Discussion 1

Discussion 1 - Thursday, June 12th

Reminders

1. Quiz 1 next **Tuesday, June 17th**
    1. Quizzes will open **at 9:30** and you will have **25 minutes** to complete it.
    2. Topics will be announced soon on Piazza.
2. Project 2 released! Due **Wednesday, June 18th**
3. If you need to make up the quiz in some way, or you need an ADS accommodation, please **email your instructor** with valid documentation

Exercises

OCaml Lists and Patern Matching

1. Write the following functions in OCaml **using recursion**:

`remove_all lst x`

- **Type:** `'a list -> 'a -> 'a list`
- **Description:** Takes in a list `lst` and returns the list `lst` without any instances of the element `x` in the same order.

```
remove_all [1;2;3;1] 1 = [2;3]
remove_all [1;2;3;1] 5 = [1;2;3;1]
remove_all [true; false; false] false = [true]
remove_all [] 42 = []
```

▼ Solution

```
let rec remove_all lst x = match lst with
    | [] -> []
    | h::t -> if x = h then (remove_all t x) else h::(remove_all t x);;
```

`index_of lst x`

- **Type:** `'a list -> 'a -> int`
- **Description:** Takes in a list `lst` and returns the index of the first instance of element `x`.
- **Notes:**
    - If the element doesn't exist, you should return `-1`
    - You can write a helper function!

```
index_of [1;2;3;1] 1 = 0
index_of [4;2;3;1] 1 = 3
index_of [true; false; false] false = 1
index_of [] 42 = -1
```

▼ Solution

```
let index_of lst x =
    let rec helper lst x i = match lst with
    | [] -> -1
    | h::t -> if x = h then i else (helper t x (i + 1))
in helper lst x 0;;
```

2. Give the type for each of the following OCaml expressions:

> **NOTE:** Feel free to skip around, there are a lot of examples! 😳

```
[2a] fun a b -> b < a

[2b] fun a b -> b + a > b - a

[2c] fun a b c -> (int_of_string c) * (b + a)

[2d] fun a b c -> (if c then a else a) * (b + a)

[2e] fun a b c -> [ a + b; if c then a else a + b ]

[2f] fun a b c -> if a b != a c then (a b) else (c < 2.0)

[2g] fun a b c d -> if a && b < c then d + 1 else b
```

▼ Solution

```
[2a] 'a -> 'a -> bool

[2b] int -> int -> bool

[2c] int -> int -> string -> int

[2d] int -> int -> bool -> int

[2e] int -> int -> bool -> int list

[2f] (float -> bool) -> float -> float -> bool

[2g] bool -> int -> int -> int -> int
```

3. Write an OCaml expression for each of the following types:

```
[3a] int * bool list

[3b] (int * float) -> int -> float -> bool list

[3c] float -> string -> int * bool
```

```
[3d] (int -> bool) -> int -> bool list

[3e] ('a -> 'b) -> 'a -> 'a * 'b list

[3f] ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c

[3g] 'a -> 'b list -> 'a -> 'a * 'a
```

▼ Solution

```
[3a] (1, [true])
(* NOTE: same thing as `int * (bool list)` *)

[3b] fun (a, b) c d -> [a + 1 = c; b +. 1.0 = d]

[3c] fun a b -> (int_of_float a, b = "a")

[3d] fun f a -> [f a; a = 2]

[3e] fun f a -> (a, [f a])

[3f] fun f g a -> g (f a)

[3g] fun a b c -> if (a = c && b = []) then (a,a) else (c,c)
```

4. Give the type of the following OCaml function:

```
let rec f p x y =
match x, y with
   | ([], []) -> []
   | ((a,b)::t1, c::t2) -> (p a c, p b c)::(f p t1 t2)
   | (_, _) -> failwith "error";;
```

▼ Solution

```
('a -> 'b -> 'c) -> ('a * 'a) list -> 'b list -> ('c * 'c) list
```

5. What values do `x`, `y`, and `z` bind to in the following code snippet?

```
let x = match ("lol", 7) with
   | ("haha", 5)  -> "one"
   | ("lol", _)   -> "two"
   | ("lol", 7)   -> "three"
;;

let y = match (2, true) with
   | (1, _)       -> "one"
   | (2, false)   -> "two"
   | (_, _)       -> "three"
   | (_, true)    -> "four"
;;

let z = match [1;2;4] with
   | []           -> "one"
   | 2::_         -> "two"
```

```
    | 1::2::t      -> "three"
    | _            -> "four"
;;
```

▼ Solution

```
x: two
y: three
z: three
```

Higher Order Functions (Map & Fold)

Consider the following higher order functions:

```
let rec map f xs =
  match xs with
    | [] -> []
    | h::t -> (f h)::(map f t)

let rec fold f a lst =
  match lst with
    | [] -> a
    | h::t -> fold f (f a h) t

let rec fold_right f lst a =
  match lst with
    | [] -> a
    | h::t -> f h (fold_right f t a)
```

Map vs Fold

Both `map` and `fold` are higher-order functions, but are used in different scenarios.

`map`

`map` is a structure-preserving operation, meaning that it applies a function to each element of a list and returns a new list of the same structure but with the new values.

`fold`

`fold` processes the structure from either the left (`fold_left`) or the right (`fold_right`), and uses an accumulator to combine the elements through a given function, ultimately reducing the structure to a single value. Note that `fold_left` is **tail-recursive**, but `fold_right` is not.

Write the following functions using either `fold` , `fold_right` , and / or `map` :

`list_square nums`

- **Type**: `int list -> int list`
- **Description:** Given a list of integers `nums` , return a list where each value is squared.
- **Examples:**

```
list_square [1; 2; 3; 4] = [1; 4; 9; 16]
list_square [0; 5; 6] = [0; 25; 36]
list_square [] = []
list_square [-3; -2; -1] = [9; 4; 1]
```

▼ Solution

```
let list_square nums = map (fun x -> x * x) nums
```

`swap_tuples tuples`

- **Type**: `(int * int) list -> (int * int) list`
- **Description:** Given a list of two element tuples, swap the first and second elements of each tuple.
- **Examples:**

```
swap_tuples [(1, 2); (3, 4)] = [(2, 1); (4, 3)]
swap_tuples [(5, 10); (7, 8)] = [(10, 5); (8, 7)]
swap_tuples [(0, 0)] = [(0, 0)]
```

▼ Solution

```
let swap_tuples tups = map (fun (a,b) -> (b,a)) tups
```

`list_product nums`

- **Type**: `int list -> int`
- **Description:** Given a list of `nums`, return the product of all elements in the list.
- **Examples:**

```
list_product [2; 5] = 10
list_product [3; 0; 2] = 0
list_product [] = 1
```

▼ Solution

```
let list_product nums = fold (fun acc elem -> acc * elem) 1 nums
```

`list_add x nums`

- **Type**: `int -> int list -> int list`
- **Description:** Given a number `x` and a list of integers `nums`, return `nums` with all of its values incremented by `x`.
- **Examples:**

```
list_add 1 [1;2;3;4] = [2;3;4;5]
list_add 3 [1;2;3;4] = [4;5;6;7]
list_add 1 [] = []
list_add (-3) [7;10] = [4;7]
```

▼ Solution!

```
let list_add x nums = map (fun num -> num + x) nums
let list_add x nums = map ((+) x) nums (* sillier version *)
```

`mold f lst`

- **Type**: `('a -> 'b) -> 'a list -> 'b list`
- **Description:** Rewrite the `map` function using `fold`
- **Examples**:

```
mold (fun x -> x = 3) [1;2;3;4] = [false;false;true;false]
mold (fun x -> x - 1) [1;2;3;4] = [0;1;2;3]
mold (fun x -> 0) [1;2;3;4] = [0;0;0;0]
mold (string_of_int) [1;2;3;4] = ["1";"2";"3";"4"]
```

- **Addendum:** What happens if we use `fold_right` instead of `fold`? How does this affect the order of iteration?

▼ Solution!

```
let mold f lst = List.rev (fold (fun a x -> (f x)::a) [] lst)
let mold f lst = fold (fun a x -> a @ [(f x)]) [] lst

(* Notice how we don't have to reverse the list! *)
let mold f lst = fold_right (fun x a -> (f x)::a) lst []
```

If we append to the accumulator using `(f x) :: a`, we are adding elements to the front of the list. Since fold processes the list from left to right, the output list will be made in reverse order. However, fold_right processes the list from right to left, which preserves the original order without needing to reverse it at the end. The order of iteration matters here!

`list_sum_product lst`

- **Type**: `int list -> int * int * bool`
- **Description**: Write a function that takes in an `int list` and returns an `int * int * bool` tuple of the following form:
  - The first element is the **sum** of the **even** indexed elements
  - The second element is the **product** of the **odd** indexed elements.
  - The third element is a boolean that will be **true** if the sum and the product are equal, otherwise **false**.
- **Note:** The list is 0 indexed, and 0 is an even index.
- **Examples**:

```
list_sum_product [] = (0,1,false)
```

```
list_sum_product [1;2;3;4] = (4,8,false)
list_sum_product [1;5;4;1] = (5,5,true)
list_sum_product [1;-2;-3;4] = (-2,-8,false)
```

▼ Solution!

```
let list_sum_product lst =
  let (sum, product, index) = fold
    (fun (even, odd, i) num ->
      if i mod 2 = 0
        then (even + num, odd, i + 1)
        else (even, odd * num, i + 1))
    (0, 1, 0) lst
  in (sum, product, sum = product);;
```

Records

Consider the following custom record type, which is similar to the return tuple of `list_sum_product`:

```
type results = {
  sum_even: int;
  product_odd: int;
  num_elements: int;
}
```

`record_sum_product lst`

- **Type**: `int list -> results`
- **Description**: Similar to the `list_sum_product` function above, but returns a `results` record with the following fields:
  - `sum_even` is the **sum** of the **even indexed** elements
  - `product_odd` is the **product** of the **odd indexed** elements.
  - `num_elements` is the number of elements in `lst`
- **Note:** The list is 0 indexed, and 0 is an even index.
- **Examples**:

```
record_sum_product [] = {sum_even = 0; product_odd = 1; num_elements = 0}
record_sum_product [1;2;3;4] = {sum_even = 4; product_odd = 8; num_elements = 4}
record_sum_product [1;5;4;1] = {sum_even = 5; product_odd = 5; num_elements = 4}
record_sum_product [1;-2;-3;4] = {sum_even = -2; product_odd = -8; num_elements = 4}
```

▼ Solution!

```
let record_sum_product lst =
  fold (fun {sum_even; product_odd; num_elements} num ->
    if num_elements mod 2 = 0
      then {
        sum_even = sum_even + num;
        product_odd;
        num_elements = num_elements + 1 }
      else {
        sum_even;
        product_odd = product_odd * num;
        num_elements = num_elements + 1 })
  {sum_even = 0; product_odd = 1; num_elements = 0} lst;;
```

Another exercise! Consider the following custom record types:

```
type weather_data = {
    temperature: float;
    precipitation: float;
    wind_speed: int;
}

type cp_weather_report = {
    days: weather_data list;
    num_of_days: float;
}
```

`average_temperature report`

- **Type:** `cp_weather_report -> float`
- **Description:** This function takes a `cp_weather_report` record, containing a list of `weather_data` records from College Park and returns the average temperature of College Park.
- **Note:** If the `num_of_days` within `cp_weather_report` is 0 then return 0.0
- **Examples:**

```
let ex1 = {
  days = [
    { temperature = 70.0; precipitation = 0.2; wind_speed = 10 };
    { temperature = 68.0; precipitation = 0.1; wind_speed = 12 };
    { temperature = 72.0; precipitation = 0.0; wind_speed = 8 };
    { temperature = 75.0; precipitation = 0.3; wind_speed = 15 }
  ];
  num_of_days = 4.0
}
average_temperature ex1 = 71.25

let ex2 = {
    days = [];
    num_of_days = 0.0
}
average_temperature ex2 = 0.0

let ex3 = {
    days = [
    { temperature = 30.0; precipitation = 0.0; wind_speed = 3 };
    { temperature = 35.0; precipitation = 0.0; wind_speed = 4 }
  ];
  num_of_days = 2.0
}
average_temperature ex3 = 32.5
```

▼ Solution!

```
let average_temperature reports =
  if reports.num_of_days = 0.0
    then 0.0
  else
    let total_temp =
      List.fold_left (fun sum day -> sum +. day.temperature) 0.0 reports.days
    in total_temp /. reports.num_of_days
```

Variant Types

Let's build a custom binary `tree` data type in OCaml! First, we will define the `tree` type:

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree
```

This recursively defines a `tree` to either be a:

- `Leaf`
- `Node` with a left sub-`tree`, a value, and a right sub-`tree`

`tree_add x tree`

- **Type**: `int -> int tree -> int tree`
- **Description**: Given an `int tree`, return a new `int tree` with the same values in the old tree incremented by `x`.
- **Examples**:

```
let tree_a = Node(Node(Leaf, 5, Leaf), 6, Leaf)
let tree_b = Node(Node(Leaf, 4, Leaf), 5, Node(Leaf, 2, Leaf))

tree_add 1 tree_a = Node(Node(Leaf, 6, Leaf), 7, Leaf)
tree_add 5 tree_b = Node(Node(Leaf, 9, Leaf), 10, Node(Leaf, 7, Leaf))
```

▼ Solution!

```
let rec tree_add x tree = match tree with
  | Leaf -> Leaf
  | Node(l, v, r) -> Node(tree_add x l, v + x, tree_add x r)
```

`tree_preorder tree`

- **Type**: `string tree -> string`
- **Description**: Given a `string tree`, return the preorder concatenation of all the strings in the tree.
- **Examples**:

```
let tree_c = Node(Node(Leaf, " World", Leaf), "Hello", Node(Leaf, "!", Leaf))
let tree_d = Node(Node(Node(Leaf, " super", Leaf), " is", Node(Leaf, " easy!", Leaf)), "Recursion"
, Node(Leaf, " 💀", Leaf))

tree_preorder tree_c = "Hello World!"
tree_preorder tree_d = "Recursion is super easy! 💀"
```

▼ Solution!

```
let rec tree_add x tree = match tree with
  | Leaf -> Leaf
  | Node(l, v, r) -> Node(tree_add x l, v + x, tree_add x r)
```

`tree_sum_product tree`

- **Type**: `int tree -> int * int`
- **Description**: Given an `int tree`, return an `int * int` tuple of the following form:
  - The first element is the **sum** of **all** numbers in the tree
  - The second element is the **product** of **all** numbers in the tree
- **Examples**:

```
let tree_a = Node(Node(Leaf, 5, Leaf), 6, Leaf)
let tree_b = Node(Node(Leaf, 4, Leaf), 5, Node(Leaf, 2, Leaf))

tree_sum_product tree_a = (11, 30)
tree_sum_product tree_b = (11, 40)
```

▼ Solution!

```
let rec tree_sum_product tree =
  match tree with
  | Leaf -> (0, 1)
  | Node(l, v, r) ->
    let (lsum, lproduct) = tree_sum_product l in
    let (rsum, rproduct) = tree_sum_product r in
    (lsum + v + rsum, lproduct * v * rproduct)
```

Review - Imperative OCaml

```
# let z = 3;;
  val z : int = 3
# let x = ref z;;
  val x : int ref = {contents = 3}
# let y = x;;
  val y : int ref = {contents = 3}
```

Here, `z` is bound to 3. It is immutable. `x` and `y` are bound to a reference. The `contents` of the reference is mutable.

```
x := 4;;
```

will update the `contents` to 4. `x` and `y` now point to the value 4.

```
!y;;
  - : int = 4
```

Here, variables y and x are aliases. In `let y = x`, variable `x` evaluates to a location, and `y` is bound to the same location. So, changing the contents of that location will cause both `!x` and `!y` to change.

Exercises

Imperative OCaml Counter

Recall: The `unit` type means that no input parameters are required for the function.

Implement a counter called `counter`, which keeps track of a value starting at 0. Then, write a function called `next: unit -> int`, which returns a new integer every time it is called.

Example

```
(* First call of next () *)
# next ();;
 : int = 1

(* First call of next () *)
# next ();;
 : int = 2
```

Solution:

▼ **Click here!**

```
# let counter = ref 0 ;;
val counter : int ref = { contents=0 }

# let next =
    fun () -> counter := !counter + 1; !counter ;;
val next : unit -> int = <fun>
```

Function argument evaluation order

What happens when we run this code?

```
let x = ref 0;;
let f _ r = r;;
f (x:=2) (!x)
```

Solution:

▼ **Click here!**

Ocaml's order of argument evaluation is not defined. On some systems it's left to right on others it's right to left.

On my system, **f** evaluates to **0**, but on your system it may evaluate to **2**!

Resources & Additional Readings

- **Spring 2023 OCaml HOF discussion (https://github.com/cmsc330-umd/spring23/tree/main/discussions/d4_hof)**
- **Spring 2023 Project Review (https://github.com/cmsc330-umd/spring23/tree/main/discussions/d5_project_review)**
- **Fall 2023 OCaml HOF discussion (https://github.com/cmsc330fall23/cmsc330fall23/tree/main/discussions/d6_ocaml_hof)**
- **Fall 2023 Python HOF + Regex discussion (https://github.com/cmsc330fall23/cmsc330fall23/tree/main/discussions/d2_hof_regex)**
- **Anwar's Imperative OCaml Slides (https://bakalian.cs.umd.edu/assets/slides/10-imperative.pdf)**

Discussion 2

Discussion 2 - Thursday, June 19th

Reminders

1. Project 3 released, due **Wednesday, July 2nd @ 11:59 PM**

Topic List

- Regular Expressions
- NFAs and DFAs

Review - Regular Expressions

There are many patterns regex can describe aside from string literals.

- **Concatenation (and)**: `ab` We use this to accept something that satisfies a and b in th er, where a and b can denote sub-regex.
  - Ex. `a` matches "a", `b` matches "b", so `ab` matches "ab"
  - Ex. `(a|b)` matches "a" or "b", `c` matches "c", so `(a|b)c` matches "ac" or "bc"
- **Union (or)**: `a|b|c` We use this to accept something from given choices. **Note** that a, b, or c can also denote sub-regex if parentheses are specified.
  - Ex. `a|b|c` matches "a" or "b" or "c"
- **Precedence (parentheses)**: `(a)` are used to enforce order of evaluation and capture groups.
  - Ex. `a|bc` matches "a" or "bc". This is the same as `a|(bc)`
  - Ex. `(a|b)c` matches "ac" or "bc"
- **Sets**: `[abc]` We use this to accept one character from the given choices.
  - Ex. `[abc]` matches "a" or "b" or "c"
- **Ranges**: `[a-z]`, `[c-k]`, `[A-Z]`, `[0-9]` We use these ranges, also known as character classes, to accept characters within a specified range (inclusive).
  - Ex. `[a-z]` matches any lowercase letter
  - Ex. `[c-k]` matches letters c to k inclusive
  - Ex. `[A-Z]` matches any uppercase letter
  - Ex. `[0-9]` matches any digit
  - Ex. `[a-z0-9]` matches any lowercase letter or digit
- **Negation**: `[^abc]` `[^a-z]` `[^0-9]` We use these to exclude a set of characters.
  - Ex. `[^abc]` matches with any character other than "a", "b", or "c"
  - Ex. `[^a-z]` matches with any character that is not a lowercase letter

  - Ex. `[^0-9]` matches with any character that is not a digit
  - Note that the use of "^" differs from the beginning of a pattern
- **Meta Characters**: `\d`, `\D`, `\s`, `\w`, `\W` We use these characters to match on any of a particular type of pattern.
  - ex. `\d` matches any digit (equivalent to `[0-9]`)
  - ex. `\D` matches any character that is not a digit (equivalent to `[^0-9]`)
  - ex. `\s` matches any whitespace character (spaces, tabs, or newlines)
  - ex. `\w` matches any alphanumeric character from the basic Latin alphabet, including the underscore (equivalent to `[A-Za-z0-9_]`)
  - ex. `\W` matches any character that is not a word character from the basic Latin alphabet (equivalent to `[^A-Za-z0-9_]`)
- **Wildcard**: `.` We use this to match on **any** single character. Note: to use a literal `.`, we must escape it, i.e. `\.`
- **Repetitions**: `a*`, `a+`, `a?`, `a{3}`, `a{4,6}`, `a{4,}`, `a{,4}`:
  - Ex. `a*` matches with 0 or more a's
  - Ex. `a+` matches with at least one a
  - Ex. `a?` matches with 0 or 1 a
  - Ex. `a{3}` matches with exactly three a's
  - Ex. `a{4,6}` matches with 4, 5, or 6 a's
  - Ex. `a{4,}` matches with at least 4 a's
  - Ex. `a{,4}` matches with at most 4 a's
  - **Note**: a can denote a sub-regex
- **Partial Match**: `a` and `abc` These patterns can match any part of a string that contains the specified characters.
  - Ex. `a` matches "a", "ab," "yay," or "apple"
  - Ex. `abc` matches "abc", "abcdefg," "xyzabcjklm," or "abc123"
  - **Note**: They do not require the specified sequence to be at the beginning or end of the string
- **Beginning of a pattern**: `^hello` The string must begin with "hello".
  - Ex. `^hello` matches with "hellocliff" but does not match with "cliffhello"
- **End of a pattern**: `bye$` The string must end with "bye".
  - Ex. `bye$` matches with "cliffbye" but does not match with "byecliff"
- **Exact Match**: `^hello$` The string must be exactly "hello".
  - Ex. `^hello$` only matches "hello" and no other string
  - **Note**: Enforces both the beginning and end of the string

> **Question**: *Can every string pattern be expressed with a regex?*

Answer: No!

There are certain string patterns that **cannot** be expressed with regex. This is because regex is memoryless; as they cannot keep track of what they have already seen.

As an example, consider a pattern that represents all palindromes, e.g. "racecar". We can't track how many of each character we have previously seen (assuming our regex engine doesn't have backreferences).

Exercises - Regular Expressions

Write a regex pattern for each of the following scenarios (or explain why you cannot):

1. **Exactly** matches a string that alternates between capital & lowercase letters, starting with capital letters. Single-character strings with just one capital letter and empty strings should be allowed.
   - Includes: "AaBbCc", "DlFsPrOa", "HiWoRlD"
   - Excludes: "aAbBcC", "aaa", "123"
2. Matches a string that contains an even number of 3s, and then an odd number of 4s.
   - Includes: "3333444", "334", "3333334444444", "4"
   - Excludes: "34", "33344", "334444", "1111222"
3. Matches a string that contains a phone number following the format (XXX)-XXX-XXXX where X represents a digit.
   - Includes: "(123)-456-7890", "(111)-222-3333"
   - Excludes: "123-456-7890", "1234567890"
4. **Exactly** matches a string email following the format [Directory ID]@umd.edu where [Directory ID] is any sequence consisting of lowercase letters (a-z), uppercase letters (A-Z), or digits (0-9) with length >= 1.
   - Includes: "colemak123@umd.edu", "ArStDhNeIo@umd.edu", "b@umd.edu"
   - Excludes: "qwerty@gmail.com", "@umd.edu"
5. Matches a string that has more 7s, 8s, and 9s than 1s, 2s, and 3s.
   - Includes: "7891", "123778899", "12789", "8"
   - Excludes: "1", "271", "12399", "831"

Solutions

▼ Click here!

1. `/^([A-Z][a-z])*([A-Z])?$/` 2. `/(33)*4(44)*/` 3. `/\([0-9]{3}\)-[0-9]{3}-[0-9]{4}/` (Note, we have to escape the parenthesis with `\`) 4. `/^[a-zA-Z0-9]+@umd\.edu$/` (Note, we have to escape the period with `\`) 5. Cannot be represented with regular expressions, since there is no memory of which numbers have been previously used.

NFAs and DFAs

Notes:

Key differences between NFA and DFA

- All DFAs are NFAs, but not all NFAs are DFAs.
- NFA can have ε-transition(s) between states.
- NFA states can have multiple transitions going out of them using the same symbol.
- DFAs are computationally cheaper to process, but often harder to read compared to NFAs.

Exercises

Regex -> NFA

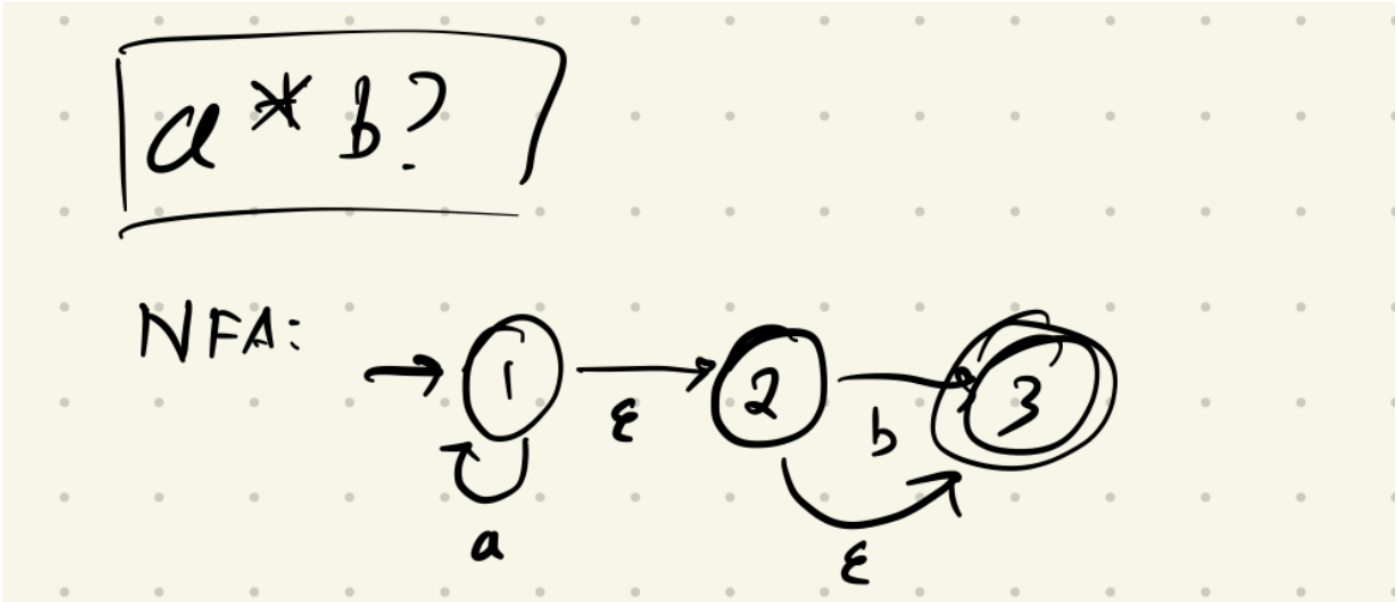
1. Consider the following regular expressions:

```
a) a*b?
b) (b|c)+
c) a*b?(b|c)+
```

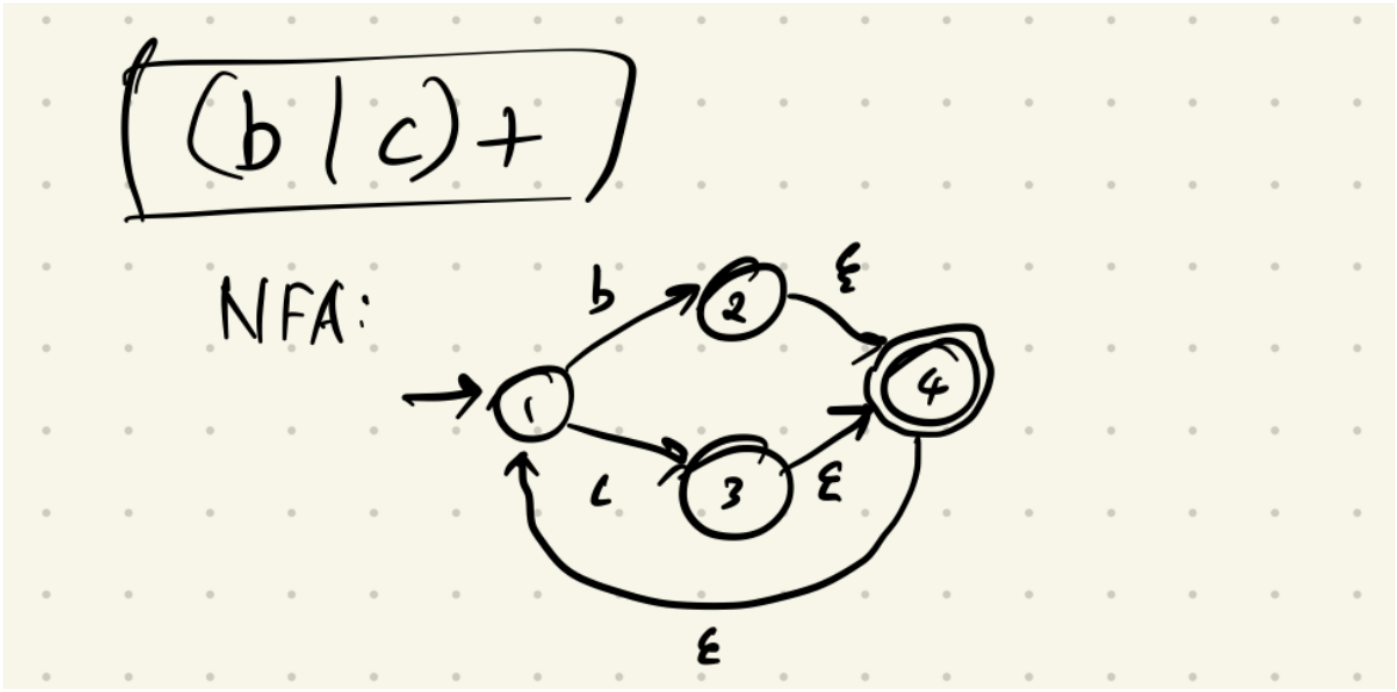
   - Convert each regex to an equivalent NFA
     - Note that there are many valid NFAs
   - Convert each NFA to its equivalent DFA
   - Compare your DFA with the person next to you
     - Are they the same?

▼ Solutions!

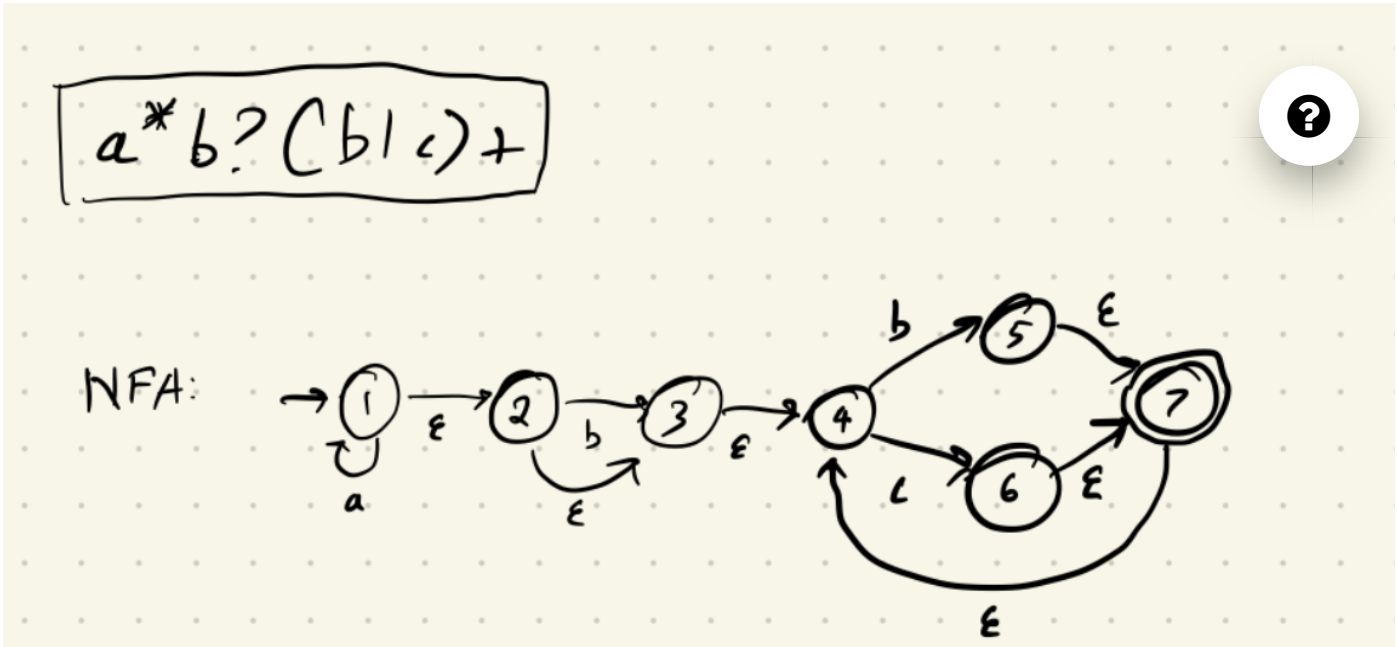
a)



b)



c)



Resources & Additional Readings

- Fall 2023 Python HOF + Regex discussion (https://github.com/cmsc330fall23/cmsc330fall23/tree/main/discussions/d2_hof_regex)
- Online Regular Expression Tester (https://regexr.com/)
- Regex Practice Problem Generator (https://apabla1.github.io/)
- Fall 2023 Discussion - NFA and DFA (https://github.com/cmsc330fall23/cmsc330fall23/tree/main/discussions/d3_nfa_dfa)
- Fall 2023 Discussion - NFA and DFA Conversion (https://github.com/cmsc330fall23/cmsc330fall23/tree/main/discussions/d4_nfa_dfa_conversion)

Discussion 3

Discussion 3 - Thursday, June 26th

Reminders

1. Project 3 due **Wednesday, July 2nd**
2. Your midterm is **Thursday, July 3rd** instead of discussion. The format will be the same as quizzes (i.e. on Gradescope, during Discussion time), but extend the entire allotted time rather than 25 minutes. Logistics post soon.

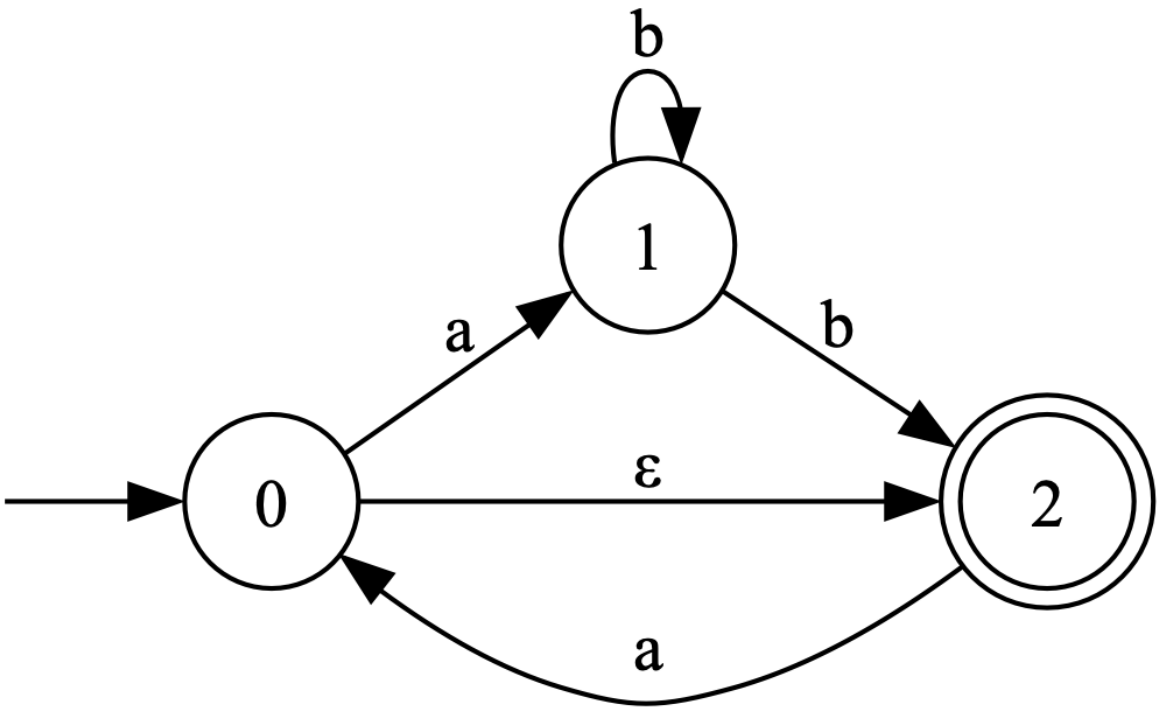
NFA/DFA Continued

Exercises

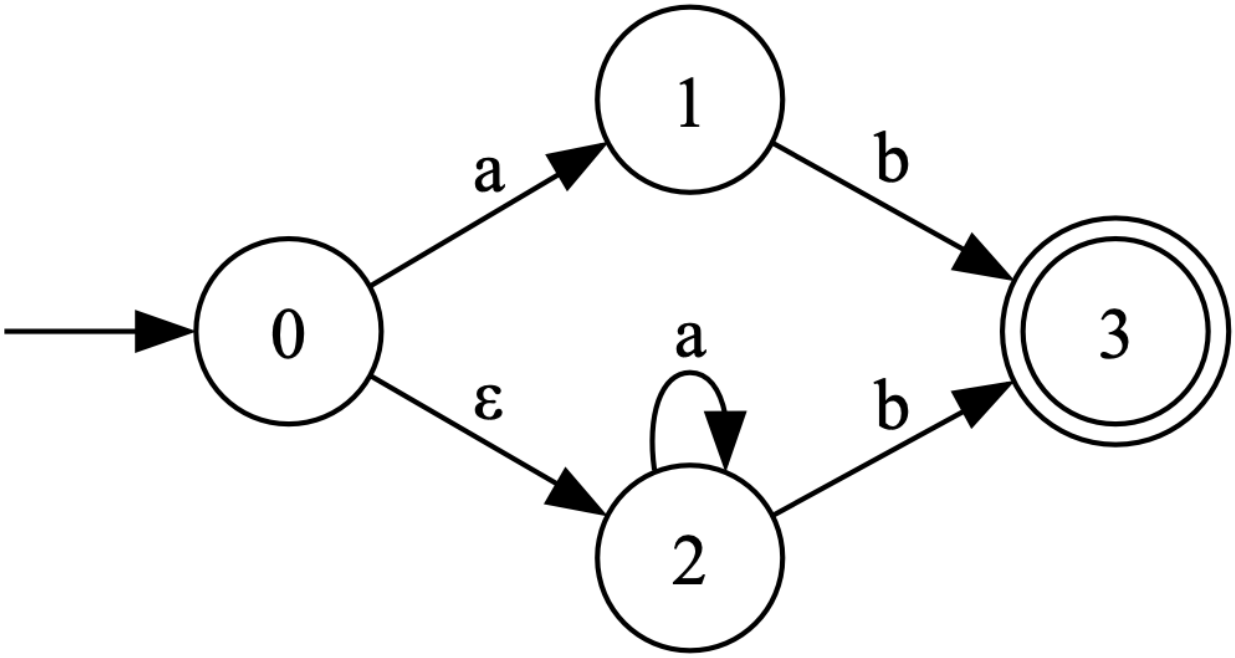
NFA -> DFA

1. Trace through the NFA -> DFA conversion algorithm using the table method for the following NFAs:

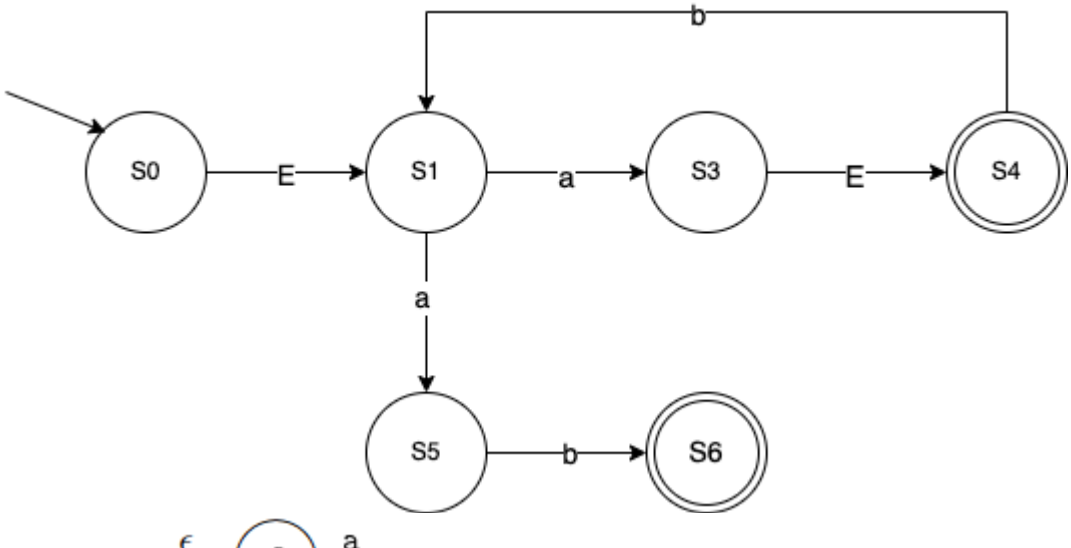
a)



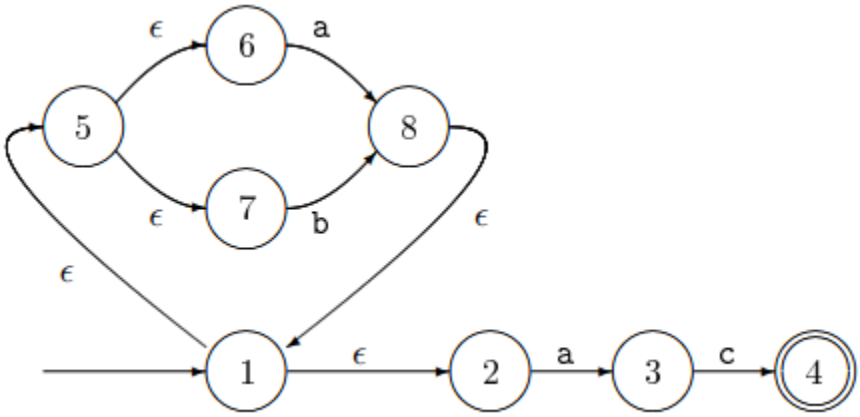
b)



c)

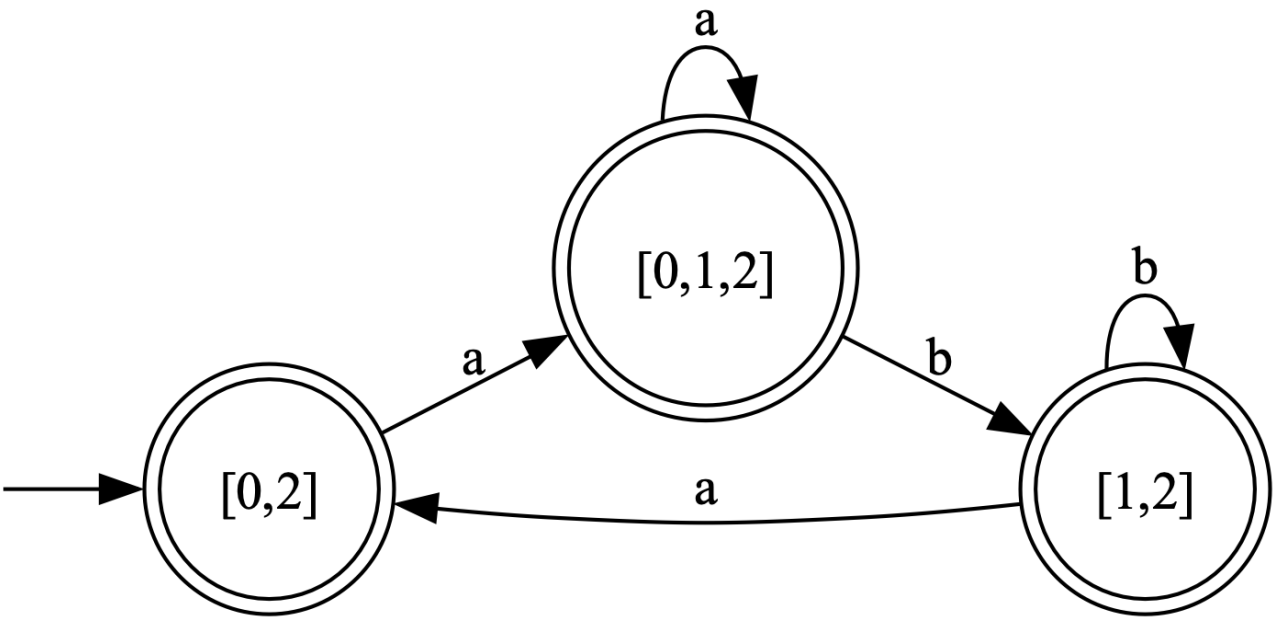


d)

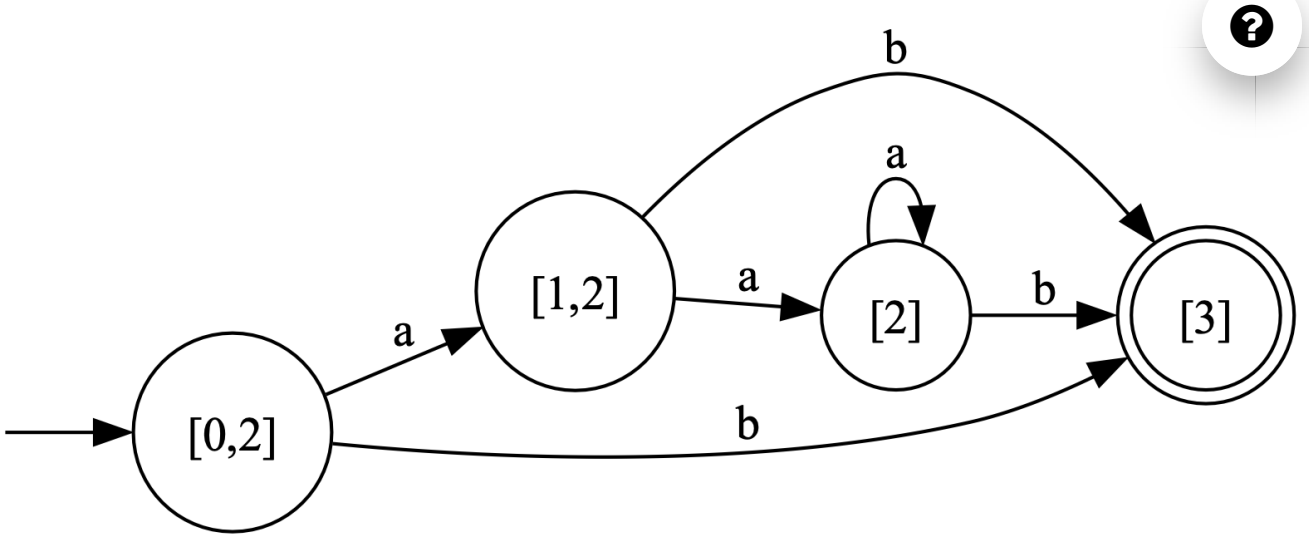


▼ Solutions!

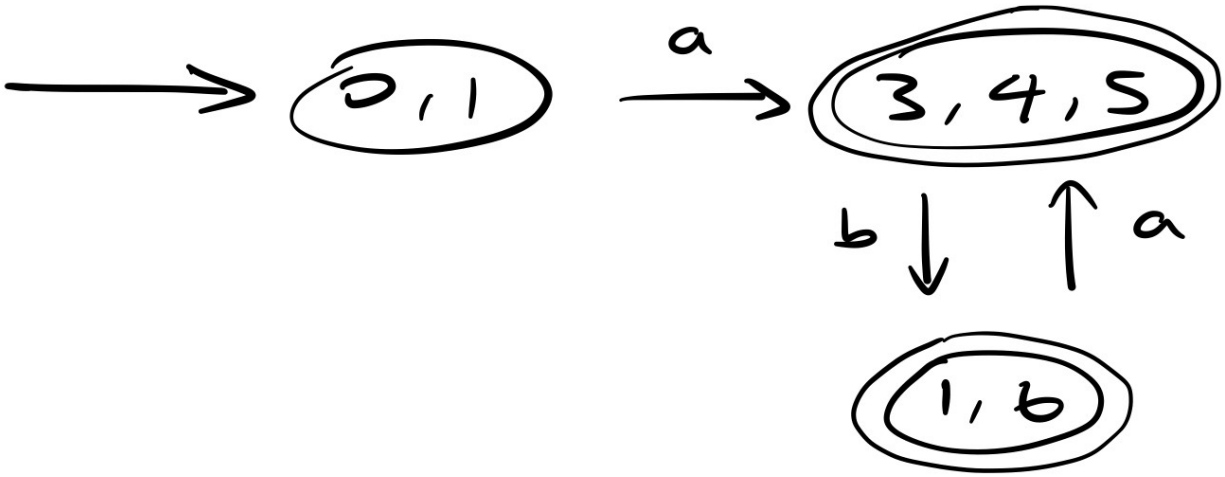
a)



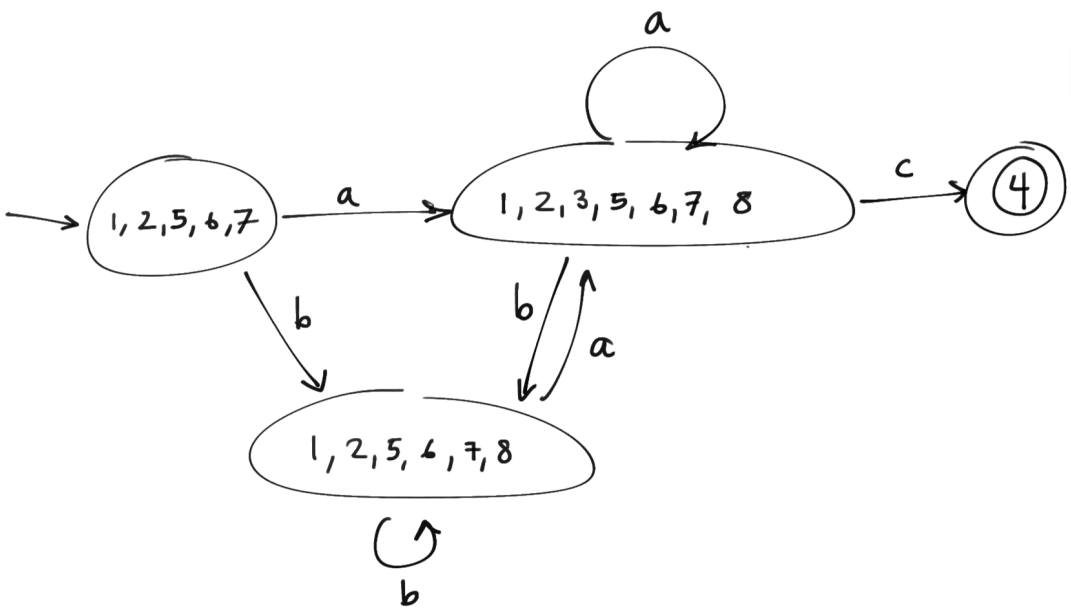
b)



c)



d)



Context Free Grammars

Exercises:

1. Consider the following grammar:

```
S -> S + S | 1 | 2 | 3
```

   - Write a leftmost derivation for the string: `1 + 2 + 3`

- Start with S and use the production rules on the LEFTMOST nonterminal ONE AT A TIME. (For a rightmost derivation, use the productions on the RIGHTMOST nonterminal.)
- ONE NONTERMINAL AT A TIME!!!! DON'T COMBINE STEPS!!!! (or you might lose credit)

- If there are 2 leftmost derivations or 2 rightmost for the same string in a grammar, what does that mean?

▼ Solution

Leftmost Derivation: S -> S + S -> S + S + S -> 1 + S + S -> 1 + 2 + S -> 1 + 2 + 3 OR S -> S + S -> 1 + S -> 1 + S + S -> 1 + 2 + S -> 1 + 2 + 3 There are two leftmost derivations for this string, which means the grammar is **ambiguous**.

1. Consider the following grammar:

```
S -> aS | T
T -> bT | U
U -> cU | ε
```

- Provide derivations for:
  - b
  - ac
  - bbc
- What language is accepted by this grammar?
- Create another grammar that accepts the same language.

▼ Solution

S -> T -> bT -> bU -> b S -> aS -> aT -> aU -> acU -> ac S -> T -> bT -> bbT -> bbU -> bbcU -> bbc This is the language of all strings equivalent to the regex `a*b*c*`. Another grammar that accepts this language is: `S -> Sc | T T -> Tb | U U -> Ua | ε`

1. Consider the expression and construct an equivalent CFG:

$a^xb^xc^y|a^x$ where x >= 0 and y >= 1

▼ Solution

`S -> A | B Union of two languages A -> CD Concatenation of two languages C -> aCb | ε Related number of 0 or more a's and b's D -> cD | c 1 or more c's B -> aB | ε 0 or more a's`

1. Consider the following grammars:

Grammar 1	Grammar 2	Grammar 3
S -> AB	S -> ASB|c	S -> Sc|AB
A -> aA|a	A -> aA|a	A -> aA|a
B -> bbB|ϵ	B -> bbB|ϵ	B -> bbB|ϵ

- Which grammar accepts both `"aaabb"` and `"aaabbcc"`?
- Which grammar is ambiguous?

▼ Solution

Grammar 3 accepts both `"aaabb"` and `"aaabbcc"`. Grammar 2 is ambiguous. For example, the string `"aac"` can be made in two ways.

1. Construct a CFG that generates strings for each of the following:

   - a^xb^y, where y = 2x.
   - a^xb^y, where y >= 3x.

   Can we represent strings of the form a^xb^xc^x, where x >= 0, with CFGs?

▼ Solution

1. `S -> aSbb | ε` 2. `S -> aSbbbB | B B -> bB | ε` We cannot accept strings of the form $a^xb^xc^x$, as there is no way for a CFG to have "memory" of multiple parts of the string, keeping the number of a's and c's equal.