CMSC330 - Organization of Programming Languages
Spring 2025- Exam 2Solutions

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

- Please write legibly. **If we cannot read your answer you will not receive credit.**
- You may use anything on the accompanying reference sheet anywhere on this exam
- Please remove the reference sheet from the exam
- 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.	10
P2.	10
P3.	10
P4.	5
P5.	20
P6.	5
P7.	5
P8.	15
P9.	20
Total	100

Problem 1: Concepts

[Total 10 pts]

	true	false
A lexer checks an input string for grammatical correctness	T	**F**
{x:{a:Bool, c:Int, b:Int} y:{d:Int}} <: {x:{a:Bool b:Int} }	**T**	F
A type-safe language like OCaml will not compile a meaningless (i.e. ill-defined) program	**T**	F
Operational Semantic and Type Checking proofs aim to prove the same thing	T	**F**
For any regular expression, there can be multiple corresponding DFAs	**T**	F
A type system can be sound but not complete	**T**	F
If B and C are subtypes of A, then B must also be a subtype of C.	T	**F**
All NFAs are DFAs	T	**F**
A string can fail at the lexer, but pass the parser.	T	**F**
An accept function that works on any NFA will also work on any DFA.	**T**	F
{x:{a:Bool, c:Int, b:Int} y:{d:Int}} <: {x:{a:Bool b:Int} }	**T**	F
A lexer checks an input string for grammatical correctness	T	**F**
A type-safe language like OCaml will not compile a meaningless (i.e. ill-defined) program	**T**	F
Operational Semantic and Type Checking proofs aim to prove the same thing	T	**F**
For any regular expression, there can be multiple corresponding DFAs	**T**	F
A type system can be sound but not complete	**T**	F
If B and C are subtypes of A, then B must also be a subtype of C.	T	**F**
All NFAs are DFAs	T	**F**
A string can fail at the lexer, but pass the parser.	T	**F**
An accept function that works on any NFA will also work on any DFA.	**T**	F
{x:{a:Bool, c:Int, b:Int} y:{d:Int}} <: {x:{a:Bool b:Int} }	**T**	F
A lexer checks an input string for grammatical correctness	T	**F**
A type-safe language like OCaml will not compile a meaningless (i.e. ill-defined) program	**T**	F
Operational Semantic and Type Checking proofs aim to prove the same thing	T	**F**

For any regular expression, there can be multiple corresponding DFAs (T) (F)

A type system can be sound but not complete (T) (F)

If B and C are subtypes of A, then B must also be a subtype of C. (T) (F)

All NFAs are DFAs (T) (F)

A string can fail at the lexer, but pass the parser. (T) (F)

An accept function that works on any NFA will also work on any DFA. (T) (F)

Operational Semantic and Type Checking proofs aim to prove the same thing (T) (F)

A lexer checks an input string for grammatical correctness (T) (F)

A type-safe language like OCaml will not compile a meaningless (i.e. ill-defined) program (T) (F)

{x:{a:Bool, c:Int, b:Int} y:{d:Int}} <: {x:{a:Bool b:Int} } (T) (F)

For any regular expression, there can be multiple corresponding DFAs (T) (F)

A type system can be sound but not complete (T) (F)

If B and C are subtypes of A, then B must also be a subtype of C. (T) (F)

All NFAs are DFAs (T) (F)

A string can fail at the lexer, but pass the parser. (T) (F)

An accept function that works on any NFA will also work on any DFA. (T) (F)

Problem 2: CFG Derivation [Total 10 pts]

$$
\begin{aligned}
S &-> SvS|SrS|T \\
T &-> cT|Td|D \\
D &-> d|fSf
\end{aligned}
$$

(a) Derive `dvddrd` using a **leftmost** derivation (do not draw a tree). If it is not possible, derive `fdvdf` using a **rightmost** derivation [8 pts]

$$
\begin{aligned}
S &-> SvS \\
&-> TvS \\
&-> DvS \\
&-> dvS \\
&-> dvSrS \\
&-> dvTrS \\
&-> dvTdrS \\
&-> dvDdrS \\
&-> dvddrS \\
&-> dvddrT \\
&-> dvddrD \\
&-> dvddrd
\end{aligned}
$$

(b) Is this grammar ambiguous? [2 pts]

(Y) Yes (N) No

$$
\begin{aligned}
S &-> ShS|SkS|T \\
T &-> pT|mT|D \\
D &-> m|fSf
\end{aligned}
$$

(c) Derive `mhmmkm` using a **leftmost** derivation (do not draw a tree). If it is not possible, derive `fmkpmf` using a **rightmost** derivation [8 pts]

$$
\begin{aligned}
S &-> ShS \\
&-> ThS \\
&-> DhS \\
&-> mhS \\
&-> mhSkS \\
&-> mhTkS \\
&-> mhTmkS \\
&-> mhDmkS \\
&-> mhmmkS \\
&-> mhmmkT \\
&-> mhmmkD \\
&-> mhmmkm
\end{aligned}
$$

(d) Is this grammar ambiguous? [2 pts]

(Y) Yes (N) No

$$\begin{aligned} S &-> SxS|SzS|T \\ T &-> wT|yT|D \\ D &-> n|fSf \end{aligned}$$

(e) Derive `nxnnzn` using a **leftmost** derivation (do not draw a tree). If it is not possible, derive `fnxwnf` using a **rightmost** derivation [8 pts]

$$\begin{aligned} S &-> T \\ &-> D \\ &-> fSf \\ &-> fSxSf \\ &-> fSxTf \\ &-> fSxwTf \\ &-> fSxwDf \\ &-> fSxwnf \\ &-> fTxwnf \\ &-> fDxwnf \\ &-> fnxwnf \end{aligned}$$

(f) Is this grammar ambiguous? [2 pts]

(Y) Yes (N) No

$$\begin{aligned} S &-> SbS|SjS|T \\ T &-> mT|oD|D \\ D &-> o|fSf \end{aligned}$$

(g) Derive `oboojo` using a **leftmost** derivation (do not draw a tree). If it is not possible, derive `fojmof` using a **rightmost** derivation [8 pts]

$$\begin{aligned} S &-> SbS \\ &-> TbS \\ &-> DbS \\ &-> obS \\ &-> obSjS \\ &-> obTjS \\ &-> oboDjS \\ &-> oboojS \\ &-> oboojT \\ &-> oboojD \\ &-> oboojo \end{aligned}$$

(h) Is this grammar ambiguous? [2 pts]

(Y) Yes (N) No

Problem 3: Lexing, Parsing, Interpreting

[Total 10 pts]

Given the following CFG, and assuming the **Ocaml** type system and semantics, at what stage of language processing would each expression **fail**? Mark **'Valid'** if the expression would be accepted by the grammar and evaluate successfully. Assume the only symbols allowed are those found in the grammar.

$$E \rightarrow \text{if } E \text{ then } E \text{ else } E \mid R$$
$$R \rightarrow R + R \mid R \,\&\&\, R \mid M$$
$$M \rightarrow M > M \mid M < M \mid S$$
$$S \rightarrow 1 \mid 2 \mid 3 \mid true \mid false \mid (E)$$

	Lexer	Parser	Evaluator	Valid
`if true then false else 7`	L	P	E	V
`if 3 then true else true && false`	L	P	E	V
`1 + 2 + 3 1`	L	P	E	V
`(&&) true false`	L	P	E	V
`if 3 > 1 then true else false`	L	P	E	V
`1 > 2 < 4`	L	P	E	V
`if true = false then true else 10`	L	P	E	V
`let x = 1 in x + 2`	L	P	E	V
`(true < 1) > 2`	L	P	E	V
`1 && 3 + 1`	L	P	E	V
`1 + 2 + 3 1`	L	P	E	V
`if 3 then true else true && false`	L	P	E	V
`(&&) true false`	L	P	E	V
`1 > 2 < 4`	L	P	E	V
`if true then false else 7`	L	P	E	V
`(true < 1) > 2`	L	P	E	V
`if 3 > 1 then true else false`	L	P	E	V
`1 && 3 + 1`	L	P	E	V
`let x = 1 in x + 2`	L	P	E	V
`if true = false then true else 10`	L	P	E	V
`if 3 > 1 then true else false`	L	P	E	V
`1 > 2 < 4`	L	P	E	V
`let x = 1 in x + 2`	L	P	E	V
`1 && 3 + 1`	L	P	E	V
`if true = false then true else 10`	L	P	E	V
`1 + 2 + 3 1`	L	P	E	V
`if 3 then true else true && false`	L	P	E	V
`(&&) true false`	L	P	E	V

Problem 4: CFG Creation

[Total 5 pts]

Write a CFG of the language that generates all even-length strings over the alphabet a, b.
Letters can be in any order, and the CFG can be ambiguous.
Example strings created by the CFG:
ab
ϵ
$abbb$
$aaaa$

$$S -> aaS \mid abS \mid baS \mid bbS \mid \epsilon$$

Problem 5: CFG Creation

[Total 5 pts]

Write a CFG of the language that generates all even-length strings over the alphabet c, d.
Letters can be in any order, and the CFG can be ambiguous.
Example strings created by the CFG:
cd
ϵ
$cddd$
$cccc$

$$S -> ccS \mid cdS \mid dcS \mid ddS \mid \epsilon$$

Problem 6: CFG Creation

[Total 5 pts]

Write a CFG of the language that generates all even-length strings over the alphabet e, f.
Letters can be in any order, and the CFG can be ambiguous.
Example strings created by the CFG:
ef
ϵ
$efff$
$eeee$

$$S -> eeS \mid efS \mid feS \mid ffS \mid \epsilon$$

Problem 7: CFG Creation

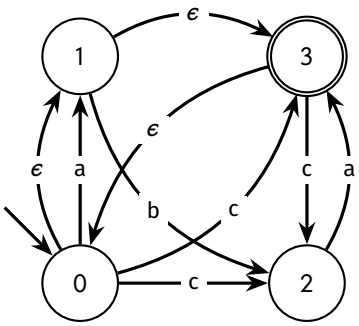
[Total 5 pts]

Write a CFG of the language that generates all even-length strings over the alphabet j, k.
Letters can be in any order, and the CFG can be ambiguous.
Example strings created by the CFG:
jk
ϵ
$jkkk$
$jjjj$

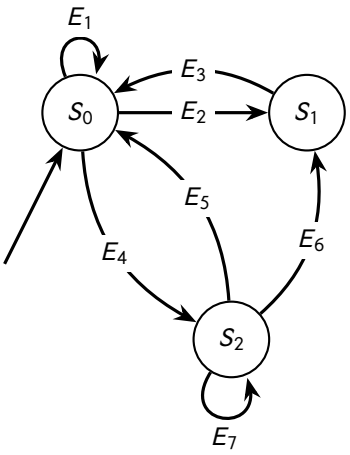
$$S -> jjS \mid jkS \mid kjS \mid kkS \mid \epsilon$$

Problem 8: NFA to DFA

[Total 20 pts]



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

(a) Which states are the final (accepting) states? Select all that apply [2 pts]

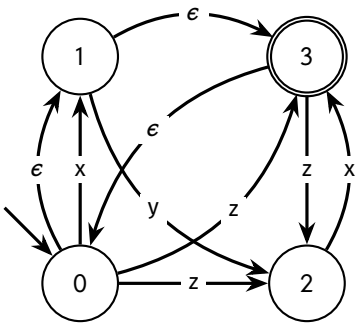
(A) State S_0 (B) State S_1 (C) State S_2

(b) What is the result of calling e-closure(1) on this NFA? [4 pts]

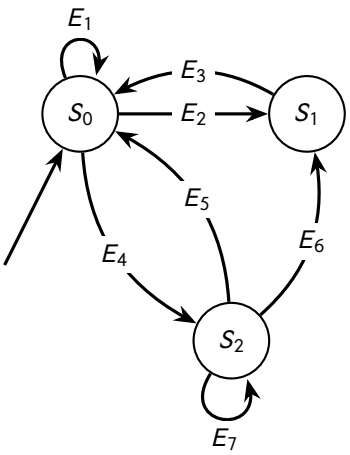
1,3,0

(c) What is the result of calling move(0,c) on this NFA? [4 pts]

2,3



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

(d) Which states are the final (accepting) states? Select all that apply [2 pts]

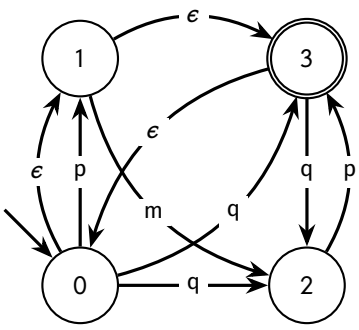
(A) State S_0 (B) State S_1 (C) State S_2

(e) What is the result of calling e-closure(1) on this NFA? [4 pts]

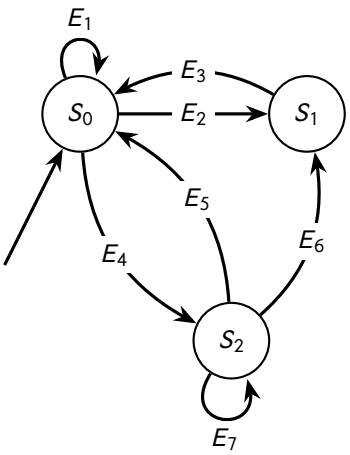
1,3,0

(f) What is the result of calling move(0,z) on this NFA? [4 pts]

2,3



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

(g) Which states are the final (accepting) states? Select all that apply [2 pts]

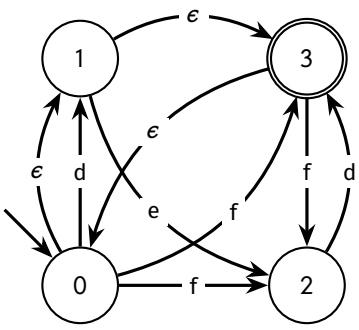
(A) State S_0 (B) State S_1 (C) State S_2

(h) What is the result of calling e-closure(1) on this NFA? [4 pts]

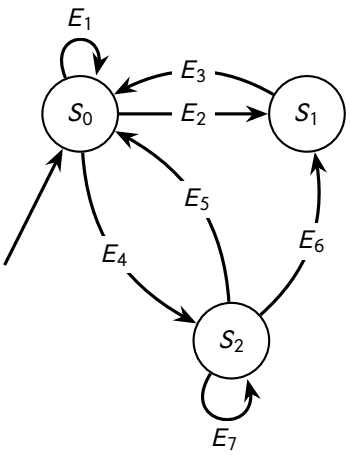
1,3,0

(i) What is the result of calling move(0,q) on this NFA? [4 pts]

2,3



Convert the above NFA to the below DFA. You **MUST** show your work to get any credit. (You will need to remove the Garbage state)



Scratch Space:

(j) Which states are the final (accepting) states? Select all that apply [2 pts]

(A) State S_0 (B) State S_1 (C) State S_2

(k) What is the result of calling e-closure(1) on this NFA? [4 pts]

1,3,0

(l) What is the result of calling move(0,f) on this NFA? [4 pts]

2,3

Problem 9: CFG Creation Pt. 2

[Total 5 pts]

Which Context Free Grammar(s) are equivalent to the regex: `[ab]+c*`
(? is a part of the regex)
Select all that apply.

(A) $S->TD$, $T->Ta|Tb|a|b$, $D->cD|\epsilon$

(B) $S->aS|bS|Sc|a|b$

(C) $S->Ta|Tb|Sc$, $T->aT|bT|\epsilon$

(D) $S->aS|Sb|Sc|\epsilon$

Which Context Free Grammar(s) are equivalent to the regex: `[ab]+c*`
(? is a part of the regex)
Select all that apply.

(A) $S->TD$, $T->Ta|Tb|a|b$, $D->cD|\epsilon$

(B) $S->aS|bS|Sc|a|b$

(C) $S->Ta|Tb|Sc$, $T->aT|bT|\epsilon$

(D) $S->aS|Sb|Sc|\epsilon$

Which Context Free Grammar(s) are equivalent to the regex: `[ab]+c*`
(? is a part of the regex)
Select all that apply.

(A) $S->TD$, $T->Ta|Tb|a|b$, $D->cD|\epsilon$

(B) $S->aS|bS|Sc|a|b$

(C) $S->Ta|Tb|Sc$, $T->aT|bT|\epsilon$

(D) $S->aS|Sb|Sc|\epsilon$

Which Context Free Grammar(s) are equivalent to the regex: `[ab]+c*`
(? is a part of the regex)
Select all that apply.

(A) $S->TD$, $T->Ta|Tb|a|b$, $D->cD|\epsilon$

(B) $S->aS|bS|Sc|a|b$

(C) $S->Ta|Tb|Sc$, $T->aT|bT|\epsilon$

(D) $S->aS|Sb|Sc|\epsilon$

Problem 10: Operational Semantics

[Total 5 pts]

Consider the following 2 Languages.

Language One

$$\frac{}{A; ABC \Rightarrow true} \quad \frac{}{A; ZED \Rightarrow false}$$

$$\frac{}{A; YEA \Rightarrow 1} \quad \frac{}{A; XYZ \Rightarrow 2}$$

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

$$\frac{A; e_1 \Rightarrow v_1 \quad A, x : v1; e_2 \Rightarrow v_2}{A; \text{erm } x \text{ is } e_1 \text{ but } e_2 \Rightarrow v_2}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 <> v_2}{A; \text{d } e_1 \text{ f } e_2 \text{ g} \Rightarrow v_3}$$

Language Two

$$\frac{}{A; LOL \Rightarrow true} \quad \frac{}{A; BRB \Rightarrow false}$$

$$\frac{}{A; FKN \Rightarrow 1} \quad \frac{}{A; PMO \Rightarrow 2}$$

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

$$\frac{A; e_1 \Rightarrow v_1 \quad A, x : v1; e_2 \Rightarrow v_2}{A; e_2 \text{ cuz } x \text{ be } e_1 \Rightarrow v_2}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 <> v_2}{A; \text{so } e_1 \text{ big } e_2 \text{ huge} \Rightarrow v_3}$$

Using OCaml as the metalanguage, convert the following **Language One sentence to a Language Two sentence**, including its value

erm they is ABC but d they f ZED g $\Rightarrow true$

so they big BRB huge cuz they be LOL

Using OCaml as the metalanguage, convert the following **Language One sentence to a Language Two sentence**, including its value

erm ts is XYZ but d YEA f ts g $\Rightarrow true$

so FKN big it huge cuz it be PMO

Using OCaml as the metalanguage, convert the following **Language Two sentence to a Language One sentence**, including its value

so they big BRB huge cuz they be LOL $\Rightarrow true$

erm they is ABC but d they f BRB g

Using OCaml as the metalanguage, convert the following **Language Two sentence to a Language One sentence**, including its value

so FKN big ts huge cuz ts be PMO $\Rightarrow true$

erm ts is XYZ but d YEA f ts g

Problem 11: Typing Proofs

[Total 15 pts]

Consider the following typing rules of OCaml:

$$\frac{}{G \vdash true : bool} \quad \frac{}{G \vdash false : bool} \quad \frac{}{G \vdash n : int} \quad \frac{G(x) = t}{G \vdash x : t}$$

$$\frac{G \vdash e_1 : t_1 \quad G, x : t_1 \vdash e_2 : t_2}{G \vdash \text{let } x \ = \ e_1 \text{ in } e_2 : t_2} \qquad \frac{G \vdash e_1 : t_1 \quad G \vdash e_2 : t_2}{G \vdash e_1 \ <> \ e_2 : bool}$$

Using OCaml as the metalanguage, prove the following sentence type checks.
IMPORTANT: you must fill in the blanks to receive credit.

let x = true in x <> false

$$\frac{\dfrac{\boxed{2} \qquad \boxed{3}}{\boxed{4}} \qquad \boxed{5}}{\boxed{6}}$$

$G \vdash$ let x = true in x <> false : $\boxed{1}$

Scratch Space:

Blank 1: $bool$

Blank 2: $G \vdash true : bool$

Blank 3: $G, x : bool \vdash x <> false : bool$

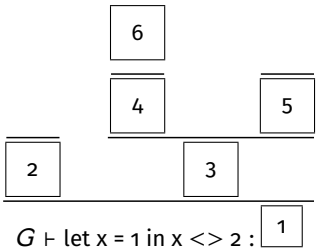
Blank 4: $G, x : bool \vdash x : bool$

Blank 5: $G, x : bool \vdash false : bool$

Blank 6: $G, x : bool(x) = bool$

Using OCaml as the metalanguage, prove the following sentence type checks.
IMPORTANT: you must fill in the blanks to receive credit.

let x = 1 in x <> 2



Scratch Space:

Blank 1: $bool$

Blank 2: $G \vdash 1 : int$

Blank 3: $G, x : int \vdash x <> 2 : bool$

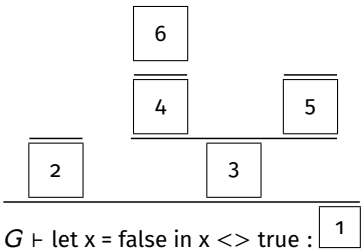
Blank 4: $G, x : int \vdash x : int$

Blank 5: $G, x : int \vdash 2 : int$

Blank 6: $G, x : int(x) = int$

Using OCaml as the metalanguage, prove the following sentence type checks.
IMPORTANT: you must fill in the blanks to receive credit.

let x = false in x <> true



Scratch Space:

Blank 1: $bool$

Blank 2: $G \vdash false : bool$

Blank 3: $G, x : bool \vdash x <> true : bool$

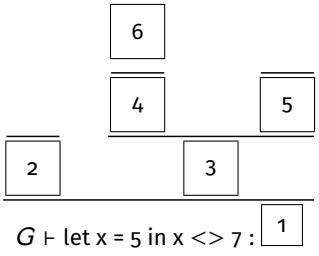
Blank 4: $G, x : bool \vdash x : bool$

Blank 5: $G, x : bool \vdash true : bool$

Blank 6: $G, x : bool(x) = bool$

Using OCaml as the metalanguage, prove the following sentence type checks.
IMPORTANT: you must fill in the blanks to receive credit.

let x = 5 in x <> 7



Scratch Space:

Blank 1: $bool$

Blank 2: $G \vdash 5 : int$

Blank 3: $G, x : int \vdash x <> 7 : bool$

Blank 4: $G, x : int \vdash x : int$

Blank 5: $G, x : int \vdash 7 : int$

Blank 6: $G, x : int(x) = int$

Problem 12: Coding

[Total 20 pts]

Restrictions: You are not allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in StdLib (string_of_int and the ^ operator are particularly useful, look at the cheat sheet).

For this question, given a **modified** subset of smallc as a AST, return a string of the equivalent OCaml Code (Recall your evaluator from Project 4).

```
type expr = Int of int | ID of string | Add of expr * expr
          | Equal of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

Given the AST:

```
let ast = Seq(Assign("x",Int(4)),
              Assign("y",If(Equal(Int(5),Int(6)),
                            ID("x"),
                            Add(Int(6),Int(8)))))
```

Expected Output:

```
to_ocaml ast = "let x = 4 in let y = if 5 = 6 then x else 6 + 8"
```

We do NOT care about spaces in the output, so the following is acceptable:

```
to_ocaml ast = "letx=4inlety=if5=6thenxelse6+8"
```

(C Program that generated AST):

```
int main(){
  x = 4;
  y = if (5 = 6){
        x
      }else{
        6 + 8
      };
}
```

You may use the following Operational Semantics if needed:

$$\frac{}{n \Rightarrow n} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow v_1 + v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 == e_2 \Rightarrow v_1 = v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if } (e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

Write your code on the next page

Here is the `ast types` again:

```
type expr = Int of int | ID of string | Add of expr * expr
          | Equal of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

The rules:

$$\frac{}{n \Rightarrow n} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow v_1 + v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 == e_2 \Rightarrow v_1 = v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if}(e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

```
(* stmt -> string *)
let rec to_ocaml ast =

        let rec string_of_expr expr = match expr with
             Int(i) -> string_of_int i
            |ID(x) -> x
            |Add(e1,e2) -> string_of_expr e1 ^ " + " ^ string_of_expr e2
            |Equal(e1,e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
            |If(e1,e2,e3) -> "if " ^ string_of_expr e1 ^ " then " ^ string_of_expr e2 ^
                             " else " ^ string_of_expr e3
        in
        match ast with
        Seq(s1,s2) -> (to_ocaml s1) ^ " in " ^ (to_ocaml s2)
        |Assign(var,e) -> "let " ^ var ^ " = " ^ (string_of_expr e)
```

Restrictions: You are not allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in StdLib (string_of_int and the ^ operator are particularly useful, look at the cheat sheet).

For this question, given a **modified** subset of smallc as a AST, return a string of the equivalent OCaml Code (Recall your evaluator from Project 4).

```
type expr = Int of int | ID of string | Add of expr * expr
          | Equal of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

Given the AST:

```
let ast = Seq(Assign("x",Int(4)),
              Assign("y",If(Equal(Int(5),Int(6)),
                            ID("x"),
                            Add(Int(6),Int(8)))))
```

Expected Output:

```
to_ocaml ast = "let x = 4 in let y = if 5 = 6 then x else 6 + 8"
```

We do NOT care about spaces in the output, so the following is acceptable:

```
to_ocaml ast = "letx=4inlety=if5=6thenxelse6+8"
```

(C Program that generated AST):

```
int main(){
  x = 4;
  y = if (5 = 6){
        x
      }else{
        6 + 8
      };
}
```

You may use the following Operational Semantics if needed:

$$\frac{}{n \Rightarrow n} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow v_1 + v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 == e_2 \Rightarrow v_1 = v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if } (e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

Write your code on the next page

Here is the `ast types` again:

```
type expr = Int of int | ID of string | Add of expr * expr
          | Equal of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

The rules:

$$\frac{}{n \Rightarrow n} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 + e_2 \Rightarrow v_1 + v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 == e_2 \Rightarrow v_1 = v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if}(e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

```
(* stmt -> string *)
let rec to_ocaml ast =

        let rec string_of_expr expr = match expr with
             Int(i) -> string_of_int i
            |ID(x) -> x
            |Add(e1,e2) -> string_of_expr e1 ^ " + " ^ string_of_expr e2
            |Equal(e1,e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
            |If(e1,e2,e3) -> "if " ^ string_of_expr e1 ^ " then " ^ string_of_expr e2 ^
                             " else " ^ string_of_expr e3
        in
        match ast with
        Seq(s1,s2) -> (to_ocaml s1) ^ " in " ^ (to_ocaml s2)
        |Assign(var,e) -> "let " ^ var ^ " = " ^ (string_of_expr e)
```

Restrictions: You are not allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in StdLib (string_of_int and the ^ operator are particularly useful, look at the cheat sheet).

For this question, given a **modified** subset of smallc as a AST, return a string of the equivalent OCaml Code (Recall your evaluator from Project 4).

```
type expr = Bool of bool | ID of string | Or of expr * expr
          | NotEqual of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

Given the AST:

```
let ast = Seq(Assign("x",Bool(true)),
              Assign("y",If(NotEqual(Bool(false),Bool(true)),
                            ID("x"),
                            Or(Bool(true),Bool(false)))))
```

Expected Output:

```
to_ocaml ast = "let x = true in let y = if false <> true then x else true || false"
```

(C Program that generated AST):

```
int main(){
  x = true;
  y = if (false != true){
        x
      } else{
        true || false
      };
}
```

You may use the following Operational Semantics if needed:

$$\frac{}{b \Rightarrow b} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \&\& e_2 \Rightarrow v_1 \&\& v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 != e_2 \Rightarrow v_1 <> v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if } (e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

Write your code on the next page

Here is the `ast types` again:

```
type expr = Bool of bool | ID of string | Or of expr * expr
          | NotEqual of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

The rules:

$$\frac{}{b \Rightarrow b} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \&\& e_2 \Rightarrow v_1 \&\& v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 != e_2 \Rightarrow v_1 <> v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if } (e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

```
(* stmt -> string *)
let rec to_ocaml ast =

        let rec string_of_expr expr = match expr with
             Bool(b) -> string_of_bool b
            |ID(x) -> x
            |And(e1,e2) -> string_of_expr e1 ^ " && " ^ string_of_expr e2
            |NotEqual(e1,e2) -> string_of_expr e1 ^ " <> " ^ string_of_expr e2
            |If(e1,e2,e3) -> "if " ^ string_of_expr e1 ^ " then " ^ string_of_expr e2 ^
                             " else " ^ string_of_expr e3
        in
        match ast with
        Seq(s1,s2) -> (to_ocaml s1) ^ " in " ^ (to_ocaml s2)
        |Assign(var,e) -> "let " ^ var ^ " = " ^ (string_of_expr e)
```

Restrictions: You are not allowed to use imperative OCaml; you can define recursive helper/helper functions below the function signatures we provided. You are not allowed to use any List module functions except the ones already given to you on the cheat sheet, otherwise you may use anything in StdLib (string_of_int and the ^ operator are particularly useful, look at the cheat sheet).

For this question, given a **modified** subset of smallc as a AST, return a string of the equivalent OCaml Code (Recall your evaluator from Project 4).

```
type expr = Bool of bool | ID of string | Or of expr * expr
          | NotEqual of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

Given the AST:

```
let ast = Seq(Assign("x",Bool(true)),
              Assign("y",If(NotEqual(Bool(false),Bool(true)),
                            ID("x"),
                            Or(Bool(true),Bool(false)))))
```

Expected Output:

```
to_ocaml ast = "let x = true in let y = if false <> true then x else true || false"
```

(C Program that generated AST):

```
int main(){
  x = true;
  y = if (false != true){
        x
      } else{
        true || false
      };
}
```

You may use the following Operational Semantics if needed:

$$\frac{}{b \Rightarrow b} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \&\& e_2 \Rightarrow v_1 \&\& v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 != e_2 \Rightarrow v_1 <> v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if } (e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

Write your code on the next page

Here is the `ast types` again:

```
type expr = Bool of bool | ID of string | Or of expr * expr
          | NotEqual of expr * expr | If of expr * expr * expr
 type stmt = Seq of stmt * stmt | Assign of string * expr
```

The rules:

$$\frac{}{b \Rightarrow b} \quad \frac{}{x \Rightarrow x} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \&\& e_2 \Rightarrow v_1 \&\& v_2} \quad \frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 != e_2 \Rightarrow v_1 <> v_2}$$

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\text{if } (e_1)\{e_2\}\text{else}\{e_3\} \Rightarrow \text{if } v_1 \text{ then } v_2 \text{ else } v_3} \quad \frac{e \Rightarrow v}{x = e; \Rightarrow \text{let } x = v} \quad \frac{e_1; \Rightarrow v_1 \quad e_2; \Rightarrow v_2}{e_1; e_2; \Rightarrow v_1 \text{ in } v_2}$$

```
(* stmt -> string *)
let rec to_ocaml ast =

        let rec string_of_expr expr = match expr with
             Bool(b) -> string_of_bool b
            |ID(x) -> x
            |And(e1,e2) -> string_of_expr e1 ^ " && " ^ string_of_expr e2
            |NotEqual(e1,e2) -> string_of_expr e1 ^ " <> " ^ string_of_expr e2
            |If(e1,e2,e3) -> "if " ^ string_of_expr e1 ^ " then " ^ string_of_expr e2 ^
                             " else " ^ string_of_expr e3
        in
        match ast with
        Seq(s1,s2) -> (to_ocaml s1) ^ " in " ^ (to_ocaml s2)
        |Assign(var,e) -> "let " ^ var ^ " = " ^ (string_of_expr e)
```

Cheat Sheet

OCaml

```
(* Map and Fold *)
(* ('a -> 'b) -> 'a list -> 'b list *)
let rec map f l = match l with
   [] -> []
  |x::xs -> (f x)::(map f xs)

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

(* ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b *)
let rec fold_right f l a = match l with
   [] -> a
  |x::xs -> f x (fold_right f xs a)
```

Structure of Regex

$$
\begin{array}{rcl}
R & \rightarrow & \emptyset \\
 & | & \sigma \\
 & | & \epsilon \\
 & | & RR \\
 & | & R|R \\
 & | & R^* \\
\end{array}
$$

```
(* OCaml Function Types *)
:: -: 'a -> 'a list -> 'a list

@ -: 'a list -> 'a list -> 'a list

+, -, *, /  -: int -> int -> int
+., -., *., /. -: float -> float -> float

&&, || -: bool -> bool -> bool
not -: bool -> bool

^ -: string -> string -> string

=>,>,=,<,<=, <> :- 'a -> 'a -> bool

string_of_int -: int -> string
string_of_bool -: bool -> string
```

NFA to DFA Algorithm (Subset Construction Algorithm)

NFA (input): $(\Sigma, Q, q_0, F_n, \delta)$, DFA (output): $(\Sigma, R, r_0, F_d, \delta_n)$

$R \leftarrow \{\}$
$r_0 \leftarrow \epsilon - \text{closure}(\sigma, q_0)$
while $\exists$ an unmarked state $r \in R$ **do**
  mark r
  for all $a \in \Sigma$ **do**
    $E \leftarrow \text{move}(\sigma, r, a)$
    $e \leftarrow \epsilon - \text{closure}(\sigma, E)$
    if $e \notin R$ **then**
      $R \leftarrow R \cup \{e\}$
    end if
    $\sigma_n \leftarrow \sigma_n \cup \{r, a, e\}$
  end for
end while
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$

Regex

*	zero or more repetitions of the preceding character or group
+	one or more repetitions of the preceding character or group
?	zero or one repetitions of the preceding character or group
.	any character
$r_1|r_2$	r_1 or r_2 (eg. a|b means 'a' or 'b')
[abc]	match any character in abc
[^r_1]	anything except r_1 (eg. [^abc] is anything but an 'a', 'b', or 'c')
[r_1-r_2]	range specification (eg. [a-z] means any letter in the ASCII range of a-z)
{n}	exactly n repetitions of the preceding character or group
{n,}	at least n repetitions of the preceding character or group
{m,n}	at least m and at most n repetitions of the preceding character or group
^	start of string
$	end of string
(r_1)	capture the pattern r_1 and store it somewhere (match group in Python)
\d	any digit, same as [0-9]
\s	any space character like \n, \t, \r, \f, or space