CMSC330 Spring 2017 Midterm 2

Name (PRINT YOUR NAME as it appears on gradescope):

_____

Discussion Time (circle one) **10am 11am 12pm 1pm 2pm 3pm**

Discussion TA (circle one) **Aaron Alex Austin Ayman Daniel Eric Greg Jake JT Sam Tal Tim Vitung**

Instructions

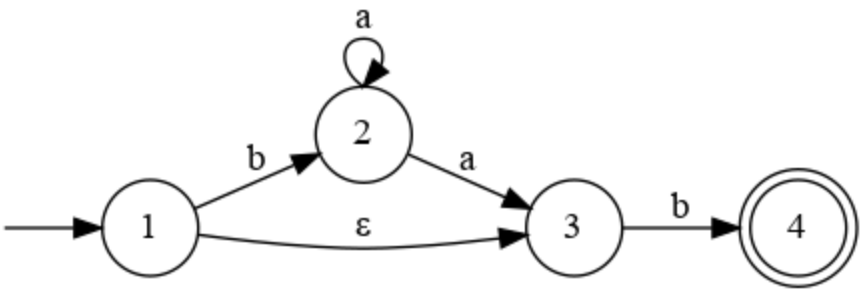
- Do not start this test until you are told to do so!
- You have 75 minutes to take this midterm.
- This exam has a total of 100 points, so allocate 45 seconds for each point.
- This is a closed book exam. No notes or other aids are allowed.
- Answer essay questions concisely in 2-3 sentences. Longer 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.

	Problem	Score
1	Finite Automata	/20
2	Context Free Grammars	/20
3	Parsing	/13
4	OCaml Programming	/10
5	PL Concepts	/15
6	Operational Semantics	/9
7	Lambda Calculus	/13
	Total	/100

1. Finite Automata (20 pts)

A. (5 pts) Construct an NFA that accepts the same language as the regular expression $ba^*|dc$.

B. (5 pts) Reduce the following NFA to a DFA:



C. (5 pts) Write a regular expression that accepts the same language as the NFA in problem B.

D. (3 pts) Can you write a regular expression for strings of length 5 or less that are palindromes (i.e., are mirror images of themselves)? Justify your answer.

E. (2 pts) True or false: there exist regular expressions that cannot be expressed as NFAs.

2. Context Free Grammars (20 pts)

A. (4 pts) Consider the following CFG, where **a** and **b** are terminals, *S* and *T* are nonterminals.

S -> **a***T*
T -> **bb***T* | **a**

Consider the following strings; circle those that are accepted by the above CFG.

abb **bba** **aa** **abbbba**

B. (3 pts) Give a regular expression that accepts the same strings as the CFG as part A.

C. Consider the following CFGs (where **and**, **true**, and **false** are terminals, and *A* and *S* are nonterminals):

CFG 1	CFG 2
S -> *S* **and** *A* | *A* *A* -> **true** | **false**	*S* -> *A* **and** *S* | *A* *A* -> **true** | **false**

a. (2 pts) Which CFG treats **and** as a left associative operator?

b. (2 pts) Which CFG *cannot* be used (as is) with a predictive parser?

D. Given the CFG:

```
S -> S*S | T
T -> a | b
```

a) (3 pts) Give a leftmost derivation for the string **a*a*b**

b) (3 pts) Give a different leftmost derivation for the string **a*a*b**

c) (3 pts) Rewrite the grammar so it is unambiguous, treating * as a left associative operator.

3. Parsing (13 pts)

$S \rightarrow$ **cd** | **b**A | A**a**
$A \rightarrow$ **d**S | ε

A. (5 pts) Calculate the first sets of the above grammar.

FIRST(S) = { }

FIRST(A) = { }

B. (8 pts) Fill in the blanks for parse functions `parse_S` and `parse_A` for the CFG shown above. Both parse functions are of type `unit -> unit`. You may use the following helpers, described in class, which have their type signatures listed next to them:

- `lookahead: unit -> string`
- `match_tok: string -> unit`
- `raise_error: unit -> unit`

```
let rec parse_S () =
    if lookahead () = “c” then
        (match_tok “c” ; match_tok “d”)

    else if                                  then


    else if                                  then


    else


;;

let rec parse_A () =

    if                                 then


    else
```

4. OCaml Programming (10 pts)

Recall the SmallC interpreter from project 3. Here are some snippets from its code:

```
type stmt =
  | NoOp
  | Seq of stmt * stmt
  | Declare of data_type * string
  | Assign of string * expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | Print of expr

let eval_stmt (e:env) (s:stmt) = match s with
...
| While(guard_expr, body) -> begin
    let guard = eval_expr e guard_expr in
    match guard with
    | Val_Bool(true) -> eval_stmt (eval_stmt e body) s
    | Val_Bool(false) -> e
    | _ -> raise (TypeError("Can't use non-bool as while guard"))
  end
...
```

Imagine a new `stmt` variant to represent a `for` loop:

```
type stmt = ... (* as above *)
| For of stmt * expr * stmt * stmt
```

The tuple elements represent the initialization, the condition, the increment, and the body, respectively. Take for example, the smallC code:

```
for(i = 0; i < 10; i = i + 1) {printf(i)}
```

In this case, the fields line up as follows:

- `i = 0` is the initialization
- `i < 10` is the condition
- `i = i + 1` is the increment
- `printf(i)` is the body

After running (an updated version of) the lexer and parser, the example code above will be represented as:

```
For (Assign (“i”, Int 0),
     Less (Id “i”, Int 5),
     Assign (“i”, Plus (Id “i”, Int 1)),
     Print (Id “i”))
```

Write the code for `eval_stmt` to handle `for` loops. The semantics must satisfy the following:

- Before the first iteration, evaluate the initialization statement
- As long as the condition is true, evaluate the body followed by the increment statement
    - If the condition is non-boolean, raise an exception

(You might have a look at problem 6.C, below, before writing the code.)

You may assume a full, correct implementation of the whole project is accessible to you, including:

- `eval_expr: env -> expr -> value`
- `eval_stmt : env -> stmt -> env`
    - Excluding `for` itself

```
let eval_stmt (e:env) (s:stmt) = match s with
      ... (* all previous statement types are handled *)
      | For (init, cond, incr, body) -> (* your code below *)
```

5. PL concepts (15 pts)

A. (2 pts) In SmallC, which stage detects if some variable **x** is not declared before its first use? Circle the answer.

Lexer Parser Interpreter

B. (2 pts) True or False: An abstract syntax tree is the same as a parse tree.

C. (2 pts) An object is best encoded by one or more of which of the following? Circle the answer.

function closure module string

D. (3 pts) The Java class `Sequence` (on the left) is partially encoded as OCaml code on the right. What code should go in the gray portion?

```
class Sequence {
  int s = 0;
  void start (int r) { s = r; }
  int next () { s++; return s; }
}

Sequence s = new Sequence();
s.start(10);
int t = s.next();
int u = s.next();
```

```
let make () =
 let s = ref 0 in
 ((fun r -> ),
  (fun ()-> s := !s + 1; !s))
;;

let (start, next) = make ();;
start 10;;
let t = next();;
let u = next();;
```

(You can write your answer below, so you don't have to write in tiny letters:)

E. (6 pts) Rewrite the `smush` function to make it **tail recursive** (without changing its type). Here, the ^ operator is string concatenation (i.e., “`hello ”` ^ “`there”` = “`hello there”`). You are welcome to write helper functions.

```
let rec smush xs = match xs with
        [] -> “”
      | h::t -> h^(smush t);;

smush [] = “”;;
smush [“this ”; “is the ”; “word”] = “this is the word”;;
```

6. Operational Semantics (9 pts)

A. (3 pts) Consider the operational semantics rules from the lecture notes for MicroOCaml, using an environment-based presentation.

$$\frac{A(x) = v}{A;\ x \Rightarrow v} \qquad A;\ n \Rightarrow n$$

$$\frac{A;\ e1 \Rightarrow v1 \quad A,x:v1;\ e2 \Rightarrow v2}{A;\ \text{let } x = e1 \text{ in } e2 \Rightarrow v2} \qquad \frac{A;\ e1 \Rightarrow n1 \quad A;\ e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2}{A;\ e1 + e2 \Rightarrow n3}$$

The following is a derivation of the program **let x = 3 in x+y** under an environment that initially maps y to 3. Fill in the three missing parts.

$$\frac{\bullet,y:3,x:3;\ x \Rightarrow 3 \qquad \bullet,y:3,x:3;\ \_\_\_\_\_ \Rightarrow 3}{\_\_\_\_\_;\ x+y \Rightarrow 6}$$

$$\bullet,y:3;\ \text{let } x = 3 \text{ in } x+y \Rightarrow \_\_\_\_\_$$

B. (3 pts) The following rule is part of the operational semantics for SmallC:

$$\frac{A;\ e \Rightarrow \text{true} \qquad A;\ s1 \Rightarrow A'}{A;\ \text{if } e\ s1\ s2 \Rightarrow A'}$$

Explain this rule, in words. Your explanation should be something of the variety *if under environment A expression e evaluates to … then …* etc.

C. (3 pts) One of the operational semantics rules for while loops in SmallC is the following

```
A; e ⇒ true
A; s ⇒ A1
A1; while e s ⇒ A2
A; while e s ⇒ A2
```

Choose the rule below that is the equivalent one for `for` loops. Here we write `for s1 e s2 s` as corresponding to the SmallC syntax `for(s1; e; s2){s}`. The `skip` statement is equivalent to a no-op.

(A)

```
A; s1 ⇒ A1
A1; e ⇒ true
A2; s ⇒ A3
A3; s2 ⇒ A3
A; for s1 e s2 s ⇒ A3
```

(B)

```
A; s1 ⇒ A1
A1; e ⇒ true
A1; s ⇒ A2
A2; s2 ⇒ A3
A3; for skip e s2 s ⇒ A4
A; for s1 e s2 s ⇒ A4
```

(C)

```
A; s1 ⇒ A1
A1; e ⇒ true
A1; s2 ⇒ A2
A2; s ⇒ A3
A3; for skip e s2 s ⇒ A4
A; for s1 e s2 s ⇒ A4
```

(D)

```
A; s1 ⇒ A1
A1; e ⇒ true
A2; for skip e s2 s ⇒ A3
A3; s2 ⇒ A4
A; for s1 e s2 s ⇒ A4
```

7. Lambda Calculus (13 pts)

A. (1 pt) True or False: The lambda calculus can encode all computable functions.

B. (2 pts) Circle all occurrences of free variables in the following λ-term.

x (λx.x (λy.x y) y)

C. (2 pts) Determine whether the following λ-terms are α-equivalent (1 point each).

(λx.λy.y x) x and (λz.λy.z y) x yes / no

λx.x λy.y z x and λv.v λy.y x v yes / no

D. (2 pts) Perform one step of β-reduction on the following λ-term. (Perform alpha-conversion if necessary.)

(λx.λy.x y) (y λy.y)

E. (5 pts) A programming language uses an evaluation strategy to determine when to evaluate the argument(s) of a function call. Reduce the following lambda expression using a call-by-value (aka *eager*) strategy and a call-by-name (aka *lazy*) strategy.

Call-by-Value
(λx.λy.x y z) (λc.c) ((λa.a) b)

Call-by-name
(λx.λy.x y z) (λc.c) ((λa.a) b)