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

Question	Points
P1	8
P2	15
P3	12
P4	10
P5	20
P6	20
Total	85

Problem 1: Language Concepts

[Total 8 pts]

	True	False
One could theoretically write project 3 in Lambda Calculus	(T)	(F)
Regular Expressions have computational power that is equivalent to Turing Machines.	(T)	(F)
If a language's grammar is changed, then the parser must be modified	(T)	(F)
`fun a b -> a b` is an example of a higher order function in Ocaml	(T)	(F)
If a function `f` is acceptable input to `fold_left`, then it is also acceptable for `fold_right`	(T)	(F)
Operational Semantics describes the meaning of language through operations that will be performed.	(T)	(F)
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`	(T)	(F)
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.	(T)	(F)

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"*.

$(a\ \lambda x.\ x\ a)((\lambda y.\ y)\ b)$

(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

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

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

$(\lambda x.\ \lambda y.\ x\ y)((\lambda b.\ b\ b)\ a)$

(A) $(\lambda x.\ \lambda y.\ x\ y)(a\ a)$
(B) $(\lambda y.\ ((\lambda b.\ b\ b)\ a)\ y)$
(C) $\lambda x.\ (y\ y)\ x$
(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.

$(a\ \lambda x.\ x\ a)((\lambda y.\ y)\ b)$

(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

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

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

$(\lambda x.\ \lambda y.\ x\ y)((\lambda b.\ b\ b)\ a)$

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

(c) **Reduce to Normal Form:** Convert the following to Beta Normal Form: $(\lambda x.(\lambda y.xa)b)(\lambda x.ax)$

(A) $\lambda x.ax$ (B) $c\ d$ (C) $b\ a$ (D) $a\ a$
(E) Can't reduce (F) infinite recursion (G) None

Problem 3: Context Free Grammars

[Total 12 pts]

Consider the following Grammar:

```
E -> aSSc
S -> aSb|bSc|T
T -> 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 abbccc

Problem 4: Lexing Parsing and evaluating

[Total 10 pts]

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

$$E \Rightarrow +\ E\ E \mid *\ E\ E \mid -\ E\ E \mid /\ E\ E \mid X$$
$$X \Rightarrow \texttt{and}\ X\ X \mid \texttt{or}\ X\ X \mid P$$
$$P \Rightarrow \texttt{true} \mid \texttt{false} \mid n \in \texttt{Positive Numbers}$$

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.

	Lexer	Parser	Evaluator	Valid
`2 * 3 + 2 3`	(L)	(P)	(E)	(V)
`^ 4 5`	(L)	(P)	(E)	(V)
`- + 1 23`	(L)	(P)	(E)	(V)
`and 2 5`	(L)	(P)	(E)	(V)
`5 exp 2 + 6`	(L)	(P)	(E)	(V)
`* 2 and true false`	(L)	(P)	(E)	(V)
`and true or false false`	(L)	(P)	(E)	(V)
`false true`	(L)	(P)	(E)	(V)
`true and false or true`	(L)	(P)	(E)	(V)
`42`	(L)	(P)	(E)	(V)

Problem 5: OCaml Programming

[Total 20 pts]

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 `even_odd_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:

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

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

```
t->  1
Leaf(1)
=> ([1],[])
   even odd
```

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.

```
let even_odd_layers t =
```

Problem 6: Operational Semantics

[Total 20 pts]

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

$$\frac{}{\text{TTT} \longrightarrow \text{TTT}} \qquad \frac{}{\text{GGG} \longrightarrow \text{GGG}}$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad v_1 = \text{GGG}}{A;\ \text{Lysine?}\ e_1 \Rightarrow \text{GGG}} \qquad \frac{A;\ e_1 \Rightarrow v_1 \qquad v_1 <> \text{GGG}}{A;\ \text{Lysine?}\ e_1 \Rightarrow \text{TTT}}$$

$$\frac{A, \text{x} : v\ (\text{x})\ =\ v}{A, \text{x} : v;\ \text{x} \Rightarrow v} \qquad \frac{A;\ e_1 \Rightarrow v_1 \qquad A, \text{x} : v_1;\ e_2 \Rightarrow v_2}{A;\ \text{ENCODE x AS}\ e_1\ ;\ e_2 \Rightarrow v_2}$$

$$\frac{A, x : v_1, y : v_2;\ x \Rightarrow v_1 \qquad A, x : v_1, y : v_2;\ y \Rightarrow v_2 \qquad A, x : v_2, y : v_1;\ e \Rightarrow v}{A, x : v_1, y : v_2;\ \text{SWAP}\ x\ y\ \text{in}\ \ e \Rightarrow v}$$

Complete the Opsem proof for the following program:

$$\texttt{A,y:TTT; ENCODE x AS GGG; SWAP x y in Lysine? x} \Rightarrow \texttt{TTT}$$

$$\dfrac{\dfrac{}{\boxed{}} \qquad \dfrac{\dfrac{\boxed{}}{A, \text{y} : \text{TTT}, \text{x} : \text{GGG}; \text{x} \Rightarrow GGG} \qquad \dfrac{\boxed{}}{\boxed{}} \qquad \dfrac{\dfrac{\boxed{}}{\boxed{}; \text{x} \Rightarrow TTT} \qquad \boxed{}}{A, \text{y} : \text{GGG}, \text{x} : \text{TTT}; \boxed{} \Rightarrow TTT}}{A, \text{y} : \text{TTT}, \text{x} : \text{GGG}; \boxed{} \Rightarrow \text{TTT}}}{A, \text{y} : \text{TTT};\ \text{ENCODE x AS GGG}\ ; \text{SWAP x y in Lysine? x} \Rightarrow \boxed{}}$$

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