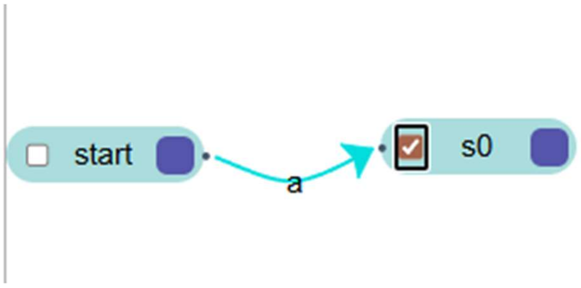
Q1)

a)Regex for Identifier: [a-zA-Z_][a-zA-Z0-9_]*

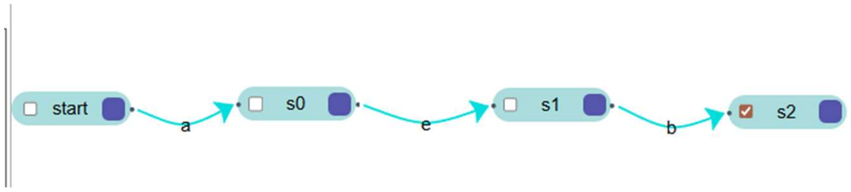
b)Regex -> NFA:

To convert a Regular Expression to NFA, we need to follow these steps:

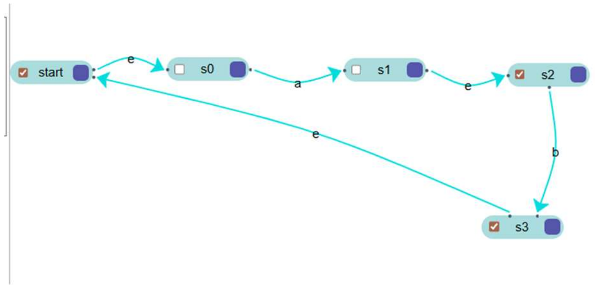
1. Create a NFA for every individual Expression, where Start state Transitions to Final state through that expression.
   Eg. For Regex "a"



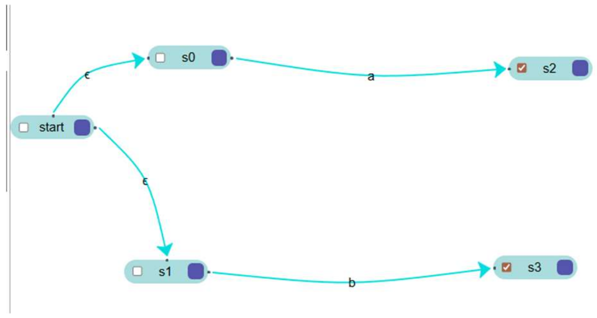
2. If there is a concatenation, then epsilon transition the Final state of the 1st one to the Start state of the 2nd one. And our new Final state will be the Final state of the 2nd one.
   Eg. (ab)



3. If there is Kleene *, then make a new Start state, which will also be the Final state, and epsilon transition new Start state to the old Start state and old Final state to the new Final/Start state.
   Eg. [ab]*



4. If there is an Union, then create a new start state, epsilon transition it to the Start state of both the expression.
   Eg. a U b



c) DFA -> NFA

To convert a DFA to NFA, we need to follow these steps:

1. Find Epsilon closure of the start state, make it a new state in our DFA.
2. Then, we need to figure out where we can go from this state on all the inputs.
3. Again, we will find the epsilon closure for our new states and add them in our DFA.
4. Repeat this process until we aren't transitioning to any further new states.
5. All the states in our DFA having the Final state from our NFA will be marked as the Final states in our DFA.

This process of combining the states with their epsilon closure is the subset construction, which is being carried through out our process.

Q2)

INT

(0x|0X)[0-9a-fA-F]+ (hex):
Hexadecimal Integers start with 0x or 0X, that's why we have contained it at the beginning of the regex. All the hexadecimals can have numbers from 0-9 and then A-F to represent numbers from 10-15, and regex for 0x is followed by regex to match this pattern.

(0)[0-7]+ (Octal):

Octal integers start with a 0, and their range is between 0-7. To represent 8 in octan, we will need to write 071. The regex makes sure that every octal is starting with a 0 and has a following number in the range of 0-7.

(0|[1-9][0-9]*)   (Decimal):

Decimals have a range between 0-10, however they can only be 0. To make sure our compiler doesn't misunderstand them for octal, our regex provide for the option only 0 compared to the option of 00 in Octal to represent the '0'.

FLOAT:
[0-9]+\.[0-9]+ (Standard)

The first part of the regex in this is for a digit, and float must have one or more digits which is indicated by the '+'. And it is followed by a '\.', which is the regex for '.', and that is again followed by a digit. So our Float is like [digit].[digit].

Eg. 10.909

([0-9]+\.[0-9]*|\.[0-9]+)(e|E)[+-]?[0-9]+ (Scientific)

In the case of scientific float, there are 3 cases: 1) [digit].e[+-][digit], 2) .[digit]e[+-][digit], 3) [digit].[digit]e[+-][digit].

In our regex, the first part doesn't has the [0-9]+ as it isn't compulsory for a float to must have a digit after the '.', which satisfies the first case. For the second case, we have a '|' (or), which lets the scanner know that it is okay if a scientific float starts with '.' (dot). And lastly, our regex won't throw an error if it encounters a float that has digits after and before the '.' (dot), which satisfies our third case.

Q3)
The Flex has two built-in variables for this purpose: yylineno and yytext.

yytlineno: This tracks the current line, and it is updated everytime the scanner encounteres '\n' because of our rule in flex(lexical.l).

yytext: This is char* in flex and it is updated everytime a pattern is matched.

So, whenever a lexical error occurs, yylineno keeps the track of the line number and yytext helps to retrieve the error tokens.

Q4) To catch these errors, the flex scanner has 2 features:

Rule Ordering: The Rules for the invalid INT and FLOAT will be placed before the Rules for the valid INT and FLOAT. This helps the scanner to differentiate these with any other Rules.

The Longest Match Principle: The flex Scanner always try to match the longest possible string of characters.

Eg. if a number is 0xZ9 it is an invalid hex, however when the scanner is interpreting it, at the first 0, it will be confused between a hex and an octal, however, as soon as it encounters 'x', it will be sure that the pattern is either for a valid or invalid hex. And we have already encountered the problem of invalid hex by placing its rule before the valid hex. So the scanner will know that 0xZ9 is an invalid hex.

Q5)

Character Streams are grouped into tokens by following the **Principle of Longest Matching**, for example, if scanner encounters something like "if(", at first 'I', it will think of it as an identifier, however it will continue to scan and encounter "if", now it can be matched with either ID or IF Rules. But scanner doesn't stop here and scans the "if(" statement, however this doesn't match any of the rules. Thus, the longest matching string was "if" and it will match with either ID or IF rule depending on their ordering(IF in our case).

Now, in order to output this we will use fprintf() and yytext, fprintf() helps us to output in our favorable format, ex. <INT, int>. And in case if there is a number we need to output, then yytext holds the actual text of the token and we can use it to output the token.

Eg 11.89 fprintf(output_dest, "(Float, %s)\n", yytext) [Line 83 and 106 of lexical.l file]

In order to remove the comments, I have written the rules to do nothing {;} whenever the scanner encounters the single line or multi line comments. [line 17 and 19 of the lexical.l file]