CS 3240 : Languages and Computation Course Mini-Project

Due date: April 20th

Guidelines:
1. This project has two phases is due on April 20th by 5pm.
2. There will be no extensions.
3. You will work in groups of three
4. Each group should submit a report and source code. If multiple source files are used, they must be tarred along with the makefile.
5. You can program in C, C++ or Java. Do not use tools (like lex and yacc) or the standard template library -- such solutions will get ZERO points.
6. Code should be properly documented with meaningful variable and function names. Short elegant code is preferred.
7. You will find the course slides on DFA/NFA/scanner/recursive descent parser useful. In addition, read the Louden book for Tiny language discussion.
8. This project is worth 100 points.
9. Provide instructions about how to compile and execute your files along with test inputs .

Objective: Consider a language called **Micro-AWK** which is a pseudo subset of the AWK scripting language. Scripts for this language process text input based on a collection of **patterns** and corresponding **actions**. Each pattern is a regular expression which is used to test a line of input. If the line of input is matched by the regular expression, the corresponding action is executed. The following is an example of a Micro-AWK script:

```
BEGIN{print("--begin output--")}
     a(a|b|c)* {substring(1,LAST); insert(END,a)}
     b(a|b|c)* {replace(a,b)}
     c(a|b|c)* {replace(a,c)}
               {print(LINE)}
END{print("--end output--")}
```

BEGIN and END are special patterns which are used to denote actions which should be executed before and after all other lines are processed (in other words, before and after the entire file/text).

An action without a pattern indicates an action that should be performed for every string.

The action **print** accepts either a string literal or the variable LINE (representing the line being modified).

The other available functions are **substring, replace, insert** and **remove.** These functions modify the current line.

substring(x,y) replaces LINE with the substring of LINE from index x (inclusive) to index y (exclusive). The special value END may be used to represent the *end* of LINE (not the last character!).

replace(a,b) modifies LINE by replacing all occurrences of **a** with the character **b**.

insert(x,a) inserts the character **a** before index **x**. If **x** is greater than END, the character should simply

be inserted at the end of LINE.

remove(a) removes all occurrences of the character **a** from LINE.

Example:
Given the above script and the following input file:

```
aabbbbababc
abbbcbcbcabcb
bbcaaaaacccb
caaabaaab
baa
```

the output would be

```
--begin output--
abbbbababca
bbbcbcbcbbcbb
bbcbbbbbcccb
ccccbcccb
bbb
--end output--
```

For this project, you will be developing a interpreter for the Micro-AWK language which does the following:

1. As input, your interpreter should accept a filename representing a Micro-AWK script and either a filename representing the text file to be processed or the actual text. If a filename is used as input, you should use the **-f** flag (additional option flags will be used as described later).

   Examples:

   $./myinterpreter [options] my_script.mawk -f my_input.data
   $./myinterpreter [options] my_script.mawk “abababababbbbabab”

   After reading its input, your interpreter should load the Micro-AWK script, parse it using a recursive decent parser and—if the script is syntactically correct—create an **abstract syntax tree** which represents the script. In order to do this, you will first need to develop a valid grammar for the language described above as well as the necessary lexical classes.

   In case a syntax error is detected, it should be notified to the user by generating the dump of the current partial AST that is built for the current sentence along with the token that has produced the error. The parser should be able to continue discarding the remainder of the current erroneous sentence and parse the remainder of Micro program. Thus, this phase should be able to catch and report multiple errors in a given Micro program. Such error reporting is required and is a part of design requirements for this project. The format for reporting an AST is (root-label label-of-1st-left-child label-of-2nd-left-child label-of-3rd-left-child ...) .

   If no syntax errors are detected, the interpreter should output the resulting AST if given the optional **-ast** flag.

2. For syntactically correct scripts, an evaluator should be developed which can walk the AST to perform the actual script interpretation. As a part of evaluation process, it will be necessary to test whether a given line of input is matched by a given pattern to determine whether or not the

corresponding action should be executed.

3. Basically, for each pattern/action pair in the Micro-AWK script, you will need to do the following:

    1. Parse the regular expression using the given grammar. Again, you should develop a scanner and recursive decent parser to generate an AST, and an evaluator to process the AST. Also, syntactic errors should be noted as described above and the final AST should be output if the **-ast** flag is set.:

        L → L1
        L1 → (L)
        <prim> → a
        <prim> → b
        <prim> → c
        L1 → <prim>*
        L1 → <prim>+
        L1 → <prim>?
        L1 → <prim>
        L → L1<L1tail>
        <L1tail> → | L<L1tail>
        <L1tail> → ε
        L → L1<L2tail>
            . L<L2tail>
        <L2tail> →
        <L2tail> → ε

    2. After the AST is created, the evaluator should walk the AST and use the algorithms described in class and the textbook to convert the regular expression to an equivalent NFA. The NFA graph should be represented as an adjacency matrix (see below). The interpreter should output the resulting adjacency matrix if the **-nfa** flag is given.

    3. Finally, use the algorithms discussed in class and the textbook to convert this NFA to an equivalent DFA (again, represented as an adjaceny matrix). If the **-dfa** flag is given, then the interpreter should output this final DFA

    Once your interpreter has converted the pattern to a DFA, a line of input can be fed to the DFA to test whether it exists in the language represented by the regular expression and, subsequently, whether the corresponding action should be executed.

4. If the **-debug** flag is set, your interpreter should step through the evaluation process after each pattern/action pair has been interpreted. Before any pairs are executed, the current line of input should be printed. After each pair has been interpreted, the modified text should be printed. When printing this output, please print using the following format:

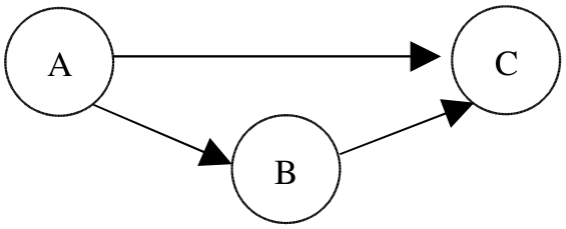
    action : string

    Also, when in debug mode, please suppress all output from the script (the debugger output will be sufficient to show that the output would have occurred)!

Notes (This section may be modified over time!):

Adjaceny Matrix:

Any NFA is a directed graph. A directed graph G consists of a set of nodes (in our case states) and directed edges (in our case, transitions). For example, in the graph below, A,B,C are nodes and 1,2,3 are edges



Any directed graph can be represented by an adjacency matrix. For example, the matrix below represents the graph. Since edge "1" connects A to B, there is a "1" in the row corresponding to "A" and the column corresponding to "B".

	A	B	C
A		1	3
B			2
C			

Similarly an NFA can be represented by an adjacency matrix. Note that more than one element can be present in a cell. For example, in the NFA if the edge from A to B is labeled a,b then you would have both "a" and "b" in the corresponding cell.