Homework – Weeks 8-10 Concepts Finite State Machines, Regular Expressions and Lexical Analysis

50.051 Programming Language Concepts

Introduction

This homework has three tasks.

This is an individual assignment.

You are expected to answer all questions and assemble your answers in a small PDF.

Some tasks may require a bit of coding. You will be expected to show your main.c file along with the PDF report, zipping both your PDF and C files together before submitting on eDimension.

The deadline for this homework is **April 11th, 2023**.

Task 1: Drawing and Coding a simple FSM.

We would like to write a finite state machine with stopping/accepting states.

This FSM will receive an input string x, consisting only of 0s and 1s, e.g. “1001”, “01010”, etc.

This FSM should consider as acceptable input strings x, any string x, which only contains even numbers of consecutive zeroes. Below are some examples of acceptable and not acceptable strings.

Input string x	Length of consecutive zeroes blocks	Acceptable input?
100	**2 (even)**	**Yes**
1001000011	**2 (even), 4 (even)**	**Yes**
111	**0 (even)**	**Yes**
10	**1 (odd, problem)**	**No**
10010001100	**2 (even, ok), 3 (odd, problem), 2 (even, ok)**	**No**

A. According to the logic of the FSM, should the empty string be considered an acceptable input?

B. Design a finite state machine using the smallest number of states possible to determine if a given input string x is acceptable or not. Remember that this FSM should not produce outputs during transitions and should have a single accepting state.
(Hint: Consider how the FSM can transition between states based on input characters)

C. Write a C program implementing the FSM designed in part B. The program should have the following features:
- The main function should prompt the user to enter an input string to be tested using scanf(). Assume that the user will only input strings consisting of 0s and 1s.
- Upon receiving the input string, the FSM should run and print either "The string is acceptable" or "The string is not acceptable."
- Implement the transition logic using a transition table, rather than if/else statements. Points will be deducted for using if/else statements in place of a transition table.
- You may refer to code examples from Weeks 8 and 9 for inspiration on implementing the FSM.
- Only use libraries introduced during Weeks 8 and 9; do not use external libraries to implement the FSM.

Remember to submit your code on eDimension in a zip file containing your PDF report and any C files required for this task.

Task 2: Some RegEx practice

Let us practice our RegEx, by answering the following questions.

A. You are working for a survey company in Singapore. You will be sending forms to a vast quantity of SG-based users, asking for their names and phone numbers, along with many survey questions regarding various topics. Every survey form will require the participants to enter their phone number, as a string of digits. Some algorithm will be used to remove all the whitespaces in the string entered by the user for you, and you should expect one of the three string formats below after the string has been cleaned.
   - *"63036600",*
   - *"+6563036600",*
   - *"006563036600".*

   We expect the users to enter only digits, no phone numbers with more than 8 digits (unless country codes are used), and no country codes other than +65 or 0065.

   Can you write a RegEx for checking if the phone number consists of eight digit and maybe a country code +65 or 0065 that might have been added to the phone number?
   - A phone number not using 8 digits (before a country code is added) shall be deemed invalid.
   - Any other country code than +65 or 0065 should be rejected.
   - We shall only check for valid characters: it is ok if our RegEx does not catch a phone number +6500000000, which is obviously fake.

B. Let us consider strings consisting of 0s and 1s only. We would like to check for strings that have the same digit in their first, third, fifth, etc. location. Have a look at the table below for some examples of acceptable and not acceptable strings.

String	Acceptable?
1	**Yes**
101	**Yes**
01000101	**Yes**
100110	**No**
101011100	**No**

Your friend Chris claims that it is impossible to describe this pattern using a RegEx. Do you agree with him, or can you provide a RegEx that works for this task?

For both questions in this Task 2, you may use the provided regex_code.c file to test your RegEx on examples of acceptable and unacceptable strings.

Task 3: A Tokenizer for a LOLBOT Code

Consider a robot, that has six possible actions:

- Start: the robot will start taking movement instructions.
- Forward: the robot will move in the direction it is facing for the next second.
- Backward: the robot will move backwards, in the opposite direction it is facing for the next second.
- Right: the robot will move in the direction, 90° to the right of the direction it is facing for the next second.
- Left: the robot will move in the direction, 90° to the left of the direction it is facing for the next second.
- Stop: the robot will stop moving and taking movement instructions.

For some crazy stupid reasons, the creators of this robot have decided that it can only be piloted, by entering instructions in a source.lolbot file, written in LOLBOT 1.0 language.

The rules of the language are as follows and relies on lexemes written in uppercase letters.

- The code should always begin with the instruction HELLOROBOT, so that the robot will execute the Start action.
- The code should always end with the instruction KTHXBYE, which will tell the robot to execute the Stop action.
- In between, four possible instructions can be written.
    - LESSGO: this instruction tells the robot to execute the action Forward and move in the direction it is facing for the next second.
    - BAKBAKBAK: this instruction tells the robot to execute the action Backward and move backwards, in the opposite direction it is facing for the next second.
    - GORITE: this instruction tells the robot to execute the action Right and move in the direction, 90° to the right of the direction it is facing for the next second.
    - MOOVLEFT: this last instruction tells the robot to execute the action Left and move in the direction, 90° to the left of the direction it is facing for the next second.

We assume that all instructions in the source.lolbot file will consist of combinations involving the above six instructions, and will be separated by whitespaces or \n sybmols, which will allow the tokenizer to produce lexemes very simply. A possible example of a source.lolbot file is shown below.

```
HELLOROBOT

LESSGO LESSGO MOOVLEFT BAKBAKBAK

GORITE

KTHXBYE
```

This source.lolbot file will then be read by a compiler, whose first step is to extract the instructions that have been entered by the user and translate them into tokens objects.

The code we have shown above will then be tokenized as

```
TOKEN(TokenType = START_TOKEN, lexeme = “HELLOROBOT”)

TOKEN(TokenType = FORWARD_TOKEN, lexeme = “LESSGO”)

TOKEN(TokenType = FORWARD_TOKEN, lexeme = “LESSGO ”)

TOKEN(TokenType = LEFT_TOKEN, lexeme = “MOOVLEFT”)

TOKEN(TokenType = BACKWARD_TOKEN, lexeme = “BAKBAKBAK”)

TOKEN(TokenType = RIGHT_TOKEN, lexeme = “GORITE”)

TOKEN(TokenType = STOP_TOKEN, lexeme = “KTHXBYE”)
```

A. Our language can be decomposed into 6 types of tokens. Out of the five big families of tokens (being keywords, identifiers, literals, operators and punctuation), which one do our six lexemes (HELLOROBOT, LESSGO, MOOVLEFT, BAKBAKBAK, GORITE and KTHXBYE) seem to fall into?

B. Does the Tokenization require any RegEx to compile my LOLBOT language?

C. Using the template provided, can you complete the Tokenizer to be used to compile our LOLBOT 1.0 language?
It should be written in C, not C++, and should compile using a simple GCC compiler.

   Feel free to have a look at the template code provided.
   Several features are currently missing:
   - The read_file() function content is missing. It should read the file given in filename and return a string of code corresponding to the content of the file.
   - The split_lexemes() function content is missing. It sould read the string of code and split the lexemes using white spaces and \n symbols as separators. It should return a struct LexemeArray, described above.
   - The create_token() function content is missing. This function should check for a token type on the given lexeme and create a Token object with the correct type and lexeme.
   - The print_token() function is incomplete. It should display something else.
   - The main() function will have to assemble all functions we defined earlier in a certain way. The test case is simple, it consists of the code below, that has been set in the source.lolbot text file.

```
HELLOROBOT

LESSGO LESSGO MOOVLEFT BAKBAKBAK

GORITE

KTHXBYE
```

And it should produce the following print upon execution.

```
TOKEN(TokenType = START_TOKEN, lexeme = “HELLOROBOT”)

TOKEN(TokenType = FORWARD_TOKEN, lexeme = “LESSGO”)

TOKEN(TokenType = FORWARD_TOKEN, lexeme = “LESSGO ”)

TOKEN(TokenType = LEFT_TOKEN, lexeme = “MOOVLEFT”)

TOKEN(TokenType = BACKWARD_TOKEN, lexeme = “BAKBAKBAK”)

TOKEN(TokenType = RIGHT_TOKEN, lexeme = “GORITE”)

TOKEN(TokenType = STOP_TOKEN, lexeme = “KTHXBYE”)
```

As an additional note, we will investigate a v1.1 of the LOLBOT language in Task 4, but the features for this 1.1 version are not to be coded for this Task 3.C. Remember to submit your code on eDimension in a zip file containing your PDF report and any C files required for this task.

Task 4: A Tokenizer for a LOLBOT Code, version 1.1

The creators of the code have decided to produce a 1.1 version of the LOLBOT language. It can still use all six commands from earlier, but they have added a new command, shown below.

```
HELLOROBOT

LESSGO

KEEPGOIN 007

KTHXBYE
```

The KEEPGOIN command will repeat the last instruction (in our example above, that would be LESSGO), for a given number of times being specified using one to three digits, entered on the same line, after the KEEPGOIN instruction. In our case, the code above will have the robot move forward for 8 seconds: 1 second because of the LESSGO instruction and 7 more seconds following from the KEEPGOIN 007 one.

A. Out of the five big families of tokens (being keywords, identifiers, literals, operators and punctuation), which one does the lexeme 007 seem to fall into?

B. In your opinion, what should be the RegEx that the compiler should use to check that digits appearing in the code, always come in groups of one to three digits?

The code below is obviously faulty for many reasons.

```
HELLOROBOT

LESSGO

KEEPGOIN 0003

BAKBAKBAK

KEEPGOIN GORITE 004

KTHXBYE

MOOVLEFT
```

C. Can you spot all the problems with this code?

D. For each of the errors you spotted in the previous question, will those errors prevent the Tokenizer from doing its job?
Or is a different part of the compiler supposed to catch each of these errors?
If so, which part of the compiler is responsible for that?

No coding is expected for this final task.

Final disclaimer

This last task was hugely inspired by the LOLCODE language.
It is a very popular esoteric programming language.
Learn more about these, here:

https://en.wikipedia.org/wiki/Esoteric_programming_language

https://github.com/justinmeza/lolcode-spec/blob/master/v1.2/lolcode-spec-v1.2.md