The construction of Fig. 2.9(c) is a poor example of this; Fig. 2.5(c) gives a better idea of what is involved. The second graph search is necessary to find the accessible subsets of $Q_{\mathcal{N}}$. These subsets form the states of $\mathcal{D}$. The transition function of $\mathcal{D}$ is given by (2.12). Equation (2.17) is a more explicit statement of (2.12).

$$\delta_{\mathcal{D}}(P,t) = \bigcup_{q \in P} \delta^*_{\mathcal{N}}(\{q\},t). \tag{2.17}$$

The start state of $\mathcal{D}$ is given by (2.14) and the final states of $\mathcal{D}$ are given by (2.15). We are now ready to express this algorithmically. Algorithm 2.2 consists mainly of two applications of Algorithm 1.3, but the terminology has been modified appropriately. (Some minor modifications have also been made to Algorithm 1.3 because we no longer wish to find whether a single node f is reachable from a given start node; instead we wish to find *all* nodes reachable from the start node.)

Algorithm 2.2 Conversion of a non-deterministic machine to a deterministic machine.

```
type non-det state = non-deterministic state;
     det state     = set of non-det state;
var  q,q',q": non-det state;
     d,d': det state;
     t: terminal;
     reached, non-det frontier: set of non-det state;
     det frontier: set of det state;
     estar: non-det state ——> set of non-det state;
          {estar(q) = δ*_N({q},Λ)}
     dnstar: non-det state × terminal ——> set of non-det state;
          {dnstar(q,t) = δ*_N({q},t)}

begin {for each q ∈ Q_N perform graph search to calculate δ*_N({q},Λ)}
for each q ∈ Q_N do
     begin non-det frontier := reached := {q};
     repeat choose and remove q' ∈ non-det frontier;
          for each q" ∈ δ_N(q',Λ) do
               begin if q" ∉ reached
                    then add q" to reached and non-det frontier;
               end;
     until non-det frontier = ∅ ;
     estar(q) := reached; { = δ*_N({q},Λ) }
     end;
```


Algorithm 2.2 (Continued)

```
{Calculate δ*_N({q},t)}
for each q ∈ Q_N do
     for each t ∈ T do
          dnstar(q,t) := estar(δ_N(estar(q),t));
{Perform graph search to find accessible states of D, simultaneously setting
δ_D and F_D.}
S_D := {S_N}; if estar(S_N) ∩ F_N ≠ ∅ then F_D := {S_D} else F_D := ∅ ;
Q_D := det frontier := {S_D};
repeat choose and remove d ∈ det frontier;
     {Construct arcs from d—each arc corresponds to a symbol t ∈ T.}
     for each t ∈ T do
          begin d' := ∅ ;
          {end-point of the arc, d', is the set of all non-deterministic states
          q' reachable under input t from some q ∈ d.}
          for each q ∈ d do d' := d' ∪ dnstar(q,t);
          if d' ∉ Q_D then begin add d' to Q_D and det frontier;
                         {final state?}
                         if F_N ∩ d' ≠ ∅
                         then F_D := F_D ∪ {d'};
                         end;
          δ_D(d,t) := d';
          end;
until det frontier = ∅ ;
end.
```

The implementation of Algorithm 2.2 provides an interesting exercise in the choice of data structures. The data structures used in the implementation of Program 1.1 are certainly *not* applicable to the second graph search in Algorithm 2.2. The main stumbling block is the representation of deterministic states and the set $Q_{\mathcal{D}}$—the *potential* size of $Q_{\mathcal{D}}$ is just too big to contemplate and simple data structures—like a Boolean array—are out of the question. There are also a lot more non-primitive operations to be considered in Algorithm 2.2 than in Algorithm 1.3. So as not to spoil the reader's enjoyment no further hints will be provided and the reader is left to suggest suitable data structures. Note that there is no single "correct" answer to this problem.

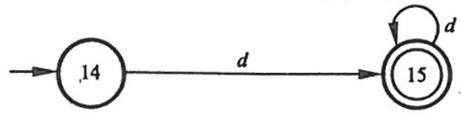
2.4 NUMBERS IN ALGOL 60

As a concrete and sizable example of the techniques introduced in Secs. 2.1 to 2.3 we shall now construct a deterministic finite-state machine recognizing

Backhouse
Syntax of Programming Languages
Theory & Practice
P.H. 1979.

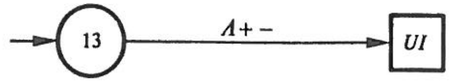
⟨*unsigned integer*⟩ ::= ⟨*digit*⟩ | ⟨*unsigned integer*⟩⟨*digit*⟩

UI



⟨*integer*⟩ ::= ⟨*unsigned integer*⟩ | + ⟨*unsigned integer*⟩ | − ⟨*unsigned integer*⟩

I

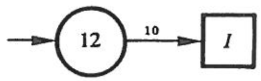


⟨*decimal fraction*⟩ ::= • ⟨*unsigned integer*⟩

DF

⟨*exponent part*⟩ ::= $_{10}$ ⟨*integer*⟩

EP



⟨*decimal number*⟩ ::= ⟨*unsigned integer*⟩ | ⟨*decimal fraction*⟩ | ⟨*unsigned integer*⟩⟨*decimal fraction*⟩

DN

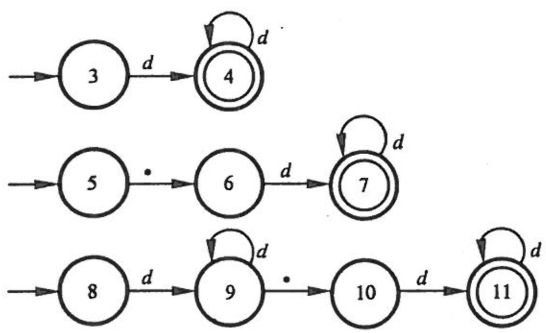
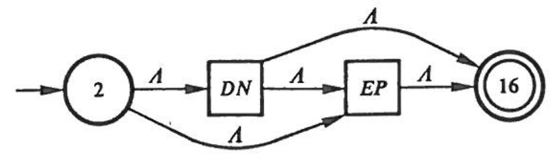


Fig. 2.10 ALGOL 60 ⟨*number*⟩

⟨*unsigned number*⟩ ::= ⟨*decimal number*⟩ | ⟨*exponent part*⟩ | ⟨*decimal number*⟩⟨*exponent part*⟩

UN



⟨*number*⟩ ::= ⟨*unsigned number*⟩ | + ⟨*unsigned number*⟩ | − ⟨*unsigned number*⟩

N

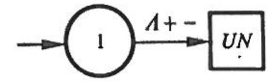


Fig. 2.10 (Continued)

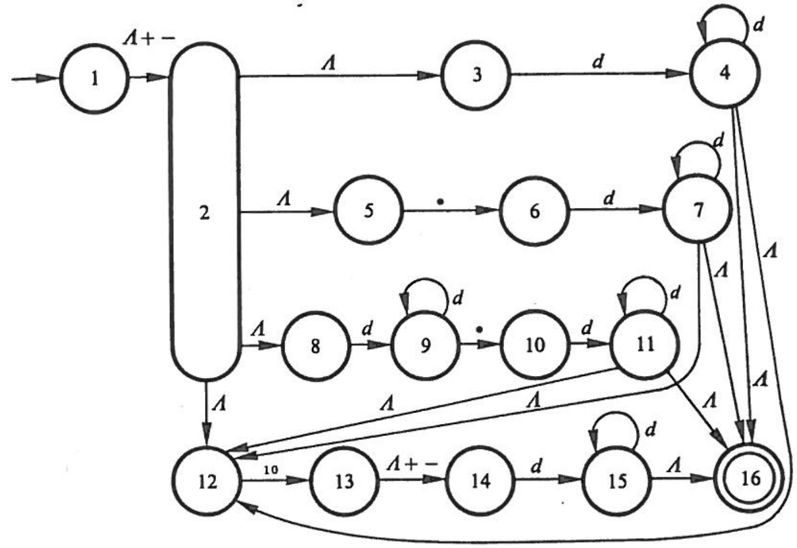


Fig. 2.11 Non-deterministic recognizer of an ALGOL 60 ⟨*number*⟩.

real numbers in ALGOL 60. To do this we have taken the BNF description of ⟨*number*⟩ directly from Sec. 2.5.1 of the "Revised Report" and converted it into a sequence of transition diagrams. These diagrams form Fig. 2.10, each diagram being preceded by the productions from which it was derived. The input alphabet is assumed to consist of the symbols $+$, $-$, $.$, $_{10}$ and d (i.e. ⟨*digit*⟩). A square box represents a diagram which appears elsewhere. Each diagram has been given a one- or two-letter name, the name being an abbreviation of the non-terminal to which it corresponds. To complete a diagram it is necessary to perform a macro-expansion of the square boxes appearing in the diagram. This has been done in Fig. 2.11, which depicts a non-deterministic machine which recognizes ALGOL 60 ⟨*number*⟩s. (Those states in Fig. 2.10 which correspond to states in Fig. 2.11 have been numbered accordingly.)

The astute reader will have observed that, in some instances, Secs. 2.1 to 2.3 do not indicate how a diagram is to be constructed. In such cases (e.g. the diagram for ⟨*unsigned number*⟩) common sense has been used to construct the diagram.

Having constructed a non-deterministic machine the next step is to construct a deterministic machine using Algorithm 2.2. Figure 2.12 depicts the machine

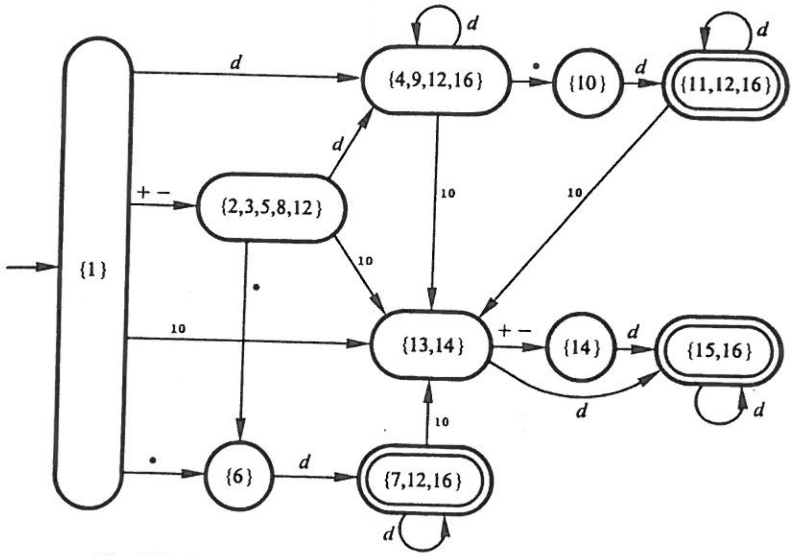


Fig. 2.12 Deterministic recognizer of an ALGOL 60 ⟨*number*⟩.

constructed in this way. (The numbering of the states in Fig. 2.11 also helps to relate the states of the deterministic machine to those of the non-deterministic machine.) For clarity the error state, ∅, has been omitted from Fig. 2.12.

Before concluding this chapter one final comment must be made about Algorithm 2.2. In Sec. 2.3 we took care to calculate only the accessible states of the deterministic machine with the objective of reducing the space required to store the transition function. Nevertheless the machine produced may well be larger than absolutely necessary. This is illustrated by Fig. 2.13 which also depicts a deterministic recognizer of ALGOL 60 ⟨*number*⟩s but has two fewer states than Fig. 2.12. This reduction has been effected by coalescing the state labeled {6} with the state labeled {10}, and the state labeled {7, 12, 16} with the state labeled {11, 12, 16}. This operation is valid because the coalesced states recognize the same languages. That is, if the start state of the machine depicted by Fig. 2.12 were redefined to be {7, 12, 16}, the language recognized by the new machine would be identical to the language recognized by the machine which has {11, 12, 16} as its start state. Similarly the language recognized by the machine with the start state {10} is identical to the language recognized by the machine with start state {6}. However, no other states of Fig. 2.12 may be coalesced without affecting the language recognized and so the machine depicted by Fig. 2.13 is minimal. In general it can be shown that, given any regular language L, there is a *unique minimal* machine which recognizes L and, more importantly, there is an efficient

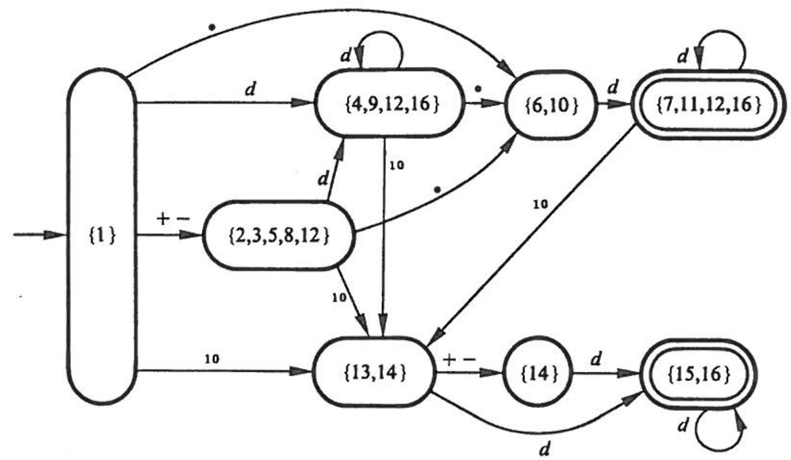


Fig. 2.13 Reduced machine.

algorithm to construct this *reduced* machine from a deterministic recognizer of L. Regrettably we shall not discuss the algorithm here primarily because, although it is easy to describe an inefficient algorithm, it is quite difficult to design and implement a respectably efficient algorithm to perform this task.

Appendix—Proof of Theorem 2.1

The statement of Theorem 2.1 is reproduced below.

Theorem 2.1 Let $\mathcal{N} = (Q, T, \delta, S, F)$ be a non-deterministic finite-state machine. Then, for all $n > 0$, $t_1, t_2, \ldots, t_n \in T$ and $P \subseteq Q$,

$$\delta^*(P, t_1 t_2 \ldots t_n) = \delta^*(\ldots \delta^*(\delta^*(P, t_1), t_2), \ldots, t_n). \tag{2.18}$$

Verbally, a proof might proceed as follows.

Let $q \in P$. A path from q to q' which spells out $t_1 t_2 \ldots t_n$ consists of a sequence of Λ-arcs followed by an arc labeled t_1 followed by a sequence of Λ-arcs followed by an arc labeled t_2, etc. On the other hand, according to (2.18) a path spelling out $t_1 t_2 \ldots t_n$ consists of (a sequence of Λ-arcs followed by an arc labeled t_1 followed by a sequence of Λ-arcs) followed by (a sequence of Λ-arcs followed by an arc labeled t_2 followed by a sequence of Λ-arcs), etc. Thus a proof of Theorem 2.1 involves, essentially, a proof that a sequence of Λ-arcs followed by a sequence of Λ-arcs is identical to a sequence of Λ-arcs. (Hence, also, our statement that the theorem is "intuitively obvious".) In the formal proof below we have introduced additional notation. Had we not done so the proof would have seemed very complex, although it is not. The moral is: think carefully about notation—it can obscure simple ideas, it can also clarify complex ideas.

Proof of Theorem 2.1 Let $q \in Q$. Let e: $2^Q \longrightarrow 2^Q$ denote the function defined by $e(P) = \delta(P, \Lambda)$. Let us also denote function application by $\cdot$. Thus $e \cdot P$ denotes $e(P)$ and $e \cdot e \cdot P$ denotes $e(e(P))$. Let $e^*: 2^Q \longrightarrow 2^Q$ be defined by $e^* \cdot P = \delta^*(P, \Lambda)$. Then, clearly,

$$e^* \cdot P \supseteq P$$

and so

$$e^* \cdot e^* \cdot P \supseteq e^* \cdot P. \tag{2.19}$$

Moreover

$$e^* \cdot P \supseteq e^* \cdot e^* \cdot P. \tag{2.20}$$

For, if $q \in e^* \cdot e^* \cdot P$, then $\exists$ integers $n, m \geq 0$ such that $q \in e^n \cdot e^m \cdot P$. (*Note*: e^m denotes $e \cdot e \cdot e \cdot \cdots \cdot e$, i.e. m applications of e.) That is,

$$q \in e^{n+m} \cdot P \subseteq e^* \cdot P.$$

From (2.19) and (2.20) we have

$$e^* \cdot e^* = e^*. \tag{2.21}$$

(That is, a sequence of Λ-arcs followed by a sequence of Λ-arcs is identical to a sequence of Λ-arcs.) Now, returning to the use of parentheses to denote function application, we have by definition

$$\delta^*(P, t_1 \ldots t_n) = e^*(\delta(\cdots e^*(\delta(e^*(\delta(e^*(P), t_1)), t_2)), \ldots, t_n)).$$

Thus, applying (2.21)

$$\begin{aligned}\delta^*(P, t_1 \ldots t_n) &= e^*(\delta(e^* \cdots (e^*(\delta(e^*(e^*(\delta(e^*(P), t_1))), t_2))), \ldots, t_n)) \\ &= \delta^*(\cdots \delta^*(\delta^*(P, t_1), t_2), \ldots, t_n).\end{aligned}$$ □

EXERCISES

2.1 Construct transition diagrams corresponding to the following regular grammars.

(a) $G_1 = (\{S,A,B,C,D\}, \{a,b,c,d\}, P, S)$, where P consists of

$$\begin{array}{ll} S \rightarrow aA & S \rightarrow B \\ A \rightarrow abS & A \rightarrow bB \\ B \rightarrow b & B \rightarrow cC \\ C \rightarrow D & \\ D \rightarrow bB & D \rightarrow d \end{array}$$

(b) $G_2 = (\{S,A,B,C,D\}, \{a,b,c,d\}, P, S)$, where P consists of

$$\begin{array}{ll} S \rightarrow Aa & S \rightarrow B \\ A \rightarrow Cc & A \rightarrow Bb \\ B \rightarrow Bb & B \rightarrow a \\ C \rightarrow D & C \rightarrow Bab \\ D \rightarrow d & \end{array}$$

Use your transition diagrams to construct a *left-linear* grammar defining $L(G_1)$ and a *right-linear* grammar defining $L(G_2)$. Check your answers firstly by constructing derivation sequences for the following sentences.

(a) of G_1: *abb aababb b cd cbb*

(b) of G_2: *a aba aabca dca*

Secondly, construct a number of strings which are *not* sentences of G_1 or G_2 and check that the grammars you have constructed do not also generate these strings. Comment on the value of these checks.

2.2 Formalize (a) the process of converting a left-linear grammar into a transition diagram, and (b) the process of converting a transition diagram into a right-linear grammar.

2.3 Construct regular expressions denoting the following languages.

(a) The set of statement lists. A statement list is a list of basic or dummy statements. Each statement in the list is separated from the next by one or

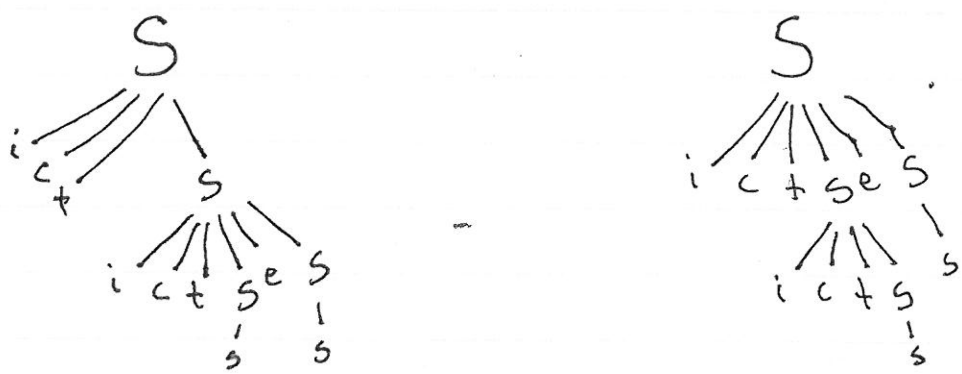
Parse Trees con't.

- Consider the following grammar:
  (context free grammar - cfg)

$S \to ictS$ // S represents start symbol
$S \to ictSeS$ (non-terminal)
$S \to s$

// lower-case letters - i, c, t are terminals.

- Consider the sentence : ictictses
  There are two distinct parse trees for this sentence.
  The above grammar is ambiguous.



associate
i = if
c = condition
t = then
e = else
s = statement

- We have two distinct program segments!

```
if cond then
   if cond then
      { statement ... some code
      }
else
   { statement ... some other code
   }
```

```
if cond then
   if cond then
      { statement ... some code
      }
else { statement ... some other code
     }
```