Programming Language Concepts
Lab 2 Instructions

Y2024 – 22/02/2024, Matthieu DE MARI

Introduction

This PDF file contains the instructions for the Lab 2 of the 50.051 Programming Language Concepts course at the Singapore University of Technology and Design.

It consists of six tasks, which build up progressively. Each task has questions leading to the expected solution, which will have to be coded.

Please submit your assignment as a Zip file. This zip may contain any number of .c files and .h files. For every .c file, there should be a .h file. Keep every file used from the lab template as they will be used for testing your code and as input to your program.

Your assignment should compile using the flags -ansi -pedantic -Wall -Werror.

Deadline: Thursday 13th March 2024, 11.59pm.

Submission should consist of: Your code files (TaskX.c and .h, if any), and a small PDF report, answering questions for each of the six tasks. Will have to be submitted on eDimension.

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

The objective in this first task is to define an FSM that will have the following features:

- Our FSM will receive an input string x, consisting of a sequence of 0 and 1, corresponding to the binary representation of an unsigned integer. The rightmost bit is the least significant bit, and the leftmost bit is the most significant bit. No sign bit is used.
- This FSM will use a number of states that is freely decided by you.
- We can safely assume that the only inputs we will see will be 0 and 1.
- The FSM will use accepting states, but will not use outputs.
- The FSM will only consider as acceptable inputs, any input string x, whose binary representation corresponds to an integer that is a multiple of 3.

Our objective is then twofold: we want to figure out the correct FSM to use first, and then wish to implement it in C. The questions below will guide you through the process.

Preliminary questions

Let us start with some preliminary questions. These will hopefully help us identify the correct FSM to use for the task at hand.

- **Question 1-A:** Consider a string x, consisting of 0 and 1 digits (e.g. 10110). Consider the string y, which is simply defined as the string x, to which a 0 has been concatenated (e.g. is x was 10110, then y is simply 101100). Assuming both x and y correspond to the binary representation of an int, what it the mathematical relationship between x and y?
- **Question 1-B:** How would the relationship between x and y change if we were to append a 1 digit to x instead of appending a 0?

For the next questions, we will reuse the notation %, from Python, which stood for modulus. Modulus a%b can be mathematically defined as the remainder of the integer division between a positive integer a and a strictly positive integer b. For instance, we have 6%3 = 0 and 7%3 = 1.

- **Question 1-C:** For any integer x, what are the three possible values that x%3 can take?
  Which one of these three values would indicate that x is a multiple of three?
- **Question 1-D:** Consider an integer x, such that x%3 = 0. Consider an integer y, defined as the string x to which we have appended a 0, as in Question 1-A.
  What would be the result of y%3 then?
- **Question 1-E:** Consider now that our integer x, such that x%3 = 1. Consider an integer y, defined as the string x to which we have appended a 0, as in Question 1-A. What would be the result of y%3 then? What would be the result of y%3 if our integer x was such that x%3 = 2 instead?
- **Question 1-F:** Consider an integer x, such that x%3 = 0. Consider an integer y, defined as the string x to which we have appended a 1, as in Question 1-B.
  What would be the result of y%3 then?
- **Question 1-G:** Consider now that our integer x, such that x%3 = 1. Consider an integer y, defined as the string x to which we have appended a 1, as in Question 1-B. What would be the result of y%3 then? What would be the result of y%3 if our integer x was such that x%3 = 2 instead?

Designing the FSM for this task

Having answered the previous questions, we would like to design and implement the FSM that will make this Task 1 work. Our FSM will use three states:

- State 0: if the input string x that has been scanned so far consists of a multiple of three (or in other words, if x%3 = 0).
- State 1: if the input string x that has been scanned so far is such that x%3 = 1.
- State 2: if the input string x that has been scanned so far is such that x%3 = 2.

This means that our states set should probably be $S = \{0, 1, 2\}$.

As our input strings x will consist of binary representations of integers, it is safe to assume that our inputs set will be $A = \{0, 1\}$.

Consider the following questions that will help you figure out the remaining elements for this FSM.

- **Question 1-H:** Assuming the empty string stands for the integer number 0, what would therefore be the starting state for our FSM, before any character has been scanned?
- **Question 1-I:** Given that we are interested in accepting inputs if and only if they are multiples of three, what would be the stopping state(s) for this FSM?
- **Question 1-J:** Given your answers to the preliminary questions (1-A to 1-G), can you define the transition table for this FSM?
- **Question 1-K:** Having now defined all the required elements for our FSM, draw its state diagram.

Implementing said FSM

Our final objective is to implement the FSM in C. Please have a look at the Task1.c file in the Code files folder attached to this PDF. As you will see, we are providing an enum and a struct for our FSM. The main function has also been coded for you and includes test cases.

Three functions prototypes are provided but have to be resolved.

- **Question 1-L:** The initFSM() function receives the FSM and should initialize its FSM object, to use the correct starting state identified in Question 1-H. Show the code for this function.
- **Question 1-M:** The processInput() function receives the FSM and a single character (0 or 1) coming from the input string. It should simply update the FSM state following the transition table in Question 1-J. You may choose to implement this using a transition table (as shown in class) or a simple if/else/switch statement of some sort. Show your code for this function.
- **Question 1-N:** The isMultipleOf3() function receives the FSM and the entire input string x. It should process the string, one character at a time, from left-to-right, and update the state of the FSM accordingly. It should return true if the FSM ends in an accepting state, and false otherwise. Note that this function receives fsm, not *fsm as the other functions. Not a typo, we want to test your ability to work with both types of functions!

This concludes Task 1.

Task 2 – Coding an FSM that implements the RegEx “a”

In this task, we will look at the implementation of the FSM, hiding behind the RegEx “a”, which accepts strings if and only if they contain the letter “a”. The code implementation will be provided and for you to study.

Preliminary questions

Let us first focus on the FSM behind this RegEx “a”.

- **Question 2-A:** What is the FSM hiding behind the RegEx “a”. Show it basic elements: states, inputs, transition table and state diagram. The FSM should use stopping states and no outputs.
- **Question 2-B:** We say that the state of an FSM is an absorbing state, if and only if, this state is impossible to leave after it has been reached.
  Conversely, we call transient state, a state that is not absorbing.
  For each of the states of your FSM, which ones are absorbing? Which ones are transient?
- **Question 2-C:** True or False – If the FSM reaches an absorbing state, we may stop scanning for the remaining inputs in our input string, interrupt the FSM and produce a decision.

Code study for Task 2

Have a look at the code in Task2.c, it contains the code describing the FSM behind the RegEx “a”. You might want to study the code, as it will be reused in the next coming tasks.

- **Question 2-D:** How would you modify the code in the function runRegex() to match your answer to Question 2-C? Show the code in your report.

This concludes Task 2.

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

Our objective in this Task is to code the FSM behind the RegEx “ab” following from the FSMs for two simple RegEx “a” and “b”, whose implementation would be based on the code shown in Task 2.

Preliminary questions

Let us first discuss the task at hand, by answering the questions below.

- **Question 3-A:** What are acceptable strings for the RegEx “ab”? Would the string “arbitrary” be acceptable for this RegEx?
- **Question 3-B:** Draw the state diagrams for both FSM implementing the RegEx “a” and “b”. You may reuse the result from Question 2-A.
- **Question 3-C:** How would you represent the state diagram for the deterministic version of the FSM implementing the RegEx “ab”? It should not contain any ϵ transitions. You may start from the non-deterministic FSM and turn it into its deterministic equivalent by reorganizing the states and transitions in a certain way, as discussed in class. Show the state diagram for your deterministic FSM.
- **Question 3-D:** Having figured out the FSM for “ab”, how would you describe it, if you were to compare it to the two FSMs behind “a” and “b”?

Implementing the FSM behind the RegEx “ab”

Having figured out the FSM behind this RegEx “ab”, our objective is now to code the FSM behind the RegEx “ab” following from the FSMs for two simple RegEx “a” and “b”. Open the code in Task3.c. You will recognize the code from Task 2, and other incomplete prototypes of functions.

- **Question 3-E:** The enum ConcatState will contain the states for the FSM behind the RegEx “ab”. Show your code for this enum.

Our FSM for the RegEx “ab” will be implemented in the struct ConcatFSM. This struct has three attributes, namely:

- currentState: Current state of the ConcatFSM.
- firstLetter: First letter to search for.
- secondLetter: Second letter to search for.

This struct is then reused in three functions: initConcatFSM(), processCharConcat() and runRegexConcat(), which follow the same intuition as the functions initFSM(), processChar() and runRegex() in Task 2. Their implementation is required for this task.

- **Question 3-F:** The function initConcatFSM() receives the ConcatFSM, as well as the FSMs behind “a” and “b”. It should update the attributes of the ConcatFSM, as described above, initializing the

ConcatFSM in the correct starting state and drawing letters information from the FSM objects, accordingly. Show your code in your report.

- **Question 3-G:** The function processCharConcat() receives our ConcatFSM and a single character. It should update the state of the ConcatFSM, according to the logic you have identified in Question 3-C. You may use a transition table or a simple if/else/switch statement, as you prefer. Show your code in your report.
- **Question 3-H:** Finally, the runRegexConcat() function receives the ConcatFSM object, after it has been initialized, and the complete input string that needs to be processed. It should use a for loop to process each character, one character at a time. It will return true if the FSM for “ab” ends in an accepting state, and false otherwise. Show your code for this function.

The main() function should remain untouched, as it contains the test cases needed for this task. Feel free to use it to control if your code is working as expected.

This concludes Task 3.

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

Our objective in this task is to code the FSM behind the RegEx “a|b” following from the FSMs for two simple RegEx “a” and “b”, whose implementation would be based on the code shown in Task 2. In a sense, it will be very similar to Task 3.

Preliminary questions

Let us first discuss the task at hand, by answering the questions below.

- **Question 4-A:** What are acceptable strings for the RegEx “a|b”?
- **Question 4-B:** How would you represent the state diagram for the deterministic version of the FSM implementing the RegEx “a|b”? It should not contain any ϵ transitions. You may start from the non-deterministic FSM and turn it into its deterministic equivalent by reorganizing the states and transitions in a certain way, as discussed in class. Show the state diagram for your deterministic FSM.
- **Question 4-C:** Having figured out the FSM for “a|b”, how would you describe it, if you were to compare it to the two FSMs behind “a” and “b”?

Implementing the FSM behind the RegEx “ab”

Have a look at the code in the Task4.c file. As in Task 3, it consists of incomplete code, describing the FSM behind the RegEx “a|b” this time.

- **Question 4-D:** The code for the enum ChoiceState is missing. Show your code.
- **Question 4-E:** The function initChoiceFSM() receives the ChoiceFSM, as well as the FSMs behind “a” and “b”. It should update the attributes of the ChoiceFSM, as described in the code comments above, initializing the ChoiceFSM in the correct starting state and drawing letters information from the FSM objects, accordingly. Show your code in your report.
- **Question 4-F:** The function processCharChoice() receives our ChoiceFSM and a single character. It should update the state of the ChoiceFSM, according to the logic you have identified in Question 4-B. You may use a transition table or a simple if/else/switch statement, as you prefer. Show your code in your report.
- **Question 4-G:** Finally, the runRegexChoice() function receives the ChoiceFSM object, after it has been initialized, and the complete input string that needs to be processed. It should use a for loop to process each character, one character at a time. It will return true if the FSM for “a|b” ends in an accepting state, and false otherwise. Show your code for this function.

The main() function should remain untouched, as it contains the test cases needed for this task. Feel free to use it to control if your code is working as expected.

This concludes Task 4.

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

Our objective in this task is to code the FSM behind the RegEx "^a*$" following from the FSMs for the simple RegEx "a", whose implementation would be based on the code shown in Task 2. In a sense, it will be very similar to Tasks 3 and 4.

Preliminary questions

Let us first discuss the task at hand, by answering the questions below.

- **Question 5-A:** What are acceptable strings for the RegEx "^a*$"? Would the string "Singapore" be acceptable for this RegEx?
- **Question 5-B:** How would you represent the state diagram for the deterministic version of the FSM implementing the RegEx "^a*$"? It should not contain any ϵ transitions.
- **Question 5-C:** Having figured out the FSM for "^a*$", how would you describe it, if you were to compare it to the FSMs behind "a"?

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

Have a look at the code in the Task5.c file. As in Tasks 3 and 4, it consists of incomplete code, describing the FSM behind the RegEx "^a*$" this time.

- **Question 5-D:** The code for the enum KleeneState is missing. Show your code.
- **Question 5-E:** The function initKleeneFSM () receives the KleeneFSM, as well as the FSM behind the RegEx "a". It should update the attributes of the KleeneFSM, as described in the code comment above, initializing the KleeneFSM in the correct starting state and drawing letters information from the FSM objects, accordingly. Show your code in your report.
- **Question 5-F:** The function processCharKleene () receives our KleeneFSM and a single character. It should update the state of the KleeneFSM, according to the logic you have identified in Question 5-B. You may use a transition table or a simple if/else/switch statement, as you prefer. Show your code in your report.
- **Question 5-G:** Finally, the runRegexKleene () function receives the KleeneFSM object, after it has been initialized, and the complete input string that needs to be processed. It should use a for loop to process each character, one character at a time. It will return true if the FSM for "^a*$" ends in an accepting state, and false otherwise. Show your code for this function.

The main() function should remain untouched, as it contains the test cases needed for this task. Feel free to use it to control if your code is working as expected.

This concludes Task 5.

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

In this final Task, we will implement the FSM behind the RegEx "^a(b|c)d*$".

- **Question 6-A:** Show the state diagram for the deterministic FSM that implements the RegEx "^a(b|c)d*$". Describe all elements of this FSM.
- **Question 6-B:** How does this state diagram resemble to the state diagrams you have produced in Questions 3-C, 4-B and 5-B?
- **Question 6-C:** Have a look at the code in Task6.c. We have provided a few function prototypes, but code is missing in many parts. We will have to implement the FSM behind the RegEx "^a(b|c)d*$". In this Task, we will not generate simple FSMs for "a", "b", "c" and "d" and reuse them as in the previous tasks. Show your code in your report. This task is less guided on purpose but should more or less follow the same intuition as in Tasks 3, 4, and 5.
- **Question 6-D:** As we have seen in class, we often use the regex.h library to create our RegEx objects. The first operation is always regcomp(&regex, expression, flags), where regex is our RegEx object, expression consists of a string describing the RegEx (e.g. "^a(b|c)d*$") and flags is a collection of flags to be used during the RegEx compilation. Having now gone through the Lab 2, what do you suspect would happen behind the scenes if we were to run the function regcomp(), using the RegEx expression "^(f|g)*h+$"?

This concludes Task 6 and Lab 2.