CMSC 330, Fall 2018 — Midterm 2

NAME ____

TEACHING ASSISTANT

Kameron Aaron Danny Chris Michael P. Justin Cameron B. Derek Kyle Hasan Shriraj Cameron M. Alex Michael S. Pei-Jo

INSTRUCTIONS

- Do not start this exam until you are told to do so.
- You have 75 minutes for this exam.
- This is a closed book exam. No notes or other aids are allowed.
- For partial credit, show all your work and clearly indicate your answers.

HONOR PLEDGE

Please copy and sign the honor pledge: "I pledge on my honor that I have not given or received any unauthorized assistance on this examination."

____

____

____

____

Section	Points
Programming Language Concepts	10
Finite Automata	23
Context-Free Grammars	18
Parsing	18
Operational Semantics	11
Lambda Calculus	13
Imperative OCaml	7
Total	100

1 Programming Language Concepts

In the following questions, circle the correct answer.

1. [1 pts] (T / F) The input to a lexer is source code and its output is an abstract syntax tree.

   Solution. False. ❏

2. [1 pts] (T / F) Any language that can be expressed by a context-free grammar can be expressed by a regular expression.

   Solution. False. ❏

3. [1 pts] (T / F) OCaml is Turing-complete.

   Solution. True. ❏

4. [1 pts] (T / F) Converting a DFA to an NFA always requires exponential time.

   Solution. False. ❏

5. [1 pts] (T / F) Recursive descent parsing requires the target grammar to be right recursive.

   Solution. True. ❏

6. [1 pts] (T / F) The SmallC parser in P4A used recursive descent.

   Solution. True. ❏

7. [1 pts] (T / F) The call-by-name and call-by-value reduction strategies can produce different normal forms for the same λ expression.

   Solution. False. ❏

8. [1 pts] (T / F / Decline to Answer) I voted last Tuesday. (All answers are acceptable.)

   Solution. Any. ❏

9. [1 pts] What language feature does the fixed-point combinator implement?

   (a) Booleans (b) Integers (c) Recursion (d) Closures

Solution. (c) ❑

10. [1 pts] What is wrong with this definition of an NFA?

```
type (’q, ’s) nfa = {
    qs : ’q list;
    sigma : ’s list;
    delta : (’q, ’s) transition list;
    q0 : ’q list;
    fs : ’q list;
}
```

(a) Allows states with multiple transitions on the same character.

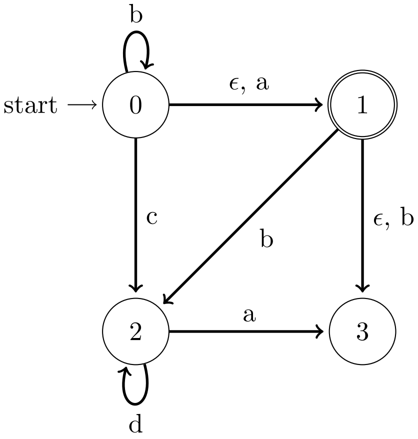
(b) Allows ε-transitions.

(c) Allows multiple final states.

(d) Allows multiple start states.

Solution. (d) ❑

2 Finite Automata



1. Use the NFA shown above to answer the following questions.

   - [2 pts] ε-closure($\{0\}$) = $\{$ $\}$

     Solution. ε-closure($\{0\}$) = $\{0, 1, 3\}$ ❑

   - [2 pts] move($\{1\}, b$) = $\{$ $\}$

     Solution. move($1, b$) = $\{2, 3\}$ ❑

2. [1 pts] (T / F) Every NFA is also a DFA.

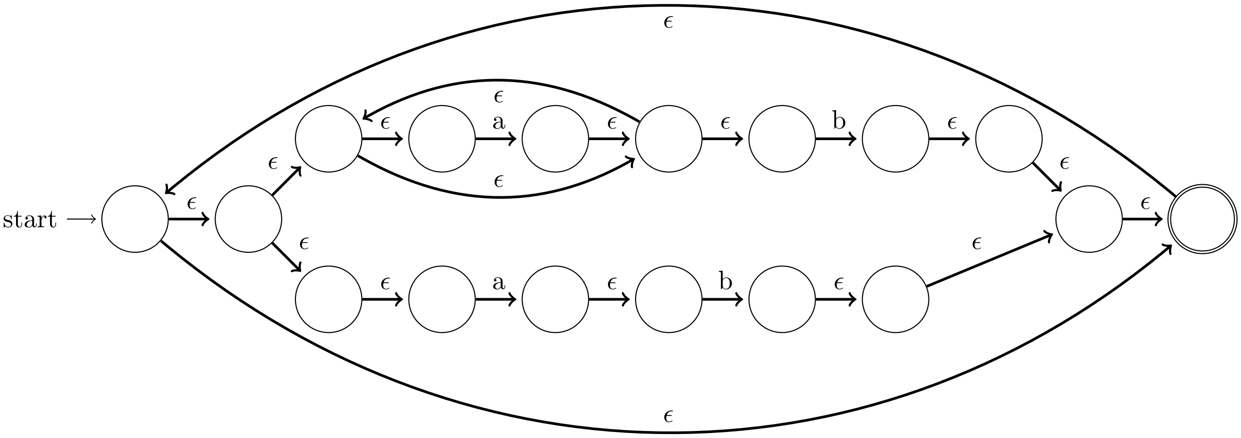
   Solution. False ❑

3. [1 pts] (T / F) Every DFA is also an NFA.

   Solution. True ❑

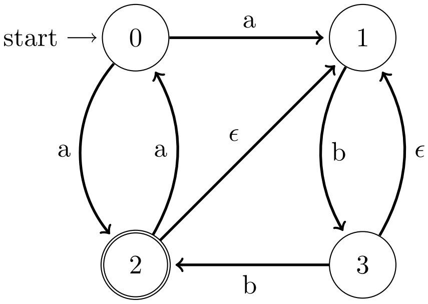
4. [5 pts] Draw an NFA that corresponds to the following regular expression: $((a^\star b) \mid (ab))^\star$

***Solution*.**

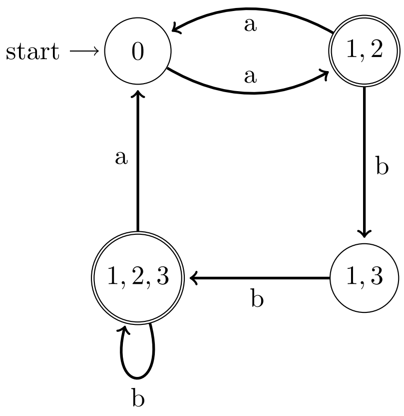


❏

5. [7 pts] Convert the following NFA into an equivalent DFA.



Solution.



❑

6. [5 pts] Circle all of the strings that will be accepted by the above **NFA**. (**Note**: Not the DFA you generated)

(a) abbaa (b) aaaa (c) abbaabb (d) abbbbaab (e) aaaaa

Solution. (a), (c), (e) ❑

3 Context-Free Grammars

1. [6 pts] Write a CFG that is equivalent to the regular expression $(wp)^{+}g^{\star}$

Solution.

$$S \to XY$$

$$X \to wpX \mid wp$$

$$Y \to gX \mid \varepsilon$$

❑

2. [6 pts] Create a CFG that generates all strings of the form $a^x b^y a^z$, where $y = x + z$ and $x, y, z \geq 0$.

Solution.

$$\begin{aligned} S &\to UL \\ U &\to aUb \mid \varepsilon \\ L &\to bLa \mid \varepsilon \end{aligned}$$

❑

3. [6 pts] Given the following grammar, where S and A denote non-terminals, give a right-most and left-most derivation of $((100, 33), 30)$. Show all steps of your derivation.

$$\begin{aligned} S &\to A \mid (S, S) \\ A &\to 100 \mid 33 \mid 30 \end{aligned}$$

Solution.

- Left-most,

$$\begin{aligned} S &\to (S, S) \\ &\to ((S, S), S) \\ &\to ((100, S), S) \\ &\to ((100, 33), S) \\ &\to ((100, 33), 30)) \end{aligned}$$

- Right-most,

$$\begin{aligned} S &\to (S, S) \\ &\to (S, S) \\ &\to (S, 30)) \\ &\to (S, 30) \\ &\to ((S, S), 30) \\ &\to ((S, 33), 30) \\ &\to ((100, 33), 30) \end{aligned}$$

❑

4 Parsing

1. [3 pts] Convert the following to a right-recursive grammar.

$$S \to S + S \mid A$$

$$A \to A * A \mid B$$

$$B \to n \mid (S)$$

***Solution**.*

$$S \to A + S \mid A$$

$$A \to B * A \mid B$$

$$B \to n \mid (S)$$

❑

2. [5 pts] What are the first sets of the non-terminals in the following grammar?

$$S \to bc \mid cA$$

$$A \to cAd \mid B$$

$$B \to wS \mid \varepsilon$$

***Solution*.**

$$\text{first}(S) = \{b, c\}$$

$$\text{first}(A) = \{c, w, \varepsilon\}$$

$$\text{first}(B) = \{w, \varepsilon\}$$

❏

3. [10 pts] Finish the definition of a recursive descent parser for the grammar below. You need not build an AST, assume all methods return unit. Note that `match_tok` takes a string.

$$S \to Abc \mid A$$

$$A \to cAd \mid e$$

```
let lookahead () : string =
  match !tok_list with
  | [] ->  raise (ParseError "no tokens")
  | h::t -> h

let match_tok (a : string) : unit =
  match !tok_list with
  | h::t when a = h -> tok_list := t
  | _ -> raise (ParseError "bad match")

let rec parse_S () : unit =
```

```
and parse_A () : unit =
```

Solution.

```
let rec parse_S () : unit =
  let () = parse_A () in
  if lookahead () = "b" then
    let () = match_tok "b" in
      match_tok "c"
    else
      ()
and parse_A () : unit =
  if lookahead () = "c" then
    let () = match_tok "c" in
    let () = parse_A () in
    match_tok "d"
  else if lookahead () = "e" then
    match_tok "e"
  else
    raise (ParseError "parse_A")
```

❏

5 Operational Semantics

$$\frac{}{A;\ \textbf{false} \Rightarrow \textbf{false}} \qquad\qquad \frac{}{A;\ \textbf{true} \Rightarrow \textbf{true}}$$

$$\frac{}{A;\ n \Rightarrow n} \qquad\qquad \frac{A(x) = v}{A;\ x \Rightarrow v}$$

$$\frac{A; e_1 \Rightarrow v_1 \qquad A, x : v_1;\ e_2 \Rightarrow v_2}{A;\ \textbf{let}\ x = e_1 \textbf{in}\ e_2 \Rightarrow v_2} \qquad \frac{A;\ e_1 \Rightarrow n_1 \qquad A;\ e_2 \Rightarrow n_2 \qquad n_3\ \textit{is}\ n_1 + n_2}{A;\ e_1 + e_2 \Rightarrow n_3}$$

$$\frac{A;\ e_1 \Rightarrow \textbf{true} \qquad A;\ e_2 \Rightarrow v}{A;\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Rightarrow v} \qquad \frac{A;\ e_1 \Rightarrow \textbf{false} \qquad A;\ e_3 \Rightarrow v}{A;\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Rightarrow v}$$

Use the above rules to fill in the given constructions.

1. [6 pts]

$$\dfrac{A;\ \boxed{\qquad\qquad} \qquad\qquad \dfrac{A;\ \boxed{\qquad\qquad} \quad A;\ \boxed{\qquad\qquad} \quad \boxed{\qquad\qquad}}{A;\ \boxed{\qquad\qquad}}}{A;\ \textbf{if}\ \boxed{\qquad\quad}\ \textbf{then}\ 10 + 3\ \textbf{else}\ 5 + 2 \Rightarrow 7}$$

Solution.

$$\dfrac{A;\ \boxed{\text{false} \Rightarrow \text{false}} \qquad\qquad \dfrac{A;\ \boxed{5 \Rightarrow 5} \quad A;\ \boxed{2 \Rightarrow 2} \quad \boxed{7\ \text{is}\ 5 + 2}}{A;\ \boxed{5 + 2 \Rightarrow 7}}}{A;\ \textbf{if}\ \boxed{\text{false}}\ \textbf{then}\ 10 + 3\ \textbf{else}\ 5 + 2 \Rightarrow 7}$$

❑

2. [5 pts]

$$\dfrac{\boxed{\qquad\qquad};\ 4 \Rightarrow 4 \qquad\qquad \dfrac{\boxed{\qquad\qquad};\ 5 \Rightarrow 5 \qquad \dfrac{\boxed{\qquad\qquad}(x) \Rightarrow 5}{\boxed{\qquad\qquad};\ x \Rightarrow 5}}{\boxed{\qquad\qquad}\ \textbf{let}\ x = 5\ \textbf{in}\ x \Rightarrow 5}}{A;\ \textbf{let}\ x = 4\ \textbf{in let}\ x = 5\ \textbf{in}\ x \Rightarrow 5}$$

Solution.

$$\dfrac{\boxed{\mathrm{A}};\ 4 \Rightarrow 4 \qquad \dfrac{\boxed{\mathrm{A,\ x = 4}};\ 5 \Rightarrow 5 \qquad \dfrac{\boxed{\mathrm{A,\ x = 4,\ x = 5}}(x) \Rightarrow 5}{\boxed{\mathrm{A,\ x = 4,\ x = 5}};\ x \Rightarrow 5}}{\boxed{\mathrm{A,\ x = 4}}\ \mathbf{let}\ x = 5\ \mathbf{in}\ x \Rightarrow 5}}{A;\ \mathbf{let}\ x = 4\ \mathbf{in}\ \mathbf{let}\ x = 5\ \mathbf{in}\ x \Rightarrow 5}$$

❑

6 Lambda Calculus

1. [2 pts] Circle all of the free variables in the following λ expression. (A variable is **free** if it is not bound by a λ abstraction.)

   $$x\ (\lambda x.\ (\lambda y.\ \lambda z.\ x\ y\ z)\ y)$$

   Solution.

   $$\underline{x}\ (\lambda x.\ (\lambda y.\ \lambda z.\ x\ y\ z)\ \underline{y})$$

   ❏

2. [2 pts] Circle all of the following where the λ expressions are α-equivalent.

   (a) $((\lambda a.\ (\lambda y.\ y\ a)\ y)$ and $(\lambda x.\ x\ y)$

   (b) $(\lambda x.\ (\lambda y.\ x\ y))$ and $(\lambda y.\ (\lambda x.\ y\ x))$

   Solution. The (a) expressions not α-equivalent, but the (b) expressions are. ❏

3. Reduce each λ expression to β-normal form (to be eligible for partial credit, show each reduction step). If already in normal form, write "normal form." If it reduces infinitely, write "reduces infinitely."

   (a) [2 pts] x (λa. λb. b a) x (λy. y) — Hint: application is left-associative.

   Solution. Normal Form ❏

   (b) [2 pts] $((\lambda x.\ x\ x)(\lambda y.\ y\ y))$

   Solution.

   $$\begin{aligned}((\lambda x.\ x\ x)(\lambda y.\ y\ y)) &\rightarrow_\beta ((\lambda y.\ y\ y)(\lambda y.\ y\ y))\\ &\rightarrow_\beta ((\lambda y.\ y\ y)(\lambda y.\ y\ y))\\ &\rightarrow_\beta \ldots\end{aligned}$$

   Reduces Infinitely ❏

(c) [2 pts] $((\lambda a.\ \lambda b.\ a\ b\ c)\ x\ y)$

Solution.

$$\begin{aligned}((\lambda a.\ \lambda b.\ a\ b\ c)\ x\ y) &\rightarrow_\beta ((\lambda b.\ x\ b\ c)\ y)\\ &\rightarrow_\beta (x\ y\ c)\end{aligned}$$

❑

4. [3 pts] Write an OCaml expression that has the same semantics as the following λ expression.

$$(\lambda a.\ \lambda b.\ a\ b)\ (\lambda x.\ x\ x)\ y$$

Solution. `(fun a -> fun b -> a b) (fun x -> x x) y` ❑

7 Imperative OCaml

1. [7 pts] Given the `mut_lst` variable, which is `'a ref list`, implement the `add` and `contains` functions which should add a given element to `mut_lst` and check if the `mut_lst` contains a specified element, respectively. You may add helpers and change the functions to be recursive.

```
let mut_lst = ref []
```

```
let add (ele : 'a) : unit =
```

```
let contains (ele : 'a) : bool =
```

Solution.

```
let add (ele : 'a) : unit =
    mut_lst := ele :: (!mut_lst)

let contains (ele : 'a) : bool =
```

```
let rec helper lst =
  match lst with
  | [] -> false
  | h :: t -> if (h = ele) then true else (helper t)
in helper !mut_lst
```

❑