CS39003 Compilers Laboratory
Autumn 2026
Assignment 3
Date of posting: 04-Aug-2026

Write your own lexeme generator

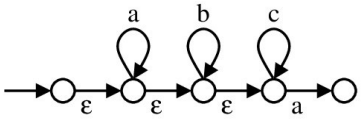
In Assignment 2, you have developed a code to convert simple patterns to NFA's. In this assignment, you solve the problem of decomposing an input string into a sequence of lexemes, each matching one of the specified patterns (or reporting that the input string cannot be so decomposed). The idea behind this assignment is that lex runs a greedy algorithm which keeps on locating the longest leftmost patterns, regardless of the rest of the string. However, for the general problem, this greedy strategy does not necessarily work. Here is an example of this situation.

Suppose that we deal with the four patterns a^+, ab^+, bc^+, and c^*. For the input string *aabbcc*, lex first finds *aa* as the longest leftmost match (generated by the pattern a^+). But then, when lex encounters two *b*'s after the initial *aa*, it cannot decompose it fully. The first *b* is unmatched. Subsequently, *bcc* is greedily discovered to be generated by bc^+. On the other hand, the same input string can be decomposed into valid lexemes as *a abb cc* and also as *a ab bcc*. Neither of these uses the longest match *aa* at the beginning.

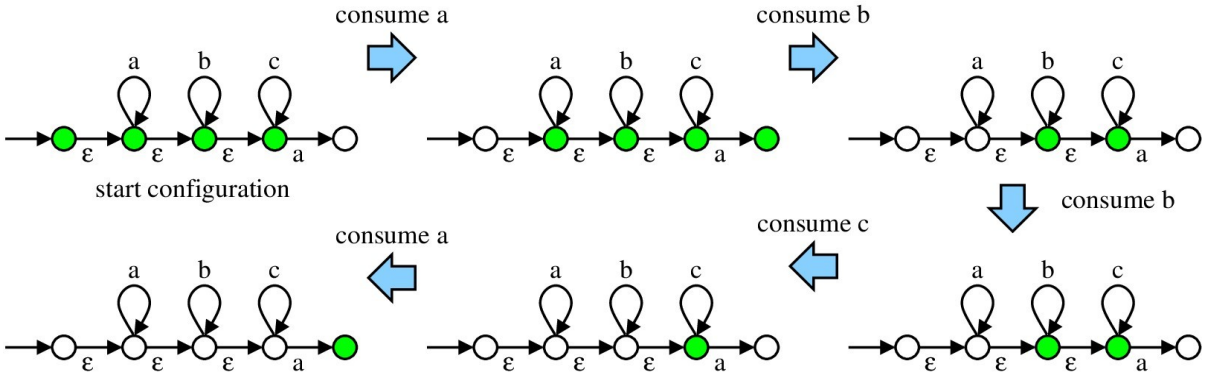
This observation has two immediate consequences. First, tokens in a programming language must be carefully designed so that the greedy algorithm of lex works. Second, a question crops up whether any *reasonably fast* algorithm (need not be greedy) can solve the general decomposition problem. This assignment addresses only the second question in the affirmative. We will use a dynamic-programming (DP) algorithm for this purpose.

Before stating our DP algorithm, let us look at how we can implement transitions in an NFA N generated as in Assignment 2. Recall that our N may have ε-transitions. This complicates the problem slightly in that at any point of time, N can be in any number of states. We keep track of all these states. To start with, we determine all the states S that N can be in at the beginning. By our construction, 0 is always the start state, so we start with $S = \{0\}$. There may be an ε-transition from this state. If so, we add the destination of that ε-transition to S. If the added state too has an ε-transition, we add its destination to S as well. We continue this until no new states can be added to S. Technically speaking, the set of states for N at the beginning is the ε-closure of $\{0\}$.

Now, suppose that at some point of time, P is the set of states that N can be in, and the next input symbol is x. We like to compute all the states Q that N can be in after consuming x from the input. We first populate Q with the defined $\delta(p,x)$ values for all p in P. We iteratively look at the ε-transitions from the states of Q, and add their destinations to Q. Technically, $\delta(P,x)$ is the ε-closure of $\{\delta(p,x) \mid p \in P\}$. Since our construction always adds ε-transitions from lower-numbered states to higher-numbered states, a single pass though the set of states suffices. The following figure illustrates this behavior of an NFA. The states that the NFA can be in are marked in color green.



(a) NFA for a*b*c*a



(b) States of the NFA on input abbca

Let $s = a_1 a_2 a_3 \ldots a_n$ be an input string that we want to decompose into valid lexemes. We iteratively consider the suffix $s_i = a_i \ldots a_n$ of s for i in the sequence $n, n-1, n-2, \ldots, 2, 1$. We traverse each of the NFA's generated from the given patterns. Let R be such a pattern, and N the NFA generated from R. We start with the start configuration of N as explained above. We make an attempt to traverse N following the input symbols a_i onward of s_i. If at some point of time, the set of states that N can be in is empty, then N has got stuck at that point, and cannot generate longer lexemes using the pattern R. Otherwise, we continue consuming symbols from the suffix s_i. In all the cases where the set of states of N contains its final state, a prefix of length $t > 0$ of s_i is discovered as a candidate for helping decompose s_i starting with an instance of the pattern R. Look at the figure on the previous page, and assume that some s_i starts with *abbca*. The transitions will continue for these five symbols. Any symbol in s_i following these five symbols will leave the NFA in the stuck configuration. Out of the five situations, only the prefixes *a* and *abbca* of s_i lead N to reach its final state (along with other states possibly). Only these two prefixes are therefore *candidates* for decomposing s_i. However, instead of greedily choosing the longer prefix *abbca*, we will do something else.

Take a candidate prefix of s_i, and let its length be t. We check whether the suffix s_{i+t} of s can also be decomposed into lexemes (under the given patterns). We do not allow $t = 0$ (even if the start configuration of N contains the final state as in the case of the patterns c^* or $c^*a^*b^*$). By the sequence of choosing i, the question for s_{i+t} is already solved. For every candidate prefix of s_i (that lets N end up in its final state), we determine whether the rest of the string (following the prefix) can be decomposed into valid lexemes. We take the longest such prefix (if any). We also maximize the length of these longest prefixes of s_i over all NFA's N defined by the given patterns. In case there is a tie in these maximum prefix lengths, we take the N defined the earliest as having generated that match. Finally, we check whether the entire string $s = s_1$ can be decomposed into valid lexemes.

In the example of the second paragraph, this algorithm will output the decomposition *a abb cc* (not the decomposition *a ab bcc* because *abb* is longer than *ab*). So the general decomposition problem is solved.

This algorithm is similar to what lex does (in a single pass) with the notable exception of taking into consideration the rest of the prefix (the underlined phrase of the last paragraph). It follows that we need an array MAXLEN of size n for storing whether each s_i has a decomposition into valid lexemes. The array may be initialized to all 0's (indicating that no pattern has been discovered so far to decompose s_i). Whenever a suitable non-empty prefix is found (as explained above), the corresponding entry in the array MAXLEN is looked at. If the length t of the newly discovered prefix is larger than that entry of MAXLEN, we replace the entry by t. A token is supposed to consist of a pattern type along with the lexeme. Use a second array MAXPAT to remember the pattern number that generates the longest prefix. This array may be initialized to an invalid pattern number (like –1). If no NFA N can decompose s_i, we will continue to have MAXLEN[i] = 0, and MAXPAT[i] = –1.

Why lex does not take this DP approach has two major reasons. The first reason is efficiency. Suppose that there are a constant number of patterns/NFA's, and each such NFA has a constant size. So the input size is dominated by the length n of the input string s. Lex's greedy algorithm works in $O(n)$ time, whereas our DP-based algorithm takes $O(n^2)$ time. For very long inputs, this blow up in the running time may be prohibitive. Second, our algorithm requires the entire input to be available from the very beginning. On the other hand, lex can proceed incrementally. So lex wins in practice. But programming-language designers must ensure that the greedy algorithm of lex works.

What you have to do is to augment your code of Assignment 2. That code already generates all the NFA's from the given patterns, and stores them globally. After specifying the patterns, the input file contains a line containing only `%%`. After this (that is, in the second section of the input), literal strings are presented for the task of decomposition, one in each non-empty line. Your lex program should ignore white spaces in the input strings (the user may insert white spaces at any point in the inputs because no ambiguities ensue).

The implementation details for sets of states (of the NFA's) are left to you.

Notice that the same strings are interpreted differently in the two sections. For example, *abc* refers to a pattern in the first section, and to a literal string in the second. In the first section, the lex program builds NFA's (one for each non-empty line), whereas in the second section, lex uses our DP algorithm for decomposing each non-empty input string. That is, the actions of lex in the two sections are different. You can achieve it by using your custom-made global variable(s). For your practice, we advocate the use of lex states instead.

Write a lex file, and optionally other C/C++ headers and/or source files. Also write a makefile with compile, run, and clean targets. Combine only these files to a tar/tgz/zip archive, and submit only that archive. Do not include machine-generated codes (like `lex.yy.c` or `a.out`) in your archive.

Sample Output

Input	Output
a + a b+ bc+ c* a* aaa bbb d+ e* f* g* d fghg*ghfh*hfgf* %% aa bb cc aa bb cc b aa bb cc aaaa bbbb cccc ddddddddddddddd dfd fghghfhfg aa bb cc aaa bbb ccc	+++ 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 +++ Input string "aabbcc" is decomposed as follows <0,a> <1,abb> <3,cc> +++ Input string "aabbccb" cannot be decomposed +++ Input string "aabbccaaaabbbbcccc" is decomposed as follows <0,a> <1,abb> <3,cc> <4,aaaabbb> <2,bcccc> +++ Input string "ddddddddddddddd" is decomposed as follows <5,ddddddddddddddd> +++ Input string "dfdfghghfhfgaabbccaaabbbccc" is decomposed as follows <5,dfd> <6,fghghfhfg> <0,a> <1,abb> <3,cc> <4,aaabbb> <3,ccc>