50.051 Programming Language Concepts

Simriti Bundhoo 1006281

Lab 2: Report

1 Task 1 – Coding an FSM that checks for multiples of 3

1.1 Preliminary questions

1.1.1 Question 1-A

Converting binary x and y values into their equivalent decimal values:
$x = 10110_2 = 22_{10}$
$y = 101100_2 = 44_{10}$
Upon appending a '0' to the binary value, it doubles. Here, $y = 2x$.

1.1.2 Question 1-B

Swapping the ending 0 with 1:
$y = 101101_2 = 45_{10}$
Upon appending a '1' to the binary value, the value doubles and increments by 1. Here, $y = 2x + 1$.

1.1.3 Question 1-C

For any integer x, the three possible values that $x\%3$ can take are 0, 1 and 2. The value 0 indicates that x is a multiple of three as $x\%3 == 0$ means there are no remainders (hence, x is a multiple of 3).

1.1.4 Question 1-D

If $x\%3 = 0$ and $y = 2x$, then $y\%3 = 0$. For example, when $x = 6$, $y = 12$ and both values are divisible by 3.

1.1.5 Question 1-E

If $x\%3 = 1$ and $y = 2x$, then $y\%3 = 2$ For example, when $x = 4$, $y = 8$. When divided by 3, x gives a remainder of 1 and y gives a remainder of 2.

If $x\%3 = 2$ and $y = 2x$, then $y\%3 = 1$ For example, when $x = 5$, $y = 10$. When divided by 3, x gives a remainder of 2 and y gives a remainder of 1.

1.1.6 Question 1-F

If $x\%3 = 0$ and $y = 2x$, then $y\%3 = 1$. For example, when $x = 3$, $y = 7$. While x is divisible by 3, y gives a remainder of 1.

1.1.7 Question 1-G

If $x\%3 = 1$ and $y = 2x$, then $y\%3 = 0$ For example, when $x = 4$, $y = 9$. While x gives a remainder of 1 , y is divisible by 3.

If $x\%3 = 2$ and $y = 2x$, then $y\%3 = 2$ For example, when $x = 5$, $y = 11$. When divided by 3, both x and y give a remainder of 2.

1.2 Designing the FSM for this task

1.2.1 Question 1-H

Before any character has been scanned, the starting state is State 0.

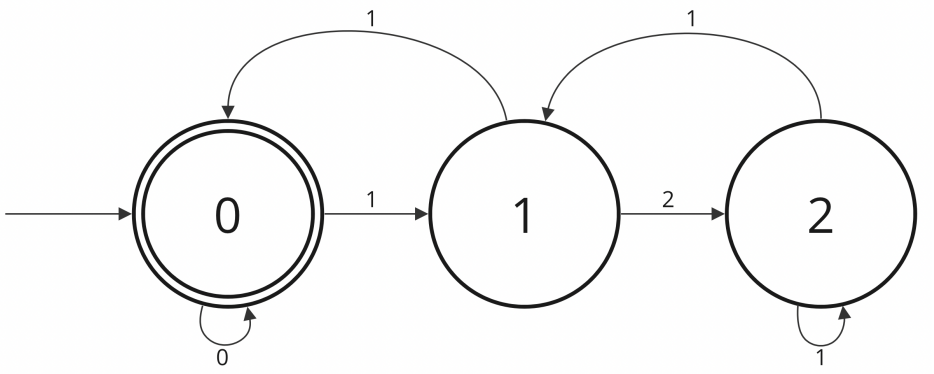
1.2.2 Question 1-I

The stopping state is State 0.

1.2.3 Question 1-J

Current State S	Current Input a	Next State s'	Output Produced y
0	0	0	Is a multiple of 3
0	1	1	Is not a multiple of 3
1	0	2	Is not a multiple of 3
1	1	0	Is not a multiple of 3
2	0	1	Is not a multiple of 3
2	1	2	Is not a multiple of 3

1.2.4 Question 1-K



1.3 Implementing said FSM

1.3.1 Question 1-L

```
void initFSM(FSM *fsm) {
    fsm->currentState = STATE_0;
}
```

1.3.2 Question 1-M

```
void processInput(FSM *fsm, char input)
{
    switch (fsm->currentState) {
        case STATE_0:
            if (input == '0') {
                fsm->currentState = STATE_0;
            } else if (input == '1') {
                fsm->currentState = STATE_1;
            }
            break;

        case STATE_1:
            if (input == '0') {
                fsm->currentState = STATE_2;
            } else if (input == '1') {
                fsm->currentState = STATE_0;
            }
            break;

        case STATE_2:
            if (input == '0') {
```

```
            fsm->currentState = STATE_1;
        } else if (input == '1') {
            fsm->currentState = STATE_2;
        }
        break;

    default:
        printf("Invalid state\n");
        break;
  }
}
```

1.3.3 Question 1-N

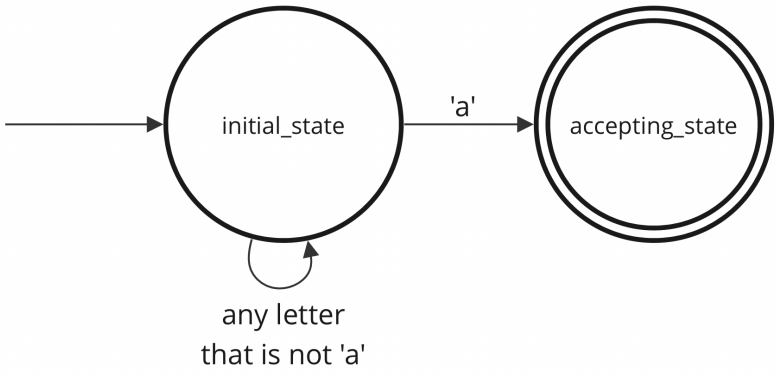
```
int isMultipleOf3(FSM fsm, char *binary) {
    int i;
    for (i = 0; binary[i] != '\0'; i++) {
        processInput(&fsm, binary[i]);
    }

    switch (fsm.currentState) {
        case STATE_0:
            return 1;
        default:
            return 0;
    }
}
```

2 Task 2 – Coding an FSM that implements the RegEx "a"

2.1 Preliminary Questions

2.1.1 Question 2-A



2.1.2 Question 2-B

The accepting_state is an absorbing state while the initial_state is transient state.

2.1.3 Question 2-C

True

2.2 Code study for Task 2

2.2.1 Question 2-D

The added code allows the function to stop processing the input string as soon as an accepting state is reached. This optimisation improves efficiency by avoiding unnecessary iterations through the entire input string once the desired state is achieved.

```
if (fsm->currentState == ACCEPTING_STATE) {
        break;
}
```

3 Task 3 – Coding a concatenation “ab” by reusing the FSMs behind two simple RegEx “a” and “b”

3.1 Preliminary questions

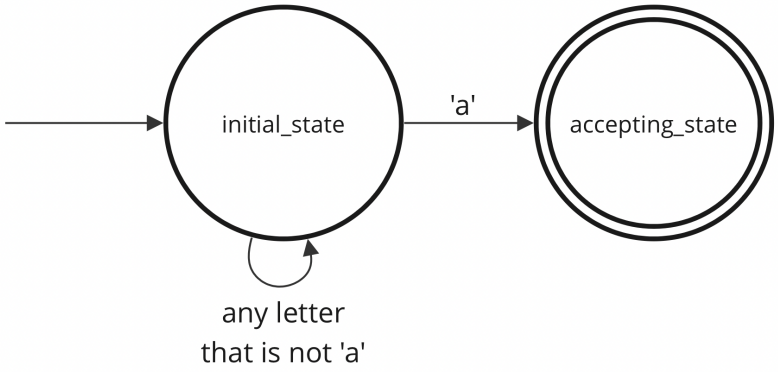
3.1.1 Question 3-A

The acceptable strings for this RegEx are those that have the character ’a’ followed immediately by ’b’. For example, ’about’ will be an acceptable string.

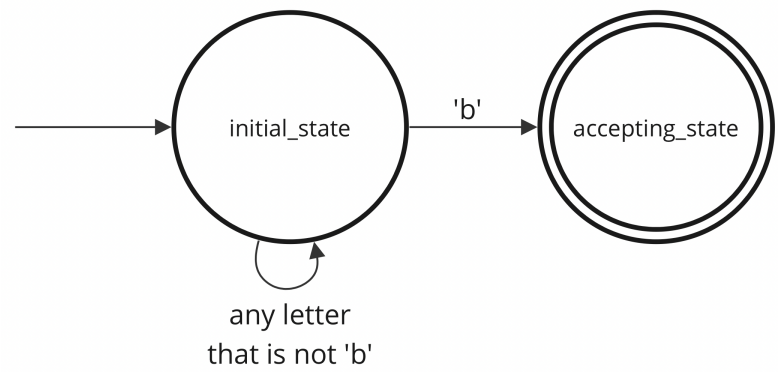
The string ’arbitrary’ is not acceptable as it does not follow the pattern mentioned (’a’ immediately followed by ’b’).

3.1.2 Question 3-B

Regex A

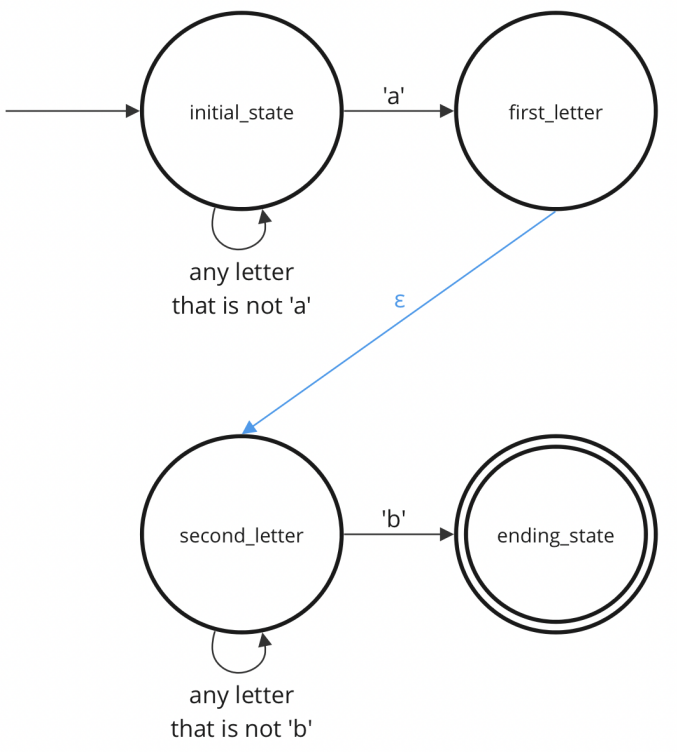


Regex B

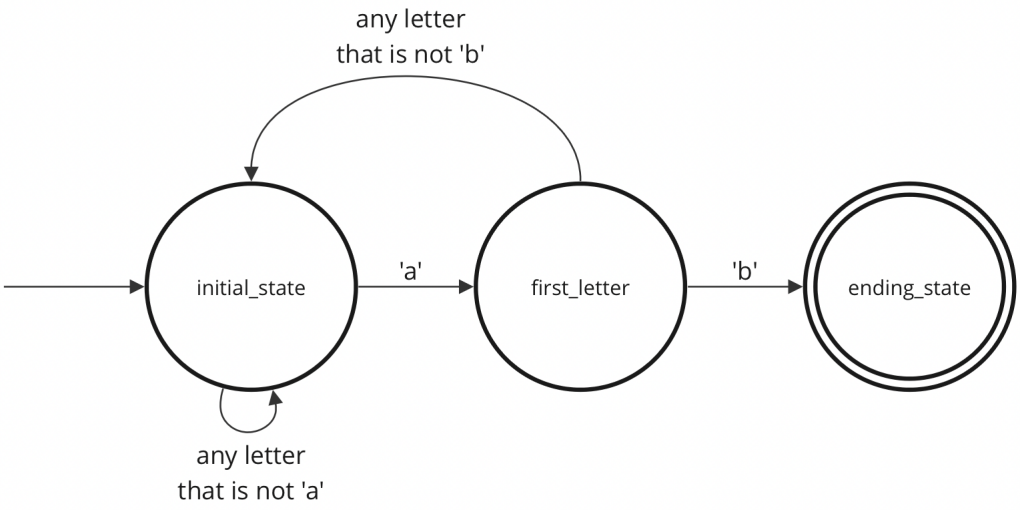


3.1.3 Question 3-C

Non-Deterministic FSM



Deterministic FSM



3.1.4 Question 3-D

- The FSM for 'ab', compared to the FSMs behind 'a' and 'b', focuses on checking the presence of a specific sequence of characters, not checking the presence of only 1 character.
- In the FSM for 'ab', the position of the second letter is very important as 'b' needs to follow 'a' immediately while in the FSMs behind 'a' and 'b', the position of 'a' or 'b' does not matter as long as they appear in the string.

3.2 Implementing the FSM behind the RegEx “ab”

3.2.1 Question 3-E

```
typedef enum {
    INITIAL_STATE_CONCAT,
    ACCEPTING_STATE_CONCAT,
    ENDING_STATE_CONCAT
} ConcatState;
```

3.2.2 Question 3-F

```
void initConcatFSM(ConcatFSM *concatFsm, FSM *fsm1, FSM *fsm2) {
    concatFsm->currentState = INITIAL_STATE_CONCAT;
    concatFsm->firstLetter = fsm1->targetLetter;
    concatFsm->secondLetter = fsm2->targetLetter;
}
```

3.2.3 Question 3-G

```
switch (concatFsm->currentState)
    {
    case INITIAL_STATE_CONCAT:
        if (input_char == concatFsm->firstLetter)
        {
            concatFsm->currentState = ACCEPTING_STATE_CONCAT;
        }
        else
        {
            concatFsm->currentState = INITIAL_STATE_CONCAT;
        }
        break;

    case ACCEPTING_STATE_CONCAT:
        if (input_char == concatFsm->secondLetter)
        {
            concatFsm->currentState = ENDING_STATE_CONCAT;
        }
        else
        {
            concatFsm->currentState = INITIAL_STATE_CONCAT;
        }
        break;

    case ENDING_STATE_CONCAT:
        break;
    }
```

3.2.4 Question 3-H

```
int runRegexConcat(ConcatFSM *concatFsm, const char *str) {
    int i;
    for (i = 0; str[i] != '\0'; i++) {
        processCharConcat(concatFsm, str[i]);
    }
    return concatFsm->currentState == ENDING_STATE_CONCAT;
}
```

4 Task 4 – Coding a choice “a|b” by reusing the FSMs behind two simple RegEx “a” and “b”

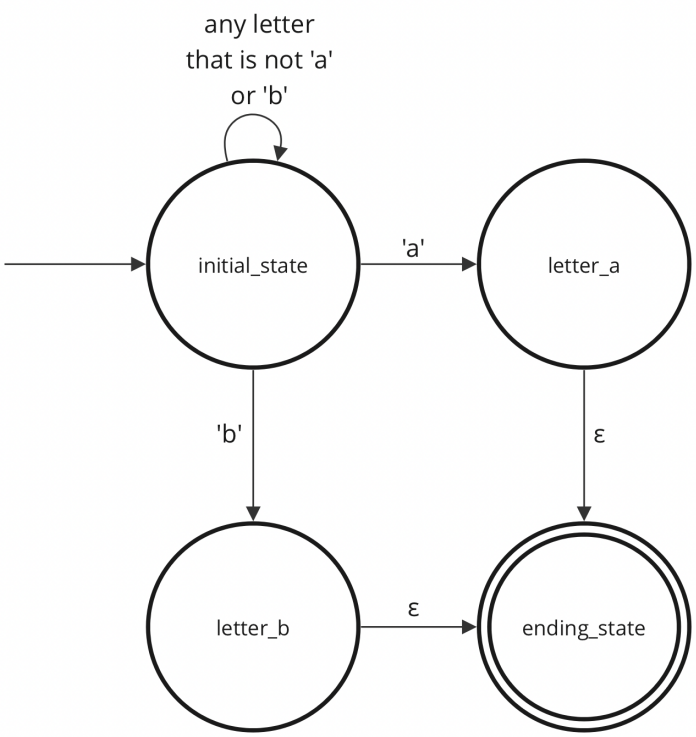
4.1 Preliminary questions

4.1.1 Question 4-A

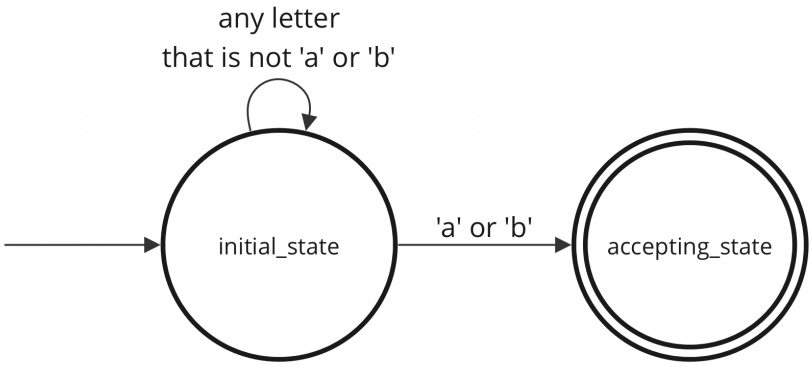
The acceptable strings for this RegEx are those that contain either ’a’, ’b’ or even both. For example: apple, bell, ball, absent.

4.1.2 Question 4-B

Non-Deterministic FSM



Deterministic FSM



4.1.3 Question 4-C

While the FSMs for ’a’ and ’b’ accept only their respective characters, the FSM for ’a|b’ accepts either ’a’ or ’b’. The FSM for ’a|b’ effectively combines the functionality of the individual FSMs for ’a’ and ’b’ into a single FSM that can handle both cases.

4.2 Implementing the FSM behind the RegEx “a|b”

4.2.1 Question 4-D

```
typedef enum {
    CHOICE_INITIAL_STATE,
    CHOICE_ACCEPTING_STATE
} ChoiceState;
```

4.2.2 Question 4-E

```
void initChoiceFSM(ChoiceFSM *choiceFsm, FSM *fsm1, FSM *fsm2) {
    choiceFsm->currentState = CHOICE_INITIAL_STATE;
    choiceFsm->firstChoice = fsm1->targetLetter;
    choiceFsm->secondChoice = fsm2->targetLetter;
}
```

4.2.3 Question 4-F

```
void processCharChoice(ChoiceFSM *choiceFsm, char input_char) {
    switch (choiceFsm->currentState) {
        case CHOICE_INITIAL_STATE:
            if (input_char == choiceFsm->firstChoice || input_char == choiceFsm->secondChoice)
            {
                choiceFsm->currentState = CHOICE_ACCEPTING_STATE;
            }
            else
            {
                choiceFsm->currentState = CHOICE_INITIAL_STATE;
            }
            break;
        case CHOICE_ACCEPTING_STATE:
            break;
    }
}
```

4.2.4 Question 4-G

```
int runRegexChoice(ChoiceFSM *choiceFsm, const char *str) {
    int i;
    for (i = 0; str[i] != '\0'; i++) {
        processCharChoice(choiceFsm, str[i]);
    }
    return choiceFsm->currentState == CHOICE_ACCEPTING_STATE;
}
```

5 Task 5 – Coding the FSM for a Kleene RegEx “^a*$” by reusing the FSMs behind the simple RegEx “a”

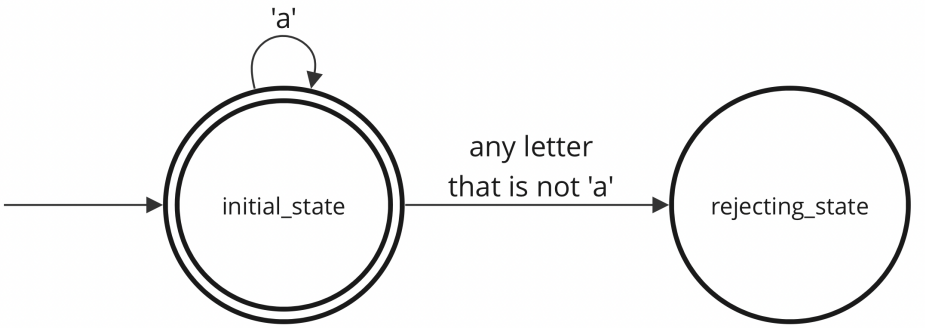
5.1 Preliminary questions

5.1.1 Question 5-A

The acceptable strings for this RegEx are those that contains zero or more occurrences of the character ”a” and only ’a’. For example, ” (empty string), a, aa, aaa.

The string ’Singapore’ would not be acceptable for this regular expression because it contains characters other than ’a’.

5.1.2 Question 5-B



5.1.3 Question 5-C

While the FSM for 'a' accepts only the specific character 'a' from any string, the FSM for '^a*$' accepts any string consisting of zero or more occurrences of 'a' from a string containing only the character 'a'.

5.2 Implementing the FSM behind the RegEx "^a*$"

5.2.1 Question 5-D

```
typedef enum {
    KLEENE_INITIAL_STATE,
    KLEENE_REJECTING_STATE
} KleeneState;
```

5.2.2 Question 5-E

```
void initKleeneFSM(KleeneFSM *kleeneFsm, FSM *fsm) {
    kleeneFsm->currentState = KLEENE_INITIAL_STATE;
    kleeneFsm->targetLetter = fsm->targetLetter;
}
```

5.2.3 Question 5-F

```
void processCharKleene(KleeneFSM *kleeneFsm, char input_char) {
    switch (kleeneFsm->currentState)
    {
    case KLEENE_INITIAL_STATE:
        if (input_char == kleeneFsm->targetLetter)
            kleeneFsm->currentState = KLEENE_INITIAL_STATE;
        else
            kleeneFsm->currentState = KLEENE_REJECTING_STATE;
        break;
    case KLEENE_REJECTING_STATE:
        break;
    }
}
```

5.2.4 Question 5-G

```
int runRegexKleene(KleeneFSM *kleeneFsm, const char *str) {
    int i;
    for (i = 0; str[i] != '\0'; i++) {
        processCharKleene(kleeneFsm, str[i]);
    }
```

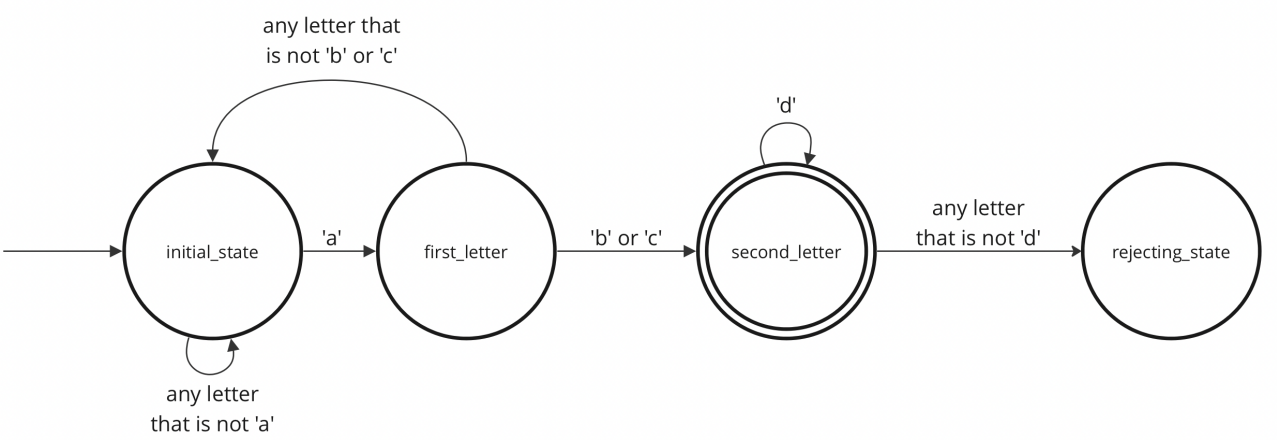
```
    return kleeneFsm->currentState == KLEENE_INITIAL_STATE;
}
```

6 Task 6 – Creating a combined FSM for the combined RegEx "^a(b|c)d*$"

6.1 Question 6-A

The string should be as follows:

- start with with the character "a"
- followed by either "b" or "c"
- followed by zero or more occurrences of the character "d"



6.2 Question 6-B

This state diagram is the combination of all the state diagrams produced in Questions 3-C, 4-B and 5-B.

6.3 Question 6-C

```
/* Define New Enum for Combined FSM */
typedef enum {
    INITIAL_STATE,
    ACCEPTING_STATE_1,
    ACCEPTING_STATE_2,
    REJECTING_STATE
} CombinedState;

/* Define Combined FSM Struct, as in previous tasks. */
typedef struct {
    CombinedState currentState;
    char firstChoice;
    char secondChoiceOption1;
    char secondChoiceOption2;
    char targetLetter;
} 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;
    combinedFsm->firstChoice = 'a';
```

```
    combinedFsm->secondChoiceOption1 = 'b';
    combinedFsm->secondChoiceOption2 = 'c';
    combinedFsm->targetLetter = 'd';
}

/* Process given input_char for the Combined FSM. */
void processCharCombined(CombinedFSM *combinedFsm, char input_char) {
    switch (combinedFsm->currentState)
    {
    case INITIAL_STATE:
        if (input_char == combinedFsm->firstChoice)
        {
            combinedFsm->currentState = ACCEPTING_STATE_1;
        }
        else
        {
            combinedFsm->currentState = INITIAL_STATE;
        }
        break;

    case ACCEPTING_STATE_1:
        if (input_char == combinedFsm->secondChoiceOption1 || input_char == combinedFsm->secondChoiceOption2)
        {
            combinedFsm->currentState = ACCEPTING_STATE_2;
        }
        else
        {
            combinedFsm->currentState = INITIAL_STATE;
        }
        break;

    case ACCEPTING_STATE_2:
        if (input_char == combinedFsm->targetLetter)
        {
            combinedFsm->currentState = ACCEPTING_STATE_2;
        }
        else
        {
            combinedFsm->currentState = REJECTING_STATE;
        }
        break;

    case REJECTING_STATE:
        break;
    }
}

/*
    Running the Combined FSM, as in previous tasks.
    - Simple for loop on all characters as before.
    - Break for loop early on rejection, if you want.
    - Return true if the FSM is in correct accepting state.
 */
int runRegexCombined(CombinedFSM *combinedFsm, const char *str) {
    int i;
    for (i = 0; str[i] != '\0'; i++) {
        processCharCombined(combinedFsm, str[i]);
        if (combinedFsm->currentState == REJECTING_STATE)
        {
            return 0;
        }
    }
    return combinedFsm->currentState == ACCEPTING_STATE_2;
}
```

6.4 Question 6-D

If we call the regcomp() function with the expression "^(f—g)*h+$", the following may be happening behind the scenes:

The function will first by analysing the the RegEx string. It will first identify the '^' as the start of the expression. Then, it will resolve (f|g) to be the string should start with 'f' or 'g'. The string will then contain zero or more occurrences of 'h' and will end with that through the '∗' symbol and the '+' operator. Upon reaching the '$', the function will understand that the RegEx ends here.

Then the function will compare the expression with the RegEx string. It will return an integer value indicating the success or failure of the compilation process. The flags will contain options to specify case sensitivity, extended syntax, or optimisation preferences.

If the expression does not behave as per the RegEx string, the compilation process would fail, and an appropriate error code would be returned.