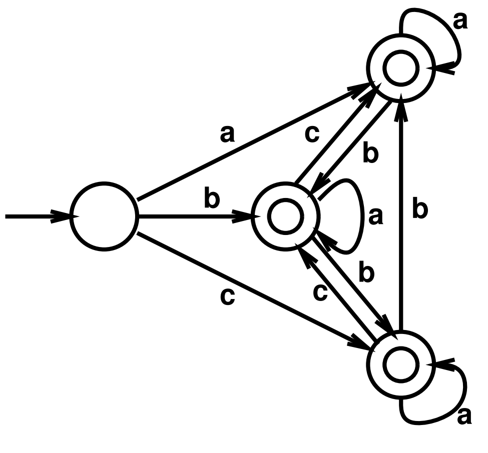
# The Theory of Computation
Bernard M.E. Moret
Addison-Wesley-Longman, 1998

*Solution Manual*
*Chapter 3, Problem 3.12*

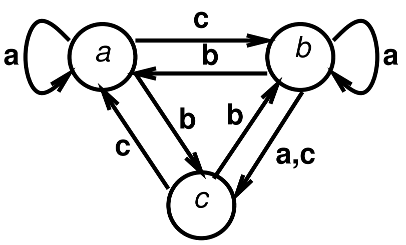
As an example, assume that the string is *abbbcc*; in a left-to-right evaluation, we ask for the value of the expression $((((a \cdot b) \cdot b) \cdot b) \cdot c) \cdot c$; in a right-to-left evaluation, we ask for the value of the expression $a \cdot (b \cdot (b \cdot (b \cdot (c \cdot c))))$.

Left-to-right evaluation of the string is straightforward for either a DFA or an NFA: we simply keep the current value of the expression stored in states (hence 3 states) and move between these states as the next symbol dictates. For instance, if we are in "state $a$" and see the input symbol $c$, we move to "state $b$," corresponding to the operation $a \cdot c = b$. The DFA has 4 states in all, as shown below. The challenge—and the reason why an NFA



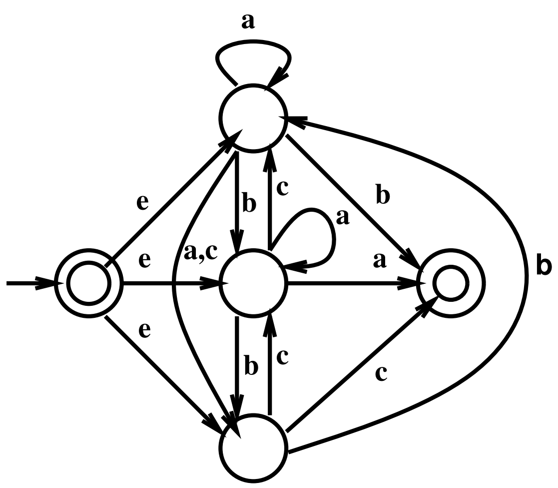

is much preferable—is the right-to-left evaluation; indeed, it may appear impossible at first, since the machine gets the symbols in the worst possible order and cannot store them. Doing the right-to-left evaluation is not too bad once one gets the key insight: the NFA will guess every intermediate result and will verify that it could have been produced as guessed once the operands are both known. In our example, the NFA first guesses the value of the entire expression, say $x$, then guesses a second operand $y$ such that

$a \cdot y = x$; $y$ is now the guessed value for the subexpression $b \cdot (b \cdot (b \cdot (c \cdot c)))$ and we proceed in the same way, guessing some second operand $z$ such that $b \cdot z = y$. For our tables, this gives the following NFA fragment, which is entered at any of its 3 states; we have labelled each state with the value that denotes the corresponding guess for the remaining substring.

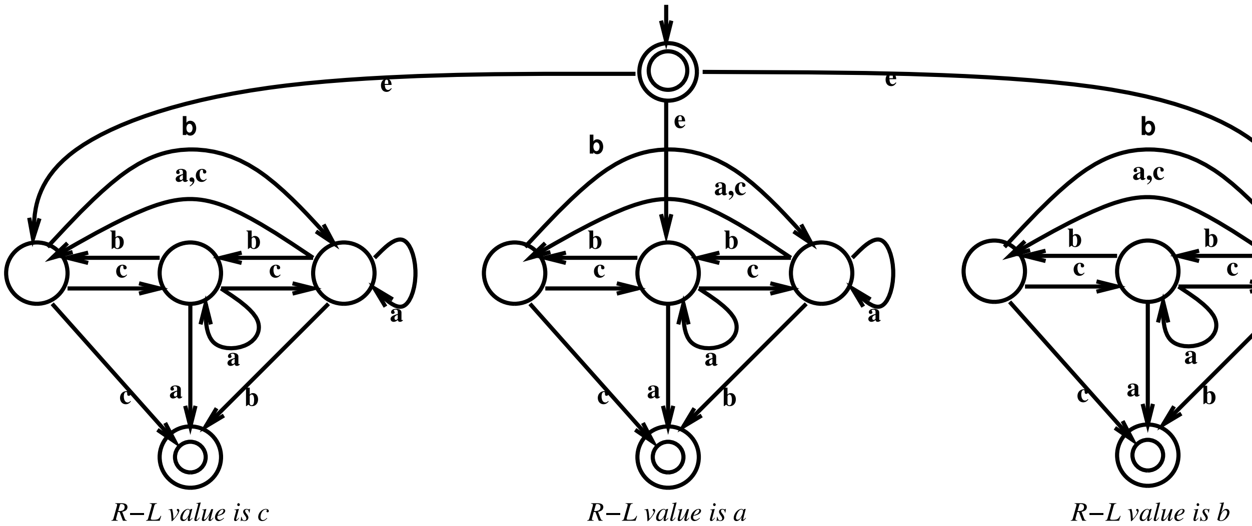



So, for instance, if we are in state “a’ (i.e., we think that the remaining substring evaluates right-to-left to $a$) and we read a $b$, we move to state “c,” because $c$ is the only value $x$ such that $a = b \cdot x$. In contrast, if we are in state “b” and we read an $a$, we can either stay in the same state or move to state “c,” because both $b$ and $c$ are values $y$ such that $b = a \cdot y$.

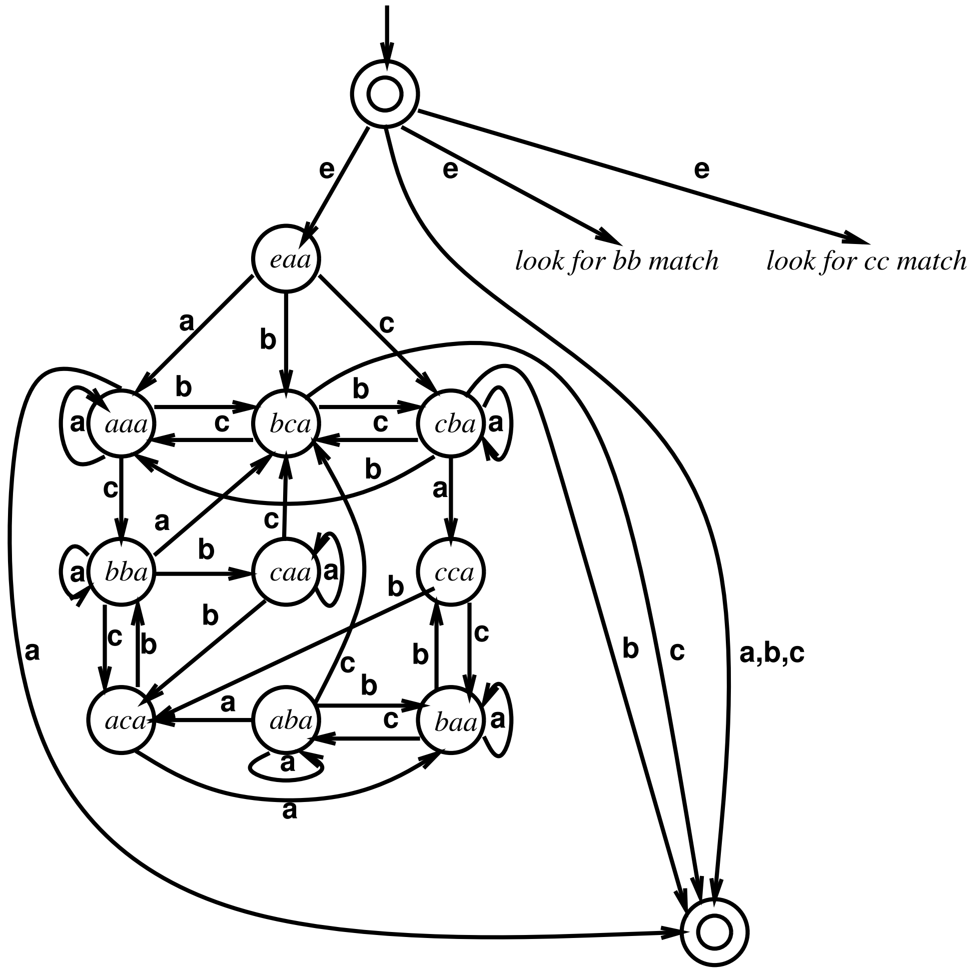
However, the behavior must change for the last symbol: at that point, there is nothing more to guess and we must just verify that the current guess can be accounted for by the last symbol. This adds another state: the NFA guesses that only one symbol is left and enters a final state, as depicted below; this time, we have added an initial state and $\epsilon$ transitions into one of the three guesses. The NFA must “remember” its first guess (the value




of the whole expression)—we can do that by having a separate copy of the NFA for each possible initial guess, three in our case, so the complete NFA now has 13 states, as given below.




The next challenge resides in doing both evaluations together, since we cannot re-read the input. Now that we know how to design a machine that will evaluate the string left-to-right and how to design another machine that will evaluate it right-to-left, how do we have them work together, at the same time, on the same input? The answer is to combine the two machines by designing a new machine whose states combine a state of the first machine with a state of the second. We do not need to combine the initial state and only 3 of the 9 possible combinations of final states are of interest (are final states of the new machines, corresponding to the same evaluation in both directions), but otherwise all combinations yield states, for a total of 32 states. The machine starts in an accepting state (for the null string); there is also a transition from the start state to a "final" accepting state under any single character input, in order to accept strings made of a single character. For all other strings, the machine now moves through $\epsilon$ transitions to one of three states, corresponding to the three guesses of the right-to-left NFA; denote these states $(\epsilon, a, a)$, $(\epsilon, b, b)$, and $(\epsilon, c, c)$, where the first part denotes the current value of the left-to-right evaluation, the second part denotes the current guess as to the right-to-left value of the remaining substring, and the third value denotes the original guess about the right-to-left value of the whole string. The first two entries exist to combine the two basic machines; the third entry serves to make three copies of the whole affair, to remember the initial guess. Thus, from state $(\epsilon, a, a)$, we can now move, on reading an $a$, to state $(a, a, a)$; on reading a $b$, to state $(b, c, a)$; and on reading a $c$, to state $(c, b, a)$. From state $(\epsilon, b, b)$, on reading an $a$, we can move to one of two states, $(a, b, b)$ or $(a, c, b)$. The resulting machine is depicted on the next page—only one third is shown, for the guess that the string will evaluate to $a$ in both directions; the other two thirds are the same, except for the transitions into the accepting states, which occur, for the first, from all states $(x, y, b)$ such that $x \cdot y = b$ and under input $y$, and, for the second, from all states $(v, w, c)$ such that $v \cdot w = c$ and under input $w$.




A DFA for this problem of course exists, but it is clearly a mess. A lower bound is somewhat hard to derive; all we can say easily is that any FA (deterministic or not) will need at least $3 \cdot 3 \cdot 3 = 27$ states: one factor of 3 is to keep information about the ongoing left-to-right evaluation, one is to keep information about the current right-to-left evaluation, and one is to remember the value guessed for the whole string. A DFA is likely to require many more states—although surely not the $2^{31}$ (!) that would be produced by a brute force mapping from our NFA.