We are at the first step of compiler: lexical analysis
Source program (a text file, list of characters). Lexical analysis will pick up words
How to actually process the regular expressions?: finite automata
Lexer uses regular expressions for stream of characters to tokens conversion.

Quick Refresher (Automata)
In automatas, we have only one starting state, which is pointed by an arrow
state machines are very common in video games. For example in Doom's source code, you could see the state machine implementation for animation
It literally implements a state machine for each of the objects in the game
For example: if you are in this state, and, if this happens, go to this state
animation can be in different states, so the program literally program a state machine
It has very practical rules, so this was why as a programmer it is useful for us.

DFA: One state at a time, NFA: Possibly multiple states at a time
You can always have multiple accepting states (both for DFA and NFA)
Regular expressions, NFAs and DFAs are equally powerful.
For the same language, you can have any of them.
There is provably minimal DFA for a language, although you can have different DFAs to represent the same language.
For any NFA you can create a DFA and any DFA is a NFA.

Chomsky hierarchy (next week)
NFA and DFA conversion:
Context free languages: these languages have grammar
Each of these languages has a corresponding machine to recognize them. For regular languages we have finite automata.

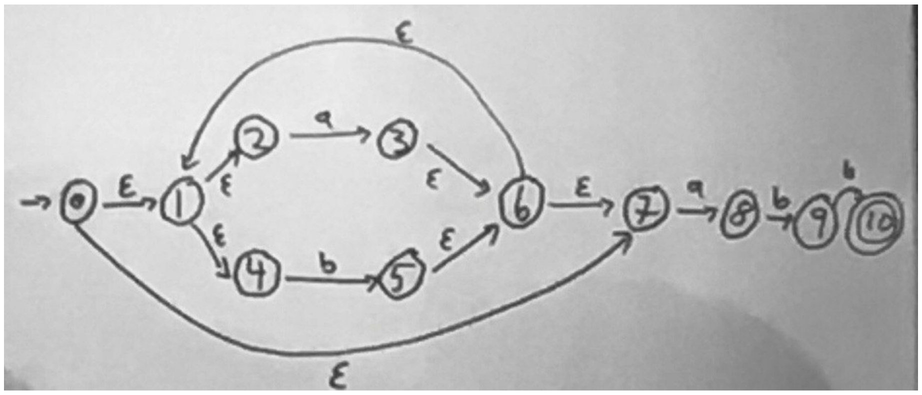
The reason why we have so many epsilon transitions, we used the systematic way of creating the NFA
You could come up with a smaller NFA by using your intuition. However, computer needs a systematic way. Hence, we have the algorithm.

Remember the operations for regular expressions: concat, union, kleene-closure
So, we have three patterns to construct NFA from regular expressions (one for each)
See the "Regular Expression to NFA" figures from lecture slides.

The algorithm to convert NFA to DFA:
- For every combination of multiple states, we assign a single DFA state (called subset construction).
- Do one step at a time, do not think about the big image: this way, applying the NFA->DFA algorithm is much easier.
- Keep track of the starting state and accepting states while applying the algorithm.

- Always check the epsilon transitions, it allows you to be at multiple states at the same time.
- You can implement transition tables by using switch/case.
- If one of the NFA states is in accepting states, the corresponding DFA state will be an accepting state.

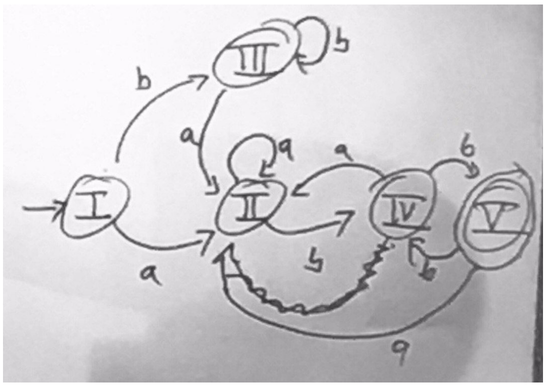
Example: We have this NFA and we want to convert it to a DFA.



The transition table is:

NFA states	DFA states	a	b
{0, 1, 2, 4, 7}	A (starting)	B	C
{1, 2, 3,4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E (accepting)	B	D

So the DFA will be:



Notes:

- Laying off the graph in the optimal way is another computer science problem.
- You can simulate the DFA by writing a program for NFA: results will be the same.
- The current state and the current input is all that matters.
- Empty entries, no possible transitions or error state: all is possible. You can catch the error in a state or you can have no transition at all. In DFA, If there is no transition for an input character while reading the input string, the string will be rejected anyway.
- Turing: any iterative program can be written as a recursive program or vice versa
  So, you can use iteration or recursion to implement FA: does not matter
- The example is included in the Dragon book (section 3.7), and it is described in more detailed and formal way.

Lexer:

You can think of the input as one long array of characters: the string in your text file (file.pl0)
Lexer reads in one character at a time and follows the state machine (ex. fgetc)
Token: abstract construct, lexeme: the string
Ex: There are many different lexemes that match the token *literal*
if has only one lexeme which is if.

Closure is while loop
Concatenation is sequence of statement
Union is conditional a|b

Tips for the first project (lexer.c):

- If your virtual machine runs slowly give it more memory.
- Make sure to read next character.
- There are three main if statements (union) in the lexer.c. 3 possibilities to start to a lexeme: alpha, digit, punctuation.
- Have a look at lexical specification
- EBNF slides: see the slides regarding it. know how it translates to regular expression operators. you are going to see EBNF notation in grammar.md
- Look grammar.md to see how buffer works.
- The only things you need to handle: identifiers, keywords, numbers, punctuation
- For alpha part: read in everything that has alphanumeric characters, then, check if the string read is reserved or not
- this language does not have floating point numbers
- digit part was implemented in class
- After you fill the buffer, do not forget to add the null terminator (‘\0’) at the end of your string in the buffer
- atoi() converts string to number
- the string representation of the number is not the same as the number representation of the number. Hence, to pass the string regarding number to new_number() func, you need to make use of atoi() func.