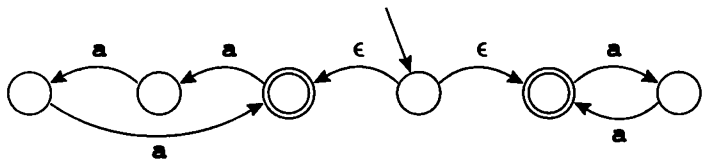
In the start state, on input character a, the automaton can move either right or left. If left is chosen, then strings of a's whose length is a multiple of three will be accepted. If right is chosen, then even-length strings will be accepted. Thus, the language recognized by this NFA is the set of all strings of a's whose length is a multiple of two or three.

On the first transition, this machine must choose which way to go. It is required to accept the string if there is *any* choice of paths that will lead to acceptance. Thus, it must "guess," and must always guess correctly.

Edges labeled with ϵ may be taken without using up a symbol from the input. Here is another NFA that accepts the same language:

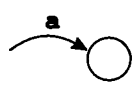


Again, the machine must choose which ϵ-edge to take. If there is a state with some ϵ-edges and some edges labeled by symbols, the machine can choose to eat an input symbol (and follow the corresponding symbol-labeled edge), or to follow an ϵ-edge instead.

CONVERTING A REGULAR EXPRESSION TO AN NFA

Nondeterministic automata are a useful notion because it is easy to convert a (static, declarative) regular expression to a (simulatable, quasi-executable) NFA.

The conversion algorithm turns each regular expression into an NFA with a *tail* (start edge) and a *head* (ending state). For example, the single-symbol regular expression **a** converts to the NFA



The regular expression **ab**, made by combining **a** with **b** using concatenation is made by combining the two NFAs, hooking the head of **a** to the tail of **b**. The resulting machine has a tail labeled by **a** and a head into which the **b** edge flows.

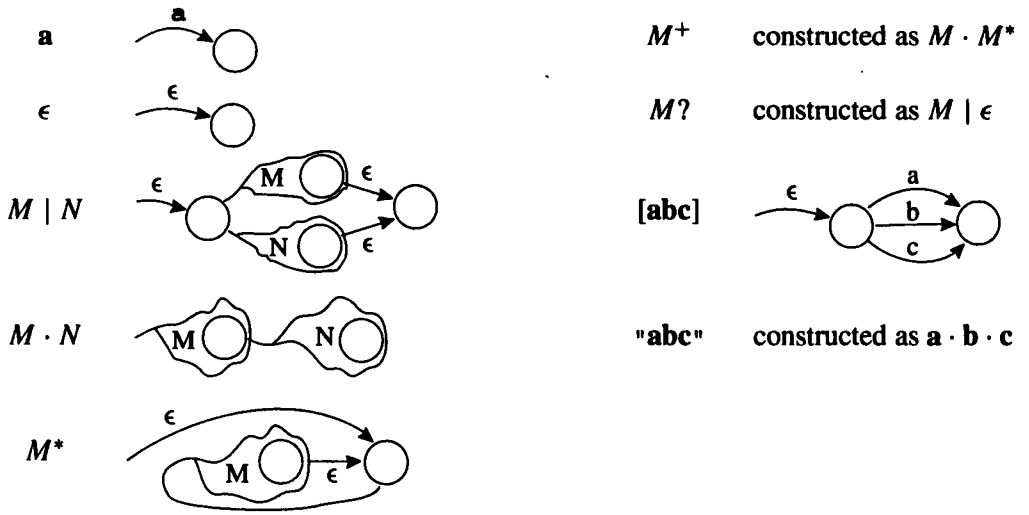
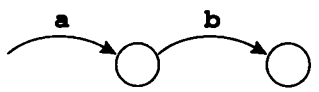


FIGURE 2.6. Translation of regular expressions to NFAs.



In general, any regular expression M will have some NFA with a tail and head:

We can define the translation of regular expressions to NFAs by induction. Either an expression is primitive (a single symbol or ϵ) or it is made from smaller expressions. Similarly, the NFA will be primitive or made from smaller NFAs.

Figure 2.6 shows the rules for translating regular expressions to nondeterministic automata. We illustrate the algorithm on some of the expressions in Figure 2.2 – for the tokens IF, ID, NUM, and **error**. Each expression is translated to an NFA, the "head" state of each NFA is marked final with a different token type, and the tails of all the expressions are joined to a new start node. The result – after some merging of equivalent NFA states – is shown in Figure 2.7.

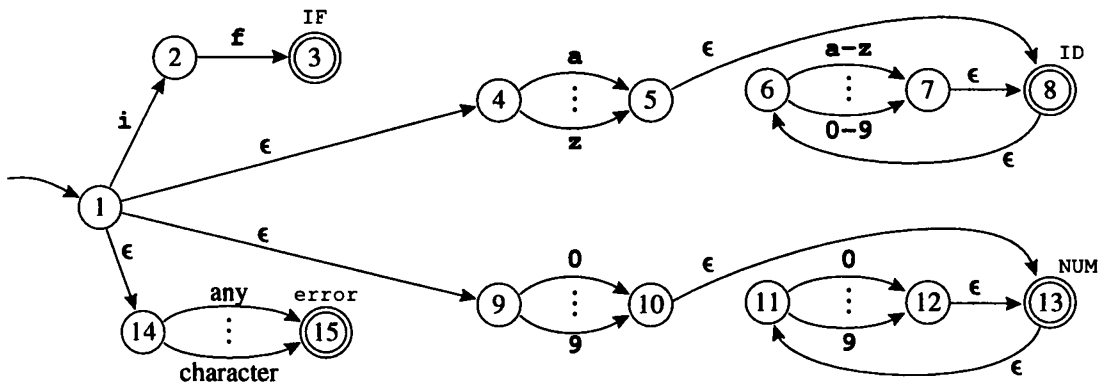


FIGURE 2.7. Four regular expressions translated to an NFA.

CONVERTING AN NFA TO A DFA

As we saw in Section 2.3, implementing deterministic finite automata (DFAs) as computer programs is easy. But implementing NFAs is a bit harder, since most computers don't have good "guessing" hardware.

We can avoid the need to guess by trying every possibility at once. Let us simulate the NFA of Figure 2.7 on the string `in`. We start in state 1. Now, instead of guessing which ϵ-transition to take, we just say that at this point the NFA might take any of them, so it is in one of the states {1, 4, 9, 14}; that is, we compute the *ϵ-closure* of {1}. Clearly, there are no other states reachable without eating the first character of the input.

Now, we make the transition on the character `i`. From state 1 we can reach 2, from 4 we reach 5, from 9 we go nowhere, and from 14 we reach 15. So we have the set {2, 5, 15}. But again we must compute ϵ-closure: from 5 there is an ϵ-transition to 8, and from 8 to 6. So the NFA must be in one of the states {2, 5, 6, 8, 15}.

On the character `n`, we get from state 6 to 7, from 2 to nowhere, from 5 to nowhere, from 8 to nowhere, and from 15 to nowhere. So we have the set {7}; its ϵ-closure is {6, 7, 8}.

Now we are at the end of the string `in`; is the NFA in a final state? One of the states in our possible-states set is 8, which is final. Thus, `in` is an ID token.

We formally define ϵ-closure as follows. Let **edge**(s, c) be the set of all NFA states reachable by following a single edge with label c from state s.

For a set of states S, **closure**(S) is the set of states that can be reached from a state in S without consuming any of the input, that is, by going only through ϵ edges. Mathematically, we can express the idea of going through ϵ edges by saying that **closure**(S) is smallest set T such that

$$T = S \cup \left(\bigcup_{s \in T} \textbf{edge}(s, \epsilon) \right).$$

We can calculate T by iteration:

$T \leftarrow S$
repeat $T' \leftarrow T$
$\quad T \leftarrow T' \cup (\bigcup_{s \in T'} \textbf{edge}(s, \epsilon))$
until $T = T'$

Why does this algorithm work? T can only grow in each iteration, so the final T must include S. If $T = T'$ after an iteration step, then T must also include $\bigcup_{s \in T'} \textbf{edge}(s, \epsilon)$. Finally, the algorithm must terminate, because there are only a finite number of distinct states in the NFA.

Now, when simulating an NFA as described above, suppose we are in a set $d = \{s_i, s_k, s_l\}$ of NFA states s_i, s_k, s_l. By starting in d and eating the input symbol c, we reach a new set of NFA states; we'll call this set **DFAedge**(d, c):

$$\textbf{DFAedge}(d, c) = \textbf{closure}(\bigcup_{s \in d} \textbf{edge}(s, c))$$

Using **DFAedge**, we can write the NFA simulation algorithm more formally. If the start state of the NFA is s_1, and the input string is $c_1, \ldots, c_k$, then the algorithm is:

$d \leftarrow$ **closure**($\{s_1\}$)
for $i \leftarrow 1$ **to** k
$\quad d \leftarrow$ **DFAedge**(d, c_i)

Manipulating sets of states is expensive – too costly to want to do on every character in the source program that is being lexically analyzed. But it is possible to do all the sets-of-states calculations in advance. We make a DFA from the NFA, such that each set of NFA states corresponds to one DFA state. Since the NFA has a finite number n of states, the DFA will also have a finite number (at most 2^n) of states.

DFA construction is easy once we have **closure** and **DFAedge** algorithms. The DFA start state d_1 is just **closure**(s_1), as in the NFA simulation algo-

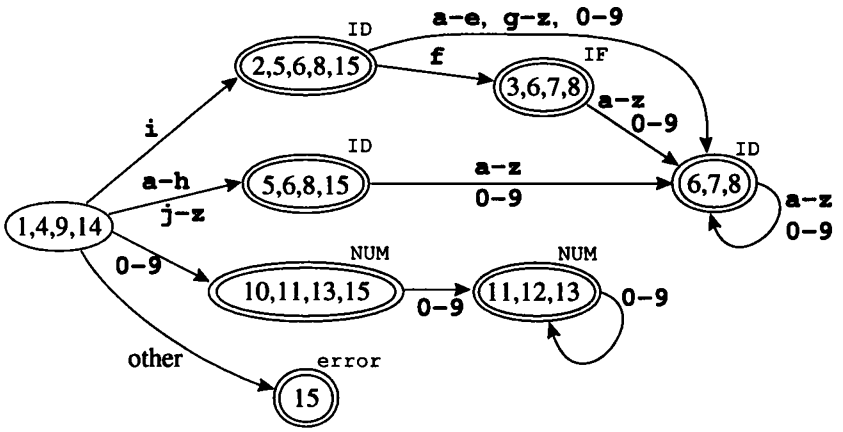


FIGURE 2.8. NFA converted to DFA.

rithm. Abstractly, there is an edge from d_i to d_j labeled with c if d_j = **DFAedge**(d_i, c). We let Σ be the alphabet.

```
states[0] ← {};      states[1] ← closure({s1})
p ← 1;     j ← 0
while j ≤ p
  foreach c ∈ Σ
    e ← DFAedge(states[j], c)
    if e = states[i] for some i ≤ p
      then trans[j, c] ← i
      else p ← p + 1
           states[p] ← e
           trans[j, c] ← p
  j ← j + 1
```

The algorithm does not visit unreachable states of the DFA. This is extremely important, because in principle the DFA has 2^n states, but in practice we usually find that only about n of them are reachable from the start state. It is important to avoid an exponential blowup in the size of the DFA interpreter's transition tables, which will form part of the working compiler.

A state d is *final* in the DFA if any NFA-state in states[d] is final in the NFA. Labeling a state *final* is not enough; we must also say what token is recognized; and perhaps several members of states[d] are final in the NFA. In this case we label d with the token-type that occurred first in the list of

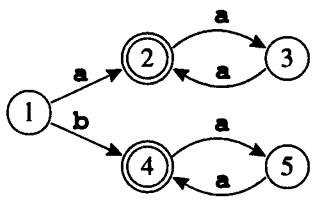
regular expressions that constitute the lexical specification. This is how *rule priority* is implemented.

After the DFA is constructed, the "states" array may be discarded, and the "trans" array is used for lexical analysis.

Applying the DFA construction algorithm to the NFA of Figure 2.7 gives the automaton in Figure 2.8.

This automaton is suboptimal. That is, it is not the smallest one that recognizes the same language. In general, we say that two states s_1 and s_2 are equivalent when the machine starting in s_1 accepts a string σ if and only if starting in s_2 it accepts σ. This is certainly true of the states labeled 5,6,8,15 and 6,7,8 in Figure 2.8; and of the states labeled 10,11,13,15 and 11,12,13. In an automaton with two equivalent states s_1 and s_2, we can make all of s_2's incoming edges point to s_1 instead and delete s_2.

How can we find equivalent states? Certainly, s_1 and s_2 are equivalent if they are both final or both non-final and for any symbol c, trans$[s_1, c]$ = trans$[s_2, c]$; 10,11,13,15 and 11,12,13 satisfy this criterion. But this condition is not sufficiently general; consider the automaton



Here, states 2 and 4 are equivalent, but trans$[2, a] \neq$ trans$[4, a]$.

After constructing a DFA it is useful to apply an algorithm to minimize it by finding equivalent states; see Exercise 2.6.

2.5 Lex: A LEXICAL ANALYZER GENERATOR

DFA construction is a mechanical task easily performed by computer, so it makes sense to have an automatic *lexical analyzer generator* to translate regular expressions into a DFA.

Lex is a lexical analyzer generator that produces a C program from a *lexical specification.* For each token type in the programming language to be lexically analyzed, the specification contains a regular expression and an *action.*

```
%{
/* C Declarations: */
#include "tokens.h"   /* definitions of IF, ID, NUM, ... */
#include "errormsg.h"
union {int ival; string sval; double fval;} yylval;
int charPos=1;
#define ADJ  (EM_tokPos=charPos, charPos+=yyleng)
%}
/* Lex Definitions: */
digits  [0-9]+
%%
/* Regular Expressions and Actions: */
if                        {ADJ; return IF;}
[a-z][a-z0-9]*            {ADJ; yylval.sval=String(yytext);
                            return ID;}
{digits}                 {ADJ; yylval.ival=atoi(yytext);
                            return NUM;}
({digits}"."[0-9]*)|([0-9]*"."{digits})     {ADJ;
                            yylval.fval=atof(yytext);
                            return REAL;}
("--"[a-z]*"\n")|(" "|"\n"|"\t")+    {ADJ;}
.                         {ADJ; EM_error("illegal character");}
```

PROGRAM 2.9. Lex specification of the tokens from Figure 2.2.

The action communicates the token type (perhaps along with other information) to the next phase of the compiler.

The output of Lex is a program in C – a lexical analyzer that interprets a DFA using the algorithm described in Section 2.3 and executes the action fragments on each match. The action fragments are just C statements that return token values.

The tokens described in Figure 2.2 are specified in Lex as shown in Program 2.9.

The first part of the specification, between the `%{···%}` braces, contains `includes` and declarations that may be used by the C code in the remainder of the file.

The second part of the specification contains regular-expression abbreviations and state declarations. For example, the declaration `digits [0-9]+` in this section allows the name `{digits}` to stand for a nonempty sequence of digits within regular expressions.

The third part contains regular expressions and actions. The actions are fragments of ordinary C code. Each action must return a value of type `int`, denoting which kind of token has been found.