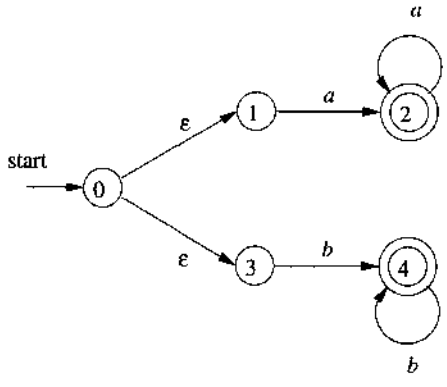

Figure 3.26: NFA accepting **aa*|bb***

1. There are no moves on input ϵ, and
2. For each state s and input symbol a, there is exactly one edge out of s labeled a.

If we are using a transition table to represent a DFA, then each entry is a single state. we may therefore represent this state without the curly braces that we use to form sets.

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers. The following algorithm shows how to apply a DFA to a string.

Algorithm 3.18: Simulating a DFA.

INPUT: An input string x terminated by an end-of-file character **eof**. A DFA D with start state s_0, accepting states F, and transition function *move*.

OUTPUT: Answer "yes" if D accepts x; "no" otherwise.

METHOD: Apply the algorithm in Fig. 3.27 to the input string x. The function $move(s, c)$ gives the state to which there is an edge from state s on input c. The function *nextChar* returns the next character of the input string x. □

Example 3.19: In Fig. 3.28 we see the transition graph of a DFA accepting the language **(a|b)*abb**, the same as that accepted by the NFA of Fig. 3.24. Given the input string *ababb*, this DFA enters the sequence of states $0, 1, 2, 1, 2, 3$ and returns "yes." □

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

Figure 3.27: Simulating a DFA

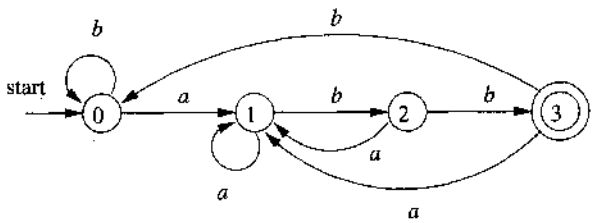

Figure 3.28: DFA accepting (a|b)*abb

3.6.5 Exercises for Section 3.6

! **Exercise 3.6.1:** Figure 3.19 in the exercises of Section 3.4 computes the failure function for the KMP algorithm. Show how, given that failure function, we can construct, from a keyword $b_1 b_2 \cdots b_n$ an $n+1$-state DFA that recognizes $.^* b_1 b_2 \cdots b_n$, where the dot stands for "any character." Moreover, this DFA can be constructed in $O(n)$ time.

Exercise 3.6.2: Design finite automata (deterministic or nondeterministic) for each of the languages of Exercise 3.3.5.

Exercise 3.6.3: For the NFA of Fig. 3.29, indicate all the paths labeled *aabb*. Does the NFA accept *aabb*?

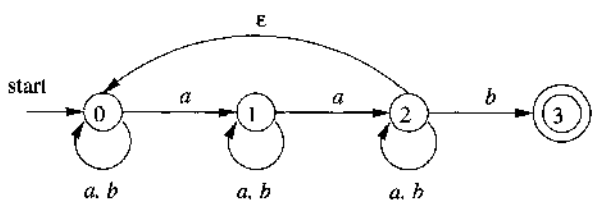

Figure 3.29: NFA for Exercise 3.6.3

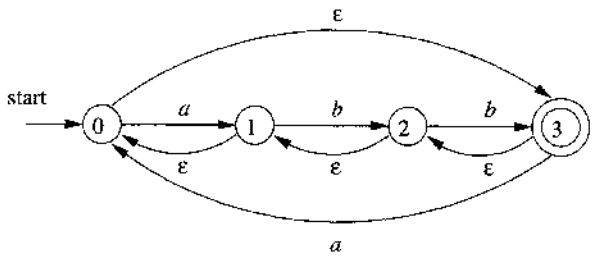

Figure 3.30: NFA for Exercise 3.6.4

Exercise 3.6.4: Repeat Exercise 3.6.3 for the NFA of Fig. 3.30.

Exercise 3.6.5: Give the transition tables for the NFA of:

a) Exercise 3.6.3.

b) Exercise 3.6.4.

c) Figure 3.26.

3.7 From Regular Expressions to Automata

The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software, as was reflected in Section 3.5. However, implementation of that software requires the simulation of a DFA, as in Algorithm 3.18, or perhaps simulation of an NFA. Because an NFA often has a choice of move on an input symbol (as Fig. 3.24 does on input a from state 0) or on ϵ (as Fig. 3.26 does from state 0), or even a choice of making a transition on ϵ or on a real input symbol, its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

In this section we shall first show how to convert NFA's to DFA's. Then, we use this technique, known as "the subset construction," to give a useful algorithm for simulating NFA's directly, in situations (other than lexical analysis) where the NFA-to-DFA conversion takes more time than the direct simulation. Next, we show how to convert regular expressions to NFA's, from which a DFA can be constructed if desired. We conclude with a discussion of the time-space tradeoffs inherent in the various methods for implementing regular expressions, and see how to choose the appropriate method for your application.

3.7.1 Conversion of an NFA to a DFA

The general idea behind the subset construction is that each state of the constructed DFA corresponds to a set of NFA states. After reading input

$a_1 a_2 \cdots a_n$, the DFA is in that state which corresponds to the set of states that the NFA can reach, from its start state, following paths labeled $a_1 a_2 \cdots a_n$.

It is possible that the number of DFA states is exponential in the number of NFA states, which could lead to difficulties when we try to implement this DFA. However, part of the power of the automaton-based approach to lexical analysis is that for real languages, the NFA and DFA have approximately the same number of states, and the exponential behavior is not seen.

Algorithm 3.20: The *subset construction* of a DFA from an NFA.

INPUT: An NFA N.

OUTPUT: A DFA D accepting the same language as N.

METHOD: Our algorithm constructs a transition table *Dtran* for D. Each state of D is a set of NFA states, and we construct *Dtran* so D will simulate "in parallel" all possible moves N can make on a given input string. Our first problem is to deal with ϵ-transitions of N properly. In Fig. 3.31 we see the definitions of several functions that describe basic computations on the states of N that are needed in the algorithm. Note that s is a single state of N, while T is a set of states of N.

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}$ ϵ-*closure*(s).
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Figure 3.31: Operations on NFA states

We must explore those sets of states that N can be in after seeing some input string. As a basis, before reading the first input symbol, N can be in any of the states of ϵ-*closure*(s_0), where s_0 is its start state. For the induction, suppose that N can be in set of states T after reading input string x. If it next reads input a, then N can immediately go to any of the states in *move*(T, a). However, after reading a, it may also make several ϵ-transitions; thus N could be in any state of ϵ-*closure*(*move*(T, a)) after reading input xa. Following these ideas, the construction of the set of D's states, *Dstates*, and its transition function *Dtran*, is shown in Fig. 3.32.

The start state of D is ϵ-*closure*(s_0), and the accepting states of D are all those sets of N's states that include at least one accepting state of N. To complete our description of the subset construction, we need only to show how

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

Figure 3.32: The subset construction

ϵ-*closure*(T) is computed for any set of NFA states T. This process, shown in Fig. 3.33, is a straightforward search in a graph from a set of states. In this case, imagine that only the ϵ-labeled edges are available in the graph. □

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ε )
                if ( u is not in ε-closure(T) ) {
                        add u to ε-closure(T);
                        push u onto stack;
                }
}
```

Figure 3.33: Computing ϵ-*closure*(T)

Example 3.21: Figure 3.34 shows another NFA accepting **(a|b)*abb**; it happens to be the one we shall construct directly from this regular expression in Section 3.7. Let us apply Algorithm 3.20 to Fig. 3.29.

The start state A of the equivalent DFA is ϵ-*closure*(0), or $A = \{0, 1, 2, 4, 7\}$, since these are exactly the states reachable from state 0 via a path all of whose edges have label ϵ. Note that a path can have zero edges, so state 0 is reachable from itself by an ϵ-labeled path.

The input alphabet is $\{a, b\}$. Thus, our first step is to mark A and compute $Dtran[A, a] = \epsilon\text{-}closure(move(A, a))$ and $Dtran[A, b] = \epsilon\text{-}closure(move(A, b))$. Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to 3 and 8, respectively. Thus, $move(A, a) = \{3, 8\}$. Also, $\epsilon\text{-}closure(\{3, 8\} = \{1, 2, 3, 4, 6, 7, 8\}$, so we conclude

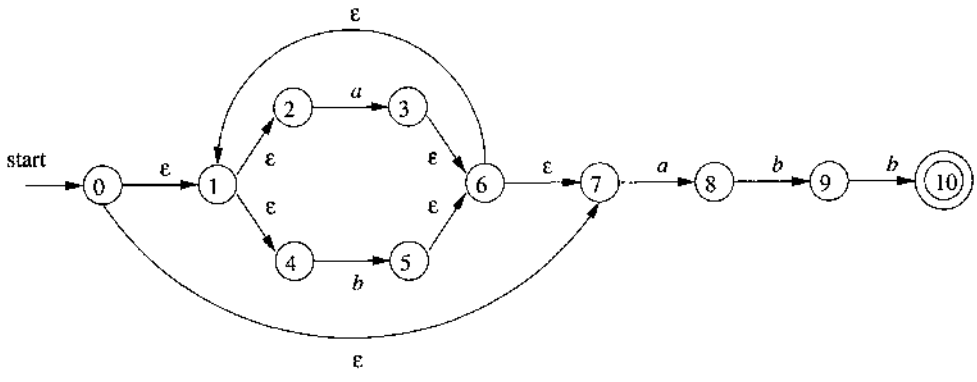

Figure 3.34: NFA N for **(a|b)*abb**

$$Dtran[A, a] = \epsilon\text{-}closure(move(A, a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Let us call this set B, so $Dtran[A, a] = B$.

Now, we must compute $Dtran[A, b]$. Among the states in A, only 4 has a transition on b, and it goes to 5. Thus,

$$Dtran[A, b] = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 6, 7\}$$

Let us call the above set C, so $Dtran[A, b] = C$.

NFA STATE	DFA STATE	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 3, 5, 6, 7, 10\}$	E	B	C

Figure 3.35: Transition table *Dtran* for DFA D

If we continue this process with the unmarked sets B and C, we eventually reach a point where all the states of the DFA are marked. This conclusion is guaranteed, since there are "only" 2^{11} different subsets of a set of eleven NFA states. The five different DFA states we actually construct, their corresponding sets of NFA states, and the transition table for the DFA D are shown in Fig. 3.35, and the transition graph for D is in Fig. 3.36. State A is the start state, and state E, which contains state 10 of the NFA, is the only accepting state.

Note that D has one more state than the DFA of Fig. 3.28 for the same language. States A and C have the same move function, and so can be merged. We discuss the matter of minimizing the number of states of a DFA in Section 3.9.6. □

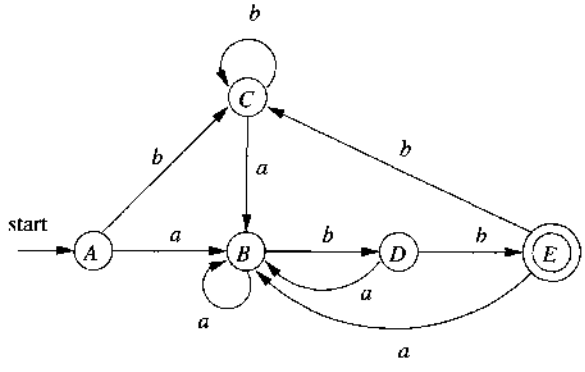

Figure 3.36: Result of applying the subset construction to Fig. 3.34

3.7.2 Simulation of an NFA

A strategy that has been used in a number of text-editing programs is to construct an NFA from a regular expression and then simulate the NFA using something like an on-the-fly subset construction. The simulation is outlined below.

Algorithm 3.22: Simulating an NFA.

INPUT: An input string x terminated by an end-of-file character **eof**. An NFA N with start state s_0, accepting states F, and transition function *move*.

OUTPUT: Answer "yes" if M accepts x; "no" otherwise.

METHOD: The algorithm keeps a set of current states S, those that are reached from s_0 following a path labeled by the inputs read so far. If c is the next input character, read by the function *nextChar*(), then we first compute $move(S, c)$ and then close that set using ϵ-*closure*(). The algorithm is sketched in Fig. 3.37. □

```
1)  S = ε-closure(s0);
2)  c = nextChar();
3)  while ( c != eof ) {
4)          S = ε-closure(move(S, c));
5)          c = nextChar();
6)  }
7)  if ( S ∩ F != ∅ ) return "yes";
8)  else return "no";
```

Figure 3.37: Simulating an NFA

3.7.3 Efficiency of NFA Simulation

If carefully implemented, Algorithm 3.22 can be quite efficient. As the ideas involved are useful in a number of similar algorithms involving search of graphs, we shall look at this implementation in additional detail. The data structures we need are:

1. Two stacks, each of which holds a set of NFA states. One of these stacks, *oldStates*, holds the "current" set of states, i.e., the value of S on the right side of line (4) in Fig. 3.37. The second, *newStates*, holds the "next" set of states S on the left side of line (4). Unseen is a step where, as we go around the loop of lines (3) through (6), *newStates* is transferred to *oldStates*.

2. A boolean array *alreadyOn*, indexed by the NFA states, to indicate which states are in *newStates*. While the array and stack hold the same information, it is much faster to interrogate *alreadyOn*$[s]$ than to search for state s on the stack *newStates*. It is for this efficiency that we maintain both representations.

3. A two-dimensional array *move*$[s, a]$ holding the transition table of the NFA. The entries in this table, which are sets of states, are represented by linked lists.

To implement line (1) of Fig. 3.37, we need to set each entry in array *alreadyOn* to `FALSE`, then for each state s in ϵ-*closure*(s_0), push s onto *oldStates* and set *alreadyOn*$[s]$ to `TRUE`. This operation on state s, and the implementation of line (4) as well, are facilitated by a function we shall call *addState*(s). This function pushes state s onto *newStates*, sets *alreadyOn*$[s]$ to `TRUE`, and calls itself recursively on the states in *move*$[s, \epsilon]$ in order to further the computation of ϵ-*closure*(s). However, to avoid duplicating work, we must be careful never to call *addState* on a state that is already on the stack *newStates*. Figure 3.38 sketches this function.

```
 9)  addState(s) {
10)      push s onto newStates;
11)      alreadyOn[s] = TRUE;
12)      for ( t on move[s, ε] )
13)          if ( !alreadyOn(t) )
14)              addState(t);
15)  }
```

Figure 3.38: Adding a new state s, which is known not to be on *newStates*

We implement line (4) of Fig. 3.37 by looking at each state s on *oldStates*. We first find the set of states *move*$[s, c]$, where c is the next input, and for each