CS39003 Compilers Laboratory
Autumn 2026
Assignment 2
Date of posting: 28-Jul-2026

Using Lex

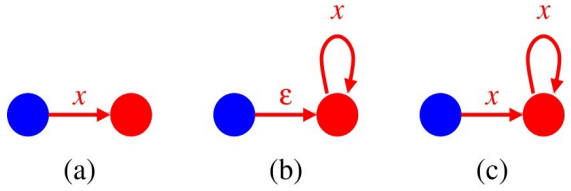
The first step in the compilation process involves decomposing the input program into a sequence of tokens. Programming languages are so designed as to minimize conflicts among different types of tokens. Ambiguities (like between keywords and identifiers) are handled in special ways. In this assignment (and the next), we plan to be completely general, and allow any amount of ambiguity across the patterns. The goal is to investigate whether lexical analysis can still be carried out in a decent amount of time.

A regular expression (pattern) consists of a concatenation of subpatterns. The subpatterns (also the pattern itself) may contain the + operator (indicating OR). In many cases (although not always), we can get rid of this operator by writing multiple patterns. For example, the pattern $a\,(b + c + c^*d)\,e + f^+$ can be written as four separate patterns *abe*, *ace*, ac^*de, and f^+. Matching any of these four patterns is equivalent to matching the original pattern. In view of this, we define a pattern as a concatenation of factors, where each factor is either a symbol (like a) or an iterated symbol (like c^* and f^+). For simplicity, we do not involve parentheses in patterns. This is a loss of generality. For example, we cannot specify patterns like $(g + h)^+$ or $(gh)^+$, because an attempt to do so will generate infinitely many patterns in our format. Using arbitrarily nested parentheses makes the language of patterns non-regular. You cannot handle such context-free constructs using lex actions alone. Lex can identify all the tokens including parentheses, but handling the parentheses would call for a stack. Later, we will use yacc's parse stack to that effect.

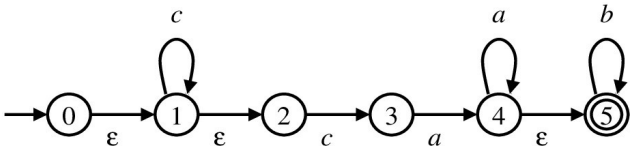
An input file consists of the restricted type of patterns with one pattern in one line. Ambiguities do not arise in the presence of white spaces (spaces and tabs, but not new lines), so we allow white spaces anywhere in the patterns. A few examples of patterns that the input file may contain follow this paragraph. For simplicity, assume that the input alphabet can contain at most eight symbols. We name them as $\Sigma = \{ a, b, c, d, e, f, g, h \}$ (so under the equivalence of the character x with the integer $x - $ 'a', the transition function can be an $s \times 8$ array, where s is the number of states of the NFA). We use the unsuperscipted letters + and * to stand for the asterate constructs $^+$ and *. In particular, our grammar interprets + as one or more occurrences (but not as the OR of two patterns).

```
a +
a b+
b c+
c*
```

Your task is to write a lex program that prepares an NFA (an ε-NFA to be more precise) for each pattern in the input file. The three basic constructs can be converted as shown in the following figure. The current state is shown in color blue. The additional constructs are shown in red. The three parts in the figure are the respective translations of the factors (a) x, (b) x^*, and (c) x^+, where x is a symbol of Σ. In addition to this generic construction, we sometimes may have to add other ε-transitions in order to ensure that for each state p and for each symbol x in $\Sigma \cup \{ \varepsilon \}$, the transition function $\delta(p,x)$ either is empty or consists of a single state. The guideline is this: if $\delta(p,x)$ is already defined, create an ε-transition from p to the next state q, and add the transition $\delta(q,x)$. These extra transitions are needed only when x or x^+ comes immediately after x^+ or x^*. If x^* comes immediately after x^+ or x^*, we do not have a problem, because the rendering of x^* already begins with an ε-transition.



For example, here is the NFA constructed from the pattern $c^*ca^+b^*$. Here, the additional transition $\delta(1,\varepsilon)$ is introduced because $\delta(1,c)$ was already defined.



Note for LA3

It is worthwhile to elaborate a point here. Although not needed in this assignment, this will be vital for LA3. Our NFA construction makes each $\delta(p,x)$ either undefined or single-valued, but the NFA at any point of time may be in multiple states. For example, if the input to the NFA N constructed from $c^*ca^+b^*$ is *cccaab*, N should loop in state 1 itself after reading the first two *c*'s, then use the ε-transition to move to state 2, and consume the third *c* to jump to state 3. If we do not use lookaheads, we cannot determine when exactly N should follow an ε-transition. Since lookaheads can be arbitrarily long, it is not a good idea to use them. Instead, we will keep track of all the states where N can be after consuming each input symbol. At the beginning, N can be in the states {0, 1}. After reading the first *c*, N will go to states {1, 2}. After reading the second and the third *c*, N will go to the states {1, 2, 3}. Then upon reading each of the two *a*'s, N will go to the states {4, 5}. After reading the last symbol *b* from the input, the state will be {5}. Since this set contains the final state, N will accept *cccaab*. On the other hand, if the input is *cccb*, then reading the three *c*'s will change the state of N to {1, 2, 3}. But there is no *b*-transition to follow from any of these states, so N will get stuck, and reject *cccb*. Finally if the input is *ccc*, the states of N at the end of the input are {1, 2, 3}. Neither of these is a final state, so N will reject *ccc* too.

Your lex program will ignore all white spaces (except new lines) and, in particular, empty lines (lines containing only spaces and/or tabs, if any). It should only try to match the three types of patterns *x*, *x**, and *x*+, and augment the current NFA as specified above. After the construction of each NFA, print that NFA in a format illustrated in the sample output (print only those $\delta(p,x)$ values that are defined). The data structure for storing an NFA is left to you. Note only that you need to store (at least) the following information against each NFA: its number of states, its *unique* initial state, its *unique* final state (the current state after the entire input pattern in a line is read), and the transition function (including the ε-transitions if any). Number the states as 0, 1, 2, … , and take 0 as the start state. Generate newer states in the order 1, 2, 3, … according to the construction explained above. For our NFA's, each $\delta(p,x)$ you need to store is either undefined (may be denoted by –1) or the number of a unique state.

If you feed your input file to lex itself, it will create one of more NFAs/DFAs for the purpose of lexical analysis. It may look like a duplication of effort why you need to build your own NFA's. The reason for doing this is two-fold. First, designing your own NFA's would furnish you with an idea how lex works. More importantly, in the next assignment, you will do something with your NFA's, that lex cannot do. Wait until then.

Write a lex program `buildNFA.l` that does the basic pattern matching needed during the NFA constructions. You may use other C/C++ file(s) containing the functions that you call (from your lex actions) to initialize, augment, and finalize each NFA. Store all the NFA's globally. You may write the `main`() function in your lex program itself or in one of your separate C/C++ source files. Also write a `makefile` with compile, run, and clean targets (so that the evaluator will not need to bother about the organization of your code across multiple files).

Combine your source code (lex and C/C++ files if any) and the makefile in a tar/tgz/zip archive. Do not include the machine-generated files (like `lex.yy.c` or `a.out`) in your archive. Submit only a single file (your archive).

Sample Output

Input file

```
1   a +
2   a b+
3   bc+
4   c*
5
6   a* aaa bbb
7
8   d+ e* f* g* d
9
10  fghg*ghfh*hfgf*
```

Note: The line numbers shown in green are not part of the input file.

Output on the input file

```
+++ Reading NFA 0 from line 1
    Number of states = 2
    Transitions
        delta(0,a) = 1
        delta(1,a) = 1
    Start state: 0
    Final state: 1

+++ Reading NFA 1 from line 2
    Number of states = 3
    Transitions
        delta(0,a) = 1
        delta(1,b) = 2
        delta(2,b) = 2
    Start state: 0
    Final state: 2

+++ Reading NFA 2 from line 3
    Number of states = 3
    Transitions
        delta(0,b) = 1
        delta(1,c) = 2
        delta(2,c) = 2
    Start state: 0
    Final state: 2

+++ Reading NFA 3 from line 4
    Number of states = 2
    Transitions
        delta(0,epsilon) = 1
        delta(1,c) = 1
    Start state: 0
    Final state: 1

+++ Reading NFA 4 from line 6
    Number of states = 9
    Transitions
        delta(0,epsilon) = 1
        delta(1,a) = 1
        delta(1,epsilon) = 2
        delta(2,a) = 3
        delta(3,a) = 4
        delta(4,a) = 5
        delta(5,b) = 6
        delta(6,b) = 7
        delta(7,b) = 8
    Start state: 0
    Final state: 8

+++ Reading NFA 5 from line 8
    Number of states = 6
    Transitions
        delta(0,d) = 1
        delta(1,d) = 1
        delta(1,epsilon) = 2
        delta(2,e) = 2
        delta(2,epsilon) = 3
        delta(3,f) = 3
        delta(3,epsilon) = 4
        delta(4,d) = 5
        delta(4,g) = 4
    Start state: 0
    Final state: 5

+++ Reading NFA 6 from line 10
    Number of states = 15
    Transitions
        delta(0,f) = 1
        delta(1,g) = 2
        delta(2,h) = 3
        delta(3,epsilon) = 4
        delta(4,g) = 4
        delta(4,epsilon) = 5
        delta(5,g) = 6
        delta(6,h) = 7
        delta(7,f) = 8
        delta(8,epsilon) = 9
        delta(9,h) = 9
        delta(9,epsilon) = 10
        delta(10,h) = 11
        delta(11,f) = 12
        delta(12,g) = 13
        delta(13,epsilon) = 14
        delta(14,f) = 14
    Start state: 0
    Final state: 14
```