First and Last Name (PRINT): ____________________________________________

9-Digit University ID: __________________________________________________

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.
- Write your 9-Digit UID at the top of EVERY PAGE.

<table>
<thead>
<tr>
<th>Section</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PL Concepts</td>
<td>/ 15</td>
</tr>
<tr>
<td>2. Finite Automata</td>
<td>/ 30</td>
</tr>
<tr>
<td>3. CFGs and Parsing</td>
<td>/ 30</td>
</tr>
<tr>
<td>4. Operational Semantics</td>
<td>/ 10</td>
</tr>
<tr>
<td>5. Lambda Calculus</td>
<td>/ 15</td>
</tr>
<tr>
<td>Total</td>
<td>/ 100</td>
</tr>
</tbody>
</table>

Please write and sign the University Honor Code below: I **pledge on my honor that I have not given or received any unauthorized assistance on this examination.**

I solemnly swear that I didn’t cheat.

Signature: __________________________
1. [15pts] PL Concepts

1 (7pts) **Circle your answers.** Each T/F question is 1 point.

- **T** F A regular expression can express all palindromes with letters A-Z, and shorter than 10 letters
- **T** F Static analysis, such as type checking, occurs before parsing
- **T** F There are multiple paths by which the same string can be accepted in a DFA
- **T** F Calling a grammar ambiguous is equivalent to saying a string may have multiple different leftmost derivations
- **T** F Using lookahead in our parser is an example of predictive parsing
- **T** F Operational semantics are analogous to interpreting a program
- **T** F Regular expressions are more powerful than DFAs (i.e., they can express more languages than DFAs can)

2 (1pts) The step below is an example of...

\[(\lambda x . x \ y) \ (\lambda z . a \ z)\]
\[(\lambda z . a \ z) \ y\]

A. \(\alpha\)-conversion

B. \(\beta\)-reduction
3  (3pts)  What is the output of the following OCaml code? (That is, what is printed)

```ocaml
let x = ref 0 in
let y = x in
  y := 1;
  print_int !x;
  print_int !y
```

**OUTPUT:** 1 1

4  (4pts)  What is printed by the following OCaml program when the parameters are passed by call-by-name and call-by-value?

```ocaml
let f x y =
  if x > 5 then (y,y) else (10,10);
let f 10 (print_string "hello"; 2);
```

**Call-by-name:** hellohello  **Call-by-value:** hello
2. [30pts] Finite Automata

1. (6pts) Which of the following strings are accepted by this NFA? *Circle all that apply.*

![NFA Diagram]

A. abcab
B. abca
C. abccc
D. aacaccaca

2. (8pts) Construct an NFA that accepts the same language as the following regular expression. There are many answers, any equivalent NFA will be accepted.

\[(a+|b^*)c?\]
3 (6pts) Answer the following questions about this NFA:

![Diagram of an NFA with states 0, 1, and 2, transitions labeled with symbols a and b, and ε]

- e-closure({0}) = {0, 1}
- e-closure(move({0, 1}, a)) = {0, 1, 2}

4 (10pts) Give a DFA equivalent to the NFA above. Any equivalent DFA will be accepted, but your answer should be clear. You may give steps for partial credit.

![Diagram of a DFA with states 0, 1, 2, and 1, 2, transitions labeled with symbols a and b, and ε]
3. [30pts] CFGs and Parsing

1. (5pts) Write a CFG that generates the following language:

\[ a^x b^y c^{x+y}, \text{ where } x, y \geq 0 \]

\[
S \rightarrow aSc \mid B \\
B \rightarrow bBc \mid \varepsilon 
\]

2. (5pts) The following CFG is ambiguous. Rewrite it so that it is not ambiguous. There are many answers, any CFG which is equivalent and is not ambiguous will be accepted. (Note: here, the terminals are: +, *, (,), a, and b.)

\[
E \rightarrow E + E \mid E \times E \mid (E) \mid a \mid b 
\]

\[
E \rightarrow T + E \mid T \\
T \rightarrow W \times T \mid W \\
W \rightarrow (E) \mid a \mid b 
\]

3. (4pts) List the FIRST SETS for each nonterminal in the following grammar (lowercase letters are terminals):

\[
S \rightarrow aB \mid Bb \mid Sc \\
B \rightarrow dB \mid d 
\]

**FIRST(S) = \{ a, d \}**

**FIRST(B) = \{ d \}**
4 (6pts) Indicate if each of the following grammars can be parsed by a recursive descent parser. If the answer is no, give a very brief explanation why.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Yes</th>
<th>No</th>
<th>If no, why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → S + S</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N → 1</td>
<td>2</td>
<td>3</td>
<td>(S)</td>
</tr>
<tr>
<td>S → aS</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B → bB</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S → Sb</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A → aAc</td>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 (10pts) Complete the OCaml implementation for a recursive-descent parser of the following context-free grammar. The implementation of `match_tok` and `lookahead` are given below:

```ocaml
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
```

NOTE: this parser takes the imperative approach. Also notice that the tokens are simply strings. So the token list for the string "abcdc" would look like `"a"; "b"; "c"; "d"; "c"`. You are not creating an AST. If the input is invalid, throw a `ParseError`.

```
S → bS | cT
T → Ra | RbR
R → dR | ε
```

Write your implementation on the next page. The CFG is repeated on the next page for your reference.
let rec parse_S () =
  if lookahead () = Some "b" then
    match_tok "b";
    parse_S ()
  else (* fill in below *)
    if lookahead () = Some "c" then
      match_tok "c";
      parse_T ()
    else
      raise (ParseError "invalid")

and rec parse_T () = (* fill in below *)
  parse_R ();
  if lookahead () = Some "a" then
    match_tok "a"
  else if lookahead () = Some "b" then
    match_tok "b";
    parse_R ()
  else
    raise (ParseError "invalid")

and rec parse_R () =
  if lookahead () = None then
    ()
  else (* fill in below *)
    if lookahead () = Some "d" then
      match_tok "d";
      parse_R ()
    else
      raise (ParseError "invalid")
4. [10pts] Operational Semantics

1. (2pts) Below is an incorrect rule for an if-then-else construct when the condition is true. Identify the mistake, and explain how to fix it. Here, the expression if $a$ then $b$ else $c$ is encoded as if-then-else $a \ b \ c$.

\[
\frac{A; e_1 \rightarrow \text{true} \quad A; e_3 \rightarrow v}{A; \text{if-then-else} \ \ e_1 \ e_2 \ e_3 \rightarrow v}
\]

The second part on the top should be $e_2$, not $e_3$.

2. (3pts) Describe what the operator \texttt{myst} does, or give its name.

\[
\frac{A; e_1 \rightarrow \text{true} \quad A; e_2 \rightarrow \text{true}}{A; \text{e1 myst e2} \rightarrow \text{true}}
\]

\[
\frac{A; e_1 \rightarrow \text{false} \quad A; e_2 \rightarrow \text{true}}{A; \text{e1 myst e2} \rightarrow \text{false}}
\]

\[
\frac{A; e_1 \rightarrow \text{true} \quad A; e_2 \rightarrow \text{false}}{A; \text{e1 myst e2} \rightarrow \text{false}}
\]

\[
\frac{A; e_1 \rightarrow \text{false} \quad A; e_2 \rightarrow \text{false}}{A; \text{e1 myst e2} \rightarrow \text{false}}
\]

The AND operator
3 (5pts) Using the following rules, show that:

\[ A; \text{let } x = 3 \text{ in let } x = 2 \text{ in } x + x \rightarrow 4 \]

\[ \frac{A; n \rightarrow n}{A; x \rightarrow v} \]

\[ \frac{A; e_1 \rightarrow v_1 \quad A; x : v_1; e_2 \rightarrow v_2}{A; \text{let } x = e_1 \text{ in } e_2 \rightarrow v_2} \]

\[ \frac{A; e_1 \rightarrow n_1 \quad A; e_2 \rightarrow n_2 \quad n_3 \text{ is } n_1 + n_2}{A; e_1 + e_2 \rightarrow n_3} \]

\[ \frac{A, x : 3, x : 2(x) = 2}{A, x : 3, x : 2 \rightarrow 2} \]

\[ \frac{A, x : 3, x : 2; x \rightarrow 2}{A, x : 3, x : 2; x \rightarrow 2 \quad 4 \text{ is } 2 + 2} \]

\[ \frac{A, x : 3; 2 \rightarrow 2}{A, x : 3 \rightarrow 3} \]

\[ \frac{A, x : 2, x : 3; x + x \rightarrow 4}{A; \text{let } x = 3 \text{ in let } x = 2 \text{ in } x + x \rightarrow 4} \]
5. [15pts] Lambda Calculus

1 (8pts) Reduce the expressions as far as possible by showing the intermediate $\beta$-reductions and $\alpha$-conversions. Make sure to show each step for full credit!

\[(\lambda x. \lambda y. x y) (\lambda y. y)x\]

\[
((\lambda x. (\lambda y. x y)) (\lambda y. y))x \\
(\lambda y. (\lambda y. y) y)x \\
(\lambda y. (\lambda z. z) y)x \\
(\lambda z. z)x \\
x
\]

\[(\lambda x. \lambda y. x y y) (\lambda x. m) n\]

\[
((\lambda x. (\lambda y. x y y)) (\lambda x. m))n \\
(\lambda y. (\lambda x. m) y y)n \\
(\lambda x. m)n n \\
((\lambda x. m)n)n \\
n n
\]
2  (7pts) Reduce the following expression to β-normal form using both call-by-name and call-by-value. Show each step, including any β-reductions and α-conversions. If there is infinite reduction, write “infinite reduction.”

\[(\lambda y.x) ((\lambda x. x x x) (\lambda z. z z z))\]

**Call-by-name:**

\[
(\lambda y.x) ((\lambda x. x x x) (\lambda z. z z z))
\]

\[x\]

**Call-by-value:**

\[
(\lambda y.x) ((\lambda x. x x x) (\lambda z. z z z))
\]

\[
(\lambda y.x) ((\lambda z. z z z) (\lambda z. z z z) (\lambda z. z z z))
\]

\[
(\lambda y.x) ((\lambda z. z z z) (\lambda z. z z z) (\lambda z. z z z) (\lambda z. z z z))
\]

\[
...
\]

**Infinite reduction**