Atul Parida
1006184

50.051 Programming Languages & Concepts

Q1.

A. This is equivalent to a bit shift one position to the left, which means $y = 2x$.

B. This would mean $y = 2x + 1$.

C. The three possible values (states) are 0, 1, and 2 where $x \% 3$ being 0 means it is a multiple of 3.

D. $y = 2x$.
If $x \% 3 = 0$ then $y \% 3$ is also 0.

E. If $x \% 3 = 1$ then $y \% 3 = (x \% 3 + x \% 3) \% 3 = 2 \% 3 = 2$
$x \% 3 = 2 \rightarrow y \% 3 = (x + x) \% 3 = ((x \% 3) + (x \% 3)) \% 3 = (2+2) \% 3 = 4 \% 3 = 1$

F. $x \% 3 = 0 \rightarrow y \% 3 = (2x + 1) \% 3 = (x \% 3 + x \% 3 + 1 \% 3) \% 3 = 1 \% 3 = 1$

G. $x \% 3 = 1 \rightarrow y \% 3 = (2x + 1) \% 3 = (x \% 3 + x \% 3 + 1) \% 3 = (1 + 1 + 1) \% 3 = 3 \% 3 = 0$

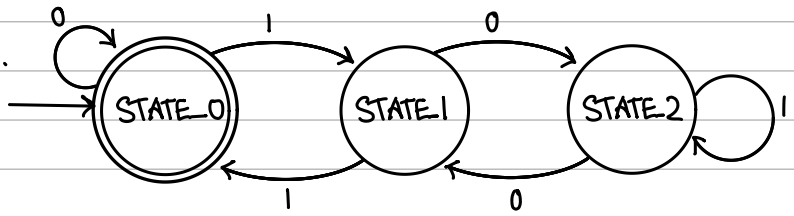
H. Start = State 0 as $0 \% 3 = 0$ (technically a multiple of 3)

I. Stopping state should be 0 as we want to accept multiples of 3.

J. $G = \{STATE\_0, STATE\_1, STATE\_2\}$
$A = \{0, 1\}$

Current	Input	Next
STATE_0	0	STATE_0
STATE_0	1	STATE_1
STATE_1	0	STATE_2
STATE_1	1	STATE_0
STATE_2	0	STATE_1
STATE_2	1	STATE_2

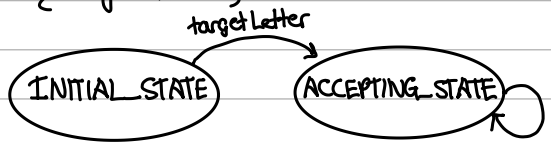
K.



Q2.

A. S = { INITIAL_STATE, ACCEPTING_STATE }

A = { targetLetter }



In this case, targetLetter is 'a'.

B. The accepting state is an absorbing state. No matter what happens input-wise, the FSM will stay in this state provided that the input contains the RegEx target.

C. If the FSM reaches an absorbing state, no further input checks are needed.

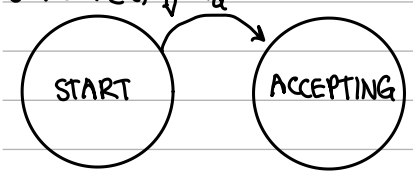
D.

```
int runRegex(FSM* fsm, const char *input_str) {
    int i;
    fsm->currentState = INITIAL_STATE;
    for(i=0; input_str[i] != '\0'; i++) {
        processChar(fsm, input_str[i]);
        if (fsm->currentState == CURRENT_STATE) {
            break;
        }
    }
    return fsm->currentState == ACCEPTING_STATE;
}
```

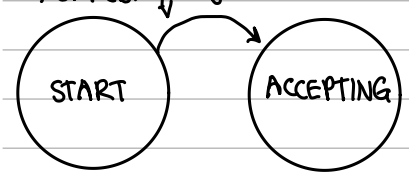
Q3.

A. Acceptable strings for RegEx "ab" is strings with a immediately followed by b. "arbitrary" does not follow this pattern.

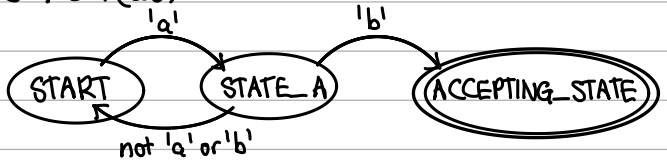
B. FSM (a)



FSM (b)



C. FSM (ab)



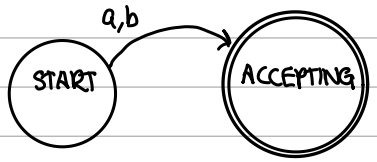
D.

FSM for 'ab' combines the FSM of 'a' and 'b' such that it searches for an 'a' and, if found, transitions to state_A where it searches for 'b'. If neither 'a' or 'b' is the character, then transition back to start. If 'b' is the character, transition to the accepting state.

Q4.

A. Acceptable strings for "(a|b)" are strings with either "a" or "b" in any position individually. "a" and "b" are both acceptable. Strings with 'ab' also pass this RegEx.

B.

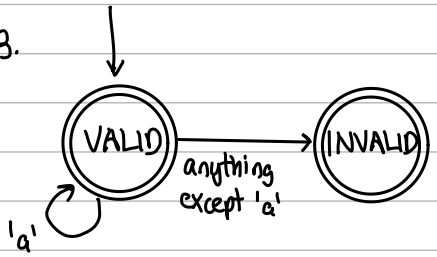


C. The FSM for "a|b" is more flexible than that of "ab", as any input with a,b or both is acceptable.

Q5.

A. Acceptable strings of "^a*$" are strings with 0 or more 'a's ONLY. Empty strings can qualify. "Singapore" would not qualify as it has characters other than a i.e. it does not start & end with a.

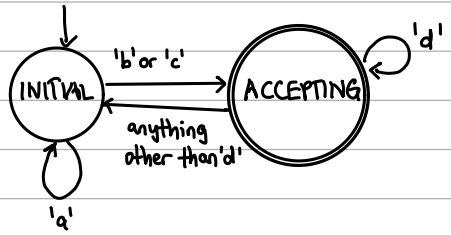
B.



C. The diagram is a simplified version of the 'a' state diagram where it only accepts strings consisting entirely of 'a'. It doesn't run unless the string is all 'a' and stops running if any other character appears, going to an invalid state. Being more restrictive, it operates more simply.

Q6.

A.



B. Initial & accepting states exist as well as transitions dependent on more than one character option – it's the most similar to the Task 5 diagram in structure.

C.

```c
typedef enum {
    INITIAL_STATE_COMBINED,
    ACCEPTING_STATE_COMBINED
} CombinedState;

/* Define Combined FSM Struct, as in previous tasks.  */
typedef struct {
    CombinedState currentState;
    char firstLetter;
    char secondLetter;
    char thirdLetter;
} CombinedFSM;

/*
    Initialize the Combined FSM, as in previous tasks.
    - For simplicity, we will not be storing letters ('a', 'b', 'c' and 'd') in attributes.
*/
void initCombinedFSM(CombinedFSM *combinedFsm) {
    combinedFsm->currentState = INITIAL_STATE_COMBINED;
}

/* Process given input_char for the Combined FSM.  */
void processCharCombined(CombinedFSM *combinedFsm, char input_char) {
    switch (combinedFsm->currentState) {
        case INITIAL_STATE_COMBINED:
            if (input_char == 'a' && combinedFsm->currentState == INITIAL_STATE_COMBINED) {
                combinedFsm->currentState = INITIAL_STATE_COMBINED;
            } else if (input_char == 'b' && combinedFsm->currentState == INITIAL_STATE_COMBINED) {
                combinedFsm->currentState = ACCEPTING_STATE_COMBINED;
            } else if (input_char == 'c' && combinedFsm->currentState == INITIAL_STATE_COMBINED) {
                combinedFsm->currentState = ACCEPTING_STATE_COMBINED;
            } else if (input_char == 'd' && combinedFsm->currentState == ACCEPTING_STATE_COMBINED) {
                combinedFsm->currentState = ACCEPTING_STATE_COMBINED;
            } else {
                combinedFsm->currentState = INITIAL_STATE_COMBINED;
            }
            break;
        case ACCEPTING_STATE_COMBINED:
            if (input_char == 'd') {
                combinedFsm->currentState = ACCEPTING_STATE_COMBINED;
            } else {
                combinedFsm->currentState = INITIAL_STATE_COMBINED;
            }
            break;
    }
}
```

D. According to the given regex:

→ It must start with either 'f' or 'g'.

→ 'f' or 's' may repeat for any number of times.

→ 'h' must be the last character.

Behind the scenes an FSM would likely be generated using flags as outputs if the RegEx is not matched. The expressions would be decomposed into that used in the Lab, as these are Kleene-complete.