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

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.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PL Concepts</td>
<td>/13</td>
</tr>
<tr>
<td>2 Finite Automata</td>
<td>/31</td>
</tr>
<tr>
<td>3 Context Free Grammars</td>
<td>/17</td>
</tr>
<tr>
<td>4 Parsing</td>
<td>/16</td>
</tr>
<tr>
<td>5 Operational Semantics</td>
<td>/10</td>
</tr>
<tr>
<td>6 Lambda Calculus</td>
<td>/13</td>
</tr>
<tr>
<td>Total</td>
<td>/100</td>
</tr>
</tbody>
</table>
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).

```ocaml
let f z = let g = let h =
  let y = ref 0 in let x = ref 1 in (fun z -> let x = z+1 in
  y := !y + z; x := !x + 1;
  !y !x+z) (fun z -> _ = (print_int z,print_int x) in
  !y) !x+z)
```

Answer:

Which of these functions is not referentially transparent?

Which function’s execution outcome depends on OCaml’s evaluation order

What is a side effect carried out by at least one of the functions?

Which function’s execution is only interesting/useful because of its side effects, not what it returns?

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

aba  abbbbbba  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

F) [5 pts] Draw a DFA that accepts strings of the form a^n b^n where 0 ≤ n ≤ 3 over Σ = { a, b }
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 \( a \), \( b \)).

\[
\begin{align*}
T \rightarrow & \; aW \mid b \\
W \rightarrow & \; b \mid bT \mid aW
\end{align*}
\]

\[\square \text{abba} \quad \square \text{aaabb} \quad \square \text{baa} \quad \square \text{aab}\]

B) [5 pts] Create a CFG for the language of all strings of the form \( n^x f^y a^z \) where \( x \geq y \geq 0 \) and \( z > 0 \).
Example strings in the language are \text{nf}a, f, \text{n}nn\text{f}a\text{a}. Example strings \textit{not} in the language are a, n, \text{f}a, \text{nf}a\text{a}.

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).

\[
\begin{align*}
S \rightarrow & \; A) \\
A \rightarrow & \; A, r \mid A, u \mid (o
\end{align*}
\]

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).

\[
\begin{align*}
E \rightarrow & \; E - E \mid N \\
N \rightarrow & \; 1 \mid 2 \mid 3
\end{align*}
\]
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("for")]. How would you fix this? (Write 1-2 sentences only.)

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

\[
\begin{align*}
\text{FIRST}(S) &= \\
\text{FIRST}(A) &= \\
\text{FIRST}(B) &= 
\end{align*}
\]

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

let rec Parse_S() =
  if lookahead() = Some "m" then
    (match_tok "m"; Parse_B())
  else (* fill-in below *)

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

  and Parse_B() =
    if lookahead() = Some "1" then
      (match_tok "1"; match_tok ";"; parse_S())
    else (* fill-in below *)
5. Operational Semantics [10 pts]

A) [5 pts] Using the rules given below, show: \texttt{let x = 1 in 1 + x \rightarrow 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.

\[
\begin{align*}
  v &::= n \\
  e &::= x \mid v \mid \text{let } x = e \text{ in } e \mid e + e
\end{align*}
\]

\[
\begin{align*}
  \text{Id} &\quad A(x) = v \\
  &\implies A; x \rightarrow v \\
  \text{Int} &\quad A; n \rightarrow n \\
  \text{Let} &\quad A; e_1 \rightarrow v_1 \quad A; x : v_1; e_2 \rightarrow v_2 \\
  &\implies A; \text{let } x = e_1 \text{ in } e_2 \rightarrow v_2 \\
  \text{Add} &\quad A; e_1 \rightarrow v_1 \quad A; e_2 \rightarrow v_2 \quad v_3 \text{ is } v_1 + v_2 \\
  &\implies A; e_1 + e_2 \rightarrow v_3
\end{align*}
\]
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.

\[
\begin{align*}
v & ::= \text{false} \mid \text{true} \\
e & ::= v \mid \text{not} \ e \mid \text{if} \ e_1 \ \text{then} \ e_2
\end{align*}
\]

\[
\begin{array}{c}
\text{true} \rightarrow \text{true} \\
\text{false} \rightarrow \text{false} \\
\text{not} \ e \rightarrow \text{false} \\
\text{not} \ e \rightarrow \text{true}
\end{array}
\]

\[
\begin{array}{c}
\text{e}_1 \rightarrow \text{true} \\
\text{if} \ e_1 \ \text{then} \ e_2 \rightarrow v \\
\text{if} \ e_1 \ \text{then} \ e_2 \rightarrow \text{true}
\end{array}
\]

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

\[
\begin{align*}
\text{type exp} &= \\
&= \text{Tru} \ (* \text{corresponds to true} *) \\
&\mid \text{Fals} \ (* \text{corresponds to false} *) \\
&\mid \text{If \ of \ exp \ * \ exp} \ (* \text{corresponds to if e}_1 \ \text{then e}_2 \ *) \\
&\mid \text{Not \ of \ exp} \ (* \text{corresponds to not e} *)
\end{align*}
\]

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

\[
\begin{align*}
\text{eval(Tru)} &= \text{Tru} \\
\text{eval(Not (Not Tru))} &= \text{Tru} \\
\text{etc.}
\end{align*}
\]

\[
\begin{align*}
\text{let rec eval e} &= \\
&= \text{match e with} \\
&\mid \text{Tru} \rightarrow \text{Tru} \\
&\text{(* \text{FILL IN REST} *)}
\end{align*}
\]
6. Lambda Calculus [13 pts]
A) [2 pts] Circle the free variables in the following λ-term:

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

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

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

- True / False The beta-normal form of $$(\lambda x. y\ z)\ z$$ 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 $$(\lambda x. y)\ z$$ 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) $$(\lambda x. (\lambda y. y\ x)\ (\lambda z. z\ z))\ (\lambda y. y\ y)$$

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