CMSC330 - Organization of Programming Languages
Spring 2024 - Final Solutions

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
- The back of the reference sheet has some scratch space on it. If you use it, you must turn in your scratch work
- 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.	15
P4.	5
P5.	6
P6.	6
P7.	6
P8.	6
P9.	10
P10.	8
P11.	18
Total	100

Problem 1: Concepts

[Total 10 pts]

(a) True/False [6 pts]

	True	False
In Ocaml, an anonymous function cannot make a recursive call to itself	[✓ T]	F
In Ocaml, an anonymous function can make a recursive call to itself	T	[✓ F]
`(fun x -> x + 1)` is alpha-equivalent to `(fun y -> y + 1)`	[✓ T]	F
`(fun x -> x + 1)` is alpha-equivalent to `(fun y -> 1 + x)`	T	[✓ F]
All statements are expressions, but not all expressions are statements	[✓ T]	F
All expressions are statements	T	[✓ F]
A Turing machine can compute everything a Finite State Machine could compute	[✓ T]	F
A Finite State Machine can compute everything a Turing Machine could compute	T	[✓ F]
Security is more than just fixing bugs	[✓ T]	F
Security only consists of fixing bugs	T	[✓ F]
Soundness implies completeness	T	[✓ F]
Completeness implies soundness	T	[✓ F]

(b) Garbage Collection [2 pts]

Version A:

Which garbage collection algorithm can successfully clean up a cyclical linked list data structure? Select all that apply.

- A Reference Counting
- [✓ B] Mark and Sweep
- [✓ C] Stop and Copy
- D Any garbage collection algorithm can clean it.
- E None of the above

Version B:

Which garbage collection algorithm would struggle to clean up a cyclical linked list data structure? Select all that apply.

- [✓ A] Reference Counting
- B Mark and Sweep
- C Stop and Copy
- D None of the Above.

(c) Type Safe [2 pts]

Given the Following Grammar, Type Rules, and Operational Semantics, is Math-ew a type safe Language?

```
M -> sq M | A
A -> !A | T
T -> true | false | H
H -> 0 | 1 | 2 | ...
```

Note: $H = \mathbb{N}$

$$\frac{}{G \vdash n : int} \qquad \frac{}{A; n \Rightarrow n}$$

$$\frac{}{G \vdash b : bool} \qquad \frac{}{A; b \Rightarrow b}$$

$$\frac{G \vdash e : int \qquad sq = (int, int)}{G \vdash sq\ e : int} \qquad \frac{A; e \Rightarrow v_1 \qquad v_2\ is\ v_1 * v_1}{A; sq\ e \Rightarrow v_2}$$

$$\frac{G \vdash e : bool \qquad ! = (bool, bool)}{G \vdash !e : bool} \qquad \frac{G \vdash e \Rightarrow v_1 \qquad v_2 = !v_1}{A;\ !e \Rightarrow v_2}$$

[✓ Yes] No

Problem 2: Regex [Total 10 pts]

If you run `ls -lh` on the command line, you get back a list of files in the current directory. Suppose `ls -lh` returned:

```
drwxrwxrwx owner1 group1 folder1
-r-xrw-r-- clyffb a330ta emails.txt
-r-------- anwarm profs  Grades.csv
---------- owner3 none   passWORDS.bin
```

Write a regex that describes each part:

(a) Directory and Permissions [4 pts]
Each line starts with either **d** (for directory) or **-** (dash if it is not a directory). It is then followed by **read (r), write (w), and execute (x)** to denote the permissions of the 3 groups: the owner, group and others. The order will always be **rwx** replacing any letter with a **-** if that group does not have that permissions. For example: `drwxr-x--x` means that this is a directory for which the owner has all three permissions, the group can only read and execute and others can only execute.

A = (d|-)((r|-)(w|-)(x|-)){3}

(b) Owner and Group Name [2 pts]
The owner and group name begins with a **lowercase** character followed by **zero or more lowercase or numeric** characters.

B = [a-z][a-z0-9]*

(c) File Name [3 pts]
File names are **at least one** character long, and can be any **alphanumeric** character, along with special characters of **dashes (-)** and **underscores(_)**.

C = ([0-9a-zA-Z_]|-)+

(d) Full Line [1 pts]

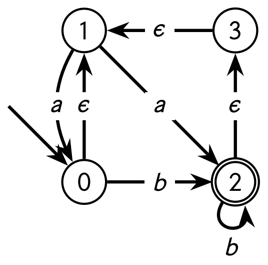
Each part is separated by **1 or more** whitespace characters between. Follow the syntax in the example lines above and use the above parts **A, B, C** to fill in the blanks to write a regex that parses the lines outputted by `ls -lh`. Write one item (either A, B, C or the appropriate regex) in each blank so that **as a whole the regex matches a full line**. (Ex: B a+ C, etc.)

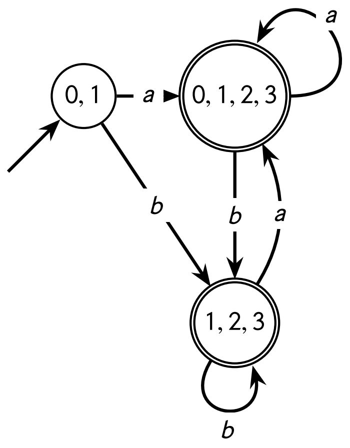
^ **A** **\s+** **B** **\s+** **B** **\s+** **C** $

Problem 3: FSM [Total 15 pts]

Convert the below NFA to a DFA. Draw a **box** around your final answer.

Version A:

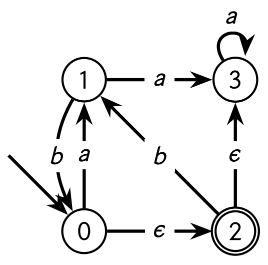


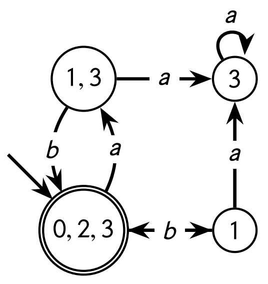


Write a CFG that describes strings accepted by the NFA above.

$S \longrightarrow aT|bT$
$T \longrightarrow aT|bT|\epsilon$

Version B:





Write a CFG that describes strings accepted by the NFA above.

$S \longrightarrow AB$
$A \longrightarrow abA|ab|\epsilon$
$B \longrightarrow bbB|bb|\epsilon$

Problem 4: Typing

[Total 5 pts]

Give the type of the following expressions. If there is a type error, put ”ERROR”

Version A:

```
(* Ocaml *)
fun x ->
    let (a,b) = x in
    fun y ->
    let a = (a+1, b > true) in
    (a::y)
```

```
// Rust
{
    let a = if false {
        true > false;
    };
    let b = true;
    (a, b)
}
```

int * bool ⟶ (int * bool) list ⟶ (int * bool) list	((), bool)

Version B:

```
(* Ocaml *)
fun x ->
    match x with (a, b) ->
    fun z ->
    let a = (a^"hello", b > false) in
    (a::z)
```

```
// Rust
{
    let a = if true {
        true > false;
    };
    let b = 0.54;
    (a, b)
}
```

string * bool ⟶ (string * bool) list ⟶ (string * bool) list	((), f64)

Problem 5: Evaluation

[Total 6 pts]

Evaluate the following expressions. It there is a compilation error, put ”ERROR”

Version A:

```
(* Ocaml *)
let rec f x = match x with
  [] -> 4
 |x::xs -> List.fold_left x (f xs) [1;2;3] in

f [(fun a b -> a * b)]
```

```
// Rust
fn f1(x: i32, y: i32) -> i32 {
    x + y
}

fn f2(x: i32, y: i32) -> i32 {
    x * y
}
...
{
    let mut x = vec![3, 2, 5];
    let mut a = true;
    for i in x.iter_mut() {
        if a {
            *i = f1(*i, *i);
            a = false;
        } else {
            *i = f2(*i, *i);
            a = true;
        }
    }
    x
};
```

24	[9, 0, 25] vector

Version B:

```
(* Ocaml *)
let rec f x = match x with
  [] -> 7
 |x::xs -> List.fold_left x (f xs) [4;2;1] in

f [(fun a b -> a - b)]
```

0

```
// Rust
fn f1(x: i32, y: i32) -> i32 {
    x * y
}

fn f2(x: i32, y: i32) -> i32 {
    x - y
}
...
{
    let mut x = vec![4, 2, 3];
    let mut a = true;
    for i in x.iter_mut() {
        if a {
            *i = f1(*i, *i);
            a = false;
        } else {
            *i = f2(*i, *i);
            a = true;
        }
    }
    x
};
```

[16, 0, 9] vector

Problem 6: Property Based Testing

[Total 6 pts]

Consider the following functions and type definitions:

```
type tree = Node of tree * int * tree | Leaf of int

(* this function is supposed to mirror a binary tree *)
(* it may or may not have a bug *)
let rec mirror tree = match tree with
    Leaf(x) -> Leaf(x)
   |Node(l,v,r) -> Node(mirror r,v, mirror l)

(* this function is supposed to count the number of nodes in a binary tree *)
(* it may or may not have a bug *)
let rec count tree = match tree with
    Leaf(x) -> 1
   |Node(l,v,r) -> count l + v
```

Below are descriptions of properties being tested and an attempted implementation of each property for the qcheck testing framework. For each property, indicate if the property is valid. If the property is valid, indicate if the property will catch the bugs in the above code **even if the function does not correctly represent the property**. If the property is invalid, put NA to catch bugs. Then indicate if the function provided correctly represents the property **not considering the bugs in the above code**.

Version A:

Property 1: Mirroring the tree should not result in the initial tree
Property 1 as a Function: `fun tree -> mirror tree <> tree`

Valid property: Y [✓ N]   Property would catch above bugs: Y N [✓ na]   Valid Property Function: [✓ Y] N

Property 2: Mirroring a tree should not change the number of nodes
Property 2 as a Function: `fun tree -> count (mirror tree) = count tree`

Valid property: [✓ Y] N   Property would catch above bugs: [✓ Y] N na   Valid Property Function: [✓ Y] N

Version B:

Property 1: Mirroring the tree should not result in the initial tree
Property 1 as a Function: `fun tree -> mirror tree = tree`

Valid property: Y [✓ N]   Property would catch above bugs: Y N [✓ na]   Valid Property Function: Y [✓ N]

Property 2: Mirroring a tree should not change the number of nodes
Property 2 as a Function: `fun tree -> count (mirror tree) <> count tree`

Valid property: [✓ Y] N   Property would catch above bugs: [✓ Y] N na   Valid Property Function: Y [✓ N]

Problem 7: Interpreters

[Total 6 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 properly. Assume the only symbols allowed are those found in the grammar. Choose only one choice for each expression.

Grammar:

$$\begin{aligned} M &\rightarrow M\ E\ +\ |\ M\ E\ -\ |\ E \\ E &\rightarrow O\ E\ /\ |\ O\ E\ *\ |\ O \\ O &\rightarrow W O >\ |\ W O <\ |\ W \\ W &\rightarrow n\ |\ b \end{aligned}$$

Note: $n \in \mathbb{Z}, b \in \{true, false\}$

In this case, + and − are prefix operators and / and ∗ are postfix operators. The opsem for this grammar is given below:

OpSem:

$$\frac{}{A; n \Rightarrow n} \qquad \frac{}{A; b \Rightarrow b}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 + v_2}{A; e_1\ e_2 + \ \Rightarrow v_3} \qquad \frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 - v_2}{A; e_1\ e_2 - \ \Rightarrow v_3}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 / v_2}{A; e_1\ e_2\ / \Rightarrow v_3} \qquad \frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 * v_2}{A; e_1\ e_2 * \Rightarrow v_3}$$

$$\frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 > v_2}{A; e_1 e_2 > \Rightarrow v_3} \qquad \frac{A; e_1 \Rightarrow v_1 \quad A; e_2 \Rightarrow v_2 \quad v_3 = v_1 < v_2}{A; e_1\ e_2 < \Rightarrow v_3}$$

	Lexer	Parser	Evaluator	Valid
`1 3 * 5 * 6 +`	L	[✓ P]	E	V
`true false not`	[✓ L]	P	E	V
`+ 1 \ 3 4`	[✓ L]	P	E	V
`* 1 2 * 7 + 6`	L	[✓ P]	E	V
`true false xor`	[✓ L]	P	E	V
`+ 1 / 3 4`	L	[✓ P]	E	V

Problem 8: Operational Semantics

[Total 6 pts]

Version A:

$$\frac{x \Rightarrow v_1 \qquad y \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \&\& v_2}{x \&\& y \Rightarrow v_3}$$

If Ocaml uses the above opsem rule, what would the following Ocaml expression print out?

```
(let _ = print_string "a" in false) && (let _ = print_string "b" in true)
```

"ab"

$$\frac{y \Rightarrow v_1 \qquad x \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \&\& v_2}{x \&\& y \Rightarrow v_3}$$

If Ocaml instead uses the above opsem rule, what would the following Ocaml expression print out?

```
(let _ = print_string "a" in false) && (let _ = print_string "b" in true)
```

"ba"

$$\frac{x \Rightarrow v_1 \qquad y \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \&\& v_2}{x \&\& y \Rightarrow v_3}$$

If Ocaml instead uses the above opsem rule, what would the following Ocaml expression print out?

```
(let _ = print_string "a" in true) && (let _ = print_string "b" in false)
```

"ab"

Version B:

$$\frac{x \Rightarrow v_1 \qquad y \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \&\& v_2}{x \&\& y \Rightarrow v_3}$$

If Ocaml uses the above opsem rule, what would the following Ocaml expression print out?

```
(let _ = print_string "c" in false) && (let _ = print_string "b" in false)
```

"cb"

$$\frac{y \Rightarrow v_1 \qquad x \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \&\& v_2}{x \&\& y \Rightarrow v_3}$$

If Ocaml instead uses the above opsem rule, what would the following Ocaml expression print out?

```
(let _ = print_string "c" in false) && (let _ = print_string "b" in false)
```

"bc"

$$\frac{x \Rightarrow v_1 \qquad y \Rightarrow v_2 \qquad v_3 \text{ is } v_1 \&\& v_2}{x \&\& y \Rightarrow v_3}$$

If Ocaml instead uses the above opsem rule, what would the following Ocaml expression print out?

```
(let _ = print_string "c" in true) && (let _ = print_string "b" in false)
```

"cb"

Problem 9: Lambda Calculus

[Total 10 pts]

(a) Reduce [5 pts]

Reduce the following lambda expression. Show every step.

Version A:

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

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

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

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

$y\ b$

Version B:

$$((\lambda a.\ (\lambda b.\ b\ a))\ b)(\lambda a.\ c\ a)$$

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

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

$(\lambda a.\ c\ a)b$

$c\ b$

(b) Free Variables: [2 pts]

Circle the free variables in the expression below:

Version A:

$$(\lambda x.(\lambda x.x\ x)\ x)\ \textcircled{x}\ (\lambda y.y\ \textcircled{f})\ \textcircled{a}$$

Version B:

$$(\lambda x.(\lambda x.\textcircled{a}\ x)\ x)\ \textcircled{a}\ (\lambda y.y\ y)\ \textcircled{x}$$

(c) Alpha Equivalence: [2 pts]

Version A:

Which of the following are alpha equivalent to the expression above, $(\lambda x.(\lambda x.x\ x)\ x)\ x\ (\lambda y.y\ f)\ a$? Select all that apply.

- A $(\lambda x.(\lambda b.b\ b)\ b)\ x\ (\lambda w.w\ f)\ a$
- B $(\lambda w.(\lambda b.b\ b)\ w)\ w\ (\lambda c.c\ f)\ a$
- C $(\lambda y.(\lambda d.d\ d)\ y)\ x\ (a\ f)$
- [✓ D] $(\lambda w.(\lambda z.z\ z)\ w)\ x\ (\lambda y.y\ f)\ a$

Version B:

Which of the following are alpha equivalent to the expression above, $(\lambda x.(\lambda x.a\ x)\ x)\ a\ (\lambda y.y\ y)\ x$? Select all that apply.

- A $(\lambda y.(\lambda x.a\ x)\ y)\ a\ (\lambda z.y\ y)\ y$
- [✓ B] $(\lambda y.(\lambda y.a\ y)\ y)\ a\ (\lambda z.z\ z)\ x$
- C $(\lambda x.(\lambda x.x\ x)\ x)\ a\ (x\ x)$
- [✓ D] $(\lambda z.(\lambda x.a\ x)\ z)\ a\ (\lambda f.f\ f)\ x$

Problem 10: Ownership and Lifetimes

[Total 8 pts]

```
1 fn main(){
2   let x = 4;
3   let y = x;
4   println!("{x},{y}");
5 }
```

Does the code compile? [✓ Y Yes] N No

If **no**, explain why not in one sentence:

```
1 fn main(){
2   let x = String::from("Hello");
3   let y = &mut x;
4   println!("{y}");
5 }
```

Does the code compile? Y Yes [✓ N No]

If **no**, explain why not in one sentence:

cannot borrow mut from an immutable value

```
1 fn main(){
2   let mut x = String::from("Hello");
3   let y = &mut x;
4   x.push_str(" world");
5   println!("{x},{y}");
6 }
```

Does the code compile? Y Yes [✓ N No]

If **no**, explain why not in one sentence:

2 mut borrows exist, lines 3 and 4

```
1  fn function<'a>(s1:&'a String,
2                  s2:&'a String,
3                  f:bool)->usize{
4      if f {s1.len()} else{s2.len()}
5  }

6  fn main(){
7      let a = String::from("hello");
8      let b = a.clone();
9      let c = function(&b,&a,true);
10     println!("{a} has length {c}");
11 }
```

Does the code compile? [✓ Y Yes] N No

If **no**, explain why not in one sentence:

```
1 fn main(){
2   let mut x = String::from("Hello");
3   let y = &mut x;
4   x.push_str(" world");
5   println!("{y}");
6 }
```

Does the code compile? Y Yes [✓ N No]

If **no**, explain why not in one sentence:

2 mut borrows attempted, lines 3 and 4 – Cannot modify x while y is borrowing the same string

---

```
1 fn main(){
2   let x = 4;
3   let y = x;
4   println!("{}, {}", x, y);
5 }
```

Does the code compile? [✓ Y Yes] N No

If **no**, explain why not in one sentence:

```
fn main(){
  let x = String::from("Hello");
  let y = &mut x;
  y.push_str(" world");
  println!("{x}");
}
```

Does the code compile? Ⓨ Yes [✓ Ⓝ No]

If **no**, explain why not in one sentence:

cannot borrow x as mutable

```
fn function<'a>(s1:&'a String,
                s2:&'a String,
                f:bool)->usize{
    if f {s1.len()} else{s2.len()}
}

fn main(){
    let a = String::from("hello");
    let b = a.clone();
    let c = function(a,b,true);
    println!("{a} has length {c}");
}
```

Does the code compile? Ⓨ Yes [✓ Ⓝ No]

If **no**, explain why not in one sentence:

arguments for function are not references

Problem 11: Coding

[Total 18 pts]

(a) Flatten: [8 pts]

Write a function that takes in a Tree and returns a linked list of the tree in pre-order. You may make helper functions.

```
type tree = TNode of tree * int * tree | Leaf of int
type llist = LNode of int * llist | Tail of int

let rec flatten tree =
    let rec append l1 l2 = match l1 with
    |Tail(x) -> LNode(x,l2)
    |LNode(x,y) -> LNode(x,append y l2)
        in match tree with
            |Leaf(x) -> Tail(x)
            |TNode(l,v,r) -> let llst = flatten l in let rlst = flatten r
                             in append LNode(v,llst) rlst
```

(b) Reachable: [10 pts]

Given a graph and a starting node, return all the reachable nodes as a list (order does not matter).

```
type node = string * int (* name of node and its value *)
type edge = node * node (* bidirectional graph *)
type graph = (node list) * (edge list)

let reachable g state =
  let (nodes,edges) = g in
  let rec onestep s =
    List.fold_left (fun a (src,dest) ->
    (* can also only check one direction *)
      if src = s then union a [dest] else if dest = s then union a [src] else a)
      [s] edges in
  let rec next states =
     match states with
     [] -> []
    | x::xs -> union (onestep x) (next xs) in
```

```
let rec reach states =
  let ret = next states in
  if eq ret states then ret else reach ret in
reach [state] (* can also remove inital state *)
```

Problem 12: Extra Credit

[Total 2 pts]

(a) Staff Stalking [1 pts]

What is your discussion TA's name and what is your discussion number?

(b) Colon Parenthesis [1 pts]

Write a poem!

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)



(* 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
```

```
(* Regex in OCaml *)
Re.Posix.re: string -> regex
Re.compile: regex -> compiled_regex

Re.exec: compiled_regex -> string -> group
Re.execp: compiled_regex -> string -> bool
Re.exec_opt: compiled_regex -> string -> group option

Re.matches: compiled_regex -> string -> string list

Re.Group.get: group -> int -> string
Re.Group.get_opt: group -> int -> string option
```

Structure of Regex

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

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

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**
$F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$
```

Rust

```
// Vectors
let vec = Vec::new();  // makes a new vector
let vec1 = vec![1,2,3]

vec.push(ele);  // Pushes the element 'ele'
                // to end of the vector 'vec'
// Strings
let string = String::from("Hello");

string.push_str(&str); // appends the str
                       // to string

vec.to_iter();  // returns an iterator for vec

string.chars()  // returns an iterator of chars
                // over the a string
```

```
iter.rev();       // reverses an iterators direction

iter.next();      // returns an Option of the next
                  // item in the iterator.

struct Building{   // example of struct
    name:String,
    floors:i32,
    locationx:f32,
    locationy:f32,
}

enum Option<T>{ Some(T); None } //enum Option type
option.unwrap();  // returns the item in an Option or
                  // panics if None
```