Assignment 3 - Parsing Using a Context Free Grammar
Due Friday, November 5, 2023

Given the context free grammar for regular expressions, write a recursive descent parser.

1. The Context Free Grammar
Implement the parser using the following grammar. Each rule will be replaced by a recursive function call.

The grammar is written in Backus–Naur form (BNF). The production rules are written differently from the CFG rules. Variables appear inside pairs of greater > and less than signs <, terminals can appear within double quotes "..." when needed for clarity, the arrow operation is replaced with the ::= symbol.

The CFG grammar for regular expressions is:

```
<regexp> ::= <concat>
<concat> ::= <term><concat> | <term> | <endofline>
<term> ::= <star> | <element>
<star> ::= <element>*
<element> ::= <group> | <char>
<group> ::= (<regexp>)
<char> ::= <alphanum> | <symbol> | <white>
<alphanum> ::= A | B | C | ... | Z |
               a | b | c | ... | z |
               0 | 1 | 2 | ... | 9
<symbol> ::= ! | " | # | $ | % | & | ' | + | , | - | . |
             / | : | ; | < | = | > | ? | @ | [ | ] | ^ |
             _ | ` | { | } | ~ | <sp> | <metachars>
<sp> ::= " "
<metachar> ::= \ | "|" |  <white>
<white> ::= <tab> | <vtab> | <nline>
<tab> ::= '\t'
<vtab> ::= '\v'
<nline> ::= '\n'
<endofline> ::= '\0'
```

2. Input
The input for the program will be a file containing the regular expression. The regular expression can contain any of the terminals listed in the grammar and the brackets ().

Escape characters will appear as their single ASCII character equivalents. For example, the newline is represented as \n in the grammar but it will appear as the single character '\n' in the file. Similarly, the tab appears as \t in the grammar but it will be a tab in the input file.

There can be newlines within the regular expression.

Assume the regular expression will not be more than 1000 characters in length.

3. Output

Add your parser to the starting code and use the print() function to display the parse tree.

There are several global variables in the code that may be useful to draw the tree. You can use these variable if you wish. You cannot add global more variables.

Print out the names of the variables for the rules that do not directly represent terminals. These are regexp, concat, term, star, element, group, and char. These are the names that should appear in the output. If a terminal matches a character in the regular expression then print match to the screen. If the terminal does not match the current character in the regex then print fail.

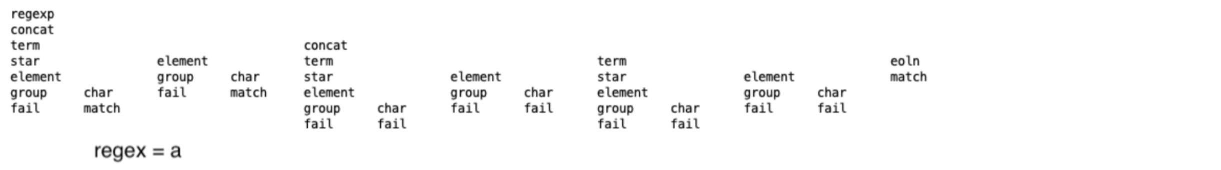
Calling a function that represents a grammar replacement rule increases the depth of the tree. Moving to each replacement rule after the first in an AND (e.g. in <term><concat> it would be <concat>) or and OR (e.g. in <star> | <element> it would be <element>) increases the width of the tree. Normally, depth can increase by 1 per replacement and with can increase by 10 per replacement.

The output when a terminal matches a character in the regex string should be "match" or "fail" (without the double quotes). Only one of these strings should be printed for each character in the regular expression. This means the bottom most element of each branch will end in either the string "pass" or "fail". These will always appear at the bottom of a column (this represents an end of a branch).

Each time you match a character in the regex to a terminal you should advance the position marker in the regex. Do not do this for the end of line character. It needs to be the last character in the string so it can be matched when at the end of the tree traversal.

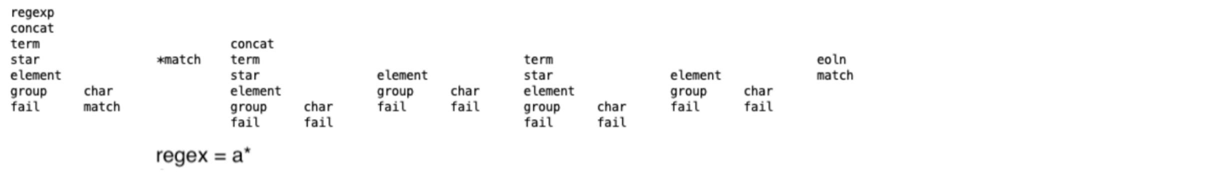
The starting code will display the first variable <regex> on the screen when it is run. Add recursive functions for the other rules.

4. Sample Output



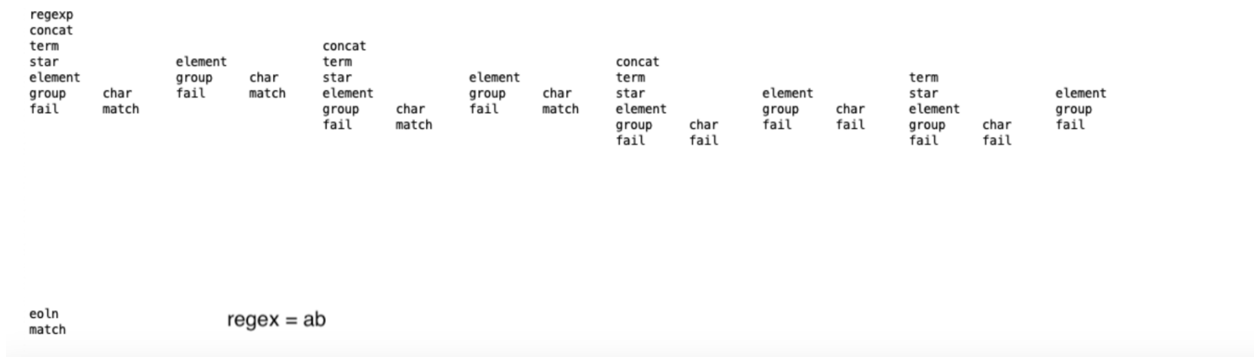
```
regexp
concat
term                            concat
star            element         term                            term                            eoln
element         group   char    star            element         star            element         match
group   char    fail    match   element         group   char    element         group   char
fail    match                   group   char    fail    fail    group   char    fail    fail
                                fail    fail                    fail    fail
```

regex = a



```
regexp
concat
term                    concat
star            *match  term                            term                            eoln
element                 star            element         star            element         match
group   char            element         group   char    element         group   char
fail    match           group   char    fail    fail    group   char    fail    fail
                        fail    fail                    fail    fail
```

regex = a*



```
regexp
concat
term                            concat
star            element         term                            concat
element         group   char    star            element         term                            term
group   char    fail    match   element         group   char    star            element         star            element
fail    match                   group   char    fail    match   element         group   char    element         group
                                fail    match                   group   char    fail    fail    group   char    fail
                                                                fail    fail                    fail    fail




eoln
match
```

regex = ab

Coding Practices
Write the code using standard stylistic practices. Use functions,
reasonable variable names, and consistent indentation.

Do not use goto statements or global variables unless they are provided in the starting code.

As usual, keep backups of your work using source control software.

Submitting the Assignment
Submit only the source code a3.c, the makefile, and readme.

Include a readme file.
If there are any parts of your program that don't work as specified
then describe them in the readme.