Review:

Formal definition of the language is: set of strings or words, so it is possibly infinite. Because of that kleene closure.
Using finite alphabet, you can create infinite strings. Just like your brain which its size is finite but you can create infinite number of sentences.

Finite automata:

Hardware gives you sequence of bytes.
No hardware in world can run C
Lexical analysis: takes each character one at a time, does the processing defined by regular languages, converts it to stream of tokens

Difference between characters and tokens: tokens are group of characters
Token is the name of all the possible set of strings that could be part of that token
Lexeme is the actual set of characters. Set of possibly infinite strings.
For example:
number is token and *3.14159* is lexeme.
literal is token and *"core dumped"* is lexeme (double quotation is hard coded in literal)
if has only one lexem which is *if*

Anything on your keyboard are characters, but they are represented as a sequence of numbers or codes. So they are just single bytes in machine representation (even the page down button!)

How to recognize all of tokens?
    By using finite automata
How does the compiler know if "life" is not the keyword "if"?
    lexer!

Regular expressions allow us to specify different kinds of tokens.
Automata is kind of mechanical way to show the regular expressions.
Automata is a very simple model of computation.
Automata has a finite set of states, an initial state, finite alphabet, set of accepting states, and transition functions.

Finite automata applications:

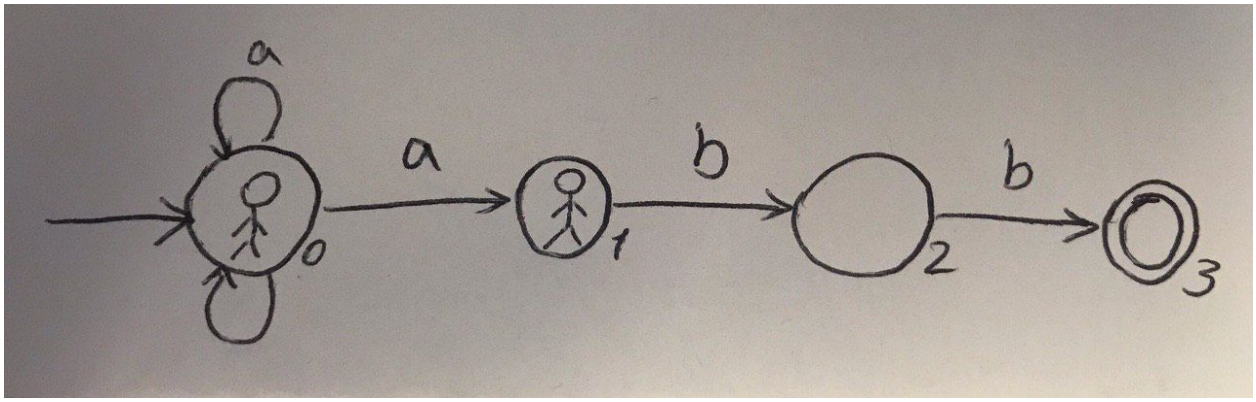
- Vending machines: famous example
- Elevators: decision logic
- Traffic lights
- Combination lock: make sure you put the numbers in the right order

Transition diagrams are like flowcharts

Some questions about Finite Automata:

- Is there any other way to show loops? If you only have finite sets of states without any loop you only can represent finite number of strings.
- We show star or kleene closure by loop in finite states.
- Double circle is accepting state (by definition of finite automata).
- You could have error states, but makes it tedious. We can implicitly assume that if valid transitions are not met, it directly goes to error state (so you do not get stuck)
- Whatever you could do with a regular expression, you could draw a finite automata for it
- It doesn't matter to show the loops clockwise or counterclockwise.

Two kinds of finite automata (deterministic vs non-deterministic)
Deterministic: we can uniquely determine which state we are in. We can't be in more than one states at a time. One state at a given time.
Non-deterministic: We can be in multiple states at the same time. If you have two transition with the same symbol it is a non deterministic automata.
Remember the example in lecture slides: what if we see "a" while we are at state 0 (the reg expression for the example is *(a|b)* abb*).
When you see "a" you can stay in state 0 or go to state 1. So you can be in either state 0 or state 1 at the same time.



This Finite Automata (Non deterministic Finite Automata) is representing "any sequence of a and b followed by abb)
So:
aabaab is **not** in this language
aabaabb is in this language

Beginning of string: '^' (caret), end of string '$'. Sometimes we use these two characters. When we see the end character ('$'), we should be at an accepting state.
Example: ^aabaabb$

What happens if we reach the accepting state but the input string is not consumed completely?
Error! Do not accept.
Is this string a valid input for this language? We search for it while doing the transitions with an input string.
If all enter to error state, it means that the string is not part of the language

grep and other regular expression tools have $ sign explicitly and it is not required to use $ at the end of the string. It will match without using $.

For compilers we want to find specific words in the language for lexical analysis.
Non-deterministic finite automata is easier to create.

What is epsilon: empty string. Transitions diagram can have transitions on epsilon.
It means that in the start state you can also be in other states automatically.
You can always read an empty string: no need to consume characters on input string

Using finite automata to implement a lexer.
The pipeline is: Regular expression -> NFA -> DFA -> Code

What are minimum operations you need to describe any regular language
Union, concatenation and closure.

Concatenation: n state of one language followed by n states of another language. So if you have regexs s and t and you want to create st (concatenation), just make the accepting state of s the starting state of t.
Union: we need epsilon transition to be in two different transition
Closure: we need loops
Have a look at the lecture slides: There are 3 figures that show how to reflect these operations

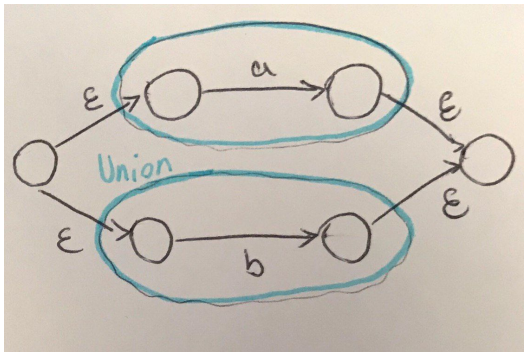
DFA: each dfa states can be a subset of states in NFA.
Read 3.6 and 3.7 in dragon book

Example: (a|b)*abb
You do the stuff in parentheses first (like math).
So first we do union,and then the closure and then concatenation.

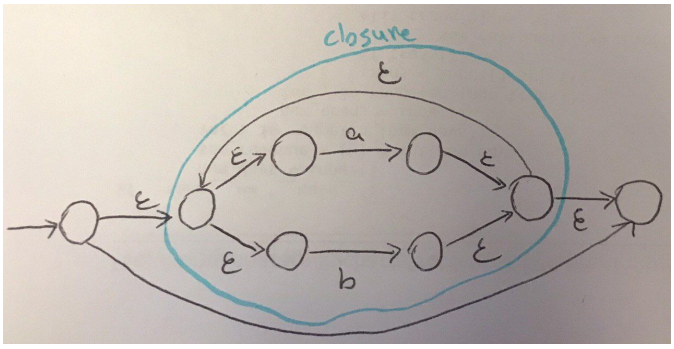
There are arbitrarily many different automata for the same language.
Also, there are many ways to write a regular expression for the same language.

Regex to NFA for (a|b)*abb:

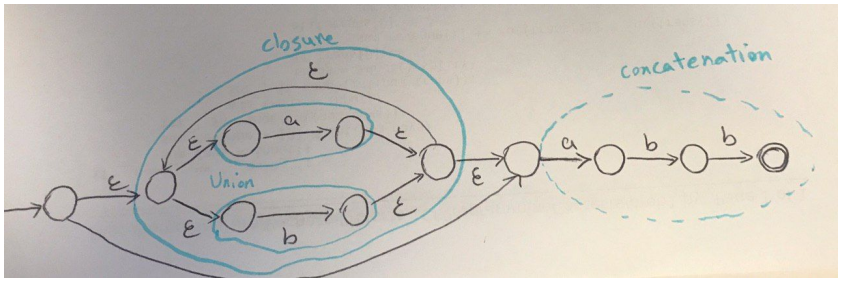
1. For union: we need two epsilon transitions to different states. So we have two patterns: one for just a and one for b.



2. Closure: one start state and one other state at the end of union pattern. And we use epsilon transition to show the closure.



3. Then we have concatenation.



NFA to DFA for (a|b)*abb:
First, number the states
(see the NFA from lecture slides or dragon book)
Initially, just take the epsilon transitions from starting state. For state 0, we can be in states: { 0, 1, 2, 4, 7 }. So, we are going to call it a new state in the DFA we are creating.
Then, using the characters in alphabet (a and b), create new states for the DFA by taking the transitions in NFA. Remember to take epsilon transitions.

Tips about programming project
Union is just conditional (if, else if..)
Kleene-closure: just a while loop (check the character in whiles condition)
The already-implemented parts are useful to understand the logic