Name: ________________________________

UID: ________________________________

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination

Signature: ________________________________

Ground Rules

• You may use anything on the accompanying reference sheet anywhere on this exam
• Please write legibly. **If we cannot read your answer you will not receive credit**
• You may not leave the room or hand in your exam within the last 10 minutes of the exam
• If anything is unclear, ask a proctor. If you are still confused, write down your assumptions in the margin

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
</tr>
<tr>
<td>P2</td>
<td>15</td>
</tr>
<tr>
<td>P3</td>
<td>12</td>
</tr>
<tr>
<td>P4</td>
<td>10</td>
</tr>
<tr>
<td>P5</td>
<td>20</td>
</tr>
<tr>
<td>P6</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>85</td>
</tr>
</tbody>
</table>
Problem 1: Language Concepts

One could theoretically write project 3 in Lambda Calculus True False
Lambda Calc is a Turing complete language True False
Regular Expressions have computational power that is equivalent to Turing Machines. True False
Regular expressions can simulate FSM, but not TM True False
If a language's grammar is changed, then the parser must be modified True False
The parser is tied to a language's grammatical structure True False
fun a b -> a b is an example of a higher order function in Ocaml True False
'a' is treated like a function here True False
If a function f is acceptable input to fold_left, then it is also acceptable for fold_right True False
The types of the functions passed into fold_left and fold_right are different True False
Operational Semantics describes the meaning of language through operations that will be performed. True False
Lexers typically identify problems with inputs that don't obey a grammar such as forgetting a closing parentheses in an expression like 91*(21 + 5 True False
Lexers look for token errors / bad characters, not errors in grammar True False
Because the Pure Lambda Calculus only has Functions, Applications, and Variables, it is not possible to encode concepts such as True and False with it. True False
lots of things including Booleans can be encoded in the Lambda Calculus True False

Problem 2: Lambda Calculus

(a) Lazy Evaluation, Single Step: Perform a single step of Beta Reduction using the Lazy / Call by Name Evaluation Strategy on the given Lambda Calculus expression. If the expression cannot be reduced, select “Beta Normal Form”.

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

A) \(a \lambda x. x a\)  
B) \(\lambda x. x ((\lambda y. y) b)\)  
C) \((a \lambda x. x a)(b)\)  
D) Beta Normal Form  
E) None of the above

(a) Lazy Evaluation, Single Step: Perform a single step of Beta Reduction using the Lazy / Call by Name Evaluation Strategy on the given Lambda Calculus expression. If the expression cannot be reduced, select “Beta Normal Form”.

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

A) \(a \lambda x. x a\)  
B) \(\lambda x. x ((\lambda y. y) b)\)  
C) \((a \lambda x. x a)(b)\)  
D) Beta Normal Form  
E) None of the above

(b) Eager Evaluation, Single Step: As before, perform a single step of Beta Reduction but this time use the Eager / Call by Value Evaluation Strategy.

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

A) \(a \lambda x. x a\)  
B) \(\lambda x. x ((\lambda y. y) b)\)  
C) \((a \lambda x. x a)(b)\)  
D) Beta Normal Form  
E) None of the above

(c) Reduce to Normal Form: Convert the following to Beta Normal Form: \((\lambda x. (\lambda y. x a)b)(\lambda x.a)\)

A) \(\lambda x. ax\)  
B) \(cd\)  
C) \(ba\)  
D) \(aa\)  
E) Can’t reduce  
F) infinite recursion  
G) None
Problem 3: Context Free Grammars

Consider the following Grammar:

\[ E \rightarrow aSSc \]
\[ S \rightarrow aSb|bSc|T \]
\[ T \rightarrow a|b|c \]

(a) Which of the following strings are grammatically correct? Select all that apply.

- A aab
- B abccaabc
- C abacbcc
- D abbac
- E None

(b) Prove that this grammar is ambiguous using the string abbcc

\[ E \rightarrow aSSc \rightarrow abScSc \rightarrow abbcSc \rightarrow abbcTc \rightarrow abbccc \]
\[ E \rightarrow aSSc \rightarrow aTSc \rightarrow abSc \rightarrow abbScc \rightarrow abbTcc \rightarrow abbccc \]

Problem 4: Lexing Parsing and evaluating

Given the following CFG, and assuming strong, static typing as is used in OCaml, at what stage of language processing would the nearby expressions fail? Mark 'Valid' otherwise.

\[ E \rightarrow + EE | * EE | - EE | / EE | X \]
\[ X \rightarrow \text{and } XX | \text{or } XX | P \]
\[ P \rightarrow \text{true } | \text{false } | n \in \text{Positive Numbers} \]

You may assume this is simple prefix notation for common mathematical and logical semantics

**Constraint:** The parser in use will reject strings that have "leftover" input that does not fit into a single parse tree.

<table>
<thead>
<tr>
<th>Expression</th>
<th>L</th>
<th>P</th>
<th>E</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 * 3 + 2 3</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>^ 4 5</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>- + 1 23</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>and 2 5</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>5 exp 2 + 6</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>* 2 and true false</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>and true or false false</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>false true</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>true and false or true</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
<tr>
<td>42</td>
<td>L</td>
<td>P</td>
<td>E</td>
<td>V</td>
</tr>
</tbody>
</table>
Problem 5: OCaml Programming

The following variant type defines a binary tree.

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
```

Write a function called `jumping_layers` that returns a `('a list * 'a list)` where the first list has all the `'_a` items from the even indexed tree layers and the second list has all the items from the odd tree layers. The order of items in the lists does not matter.

Examples:

```
let even_odd_layers t =
  let rec jl t iseven =
    match t with
      | Leaf(x) -> if iseven then ([x],[]) else ([],x)
      | Node(l,v,r) -> let e1,o1 = jl l (not iseven) in
          let e2,o2 = jl r (not iseven) in
          if iseven then (v::(e1@e2),o1@o2)
                    else (e1@e2,v::(o1@o2))
    in jl t true
```

```
t-> 1
   / 
  2 3
Node(Leaf(2),
1,        Node(Node(Leaf(4),
Leaf(3)) 2,
        4  5
Leaf(5)),
=> ([1], [2,3])
  even   odd
=> ([1,4,5], [2,3])
  even   odd
```

You may define recursive helper function(s) as you find them useful.

```
let even_odd_layers t =
  let rec jl t iseven =
    match t with
      | Leaf(x) -> if iseven then ([x],[]) else ([],x)
      | Node(l,v,r) -> let e1,o1 = jl l (not iseven) in
          let e2,o2 = jl r (not iseven) in
          if iseven then (v::(e1@e2),o1@o2)
                    else (e1@e2,v::(o1@o2))
    in jl t true
```
Problem 6: Operational Semantics

Consider the following rules for RNACODE, using OCaml as the Metalanguage:

\[
\begin{align*}
\text{TTT} & \rightarrow \text{TTT} \\
A; e_1 \Rightarrow v_1 & \quad v_1 = \text{GGG} \\
A; \text{Lysine? } e_1 & \Rightarrow \text{GGG} \\
A; x : v \ (x) = v & \\
A; x : v ; x \Rightarrow v \\
A; e_1 & \Rightarrow v_1 \\
A; \text{ENCODE x AS } e_1 ; e_2 & \Rightarrow v_2 \\
A; \text{SWAP } x y \text{ in } e & \Rightarrow v
\end{align*}
\]

Complete the Opsem proof for the following program:

\[
A, y : \text{TTT}; \text{ENCODE x AS GGG}; \text{SWAP x y in Lysine? } x \Rightarrow \text{TTT}
\]
Cheat Sheet

OCaml

(* Lists *)
let lst = []
let lst = [1;2;3;4]

(* :: (cons) has type 'a->'a list -> 'a list *)
1::2::3::[] = [1;2;3]

(* @ (append) has type 'a list -> 'a list-> 'a list *)
[1;2;3] @ [4;5;6] = [1;2;3;4;5;6]

(* variants *)
type linkedlist = Cons of int * linkedlist | Nil
Cons(1,Cons(2,Cons(3,Nil)))

(* Anonymous Functions *)
(fun a b c -> a + b + c *)

(* Map and Fold *)
let rec map f l = match l with
  [] -> []
  | x::xs -> (f x)::(map f t)

let rec fold_left f a l = match l with
  [] -> a
  | x::xs -> fold_left f (f a x) xs

let rec fold_right f l a = match l with
  [] -> a
  | x::xs -> f x (fold_right f xs a)

Lambda Calc Encodings

We will give you the encodings that you will need. They may or may not look like/include the following:

\( \lambda x. \lambda y. x \) = true
\( \lambda x. \lambda y. y \) = false
\( e_1 e_2 e_3 \) = if \( e_1 \) then \( e_2 \) else \( e_3 \)