CMSC330 Spring 2019 Midterm 2
11:00am / 12:15pm / 2:00pm

Solution

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	/13
2	Finite Automata	/31
3	Context Free Grammars	/17
4	Parsing	/16
5	Operational Semantics	/10
6	Lambda Calculus	/13
	Total	/100

1. PL concepts [13 pts]

A) [5 pts] Circle true or false for each of the following 5 questions (1 point each)

True / ***False***	In OCaml, if an exception is thrown, then the executing program will terminate
True / False	OCaml variables are immutable
True / ***False***	If x and y are aliases, changing the content in the location referenced by x will cause it to no longer be an alias of y
True / False	If a lambda calculus expression reduces to a beta-normal form using call-by-value order, then it will also do so using call-by-name
True / False	You can create a cyclic data structure in OCaml (i.e., one that points to itself)

B) [4 pts] Consider the following OCaml definitions for `f`, `g`, and `h` (each is a `int -> int` function).

```
let f z =
  let y = ref 0 in
  y := !y + z;
  !y
```

```
let g =
  let x = ref 1 in
  (fun z ->
    x := !x + 1;
    !x+z)
```

```
let h =
 (fun z -> let x = z+1 in
   let _ = (print_int z,print_int x) in
   0)
```

	Answer:
Which of these functions is not *referentially transparent*?	***either g or h***
Which function's execution outcome *depends on OCaml's evaluation order*	***h***
What is a *side effect* carried out by at least one of the functions?	***Printing or incrementing***
Which function's execution is *only* interesting/useful because of its side effects, not what it returns?	***h***

C) [4 pts] Check the box next to each function that is *tail recursive* (they all type check and run properly).

☐
```
let rec sum lst =
  match lst with
    [] -> 0
  | h::t-> h + sum t
```

☑
```
let rec max lst r =
  match lst with
    [] -> r
  | h::t ->
      if r>h then max t r
             else max t h
```

☑
```
let rec pow2 x =
  if x = 1 then true
  else
    let y = x/2 in
    if y*2 = x then pow2 y
    else false
```

☐
```
let rec prod lst =
  match lst with
    [] -> 1
  | h::t -> (prod t) * h
```

2. Finite Automata [31 pts]

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

True / ***False*** NFAs are more expressive than DFAs (i.e., they can describe more languages)

True / ***False*** Every CFG has an equivalent NFA

True / ***False*** Every formal language has a unique DFA that generates it

True / ***False*** Regexes are more expressive (can generate more languages) than DFAs

B) [6 pts] For each of the following statements, check the DFA box if it's true for DFAs, and the NFA box for NFAs. You may check neither or both boxes.

☐DFA ☑NFA Can transition to multiple states at once with a symbol

☐DFA ☑NFA Can have epsilon transitions

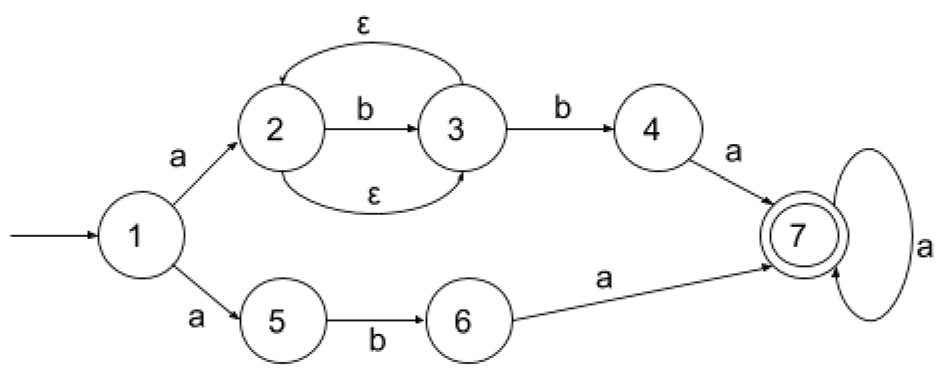
☑DFA ☑NFA Can have multiple final states

☐DFA ☐NFA Always has at least one final state

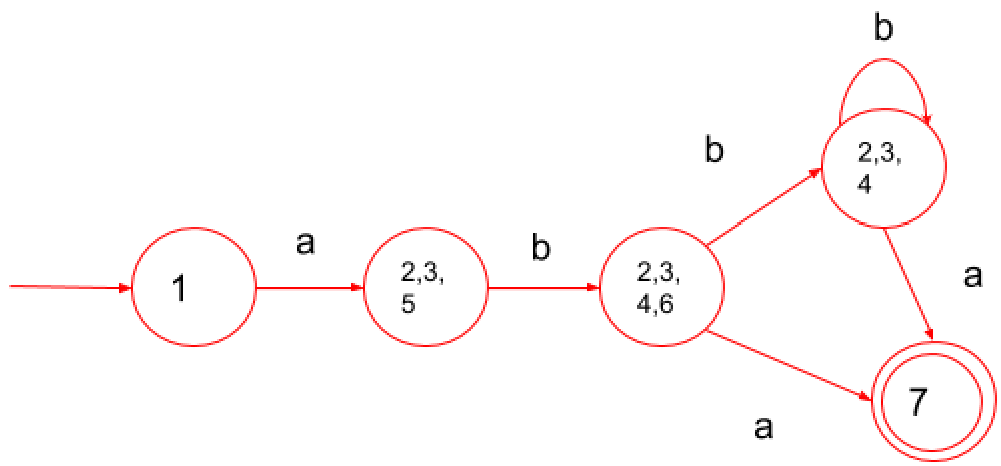
☐DFA ☑NFA Easy to translate directly from a regular expression

☑DFA ☑NFA Can accept an empty string

C) [6 pts] Draw a DFA that is equivalent to the following NFA.



Solution:

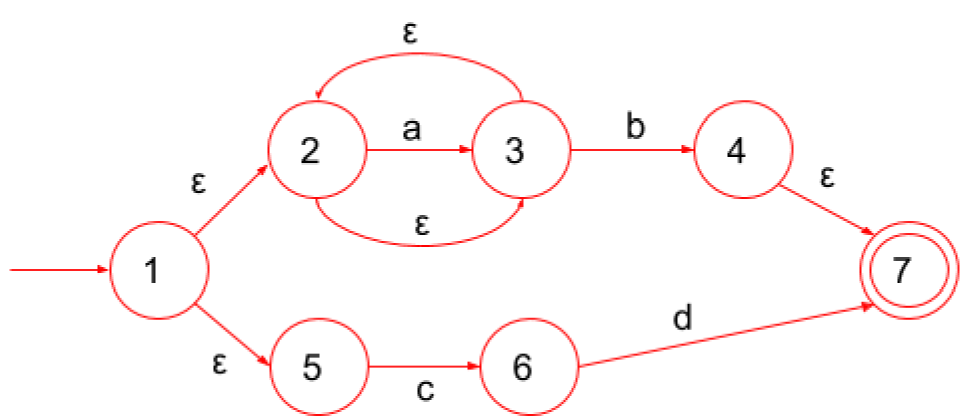


D) [4 pts] Circle any of the following strings that would be accepted by the nfa from the previous problem.

aba ***abbbbba*** **aa** ***abaa***

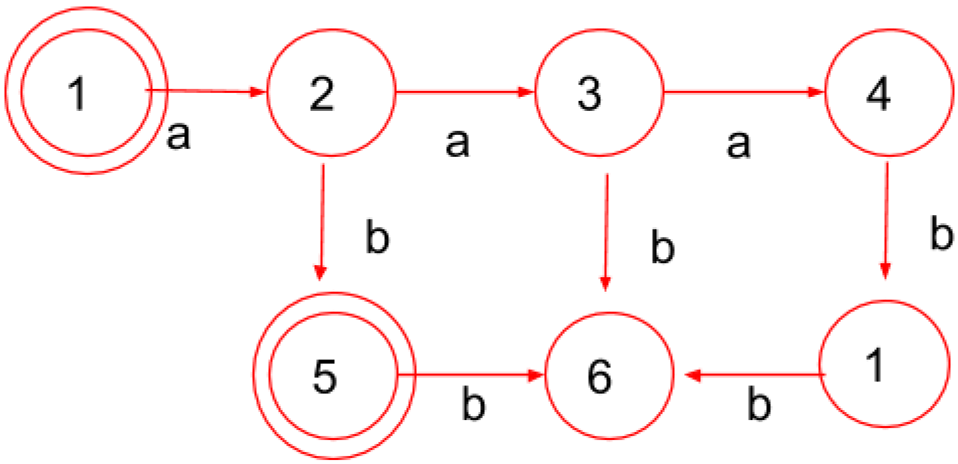
E) [6 pts] Draw an NFA that accepts the same language as the regex (**a***b**)|(**cd**). Here are some examples this NFA will accept: **b**, **ab**, **cd**, **aab**, **aaaaab**

Solution:



F) [5 pts] Draw a DFA that accepts strings of the form $\mathbf{a}^n\mathbf{b}^n$ where $0 \le n \le 3$ over $\Sigma = \{ \mathbf{a},\mathbf{b} \}$

Solution:



3. Context Free Grammars [17 pts]

A) [4 pts] Check the box next to the strings that are accepted by the following CFG. Note that here and below all nonterminals are in italics (like *T* and *W*) and terminals are in bold (like **a**, **b**).

$T \rightarrow \mathbf{a}W \mid \mathbf{b}$
$W \rightarrow \mathbf{b} \mid \mathbf{b}T \mid \mathbf{a}W$

☐ **abba** ☑ **aaabb** ☐ **baa** ☑ **aab**

B) [5 pts] Create a CFG for the language of all strings of the form $n^x f^z a^y$ where $x \geq y \geq 0$ and $z > 0$. Example strings in the language are **nfa**, **f**, **nnnfaa**. Example strings *not* in the language are **a**, **n**, **fa**, **nfaa**.

Solution:

S -> nSa | nS | A
A -> fA | f

C) [4 pts] Rewrite the following grammar so that it can be parsed by a recursive descent parser. Note that parentheses and commas, below, are terminals (along with **r**, **u**, and **o**).

$S \rightarrow A$**)**
$A \rightarrow A$**,r** | A**,u** | **(o**

Solution:

S → A)
A → (oB
B → ,rB | ,uB | ε

D) [4 pts] The following CFG is ambiguous. Rewrite the grammar to remove the ambiguity. Note that minus sign is a terminal (along with **1**, **2**, and **3**).

$E \rightarrow E$ **-** $E \mid N$
$N \rightarrow$ **1** | **2** | **3**

Solution:

E → N - E | N
N → 1 | 2 | 3

4. Parsing and Scanning [16 pts]

A) [3 pts] Recall the scanner for SmallC. Suppose, when you tokenize the variable “`for2`”, your tokenizer returned `[Tok_ID("for");Tok_Int(2)]` instead of `[Tok_ID("for2")]`. How would you fix this? (Write 1-2 sentences only.)

Solution:
Use a regular expression that captures all IDs (i.e. upper & lower letters and digits). Then check the captured term against a list of keywords. If the term doesn't match any keyword, it must be just an ID.

B) [5 pts] Consider the following CFG. Compute the first sets for each nonterminal.

$S \rightarrow mB \mid aA$
$A \rightarrow cS \mid \varepsilon$
$B \rightarrow 1\#S \mid dB \mid St \mid Ao$

FIRST(S) = *{m, a}*

FIRST(A) = *{c, ε}*

FIRST(B) = *{1, d, m, a, c, o}*

C) [8 pts] Complete the implementation for a recursive-descent parser for the provided CFG, given on the next page. Write your answer on the next page.

(scratch space, do not write your final answer here)

```
exception ParseError of string

let tok_list = ref [];;

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

let lookahead () = match !tok_list with
|[] -> None
|(h::t) -> Some h
```

$S \rightarrow mB \mid aA$

$A \rightarrow cS \mid \varepsilon$

$B \rightarrow 1\#S \mid dB \mid St \mid Ao$

```
let rec Parse_S() =
      if lookahead() = Some "m" then
            (match_tok "m"; Parse_B())
      else (* fill-in below *)
      if lookahead() = Some "a" then
            (match_tok "a"; Parse_A())
      else
            raise(Parse Error "not valid input")


and Parse_A() =
      if lookahead() = Some "c" then (* fill-in below *)
            (match_tok "c"; parse_S())
      else
            ()


and Parse_B() =
      if lookahead() = Some "1" then
            (match_tok "1"; match_tok "#"; parse_S())
      else (* fill-in below *)
      if lookahead() = Some "d" then
            (match_tok "d"; Parse_B())
      else if lookahead() = Some "m" || lookahead = Some "a" then
            (parse_S(); match_tok"t")
      else if lookahead() = Some "c" || lookahead() = Some "o" then
            (parse_A(); match_tok "o")
      else
            raise(Parse Error "not valid input")
```

5. Operational Semantics [10 pts]

A) [5 pts] Using the rules given below, show: `let x = 1 in 1 + x ⟶ 2`

In the rules, *e* refers to an expression whose abstract syntax tree (AST) is defined by the following grammar, where *x* is an arbitrary identifier and *n* is an integer.

v ::= *n*
e ::= *x* | *v* | let x = *e* in *e* | *e* + *e*

$$\text{Id}\,\frac{A(x) = v}{A; x \longrightarrow v} \qquad \text{Int}\,\frac{}{A; n \longrightarrow n}$$

$$\text{Let}\,\frac{A; e1 \longrightarrow v1 \quad A, x : v1; e2 \longrightarrow v2}{A; \text{let } x = e1 \text{ in } e2 \longrightarrow v2} \qquad \text{Add}\,\frac{A; e1 \longrightarrow v1 \quad A; e2 \longrightarrow v2 \quad v3 \text{ is } v1 + v2}{A; e1 + e2 \longrightarrow v3}$$

Solution:

$$\textit{Let}\,\cfrac{\textit{Int}\,\cfrac{}{A; 1 \longrightarrow 1} \qquad \textit{Add}\,\cfrac{\textit{Int}\,\cfrac{}{A, x : 1; 1 \longrightarrow 1} \quad \textit{Id}\,\cfrac{A, x : 1(x) = 1}{A, x : 1; x \longrightarrow 1} \quad \textit{2 is 1 + 1}}{A, x : 1; 1 + x \longrightarrow 2}}{A; \textit{let } x = 1 \ \textit{in } 1 + x \longrightarrow 2}$$

B) [5 pts] Below are operational semantics rules for a simple language, where the abstract syntax tree for expressions e and values *v* defined as follows.

v ::= false | true
e ::= v | not *e* | if *e1* then *e2*

$$\text{true}\,\frac{}{true \longrightarrow true} \qquad \text{false}\,\frac{}{false \longrightarrow false} \qquad \text{nottrue}\,\frac{e \rightarrow true}{not\ \ e \longrightarrow false} \qquad \text{notfalse}\,\frac{e \rightarrow false}{not\ \ e \longrightarrow true}$$

$$\text{Iftrue}\,\frac{\begin{array}{c} e1 \rightarrow true \\ e2 \longrightarrow v \end{array}}{if\ \ e1\ \ then\ \ e2 \longrightarrow v} \qquad \text{Iffalse}\,\frac{e1 \rightarrow false}{if\ \ e1\ \ then\ \ e2 \longrightarrow true}$$

Write a function `eval` of type `exp -> exp`, where `exp` is the OCaml representation of *e*:

```
type exp =
    Tru                         (* corresponds to true *)
  | Fals                        (* corresponds to false *)
  | If of exp * exp             (* corresponds to if e1 then e2 *)
  | Not of exp                  (* corresponds to not e *)
```

The `eval` function evaluates an expression in a manner consistent with the rules. For example:

```
eval(Tru) = Tru
eval(Not (Not Tru)) = Tru
```
etc.

```
let rec eval e =
  match e with
  | Tru -> Tru
   (* FILL IN REST *)
  | Fals -> Fals
  | If (e1, e2) -> if (eval e1) = Tru then
                     (eval e2)
                   else
                     Tru
  | Not e' -> if (eval e') = Tru then
                 Fals
               else
                 Tru
```

6. Lambda Calculus [13 pts]

A) [2 pts] Circle the **free variables** in the following λ-term:

`λx. y (λz.z y x) z`

B) [2 pts] Write a lambda calculus term that is *α-equivalent* to the one above.

Solution:

Examples: ***λx. y (λz.z y x) z***
λa. y (λb.b y a) z

C) [4 pts] Circle true or false for the following questions (1 point each)

True / False	The beta-normal form of `(λx.y z) z` is `y z`
True / False	The fixpoint combinator Y is used in lambda calculus to achieve recursion
True / ***False***	A *Church numeral* is the encoding of a real number as a lambda calculus term
True / ***False***	The expression `(λx. y) z` encodes `let x = y in z`

D) [5 pts] Reduce the following lambda expressions into beta-normal form. Show each beta reduction. If already in normal form or infinite reduction, write "normal form" or"infinite reduction", respectively.

1) `(λx. (λy. y x) (λz. x z)) (λy. y y)`

```
=> (λ x. (λ z. x z) x) (λy. y y)
=> (λ x. x x) (λ y. y y)
=> (λ y. y y) (λ y. y y)
Infinite reduction
```

2) `(λx. x y z) (λy. z)`

```
=> (λy. z) y z
=> z z
```