx**, ...

4. Parsing [17 pts]

A. [2 pts] Circle whether the following are true or false

   a. True / false   Recursive descent parsing works bottom-up

   b. True / false   Recursive descent parsing is a kind of predictive parsing

B. [2 pts] Name two features of a grammar that make it unsuitable for recursive descent parsing.

Now Consider the following context-free grammar (CFG):

S → A**c** | **d**S
A → **a**BA | ε
B → **b**B | **c**

C. [1 point] Circle the correct answer about the CFG definition for nonterminal B.

   a. B is left recursive
   b. B is right recursive
   c. B is ambiguous
   d. None of the above

D. [4 points] What are the FIRST SETS of each of the nonterminals in the grammar?

E. (8 points) Complete the implementation for a recursive-descent parser for the CFG.

$S \rightarrow A\mathbf{c} \mid \mathbf{d}S$
$A \rightarrow \mathbf{a}BA \mid \varepsilon$
$B \rightarrow \mathbf{b}B \mid \mathbf{c}$

```
exception ParseError of string
let tok_list = ref [];; (* filled in by scanner *)
let lookahead () =
  match !tok_list with
      [] -> None
    | (h::t) -> Some h
let match_tok a =
  match !tok_list with
    | (h::t) when a = h -> tok_list := t
    | _ -> raise (ParseError "bad match")

let rec parse_S( ) =
  if (lookahead() = Some "a") || (lookahead() = Some "c") then
    (parse_A();
     match_tok "c")
  else (* FILL IN - 4 pts *)
```

```
and parse_A( ) = (* FILL IN - 4 pts *)
```

```
and parse_B() =
  if lookahead() = Some "b" then
    (match_tok "b";
     parse_B())
  else if lookahead() = Some "c" then
    match_tok "c"
  else raise (ParseError "bad match")
```

5. Operational Semantics [10 pts]

A. [3 pts] Describe in English what the operator `myst` does, or give its usual name (you have seen it before).

$$\text{Mystery}(1): \frac{A;\ e_1 \Rightarrow \mathit{true} \quad A;\ e_2 \Rightarrow \mathit{true}}{A;\ e_1\ \mathit{myst}\ e_2 \Rightarrow \mathit{false}} \qquad \text{Mystery}(2): \frac{A;\ e_1 \Rightarrow \mathit{false} \quad A;\ e_2 \Rightarrow \mathit{false}}{A;\ e_1\ \mathit{myst}\ e_2 \Rightarrow \mathit{false}}$$

$$\text{Mystery}(3): \frac{A;\ e_1 \Rightarrow \mathit{true} \quad A;\ e_2 \Rightarrow \mathit{false}}{A;\ e_1\ \mathit{myst}\ e_2 \Rightarrow \mathit{true}} \qquad \text{Mystery}(4): \frac{A;\ e_1 \Rightarrow \mathit{false} \quad A;\ e_2 \Rightarrow \mathit{true}}{A;\ e_1\ \mathit{myst}\ e_2 \Rightarrow \mathit{true}}$$

B. [3 pts] Below are incorrect rules for conditionals. Circle the key part of each rule that is incorrect. Feel free to explain, for clarity.

$$\text{Bad-If-True}\frac{\begin{array}{c} A; e \Rightarrow \texttt{true} \\ A; s_1 \Rightarrow A_1 \\ A_1; s_2 \Rightarrow A_2 \end{array}}{A; \texttt{if}\ e\ s_1\ s_2 \Rightarrow A_1} \qquad \text{Bad-If-False}\frac{\begin{array}{c} A; e \Rightarrow \texttt{false} \\ A; s_1 \Rightarrow A_1 \\ A_1; s_2 \Rightarrow A_2 \end{array}}{A; \texttt{if}\ e\ s_1\ s_2 \Rightarrow A_2}$$

C. [4 pts] The statement **`s unless e`** will execute statement *s* if *e* evaluates to `false` and has no effect if ***e*** evaluates to **`true`**. Implement the semantics for **`unless`** by filling in the boxes below. (Like in SmallC, you can assume that expressions have no effect on the environment.)

$$\text{Unless-True}\ \frac{\boxed{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}{A; s\ \texttt{unless}\ e \Rightarrow \boxed{\qquad}}$$

$$\text{Unless-False}\ \frac{\boxed{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}{A; s\ \texttt{unless}\ e \Rightarrow \boxed{\qquad}}$$

6. Lambda Calculus [12 pts]

A. [2 pts] Circle all occurrences of **free variables** in the following λ-term:

λx. z (λy. x y) y   x

B. [2 pts] Circle whether the following statements are true or false

a. True / False   λx.λy.y x is alpha-equivalent to λf.λn.f n

b. True / False   λx.λy.y x is alpha-equivalent to (λx.λy.y) x

C. Reduce each lambda expression to beta-normal form (to be eligible for for partial credit, show each reduction step). If already in normal form, write "normal form."

a) [2 pts] (λz.λy.z) x

b) [2 pts] (λx.λx.x x) y

c) [3 pts] (λz.(λx. z) z) (λy.x y)

D. [2 pts] Which of the following lambda terms has the same semantics as this bit of OCaml code: (circle exactly one)

```
let func x = (fun y -> y x)
in func a b
```

a) (λy. y x) a b

b) (λx. (λy. y x) a b)

c) (λx. (λy. y x)) a b

d) (x (λy. y x)) a b

7. FP & Objects, Tail Recursion [10 pts]

A. [5 pts] Given the Java class `Point` on the left, write the OCaml encoding of the `Point` class on the right. (*Hint*: `make()` returns a tuple of 3 functions, as shown in the code at the bottom of the righthand-side.)

<pre>class Point {
   private int x=0,y=0;
   void set(int x, int y){
     this.x = x;
     this.y = y;
    }
   int getX(){return x;}
   Int getY(){return y;}
}
Point p = new Point();
p.set(2,6);
int x = p.get(x);</pre>
	<pre>let make () =
	
	let (set,getx,gety) = make ();;
	set 2 6;;
	let x = getx ();</pre>

B. [5 pts] Write a **tail-recursive** version of the `sum` function, which sums all elements of a list, having type `int list -> int`.