2.4.4 DFA to Minimal DFA

As the final step in the RE→DFA construction, we can employ an algorithm to minimize the number of states in the automaton. The subset construction can produce a DFA that has a large set of states. While the size of the DFA does not affect its asymptotic complexity, it does determine the recognizer’s footprint in memory. On modern computers, the speed of memory accesses often governs the speed of computation. A smaller recognizer may fit better into the processor’s lowest level of cache memory, producing faster average accesses.

To reduce the size of the DFA and, thus, its transition table, the scanner generator can apply a DFA minimization algorithm. The best known and asymptotically fastest algorithm, Hopcroft’s algorithm, constructs a minimal DFA from an arbitrary DFA by grouping together states into sets that are equivalent. Two DFA states are equivalent when they produce the same behavior on any input string. The algorithm finds the largest possible sets of equivalent states; each set becomes a state in the minimal DFA.

The algorithm constructs a set partition, \( P = \{ p_1, p_2, p_3, \ldots, p_m \} \) of the DFA states. Each \( p_i \) contains a set of equivalent DFA states. More formally, it constructs a partition with the smallest number of sets, subject to the following two rules:

1. \( \forall c \in \Sigma, \text{ if } d_i, d_j \in p_i; d_i \xrightarrow{c} d_i, d_j \xrightarrow{c} d_j; \text{ and } d_k \in p_i; \text{ then } d_j \in p_i. \)

2. If \( d_i, d_j \in p_i \) and \( d_i \in D_\text{accept}, \text{ then } d_j \in D_\text{accept}. \)

Rule 1 mandates that two states in the same set must, for every character \( c \in \Sigma \), transition to states that are, themselves, members of a single set in the partition. Rule 2 rules states that any single set contains either accepting states or non-accepting states, but not both.

These two properties not only constrain the final partition, \( P \), but they also lead to a construction for \( P \). The algorithm starts with the coarsest partition on behavior, \( P_0 = \{ D_\text{accept}, \{ D - D_\text{accept} \} \} \). It then iteratively “refines” the partition until both properties hold true for each set in \( P \).

To refine the partition, the algorithm splits sets based on the transitions out of DFA states in the set.

Figure 2.8 shows how the algorithm uses transitions to split sets in the partition. In panel (a), all three DFA states in set \( p_1 \) have transitions to DFA states in \( p_2 \) on the input character \( a \). Specifically, \( d_i \xrightarrow{a} d_x \), \( d_j \xrightarrow{a} d_y \), and \( d_k \xrightarrow{a} d_z \). Since \( d_i, d_j, d_k \in p_1 \), and \( d_x, d_y, d_z \in p_2 \), sets \( p_1 \) and \( p_2 \) conform to rule 1. Thus, the states in \( p_1 \) are behaviorally equivalent on \( a \), so \( a \) does not induce the algorithm to split \( p_1 \).

By contrast, panel (b) shows a situation where the character \( a \) induces a split in set \( p_1 \). As before, \( d_i \xrightarrow{a} d_x \), \( d_j \xrightarrow{a} d_y \), and \( d_k \xrightarrow{a} d_z \), but \( d_x \in p_2 \) while \( d_y, d_z \in p_3 \). This situation violates rule 1, so \( a \) induces the
algorithm to split \( p_1 \) into two sets, \( p_4 = \{ d_i \} \) and \( p_5 = \{ d_j, d_k \} \), shown in panel (c).

The algorithm, shown in Figure 2.9, builds on these ideas. Given an arbitrary DFA, it constructs a partition that represents the minimal DFA. To simplify the exposition, it uses two copies of the partition. At each stage, \( \text{Partition} \) holds the current approximation to the minimal DFA, while the algorithm builds the next approximation in \( \text{NextPartition} \).

To start, the algorithm constructs the coarsest partition consistent with rule 2, \( \{ D_A, \{ D - D_A \} \} \). This choice has two consequences. First, since each set in the final partition is constructed by splitting a set in an earlier approximation, it ensures that no set in the final partition will contain both accepting and nonaccepting states. Second, by choosing the largest sets consistent with rule 2, it imposes the minimum constraints on the splitting process which, in turn, can lead to larger sets in the final partition. (Larger sets means fewer states in the final DFA.)

The algorithm operates from a worklist of states, starting with the initial partition \( \{ D_A, \{ D - D_A \} \} \). It repeatedly picks a set \( s \) from the worklist and uses that set to refine the partition in \( \text{NextPartition} \) by splitting sets based on their transitions into \( s \).

To identify states that must split because of a transition into set \( s \) on some character \( c \), the algorithm inverts the transition function. It computes the set of DFA states that can reach a state in set \( s \) on a transition labeled \( c \) and assigns that set to \( \text{Image} \). It then systematically examines each set \( q \) that has a state in \( \text{Image} \) to see if \( \text{Image} \) induces a split in \( q \). If \( \text{Image} \) divides \( q \) into non-empty sets \( q_1 \) and \( q_2 \), it removes \( q \) from both \( \text{Partition} \) and \( \text{NextPartition} \) and then adds both \( q_1 \) and \( q_2 \) to \( \text{NextPartition} \).

All that remains, in processing \( q \) with respect to \( c \), is to update the worklist. If \( q \) is on the worklist, then the algorithm replaces \( q \) with both \( q_1 \) and \( q_2 \). The rationale is simple: \( q \) was on the worklist for some potential effect; that effect might be from some character other than \( c \).
2.4 From Regular Expression to Scanner

\[
\begin{align*}
\text{Partition} & \leftarrow \{D_A, \{D - D_A\}\} \\
\text{Worklist} & \leftarrow \{D_A, \{D - D_A\}\}
\end{align*}
\]

\(\text{while} (\text{Worklist} \neq \emptyset)\)

- select a set \(s\) from Worklist and remove it

- for each character \(c \in \Sigma\)
  
  - Image \(\leftarrow \{x \mid \delta(x, c) \in s\}\)

  - for each set \(q \in \text{Partition}\) that has a state in Image
    
    - \(q_1 \leftarrow q \cap \text{Image}\)
    
    - \(q_2 \leftarrow q - q_1\)

  - if \(q_1 \neq \emptyset\) and \(q_2 \neq \emptyset\) then  // split \(q\) around \(s\) and \(c\)
    
    - remove \(q\) from Partition
    
    - Partition \(\leftarrow\) Partition \(\cup\) \(q_1\) \(\cup\) \(q_2\)

  - if \(q \in \text{Worklist}\) then  // and update the Worklist
    
    - remove \(q\) from Worklist
    
    - WorkList \(\leftarrow\) WorkList \(\cup\) \(q_1\) \(\cup\) \(q_2\)

- else if \(|q_1| \leq |q_2|\)
  
  - then WorkList \(\leftarrow\) WorkList \(\cup\) \(q_1\)
  
  - else WorkList \(\leftarrow\) WorkList \(\cup\) \(q_2\)

  - if \(s = q\) then  // need another \(s\)
    
    - break

\textbf{Figure 2.9}  DFA Minimization Algorithm

so all of the DFA states in \(q\) need to be represented on the worklist.

If, on the other hand, \(q\) is not on the worklist, then the only effect that splitting \(q\) can have on other sets is to split them. Assume that some set \(r\) has transitions on letter \(e\) into \(q\). Dividing \(q\) might create the need to split \(r\) into sets that transition to \(q_1\) and \(q_2\). In this case, either of \(q_1\) or \(q_2\) will induce the split, so the algorithm can choose between them. Using the smaller set will lead to faster execution; for example, computing Image takes time proportional to the size of the set.

To construct the new DFA from the final Partition, we can create a state to represent each set \(p_i \in \text{Partition}\) and add the appropriate transitions between these new representative states. For the state representing \(p_m\), we add a transition to the state representing \(p_n\) on character \(c\) if some \(d_j \in p_m\) has a transition on \(c\) to some \(d_k \in p_n\). The construction ensures that, if \(d_j \xrightarrow{c} d_k\), where \(d_j \in p_m\) and \(d_k \in p_n\), then every state in \(p_m\) has a similar transition on \(c\) to a state in \(p_n\). If this condition did not hold, the algorithm would have split \(p_m\) around the transitions on \(c\). The resulting DFA is minimal; the proof is beyond our scope.


<table>
<thead>
<tr>
<th>Step</th>
<th>Partition on Entry</th>
<th>Worklist</th>
<th>s</th>
<th>c</th>
<th>Image</th>
<th>q</th>
<th>q₁</th>
<th>q₂</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(p₀: {s₀, s₃}, \ p₁: {s₀, s₁, s₂, s₄})</td>
<td>(p₀, p₁)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>(p₀: {s₀, s₃}, \ p₁: {s₀, s₁, s₂, s₄})</td>
<td>(p₁)</td>
<td>(p₀)</td>
<td>e</td>
<td>(s₂, s₄)</td>
<td>(p₁)</td>
<td>(s₂, s₄)</td>
<td>(s₀, s₁)</td>
<td>split (p₁ \rightarrow p₂, p₃)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p₂, p₃)</td>
<td>(p₀)</td>
<td>f</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p₂, p₃)</td>
<td>(p₀)</td>
<td>i</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>(p₀: {s₀, s₃}, \ p₂: {s₂, s₄}), (p₃: {s₀, s₁})</td>
<td>(p₂)</td>
<td>(p₂)</td>
<td>e</td>
<td>(s₁)</td>
<td>(p₃)</td>
<td>(s₁)</td>
<td>(s₀)</td>
<td>split (p₃ \rightarrow p₄, p₅)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p₄, p₅)</td>
<td>(p₂)</td>
<td>f</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p₄, p₅)</td>
<td>(p₂)</td>
<td>i</td>
<td>(s₁)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td>3</td>
<td>(p₀: {s₀, s₃}, \ p₂: {s₂, s₄}), (p₄: {s₁}), (p₅: {s₀})</td>
<td>(p₅)</td>
<td>(p₄)</td>
<td>e</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p₅)</td>
<td>(p₄)</td>
<td>f</td>
<td>(s₀)</td>
<td>(p₅)</td>
<td>(s₀)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(p₅)</td>
<td>(p₄)</td>
<td>i</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>(p₀: {s₀, s₃}, \ p₂: {s₂, s₄}), (p₄: {s₁}), (p₅: {s₀})</td>
<td>(∅)</td>
<td>(p₅)</td>
<td>e</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(∅)</td>
<td>(p₅)</td>
<td>f</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(∅)</td>
<td>(p₅)</td>
<td>i</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>(∅)</td>
<td>none</td>
</tr>
</tbody>
</table>

(a) Iterations of Hopcroft's Algorithm on the Original DFA for \(\text{“fee | fie”}\)

(b) Original DFA for \(\text{“fee | fie”}\)

(c) Minimal DFA for \(\text{“fee | fie”}\)

**Figure 2.10** Applying the DFA Minimization Algorithm

**Examples**

As a first example, consider the DFA for \(\text{“fee | fie”}\) shown in Figure 2.10.b. Panel (a) shows the progress of Hopcroft’s algorithm on this DFA.

The first line, step 0, shows the initial configuration of the algorithm. Both *Partition* and *Worklist* contain two sets: \(\{D_A, \{D - D_A\}\}\). \(D_A\) is labeled \(p₀\) while \(\{D - D_A\}\) is labeled \(p₁\).

The algorithm enters the while loop and removes \(p₀\) from *Worklist*; it becomes \(s\). The algorithm iterates over the characters in \(\Sigma\), in the order \(e, f,\) and \(i\). For \(e\), \(p₀\) splits \(p₁\) into two sets: \(p₂: \{s₀, s₁\}\) and \(p₃: \{s₂, s₄\}\). The algorithm removes \(p₁\) from *Partition* and adds \(p₂\) and \(p₃\). Next, it removes \(p₁\) from *Worklist* and adds \(p₂\) and \(p₃\). For \(f\) and \(i\), no edges...
enter \( p_0 \). Thus, \( \text{Image} \) is empty and no splits occur.

The second iteration proceeds in a similar fashion. It chooses \( p_2 \). The character \( e \) splits \( p_3 \) and causes an update to both \( \text{Partition} \) and \( \text{Worklist} \). For \( f \), the \( \text{Image} \) set is empty. For \( i \), the \( \text{Image} \) set contains \( s_i \). Because the algorithm already split \( p_3 \) around \( e \), this situation does not cause a split. It will, however, add a transition to the final DFA.

The third iteration chooses \( p_4 \) from the worklist. Both \( e \) and \( i \) produce empty \( \text{Image} \) sets. With \( f \), the \( \text{Image} \) set contains \( s_0 \). Because \( p_5 \), which contains \( s_0 \), is a singleton set, it cannot be split. This situation, however, will add a transition to the final DFA.

The fourth and final iteration takes \( p_5 \) from the worklist. For each of \( e \), \( f \), and \( i \), the \( \text{Image} \) set is empty. Thus, the iteration splits no sets. It adds no transitions. It finishes with an empty worklist. The resulting minimal DFA is shown Figure 2.10.c.

As a second example, consider the DFA for \( a (b \mid c)^* \) produced by Thompson’s construction and the subset construction, shown in Figure 2.11.a. The initial partition is \( \{ p_0 : \{ s_1, s_2, s_3 \}, p_1 : \{ s_0 \} \} \).

The algorithm first selects \( p_0 \) and examines each of \( a \), \( b \), and \( c \). For \( a \), \( \text{Image} \) contains \( s_0 \) which is in a singleton set, \( p_1 \). Thus, \( a \) introduces a transition for the final DFA, but no split. For both \( b \) and \( c \), \( \text{Image} \) is \( \{ s_1, s_2, s_3 \} \), which is exactly \( p_0 \). Thus, \( q_2 \) is empty and no splits occur.

Next the algorithm removes \( p_1 \) and examines each of \( a \), \( b \), and \( c \). Since no transitions enter \( p_1 \), the \( \text{Image} \) set is empty for each letter. No further splits occur. The original two set partition is the final partition. The final DFA has two states, as shown in the margin. Recall that this is the DFA that we suggested a human would derive. After minimization, the automatic techniques produce the same result.

### 2.4.5 Using a DFA as a Scanner

The tools in the three previous sections provide an algorithmic path from an RE to a minimal DFA. As we saw in Figure 2.2, a DFA can be simulated with a simple table-driven skeleton. Taken together, these suggest that we can automate scanner construction by taking REs for all of the words in a programming language, combining them into a single RE, and using the resulting DFA as a scanner. Reality, however,
is more complex. Scanners and DFAs differ in two critical ways that affect how we formulate and build RE-based scanners.

**Model of Execution**

A DFA reads all of its input and accepts the input if its last state is a final state. That is, a DFA tries to find one word. By contrast, a scanner reads enough input to find the next word in the input stream. The scanner leaves the input stream in a state from which it can find the next word.

This difference necessitates a new model of execution. Rather than exhausting the input stream, the scanner simulates the DFA until it hits an error—that is, until it is in some state $d_i$ with input character $c$ such that $\delta(d_i, c) = s_e$, the error state. We also define $\delta(d_i, \text{eof}) = s_e, \forall d_i \in D$.

If $d_i$ is an accepting state, $d_i \in D_A$, the scanner has found a word. If $d_i$ is not an accepting state, the scanner may have passed through such a state on its way to $d_i$. To determine if it did, the scanner must back up, one character at a time until it either reaches an accepting state or it exhausts the lexeme.

This scheme adds some work to the implementation. The scanner must either record states or invert $\delta$. Either approach takes time proportional to the number of scanned characters. A character may be scanned multiple times; Section 2.5.1 shows a method for avoiding the worst case of this behavior.

**Finding Syntactic Categories**

A DFA returns a binary answer: it either accepts or rejects the input. By contrast, a scanner returns a token, $\langle \text{lexeme}, \text{category} \rangle$, that gives the spelling and syntactic category of the next word. It indicates an error with an invalid token.

If we construct the DFA so that each final state maps to a single category, then the scanner can find the category with a simple table lookup. However, this scheme requires that we build the DFA in a way that preserves the mapping of final states to categories.

Most scanner generators take, as input, a list of REs, $r_1, r_2, \ldots, r_k$, each of which defines the spelling of some category. The obvious way to build a single DFA is to construct a single RE, $(r_1 \mid r_2 \mid \ldots \mid r_k)$, and construct a DFA from this RE. However, Thompson’s construction will immediately unify the final states, destroying the mapping from $d_i \in D_A$ to categories.

Instead, the scanner generator can build an NFA for each rule, using Thompson’s construction. It can join those NFAs into a single DFA, with a new start state and $\epsilon$-transitions, and use the subset construction to build a DFA that simulates the NFA. This process preserves the final states for each rule.
Identifying Keywords

Most programming languages reserve the keywords that identify critical parts of the syntax, words such as if, then, and while. In a typical scanner and parser, each keyword has its own syntactic category—a category with only one word. The compiler writer faces a choice: specify each keyword with its own rule, or fold keywords into the rule for identifiers and recognize them with some other mechanism. Either approach works and can lead to an efficient scanner.

With a separate rule for each keyword, the scanner can return the appropriate category using the same mechanism used for other categories, such as number or identifier. The extra rules may add minor cost to scanner generation, for the extra rules and extra states, and may produce a DFA with more states. However, since the process produces a DFA, the resulting scanner will still require $O(1)$ time per character.

As an alternative, most scanners build a table of all identifier names. This table serves as a start on the compiler’s symbol table and as a way to map identifier names into small integers so that they can be represented and compared efficiently (see Section 4.4). If the compiler writer pre-loads the symbol table with the keywords and their syntactic categories, the scanner will find the keywords as previously seen and categorized identifiers, and will return the appropriate category for each.

If two rules share a common prefix, the subset construction will merge their final states. When the subset construction merges final states, the scanner generator must decide which syntactic category it will return for that final state. In practice, scanner generators let the compiler writer specify a precedence among syntactic categories. The scanner generator assigns to the final state the category with the highest precedence.

Minimization poses another challenge. Hopcroft’s algorithm immediately combines all of the final states into a single partition, destroying the property that final states map to syntactic categories. If, however, the scanner generator constructs an initial partition that places the final states for each syntactic category in a distinct set in the final partition, then the rest of the algorithm will maintain that property.

The resulting DFA may be larger than the minimal DFA that results from grouping all final states into the same partition. However, the larger DFA has the property that the compiler needs: each final state maps to a specific syntactic category.
The Role of Whitespace

Programmers often refer to blanks and tabs, when used to format code, as whitespace. In most languages, whitespace has no intrinsic meaning. Scanners for these languages typically recognize and discard whitespace. The primary impact of whitespace arises from its absence in the regular expressions that define words in the language.

For example, the fact that the RE for an identifier or keyword name does not include blank or tab forces an RE-based scanner to separate do and i in a sentence such as:

```plaintext
do i = 1 to 100
```

For similar reasons, the RE for identifier does not contain +, -, *, or /. This fact ensures that “a * b” scans the same as “a*b”.

**Fortran** In FORTRAN 66 blanks are not significant. That is, “n a m e” and “name” refer to the same identifier. This rule complicates scanning. The header of a FORTRAN do loop might read:

```plaintext
do 10 i = 1,100
```

where 10 is the label of the last statement in the loop body, i is the loop’s index variable, and i’s value runs from 1 to 100. (The increment defaults to one unless specified.)

Of course, do10i is a valid variable name. To differentiate between these two statements:

```plaintext
do 10 i = 1
   do 10 i = 1,100
```

a scanner must read beyond the = and 1 to the comma. The comma proves that the second statement is a loop header, and the scanner can separate do10i into three words, do, 10, and i. Few, if any languages, have followed FORTRAN’s example.

**Python** Python takes the opposite approach: not only are blanks significant, but the number of blanks at the start of a line determines the meaning of a Python program. Rather than using bracket constructs, such as `{` and `}` or `begin` and `end`, to indicate block structure, Python relies on changes in indentation.

A simple way of handling leading blanks in Python is to add a rule that recognizes an end-of-line followed by zero or more blanks. The scanner can then test the length of the lexeme. If its length is identical to the previous token in this category, it returns the result of calling the scanner again. If its length differs, the scanner can return a category indicating the start of a block or the end of a block, as appropriate.