

Implementing the Lookahead Operator

Recall from Section 3.4 that the lookahead operator $/$ is necessary in some situations, since the pattern that denotes a particular token may need to describe some trailing context for the actual lexeme. When converting a pattern with $/$ to an NFA, we can treat the $/$ as if it were ϵ , so that we do not actually look for $/$ on the input. However, if a string denoted by this regular expression is recognized in the input buffer, the end of the lexeme is not the position of the NFA's accepting state. Rather it is at the last occurrence of the state of this NFA having a transition on the (imaginary) $/$.

Example 3.20. The NFA recognizing the pattern for **IF** given in Example 3.12 is shown in Fig. 3.38. State 6 indicates the presence of keyword **IF**; however, we find the token **IF** by scanning backwards to the last occurrence of state 2. \square

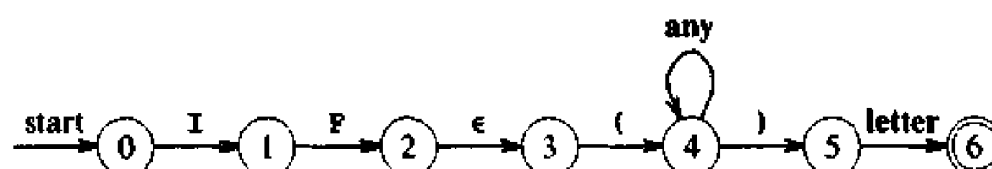


Fig. 3.38. NFA recognizing Fortran keyword **IF**.

3.9 OPTIMIZATION OF DFA-BASED PATTERN MATCHERS

In this section, we present three algorithms that have been used to implement and optimize pattern matchers constructed from regular expressions. The first algorithm is suitable for inclusion in a Lex compiler because it constructs a DFA directly from a regular expression, without constructing an intermediate NFA along the way.

The second algorithm minimizes the number of states of any DFA, so it can be used to reduce the size of a DFA-based pattern matcher. The algorithm is efficient; its running time is $O(n \log n)$, where n is the number of states in the DFA. The third algorithm can be used to produce fast but more compact representations for the transition table of a DFA than a straightforward two-dimensional table.

Important States of an NFA

Let us call a state of an NFA *important* if it has a non- ϵ out-transition. The subset construction in Fig. 3.25 uses only the important states in a subset T when it determines ϵ -closure(move(T, a)), the set of states that is reachable from T on input a . The set move(s, a) is nonempty only if state s is important. During the construction, two subsets can be identified if they have the same important states, and either both or neither include accepting states of the NFA.

When the subset construction is applied to an NFA obtained from a regular expression by Algorithm 3.3, we can exploit the special properties of the NFA to combine the two constructions. The combined construction relates the important states of the NFA with the symbols in the regular expression. Thompson's construction builds an important state exactly when a symbol in the alphabet appears in a regular expression. For example, important states will be constructed for each a and b in $(a|b)^*abb$.

Moreover, the resulting NFA has exactly one accepting state, but the accepting state is not important because it has no transitions leaving it. By concatenating a unique right-end marker $\#$ to a regular expression r , we give the accepting state of r a transition on $\#$, making it an important state of the NFA for $r\#$. In other words, by using the augmented regular expression $(r)\#$ we can forget about accepting states as the subset construction proceeds; when the construction is complete, any DFA state with a transition on $\#$ must be an accepting state.

We represent an augmented regular expression by a syntax tree with basic symbols at the leaves and operators at the interior nodes. We refer to an interior node as a *cat-node*, *or-node*, or *star-node* if it is labeled by a concatenation, $|$, or $*$ operator, respectively. Figure 3.39(a) shows a syntax tree for an augmented regular expression with cat-nodes marked by dots. The syntax tree for a regular expression can be constructed in the same manner as a syntax tree for an arithmetic expression (see Chapter 2).

Leaves in the syntax tree for a regular expression are labeled by alphabet symbols or by ϵ . To each leaf not labeled by ϵ we attach a unique integer and refer to this integer as the *position* of the leaf and also as a position of its symbol. A repeated symbol therefore has several positions. Positions are shown below the symbols in the syntax tree of Fig. 3.39(a). The numbered states in the NFA of Fig. 3.39(c) correspond to the positions of the leaves in the syntax tree in Fig. 3.39(a). It is no coincidence that these states are the important states of the NFA. Non-important states are named by upper case letters in Fig. 3.39(c).

The DFA in Fig. 3.39(b) can be obtained from the NFA in Fig. 3.39(c) if we apply the subset construction and identify subsets containing the same important states. The identification results in one fewer state being constructed, as a comparison with Fig. 3.29 shows.

From a Regular Expression to a DFA

In this section, we show how to construct a DFA directly from an augmented regular expression $(r)\#$. We begin by constructing a syntax tree T for $(r)\#$ and then computing four functions: *nullable*, *firstpos*, *lastpos*, and *followpos*, by making traversals over T . Finally, we construct the DFA from *followpos*. The functions *nullable*, *firstpos*, and *lastpos* are defined on the nodes of the syntax tree and are used to compute *followpos*, which is defined on the set of positions.

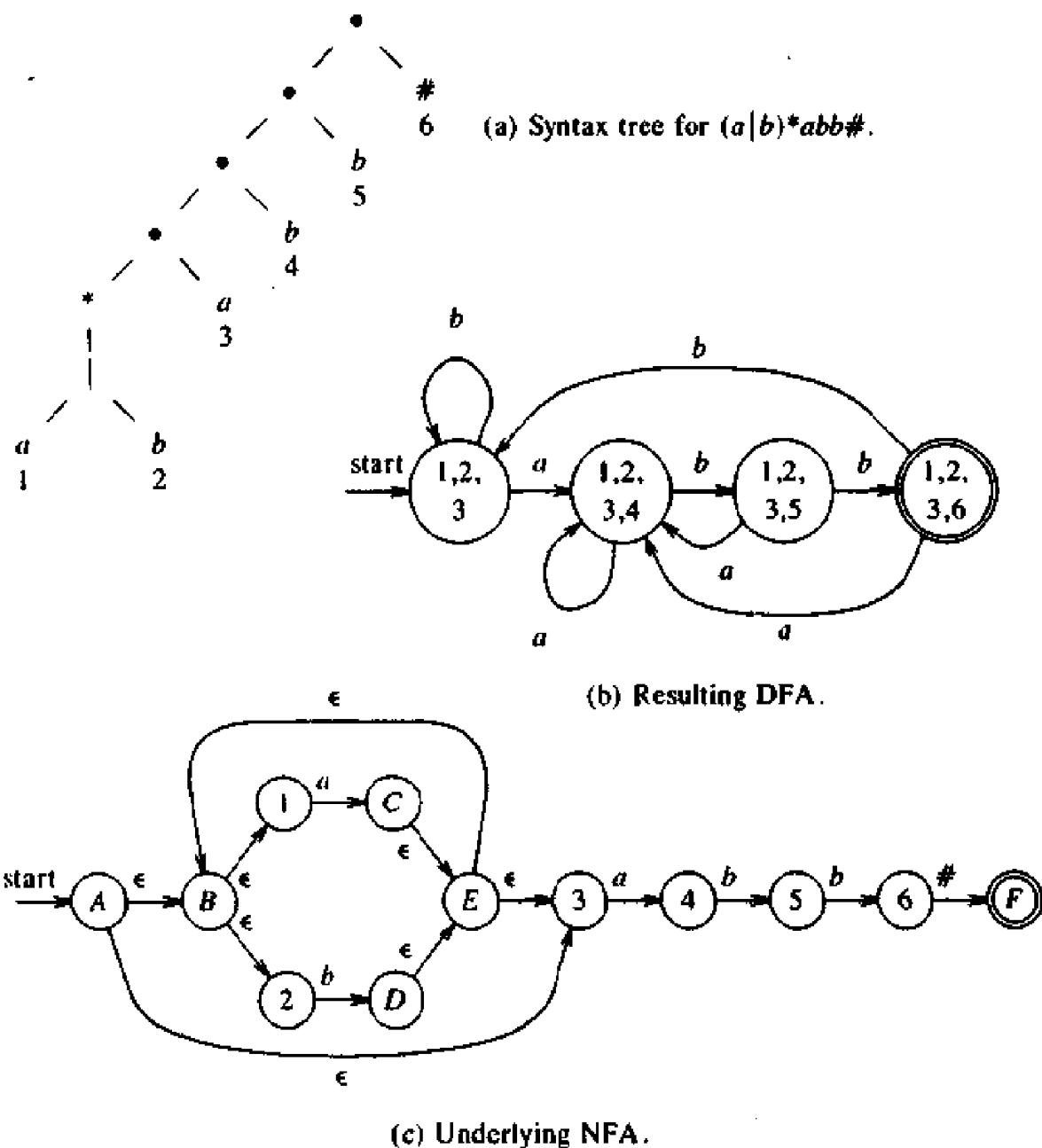


Fig. 3.39. DFA and NFA constructed from $(a|b)^*abb\#$.

Remembering the equivalence between the important NFA states and the positions of the leaves in the syntax tree of the regular expression, we can short-circuit the construction of the NFA by building the DFA whose states correspond to sets of positions in the tree. The ϵ -transitions of the NFA represent some fairly complicated structure of the positions; in particular, they encode the information regarding when one position can follow another. That is, each symbol in an input string to a DFA can be matched by certain positions. An input symbol c can only be matched by positions at which there is a c , but not every position with a c can necessarily match a particular occurrence of c in the input stream.

The notion of a position matching an input symbol will be defined in terms

of the function *followpos* on positions of the syntax tree. If i is a position, then *followpos*(i) is the set of positions j such that there is some input string $\cdots cd \cdots$ such that i corresponds to this occurrence of c and j to this occurrence of d .

Example 3.21. In Fig. 3.39(a), *followpos*(1) = {1, 2, 3}. The reasoning is that if we see an a corresponding to position 1, then we have just seen an occurrence of $a|b$ in the closure $(a|b)^*$. We could next see the first position of another occurrence of $a|b$, which explains why 1 and 2 are in *followpos*(1). We could also next see the first position of what follows $(a|b)^*$, that is, position 3. \square

In order to compute the function *followpos*, we need to know what positions can match the first or last symbol of a string generated by a given subexpression of a regular expression. (Such information was used informally in Example 3.21.) If r^* is such a subexpression, then every position that can be first in r follows every position that can be last in r . Similarly, if rs is a subexpression, then every first position of s follows every last position of r .

At each node n of the syntax tree of a regular expression, we define a function *firstpos*(n) that gives the set of positions that can match the first symbol of a string generated by the subexpression rooted at n . Likewise, we define a function *lastpos*(n) that gives the set of positions that can match the last symbol in such a string. For example, if n is the root of the whole tree in Fig. 3.39(a), then *firstpos*(n) = {1, 2, 3} and *lastpos*(n) = {6}. We give an algorithm for computing these functions momentarily.

In order to compute *firstpos* and *lastpos*, we need to know which nodes are the roots of subexpressions that generate languages that include the empty string. Such nodes are called *nullable*, and we define *nullable*(n) to be true if node n is nullable, false otherwise.

We can now give the rules to compute the functions *nullable*, *firstpos*, *lastpos*, and *followpos*. For the first three functions, we have a basis rule that tells about expressions of a basic symbol, and then three inductive rules that allow us to determine the value of the functions working up the syntax tree from the bottom; in each case the inductive rules correspond to the three operators, union, concatenation, and closure. The rules for *nullable* and *firstpos* are given in Fig. 3.40. The rules for *lastpos*(n) are the same as those for *firstpos*(n), but with c_1 and c_2 reversed, and are not shown.

The first rule for *nullable* states that if n is a leaf labeled ϵ , then *nullable*(n) is surely true. The second rule states that if n is a leaf labeled by an alphabet symbol, then *nullable*(n) is false. In this case, each leaf corresponds to a single input symbol, and therefore cannot generate ϵ . The last rule for *nullable* states that if n is a star-node with child c_1 , then *nullable*(n) is true, because the closure of an expression generates a language that includes ϵ .

As another example, the fourth rule for *firstpos* states that if n is a cat-node with left child c_1 and right child c_2 , and if *nullable*(c_1) is true, then

$$\text{firstpos}(n) = \text{firstpos}(c_1) \cup \text{firstpos}(c_2)$$

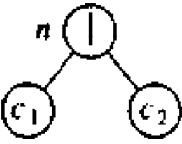
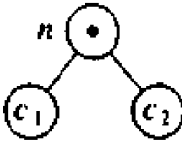

NODE n	$nullable(n)$	$firstpos(n)$
n is a leaf labeled ϵ	true	\emptyset
n is a leaf labeled with position i	false	$\{i\}$
	$nullable(c_1) \text{ or } nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$
	$nullable(c_1) \text{ and } nullable(c_2)$	if $nullable(c_1)$ then $firstpos(c_1) \cup firstpos(c_2)$ else $firstpos(c_1)$
	true	$firstpos(c_1)$

Fig. 3.40. Rules for computing *nullable* and *firstpos*.

otherwise, $firstpos(n) = firstpos(c_1)$. What this rule says is that if in an expression rs , r generates ϵ , then the first positions of s “show through” r and are also first positions of rs ; otherwise, only the first positions of r are first positions of rs . The reasoning behind the remaining rules for *nullable* and *firstpos* are similar.

The function *followpos*(i) tells us what positions can follow position i in the syntax tree. Two rules define all the ways one position can follow another.

1. If n is a cat-node with left child c_1 and right child c_2 , and i is a position in $lastpos(c_1)$, then all positions in $firstpos(c_2)$ are in *followpos*(i).
2. If n is a star-node, and i is a position in $lastpos(n)$, then all positions in $firstpos(n)$ are in *followpos*(i).

If *firstpos* and *lastpos* have been computed for each node, *followpos* of each position can be computed by making one depth-first traversal of the syntax tree.

Example 3.22. Figure 3.41 shows the values of *firstpos* and *lastpos* at all nodes in the tree of Fig. 3.39(a); *firstpos*(n) appears to the left of node n and *lastpos*(n) to the right. For example, *firstpos* at the leftmost leaf labeled a is $\{1\}$, since this leaf is labeled with position 1. Similarly, *firstpos* of the second leaf is $\{2\}$, since this leaf is labeled with position 2. By the third rule in Fig. 3.40, *firstpos* of their parent is $\{1, 2\}$.

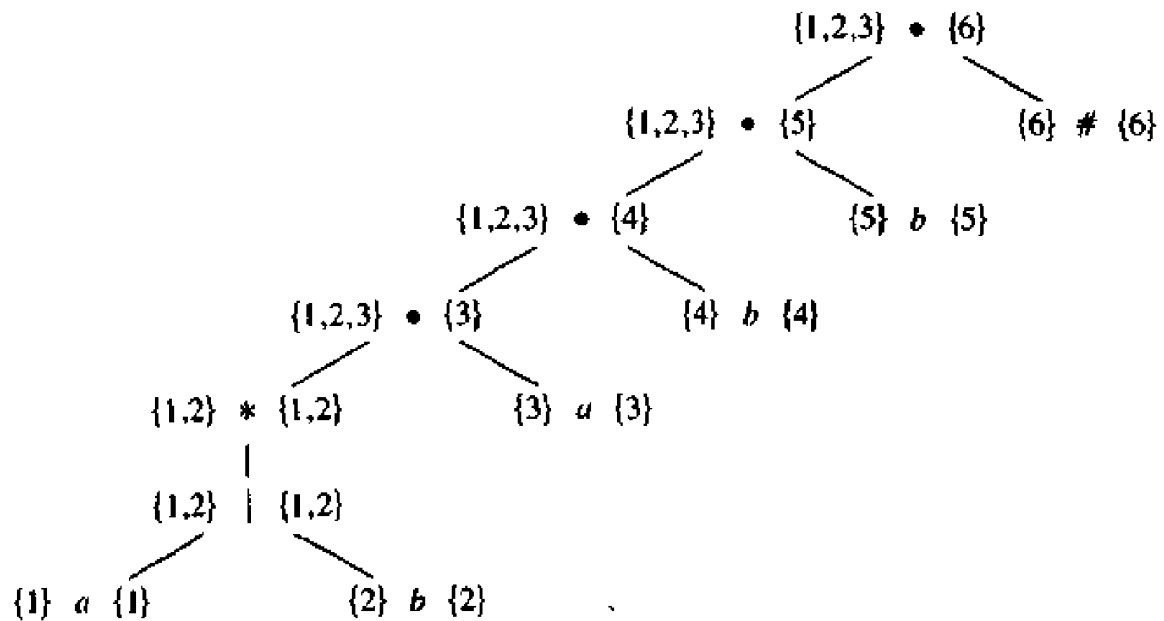


Fig. 3.41. *firstpos* and *lastpos* for nodes in syntax tree for $(a|b)^*abb\#$.

The node labeled $*$ is the only nullable node. Thus, by the if-condition of the fourth rule, *firstpos* for the parent of this node (the one representing expression $(a|b)^*a$) is the union of $\{1, 2\}$ and $\{3\}$, which are the *firstpos*'s of its left and right children. On the other hand, the else-condition applies for *lastpos* of this node, since the leaf at position 3 is not nullable. Thus, the parent of the star-node has *lastpos* containing only 3.

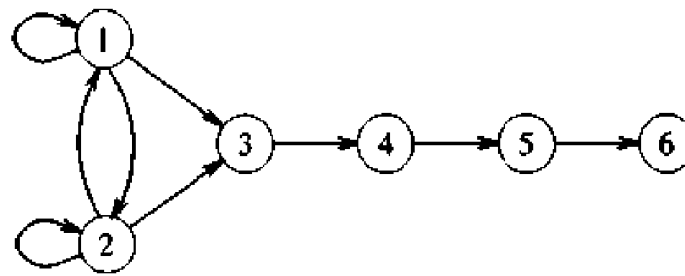
Let us now compute *followpos* bottom up for each node of the syntax tree of Fig. 3.41. At the star-node, we add both 1 and 2 to *followpos*(1) and to *followpos*(2) using rule (2). At the parent of the star-node, we add 3 to *followpos*(1) and *followpos*(2) using rule (1). At the next cat-node, we add 4 to *followpos*(3) using rule (1). At the next two cat-nodes we add 5 to *followpos*(4) and 6 to *followpos*(5) using the same rule. This completes the construction of *followpos*. Figure 3.42 summarizes *followpos*.

NODE	<i>followpos</i>
1	$\{1, 2, 3\}$
2	$\{1, 2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$
6	-

Fig. 3.42. The function *followpos*.

We can illustrate the function *followpos* by creating a directed graph with a node for each position and directed edge from node i to node j if j is in *followpos*(i). Figure 3.43 shows this directed graph for *followpos* of Fig.

3.42.

Fig. 3.43. Directed graph for the function *followpos*.

It is interesting to note that this diagram would become an NFA without ϵ -transitions for the regular expression in question if we:

1. make all positions in *firstpos* of the root be start states,
2. label each directed edge (i, j) by the symbol at position j , and
3. make the position associated with $\#$ be the only accepting state.

It should therefore come as no surprise that we can convert the *followpos* graph into a DFA using the subset construction. The entire construction can be carried out on the positions, using the following algorithm. \square

Algorithm 3.5. Construction of a DFA from a regular expression r .

Input. A regular expression r .

Output. A DFA D that recognizes $L(r)$.

Method.

1. Construct a syntax tree for the augmented regular expression $(r)\#$, where $\#$ is a unique endmarker appended to (r) .
2. Construct the functions *nullable*, *firstpos*, *lastpos*, and *followpos* by making depth-first traversals of T .
3. Construct $Dstates$, the set of states of D , and $Dtran$, the transition table for D by the procedure in Fig. 3.44. The states in $Dstates$ are sets of positions; initially, each state is "unmarked," and a state becomes "marked" just before we consider its out-transitions. The start state of D is *firstpos*(*root*), and the accepting states are all those containing the position associated with the endmarker $\#$. \square

Example 3.23. Let us construct a DFA for the regular expression $(a|b)^*abb$. The syntax tree for $((a|b)^*abb)\#$ is shown in Fig. 3.39(a). *nullable* is true only for the node labeled $*$. The functions *firstpos* and *lastpos* are shown in Fig. 3.41, and *followpos* is shown in Fig. 3.42.

From Fig. 3.41, *firstpos* of the root is $\{1, 2, 3\}$. Let this set be A and

```

initially, the only unmarked state in  $Dstates$  is  $firstpos(root)$ ,
  where  $root$  is the root of the syntax tree for  $(r)\#$ ;
while there is an unmarked state  $T$  in  $Dstates$  do begin
  mark  $T$ ;
  for each input symbol  $a$  do begin
    let  $U$  be the set of positions that are in  $followpos(p)$ 
      for some position  $p$  in  $T$ ,
      such that the symbol at position  $p$  is  $a$ ;
    if  $U$  is not empty and is not in  $Dstates$  then
      add  $U$  as an unmarked state to  $Dstates$ ;
       $Dtran[T, a] := U$ 
    end
  end
end
end

```

Fig. 3.44. Construction of DFA.

consider input symbol a . Positions 1 and 3 are for a , so let $B = followpos(1) \cup followpos(3) = \{1, 2, 3, 4\}$. Since this set has not yet been seen, we set $Dtran[A, a] := B$.

When we consider input b , we note that of the positions in A , only 2 is associated with b , so we must consider the set $followpos(2) = \{1, 2, 3\}$. Since this set has already been seen, we do not add it to $Dstates$, but we add the transition $Dtran[A, b] := A$.

We now continue with $B = \{1, 2, 3, 4\}$. The states and transitions we finally obtain are the same as those that were shown in Fig. 3.39(b). \square

Minimizing the Number of States of a DFA

An important theoretical result is that every regular set is recognized by a minimum-state DFA that is unique up to state names. In this section, we show how to construct this minimum-state DFA by reducing the number of states in a given DFA to the bare minimum without affecting the language that is being recognized. Suppose that we have a DFA M with set of states S and input symbol alphabet Σ . We assume that every state has a transition on every input symbol. If that were not the case, we can introduce a new "dead state" d , with transitions from d to d on all inputs, and add a transition from state s to d on input a if there was no transition from s on a .

We say that string w *distinguishes* state s from state t if, by starting with the DFA M in state s and feeding it input w , we end up in an accepting state, but starting in state t and feeding it input w , we end up in a nonaccepting state, or vice versa. For example, ϵ distinguishes any accepting state from any nonaccepting state, and in the DFA of Fig. 3.29, states A and B are distinguished by the input bb , since A goes to the nonaccepting state C on input bb , while B goes to the accepting state E on that same input.