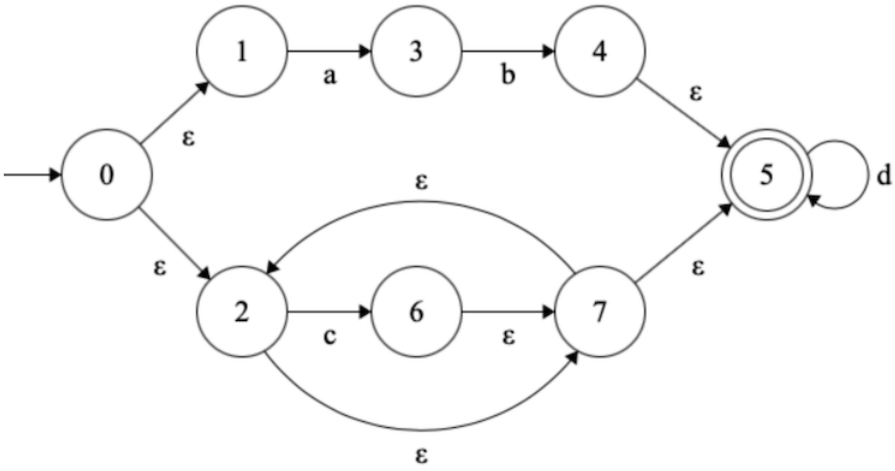
# CMSC 330 Exam 2 Fall 2021 Solutions

## Q1. Introduction

...

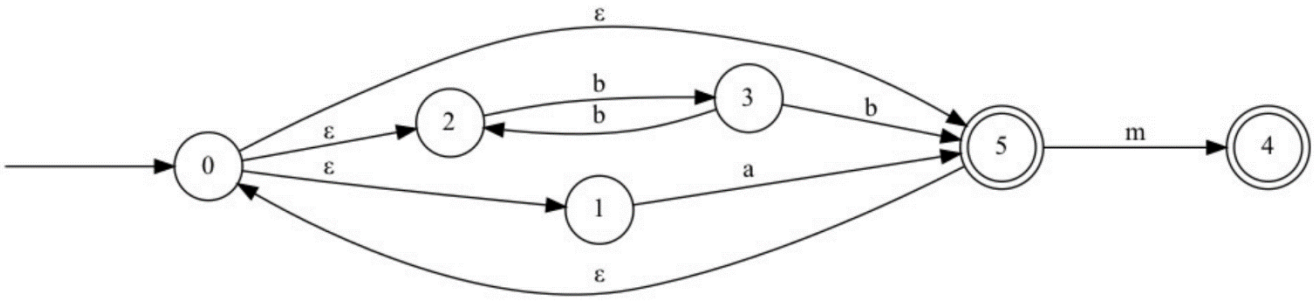
## Q2. NFA/DFA

Q2.1. Which strings will this NFA accept?




ab, ddddd

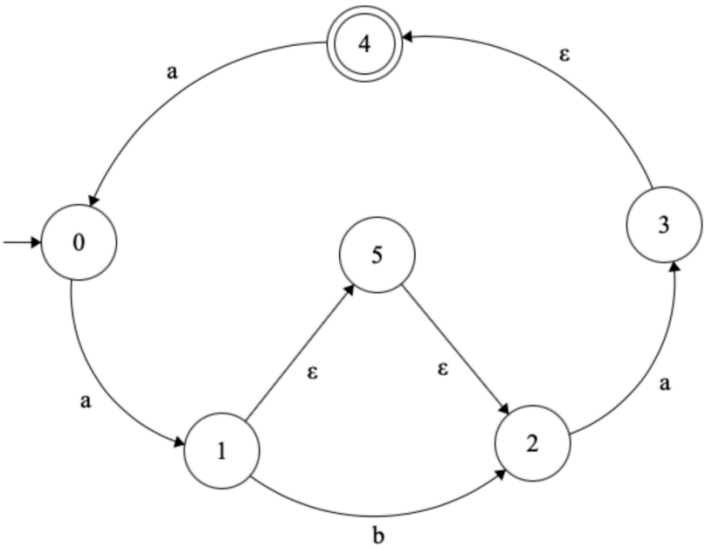
Q2.2. What regular expression does this NFA correspond to?



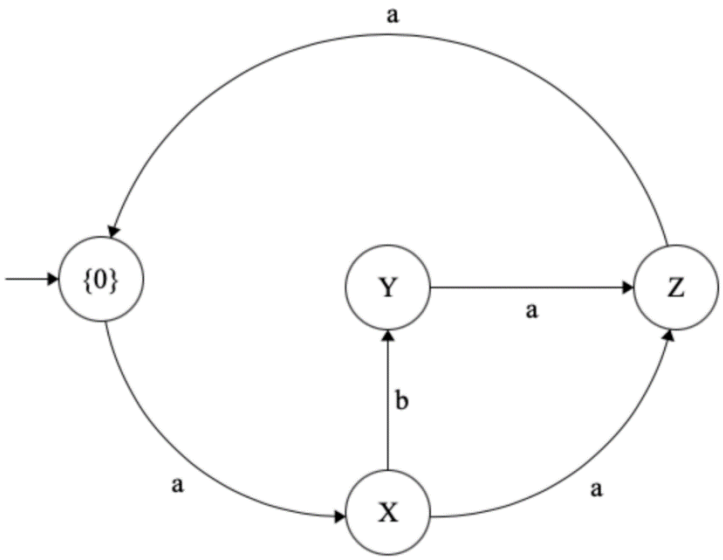

(a|bb)*m?

## Q3. NFA to DFA

Consider the following NFA:




Using subset construction, the above NFA can be converted to the DFA.




Q3.1. In this DFA, which states from the original NFA make up the state X? 1, 2, 5

Q3.2. In this DFA, which states from the original NFA make up the state Y? 2

Q3.3. In this DFA, which states from the original NFA make up the state Z? 3, 4

Q3.4. In this DFA, which states are final? Z

## Q4. Context-Free Grammars

Q4.1. Write a CFG over the alphabet `Σ = {0, 1}` that recognizes strings that start with a `1`, end with a `0`, and have any number of `0`s or `1`s in between.

```
S -> 1T0
T -> 1T | 0T | ε
```

Q4.2. Consider the following CFG

```
S -> S + T | T
T -> P * T | P
P -> 1 | 2 | 3 | (S)
```

Notice that this grammar is left recursive, write down every rule (with or without any changes) so that it is right recursive.

```
S -> T + S | T
T -> P * T | P
P -> 1 | 2 | 3 | (S)
```

Q4.3. Find the FIRST sets for each non-terminal in the following grammar:

```
S -> TU | Ua
T -> da | Ub
U -> g | ε
```

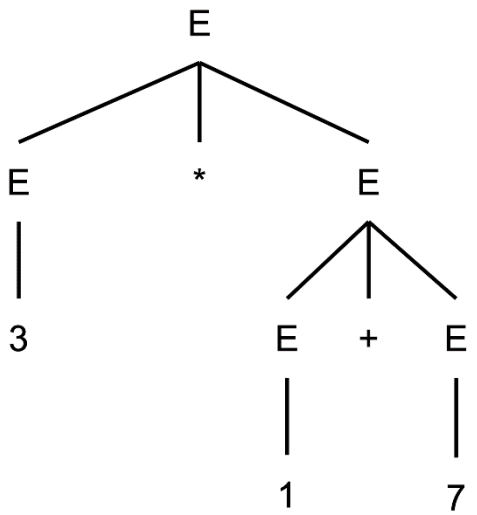
FIRST(S) = `{d, g, a, b}`

FIRST(T) = `{d, g, b}`

FIRST(U) = `{g, ε}`

## Q5. Parsing

Q5.1.




Write an `exp` value to describe the corresponding AST of the above parse tree.

```
type exp =
    | Num of int
    | Plus of exp * exp
    | Mul of exp * exp
```

Mul (Num 3, Plus (Num 1, Num 7))

Q5.2. You will implement a recursive descent parser for the following CFG:

```
S -> TU | a
T -> da | Ub
U -> g
```

You should use these functions and definitions:

```
type token =
    | Tok_a
    | Tok_b
    | Tok_d
    | Tok_g

(* Note that these are imperative implementations.
   You may assume that `tok_list` has been filled by a lexer. *)

let tok_list = ref []

let match_tok x =
    match !tok_list with
    | h :: t when h = x -> tok_list := t
    | _ -> raise (ParseError "bad match")

let lookahead () =
    match !tok_list with
    | [] -> None
    | h :: t -> Some h
```

Your functions should return the unit value `()` if they successfully parse, otherwise they should `raise (ParseError "message")`. The contents of the message string do not matter.

Enter your code for the functions below:

```
let rec parse_S () =
      match lookahead () with
      | Some Tok_a -> match_tok Tok_a
      | _ -> parse_T (); parse_U ()

and parse_T () =
      match lookahead () with
      | Some Tok_d -> match_tok Tok_d; match_tok Tok_a
      | _ -> parse_U (); match_tok Tok_b

and parse_U () =
      match lookahead () with
      | Some Tok_g -> match_tok Tok_g
      | _ -> raise (ParseError “bad input”)
```

## Q6. Operational Semantics

Consider the following grammar of expressions in a new programming language:

$$
\begin{array}{rcl}
e & ::= & x \\
& | & 1 \\
& | & 0 \\
& | & x \texttt{ = } e \texttt{ in } e \\
& | & \blacksquare\, e \\
& | & e \bullet e \\
& | & (e)
\end{array}
\qquad\qquad
\begin{array}{rcl}
v & ::= & 1 \\
& | & 0
\end{array}
$$

(The $x$ represents variable names. We use $e$ to mean *expressions* and $v$ to mean *values*. Parentheses are only used to prevent ambiguity, e.g., the term $(\blacksquare\, e) \bullet e$ is distinct from the term $\blacksquare\, (e \bullet e)$, but the parens have no other purpose.)

Answer the questions about this language with the following operational semantics. Note that each semantic rule is numbered for reference in your answers.

$$\frac{}{A;\ 1 \Rightarrow 1}\ (1) \qquad \frac{}{A;\ 0 \Rightarrow 0}\ (2) \qquad \frac{A(x) = v}{A;\ x \Rightarrow v}\ (3)$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad v_2 \text{ is 0 if } v_1 \text{ is 1, otherwise } v_2 \text{ is 1}}{A;\ \blacksquare\, e_1 \Rightarrow v_2}\ (4)$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad A;\ e_2 \Rightarrow v_2 \qquad v_3 \text{ is 0 if } v_1 = v_2, \text{ otherwise } v_3 \text{ is 1}}{A;\ e_1 \bullet e_2 \Rightarrow v_3}\ (5)$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad A, x : v_1;\ e_2 \Rightarrow v_2}{A;\ x \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow v_2}\ (6)$$

Q6.1. Rules 1 and 2 (but none of the others) are examples of axioms.

Q6.2. The semantics given are big-step.

Q6.3. The "A" used in operational semantics is called an environment.

Q6.4.

Using the given operational semantics, you must evaluate the following expression:

$$x \;=\; 0 \;\texttt{in}\; 1 \bullet (\blacksquare\, x)$$

First, fill in the holes in the proof tree shown below by writing the number of the rule for each hole. Then, tell us what the expression evaluates to (the $?$ in the bottom judgment).

**ONLY WRITE THE RULE'S NUMBER, DO NOT WRITE THE RULE'S TEXT.**

$$\dfrac{\dfrac{}{\;}\,(b) \qquad \dfrac{\dfrac{}{\;}\,(d) \qquad \dfrac{\dfrac{}{\;}\,(f)}{\;}\,(e)}{A, x : 0;\; 1 \bullet (\blacksquare\, x) \Rightarrow\; ?}\,(c)}{A;\; x \;=\; 0 \;\texttt{in}\; 1 \bullet (\blacksquare\, x) \Rightarrow\; ?\;(g)}\,(a)$$

**Hints**

1. Each rule is used exactly once.
2. Proof trees are filled from the bottom up and from left to right.
3. The result in part (g) should be a *value*. There are only two possible values in this language!

(a) 6
(b) 2
(c) 5
(d) 1
(e) 4
(f) 3
(g) (Result) 0

## Q7. Lambda Calculus

In your answers for this section, you may write the lambda symbol as λ, \, or L, but please be consistent!

Q7.1. Reduce the expression as far as possible (Show your work for partial credits):

```
(λx. λx. y x) (λx. y x) x
= (λx. y x) x
= y x
```

Q7.2. Perform an α-conversion to the following expression:

```
(λy. λz. y z) (λa. y) x
= (λw. λz. w z) (λa. y) x
```

Then, apply as many β-reductions as possible to it without performing any other α-conversions.

```
= (λz. (λa. y) z) x
= (λa. y) x
= y
```

Q7.3. Given the following definitions:

```
true  = λx. λy. x
false = λx. λy. y
and   = λx. λy. x y false
```

Prove that and true false is equivalent to false. (Show all steps involving β-reduction and substitution/replacement for full credit.)

```
and true false
= (λx. λy. x y false) true false
= (λy. true y false) false
= true false false
= (λx. λy. x) false false
= (λy. false) false
= false
```

## Q8. Rust

Q8.1.

```
let a = String::from("cmsc330");
let b = a;
let c = &b;
let d = c;
```

Which variable is the owner of the string “cmsc330”? b

Q8.2. Fill in the 3 blanks such that the program, when run, outputs:

```
0
30
330
C330
SC330
MSC330
```

Consider the following Rust code:

```
fn foo(s: String) {   // #1
  let mut i = s.len();
  while i > 0 {
    println!("{}", &s[i...]); //#2
    i = i - 1;
  }
}
fn main() {
  let s = String::from("CMSC330");
  foo(s); //#3
}
```

**OR any other equivalent solution!**