CMSC330 - Organization of Programming Languages
Fall 2023 - Exam 2

CMSC330 Course Staff
University of Maryland
Department of Computer Science

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**  
[Total 8 pts]

One could theoretically write project 3 in Lambda Calculus  

Regular Expressions have computational power that is equivalent to Turing Machines.  

If a language's grammar is changed, then the parser must be modified  

`fun a b -> a b` is an example of a higher order function in Ocaml  

If a function `f` is acceptable input to `fold_left`, then it is also acceptable for `fold_right`  

Operational Semantics describes the meaning of language through operations that will be performed.  

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`  

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.

**Problem 2: Lambda Calculus**  
[Total 15 pts]

(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”.

(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.

(c) **Reduce to Normal Form:** Convert the following to Beta Normal Form: `((λx.(λy.x)a)b)(λx.ax)`
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

Problem 4: Lexing Parsing and evaluating

Given the following CFG, and assuming Ocaml's typing, at what stage of language processing would the nearby expressions fail? Mark 'Valid' otherwise.

\[ E \Rightarrow + \ E \ E | \ast \ E \ E | - \ E \ E | / \ E \ E | X \]
\[ X \Rightarrow \text{and} \ X \ X | \text{or} \ X \ X | P \]
\[ P \Rightarrow \text{true} | \text{false} | n \in \text{Positive Numbers} \]

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.

**Constraint:** You may assume there are tokens for only the terminal characters.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Lexer</th>
<th>Parser</th>
<th>Evaluator</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.

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

Write a function called `even_odd_layers` that returns a `list 'a * list 'a` where the first list has all the `list '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:

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

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

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

*HINT: Higher-order functions are not so useful for this problem in favor a more tailored approach.*

```ocaml
let even_odd_layers t =
```
**Problem 6: Operational Semantics**

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

\[
\begin{align*}
TTT & \rightarrow TTT \\
A; e_1 \Rightarrow \nu_1 & \quad \nu_1 = GGG \\
\hline
A; \text{Lysine? } e_1 & \Rightarrow GGG \\
A, x : \nu(x) = \nu & \Rightarrow A, x : \nu ; x \Rightarrow \nu \\
A, x : \nu_1, y : \nu_2 ; x \Rightarrow \nu_1 & \quad A, x : \nu_1, y : \nu_2 ; y \Rightarrow \nu_2 & \quad A, x : \nu_2, y : \nu_1 ; e \Rightarrow \nu \\
\hline
A, x : \nu_1, y : \nu_2 ; \text{SWAP } x \text{ y in } e & \Rightarrow \nu \\
\end{align*}
\]

Complete the Opsem proof for the following program:

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

**OCaml**

(* Lists *)

```ocaml
let lst = []
let lst = [1; 2; 3; 4]
```

(* :: (cons) has type 'a->'a list -> 'a list *)

```ocaml
1::2::3::[] = [1;2;3]
```

(* @ (append) has type 'a list -> 'a list -> 'a list *)

```ocaml
[1;2;3]@[4;5;6] = [1;2;3;4;5;6]
```

(* variants *)

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

(* Anonymous Functions *)

```ocaml
(fun a b c -> a + b + c *)
```

(* Map and Fold *)

```ocaml
let rec map f l = match l with
  [] -> []
| x::xs -> (f x)::(map f t)
```

```ocaml
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.\y.x = true
\x.\y.y = false
\ e_1 \ e_2 \ e_3 = if \ e_1 \ then \ e_2 \ else \ e_3
```