Lexical Analysis

2

We start this chapter by describing the task of lexical analysis. Then we present regular expressions as specifications for this task. Regular expressions can be automatically converted into nondeterministic finite automata, which implement lexical analysis. Nondeterministic finite automata can be made deterministic, which is preferred for implementing lexical analyzers, often called *scanners*. Another transformation on the resulting deterministic finite automata attempts to reduce the sizes of the automata. These three steps together make up an automatic process generating lexical analyzers from specifications. Another module working in close cooperation with such a finite automaton is the *screener*. It filters out keywords, comments, etc., and may do some bookkeeping or conversion.

2.1 The Task of Lexical Analysis

Let us assume that the source program is stored in a file. It consists of a sequence of characters. Lexical analysis, i.e., the scanner, reads this sequence from left to right and decomposes it into a sequence of lexical units, called *symbols*. Scanner, screener, and parser may work in an integrated way. In this case, the parser calls the combination scanner-screener to obtain the next symbol. The scanner starts the analysis with the character that follows the end of the last found symbol. It searches for the longest prefix of the remaining input that is a symbol of the language. It passes a representation of this symbol on to the screener, which checks whether this symbol is relevant for the parser. If not, it is ignored, and the screener reactivates the scanner. Otherwise, it passes a possibly transformed representation of the symbol on to the parser.

The scanner must, in general, be able to recognize infinitely many or at least very many different symbols. The set of symbols is, therefore, divided into finitely many classes. One *symbol class* consists of symbols that have a similar syntactic role. We distinguish:

- The *alphabet* is the set of characters that may occur in program texts. We use the letter Σ to denote alphabets.

R. Wilhelm, H. Seidl, S. Hack, *Compiler Design*, DOI 10.1007/978-3-642-17540-4_2,

- A *symbol* is a word over the alphabet Σ. Examples are *xyz*12, 125, class, "*abc*".
- A *symbol class* is a set of symbols. Examples are the set of identifiers, the set of *int*-constants, and the set of character strings. We denote these by Id, Intconst, and String, respectively.
- The *representation of a symbol* comprises all of the mentioned information about a symbol that may be relevant for later phases of compilation. The scanner may represent the word *xyz*12 as pair (Id, "*xyz*12"), consisting of the name of the class and the found symbol, and pass this representation on to the screener. The screener may replace "*xyz*12" by the internal representation of an identifier, for example, a unique number, and then pass this on to the parser.

2.2 Regular Expressions and Finite Automata

2.2.1 Words and Languages

We introduce some basic terminology. We use Σ to denote the *alphabet*, that is a finite, nonempty set of characters. A *word* x over Σ of length n is a sequence of n characters from Σ. The *empty word* ε is the empty sequence of characters, i.e., the sequence of length 0. We consider individual characters from Σ as words of length 1.

Σ^n denotes the set of words of length n for $n \geq 0$. In particular, $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^1 = \Sigma$. The set of all words is denoted as Σ^*. Correspondingly, Σ^+ is the set of *nonempty* words, i.e.,

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n \qquad \text{and} \qquad \Sigma^+ = \bigcup_{n \geq 1} \Sigma^n.$$

Several words can be concatenated to a new word. The *concatenation* of the words x and y puts the sequence of characters of y after the sequence of characters of x, i.e.,

$$x \,.\, y = x_1 \ldots x_m y_1 \ldots y_n,$$

if $x = x_1 \ldots x_m$, $y = y_1 \ldots y_n$ for $x_i, y_j \in \Sigma$.

Concatenation of x and y produces a word of length $n + m$ if x and y have lengths n and m, respectively. Concatenation is a binary operation on the set Σ^*. In contrast to addition of numbers, concatenation of words is not *commutative*. This means that the word $x \,.\, y$ is , in general, different from the word $y \,.\, x$. Like addition of numbers, concatenation of words is *associative*, i.e.,

$$x \,.\, (y \,.\, z) = (x \,.\, y) \,.\, z \qquad \text{for all } x, y, z \in \Sigma^*$$

The empty word ε is the *neutral* element with respect to concatenation of words, i.e.,

$$x \,.\, \varepsilon = \varepsilon \,.\, x = x \qquad \text{for all } x \in \Sigma^*.$$

In the following, we will write xy for $x \,.\, y$.

For a word $w = xy$ with $x, y \in \Sigma^*$ we call x a *prefix* and y a *suffix* of w. Prefixes and suffixes are special *subwords*. In general, word y is a subword of word w, if $w = xyz$ for words $x, z \in \Sigma^*$. Prefixes, suffixes, and, in general, subwords of w are called *proper*, if they are different from w.

Subsets of Σ^* are called (formal) *languages*. We need some operations on languages. Assume that $L, L_1, L_2 \subseteq \Sigma^*$ are languages. The *union* $L_1 \cup L_2$ consists of all words from L_1 and L_2:

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}.$$

The *concatenation* $L_1.L_2$ (abbreviated L_1L_2) consists of all words resulting from concatenation of a word from L_1 with a word from L_2:

$$L_1 \,.\, L_2 = \{xy \mid x \in L_1, y \in L_2\}.$$

The *complement* $\overline{L}$ of language L consists of all words in Σ^* that are not contained in L:

$$\overline{L} = \Sigma^* - L.$$

For $L \subseteq \Sigma^*$ we denote L^n as the n-times concatenation of L, L^* as the union of arbitrary concatenations, and L^+ as the union of nonempty concatenations of L, i.e.,

$$\begin{array}{lcll} L^n & = & \{w_1 \ldots w_n \mid w_1, \ldots, w_n \in L\} & \\ L^* & = & \{w_1 \ldots w_n \mid \exists n \geq 0.\ w_1, \ldots, w_n \in L\} & = \bigcup\limits_{n \geq 0} L^n \\ L^+ & = & \{w_1 \ldots w_n \mid \exists n > 0.\ w_1, \ldots, w_n \in L\} & = \bigcup\limits_{n \geq 1} L^n \end{array}$$

The operation (_)* is called *Kleene star*.

Regular Languages and Regular Expressions

The languages described by symbol classes as they are recognized by the scanner are nonempty *regular languages*.

Each nonempty regular language can be constructed starting with singleton languages and applying the operations union, concatenation, and Kleene star. Formally, the set of all *regular languages* over an alphabet Σ is inductively defined by:

- The empty set $\emptyset$ and the set $\{\varepsilon\}$, consisting only of the empty word, are regular.
- The sets $\{a\}$ for all $a \in \Sigma$ are regular over Σ.
- If R_1 and R_2 are regular languages over Σ, so are $R_1 \cup R_2$ and R_1R_2.
- If R is regular over Σ, then so is R^*.

According to this definition, each regular language can be specified by a regular expression. *Regular expressions* over Σ and the regular languages described by them are also defined inductively:

- $\emptyset$ is a regular expression over Σ, which specifies the regular language $\emptyset$.
  ε is a regular expression over Σ, and it specifies the regular language $\{\varepsilon\}$.

- For each $a \in \Sigma$, a is a regular expression over Σ that specifies the regular language $\{a\}$.
- If r_1 and r_2 are regular expressions over Σ that specify the regular languages R_1 and R_2, respectively, then $(r_1 \mid r_2)$ and $(r_1 r_2)$ are regular expressions over Σ that specify the regular languages $R_1 \cup R_2$ and $R_1 R_2$, respectively.
- If r is a regular expression over Σ, that specifies the regular language R, then r^* is a regular expression over Σ that specifies the regular language R^*.

In practical applications, $r?$ is often used as abbreviation for $(r \mid \varepsilon)$ and sometimes also r^+ for the expression (rr^*).

In the definition of regular expressions, we assume that the symbols for the empty set and the empty word are not contained in Σ, and the same also holds for parentheses (and)i, and the operators | and $*$, and also ? and $+$. These characters belong to the specification language for regular expressions and not to the languages denoted by the regular expressions. They are called *metacharacters*. The set of representable characters is limited, though, so that some metacharacters may also occur in the specified languages. A programming system for generating scanners which uses regular expressions as specification language, must provide a mechanism to distinguish for a character between when it is used as a metacharacter and when as a character of the specified language. One such mechanism relies on an *escape character*. In many specification formalisms for regular languages, the character \ is used as escape character. If, for example, the metacharacter | is also included in the alphabet, then every occurrence of | as an alphabet character is preceded with a \. So, the set of all sequences of | is represented by $\backslash|^*$.

As in programming languages, we introduce operator precedences to save on parentheses: The ?-operator has the highest precedence, followed by the Kleene star $(\_)^*$, and then possibly the operator $(\_)^+$, then concatenation, and finally the alternative operator |.

Example 2.2.1 The following table lists a number of regular expressions together with the specified languages, and some or even all of their elements.

Regular expression	Described language	Elements of the language
$a \mid b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbba, \ldots$
$(ab)^*$	$\{ab\}^*$	$\varepsilon, ab, abab, \ldots$
$abba$	$\{abba\}$	$abba$ □

Regular expressions that contain the empty set as symbol can be simplified by repeated application of the following equalities:

$$\begin{array}{lcl} r \mid \emptyset & = & \emptyset \mid r = r \\ r \,.\, \emptyset & = & \emptyset \,.\, r = \emptyset \\ \emptyset^* & = & \epsilon \end{array}$$

The equality symbol = between two regular expressions means that both specify the same language. We can prove:

Lemma 2.2.1 For every regular expression r over alphabet Σ, a regular expression r' can be constructed which specifies the same language and additionally has the following properties:
1. If r is a specification of the empty language, then r' is the regular expression $\emptyset$;
2. If r is a specification of a nonempty language, then r' does not contain the symbol $\emptyset$. □

Our applications only have regular expressions that specify nonempty languages. A symbol to describe the empty set, therefore, need not be included into the specification language of regular expressions. The empty word, on the other hand, cannot so easily be omitted. For instance, we may want to specify that the sign of a number constant is *optional*, i.e., may be present or absent. Often, however, specification languages used by scanners do not provide a dedicated metacharacter for the empty word: The ?-operator suffices in all practical situations. In order to remove explicit occurrences of ε in a regular expression by means of ?, the following equalities can be applied:

$$\begin{array}{lllll} r \mid \varepsilon & = & \varepsilon \mid r & = & r? \\ r \,.\, \varepsilon & = & \varepsilon \,.\, r & = & r \\ \varepsilon^* & = & \varepsilon? & = & \varepsilon \end{array}$$

We obtain:

Lemma 2.2.2 For every regular expression r over alphabet Σ, a regular expression r' (possibly containing ?) can be constructed which specifies the same language as r and additionally has the following properties:
1. If r is a specification of the language $\{\varepsilon\}$, then r' is the regular expression ε;
2. If r is a specification of a language different from $\{\varepsilon\}$, then r' does not contain ε. □

Finite Automata

We have seen that regular expressions are used for the specification of symbol classes. The implementation of recognizers uses finite automata (FA). Finite automata are acceptors for regular languages. They maintain one state variable that can only take on finitely many values, the *states* of the FA. According to Fig. 2.1, an FA has an input tape and an input head, which reads the input on the tape from left to right. The behavior of the FA is specified by means of a *transition relation* Δ.

Formally, a *nondeterministic finite automaton (with ε-transitions)* (or FA for short) is represented as a tuple $M = (Q, \Sigma, \Delta, q_0, F)$, where
- Q is a finite set of *states*,
- Σ is a finite alphabet, the *input alphabet*,
- $q_0 \in Q$ is the *initial state*,

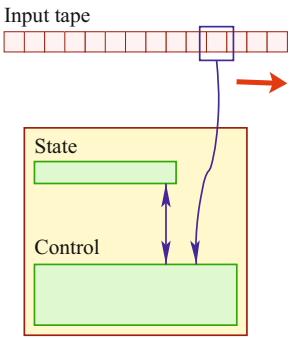


Fig. 2.1 Schematic representation of an FA

- $F \subseteq Q$ is the set of *final states*, and
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the *transition relation*.

A transition $(p, x, q) \in \Delta$ expresses that M can change from its current state p into the state q. If $x \in \Sigma$ then x must be the next character in the input, and after reading x the input head is moved by one character. If $x = \varepsilon$ then no character of the input is read upon this transition. The input head remains at its actual position. Such a transition is called an *ε-transition*.

Of particular interest for implementations are FAs without ε-transitions, which in addition have in each state exactly one transition under each character. Such a finite automaton is called a *deterministic finite automaton* (DFA). For such a DFA the transition relation Δ is a *function* $\Delta : Q \times \Sigma \to Q$.

We describe the behavior of a DFA in comparison with a DFA used as a scanner. The description of the behavior of a scanner is put into boxes. A DFA checks whether given input words are contained in a language or not. It accepts the input word if it arrives in a final state after reading the whole word.

> A DFA used as a scanner decomposes the input word into a sequence of subwords corresponding to *symbols* of the language. Each symbol drives the DFA from its initial state into one of its final states.

The DFA starts in its initial state. Its input head is positioned at the beginning of the input head.

> A scanner's input head is always positioned at the first not-yet-consumed character.

It then makes a number of steps. Depending on the actual state and the next input symbol, the DFA changes its state and moves its input head to the next character.

The DFA accepts the input word when the input is exhausted, and the actual state is a final state.

Quite analogously, the scanner performs a number of steps. It reports that it has found a symbol or that it has detected an error. If in the actual state no transition under the next input character in the direction of a final state is possible, the scanner rewinds to the last input character that brought it into a final state. The class corresponding to this final state is returned together with the consumed prefix of the input. Then the scanner restarts in the initial state with its input head positioned at the first input character not yet consumed. An error is detected if while rewinding, no final state is found.

Our goal is to derive an implementation of an acceptor of a regular language out of a specification of the language, that is, to construct out of a regular expression r a DFA that accepts the language described by r. In a first step, a *nondeterministic* FA for r is constructed that accepts the language described by r.

An FA $M = (Q, \Sigma, \Delta, q_0, F)$ starts in its initial state q_0 and nondeterministically performs a sequence of steps, a *computation*, under the given input word The input word is accepted if the computation leads to a final state, The future behavior of an FA is fully determined by its actual state $q \in Q$ and the remaining input $w \in \Sigma^*$. This pair (q, w) makes up the *configuration* of the FA. A pair (q_0, w) is an *initial configuration*. Pairs (q, ε) such that $q \in F$ are *final configurations*.

The *step*-relation $\vdash_M$ is a binary relation on configurations. For $q, p \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $w \in \Sigma^*$, it holds that $(q, aw) \vdash_M (p, w)$ if and only if $(q, a, p) \in \Delta$ and $a \in \Sigma \cup \{\varepsilon\}$. $\vdash_M^*$ denotes the reflexive, transitive hull of the relation $\vdash_M$. The language accepted by the FA M is defined as

$$L(M) = \{w \in \Sigma^* | (q_0, w) \vdash_M^* (q_f, \varepsilon) \text{ with } q_f \in F\}.$$

Example 2.2.2 Table 2.1 shows the transition relation of an FA M in the form of a two-dimensional matrix T_M. The states of the FA are denoted by the numbers $0, \ldots, 7$. The alphabet is the set $\{0, \ldots, 9, ., E, +, -\}$. Each row of the table describes the transitions for one of the states of the FA. The columns correspond to the characters in $\Sigma \cup \{\varepsilon\}$. The entry $T_M[q, x]$ contains the set of states p such that $(q, x, p) \in \Delta$. The state 0 is the initial state, $\{1, 4, 7\}$ is the set of final states. This FA recognizes unsigned *int*- and *float*-constants. The accepting (final) state 1 can be reached through computations on *int*-constants. Accepting states 4 and 6 can be reached under *float*-constants. □

An FA M can be graphically represented as a finite *transition diagram*. A transition diagram is a finite, directed, edge-labeled graph. The vertices of this graph correspond to the states of M, the edges to the transitions of M. An edge from

Table 2.1 The transition relation of an FA to recognize unsigned *int*- and *float*-constants. The first column represents the identical columns for the digits $i = 0, \ldots, 9$; the fourth the ones for $+$ and $-$

T_M	i	.	E	$+, -$	ε
0	$\{1, 2\}$	$\{3\}$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{4\}$
2	$\{2\}$	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$
3	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{4\}$	$\emptyset$	$\{5\}$	$\emptyset$	$\{7\}$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\{6\}$	$\{6\}$
6	$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7	$\{7\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

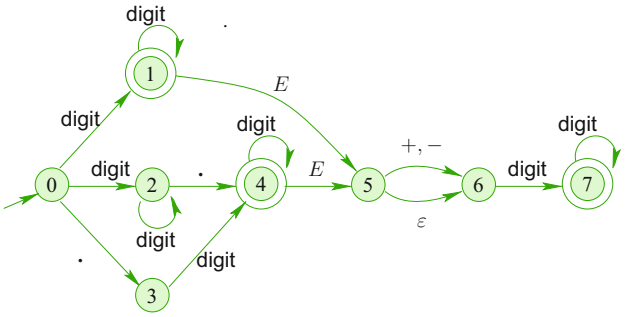


Fig. 2.2 The transition diagram for the FA of Example 2.2.2. The character digit stands for the set $\{0, 1, \ldots, 9\}$, an edge labeled with digit for edges labeled with $0, 1, \ldots 9$ with the same source and target vertices

p to q that is labeled with x corresponds to a transition (p, x, q). The start vertex of the transition diagram, corresponding to the initial state, is marked by an arrow pointing to it. The *end vertices*, corresponding to final states, are represented by doubly encircled vertices. A w-*path* in this graph for a word $w \in \Sigma^*$ is a path from a vertex q to a vertex p, such that w is the concatenation of the edge labels. The language accepted by M consists of all words in $w \in \Sigma^*$, for which there exists a w-path in the transition diagram from q_0 to a vertex $q \in F$.

Example 2.2.3 Figure 2.2 shows the transition diagram corresponding to the FA of example 2.2.2. □

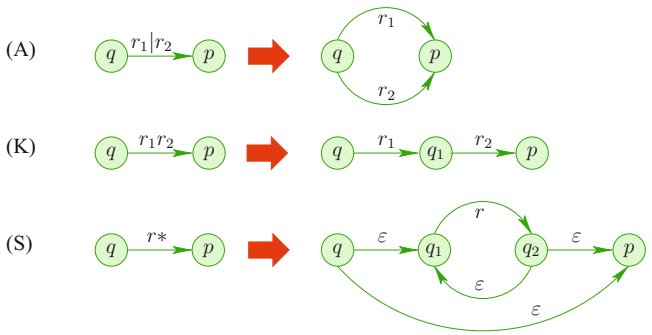


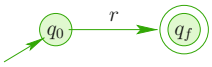
Fig. 2.3 The rules for constructing an FA from a regular expression.

Acceptors

The next theorem guarantees that every regular expression can be compiled into an FA that accepts the language specified by the expression.

Theorem 2.2.1 For each regular expression r over an alphabet Σ, an FA M_r with input alphabet Σ can be constructed such that $L(M_r)$ is the regular language specified by r.

We now present a method that constructs the transition diagram of an FA for a regular expression r over an alphabet Σ. The construction starts with one edge leading from the initial state to a final state. This edge is labeled with r.



While the expression r is decomposed according to its syntactical structure, an implementing transition diagram is built up. The transformation rules are given in Fig. 2.3. The rules are applied until all remaining edges are labeled with $\emptyset$, ε or characters from Σ. Then, the edges labeled with $\emptyset$ are removed.

The application of a rule replaces the edge whose label is matched by the label of the left side with a corresponding copy of the subgraph of the right side. Exactly one rule is applicable for each operator. The application of the rule removes an edge labeled with a regular expression r and inserts new edges that are labeled with the argument expressions of the outermost constructor in r. The rule for the Kleene star inserts additional ε-edges. This method can be implemented by the following program snippet if we take natural numbers as states of the FA:

$trans \leftarrow \emptyset$;
$count \leftarrow 1$;
generate$(0, r, 1)$;
return $(count, trans)$;

The set *trans* globally collects the transitions of the generated FA, and the global counter *count* keeps track of the largest natural number used as state. A call to a procedure generate for (p, r', q) inserts all transitions of an FA for the regular expression r' with initial state p and final state q into the set *trans*. New states are created by incrementing the counter *count*. This procedure is recursively defined over the structure of the regular expression r':

```
void generate (int p, Exp r', int q) {
    switch (r') {
      case (r1 | r2) :  generate(p, r1, q);
                        generate(p, r2, q); return;
      case (r1.r2) :    int q1 ← ++count;
                        generate(p, r1, q1);
                        generate(q1, r2, q); return;
      case r1* :        int q1 ← ++count;
                        int q2 ← ++count;
                        trans ← trans ∪ {(p, ε, q1), (q2, ε, q), (q2, ε, q1)}
                        generate(q1, r1, q2); return;
      case ∅ :          return;
      case x :          trans ← trans ∪ {(p, x, q)}; return;
    }
}
```

Exp denotes a datatype for regular expressions over the alphabet Σ. We have used a JAVA-like programming language as implementation language. The *switch*-statement was extended by *pattern matching* to elegantly deal with structured data such as regular expressions. This means that patterns are not only used to select between alternatives but also to identify substructures.

A procedure call generate$(0, r, 1)$ terminates after n rule applications, where n is the number of occurrences of operators and symbols in the regular expression r. If l is the value of the counter after the call, the generated FA has $\{0, \ldots, l\}$ as set of states, where 0 is the initial state and 1 the only final state. The transitions are collected in the set *trans*. The FA M_r can be computed in linear time.

Example 2.2.4 The regular expression $a(a|0)^*$ over the alphabet $\{a, 0\}$ describes the set of words $\{a, 0\}^*$ beginning with an a. Figure 2.4 shows the construction of the state diagram of a FA that accepts this language.

□

The Subset Construction

For implementations, *deterministic* finite automata are preferable to nondeterministic finite automata. A deterministic finite automaton M has no transitions under ε and for each pair (q, a) with $q \in Q$ and $a \in \Sigma$, it has exactly one successor state. So, for each state q in M and each word $w \in \Sigma^*$ it has exactly one w-path in the

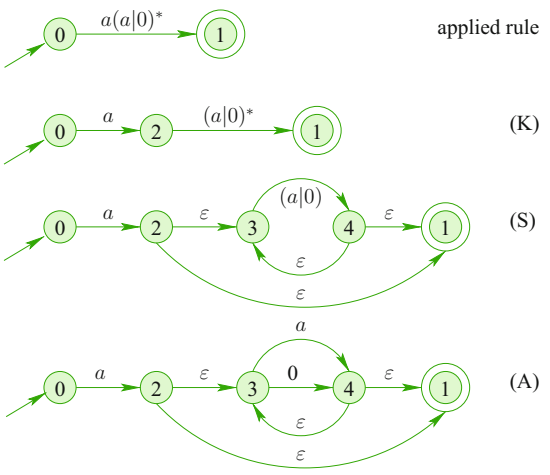


Fig. 2.4 Construction of a transition diagram for the regular expression $a(a \mid 0)^*$

transition diagram of M starting in q. If q is chosen as initial state of M then w is in the language of M if and only if this path leads to a final state of M. Fortunately, we have Theorem 2.2.2.

Theorem 2.2.2 For each FA a DFA can be constructed that accepts the same language. □

Proof The proof provides the second step of the generation procedure for scanners. It uses the *subset construction*. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be an FA. The goal of the subset construction is to construct a DFA $\mathcal{P}(M) = (\mathcal{P}(Q), \Sigma, \mathcal{P}(\Delta), \mathcal{P}(q_0)\mathcal{P}(F))$ that recognizes the same language as M. For a word $w \in \Sigma^*$ let $\mathsf{states}(w) \subseteq Q$ be the set of all states $q \in Q$ for which there exists a w-path leading from the initial state q_0 to q. The DFA $\mathcal{P}(M)$ is given by:

$$
\begin{array}{lcll}
\mathcal{P}(Q) & = & \{\mathsf{states}(w) \mid w \in \Sigma^*\} & \\
\mathcal{P}(q_0) & = & \mathsf{states}(\varepsilon) & \\
\mathcal{P}(F) & = & \{\mathsf{states}(w) \mid w \in L(M)\} & \\
\mathcal{P}(\Delta)(S, a) & = & \mathsf{states}(wa) & \text{for } S \in \mathcal{P}(Q) \text{ and } a \in \Sigma \text{ if } S = \mathsf{states}(w)
\end{array}
$$

We convince ourselves that our definition of the transition function $\mathcal{P}(\Delta)$ is *reasonable*. For this, we show that for words $w, w' \in \Sigma^*$ with $\mathsf{states}(w) = \mathsf{states}(w')$ it holds that $\mathsf{states}(wa) = \mathsf{states}(w'a)$ for all $a \in \Sigma$. It follows that M and $\mathcal{P}(M)$ accept the same language.

We need a systematic way to construct the states and the transitions of $\mathcal{P}(M)$. The set of final states of $\mathcal{P}(M)$ can be constructed – once the set of states of $\mathcal{P}(M)$ is known, because it holds that:

$$\mathcal{P}(F) = \{A \in \mathcal{P}(M) \mid A \cap F \neq \emptyset\}$$

For a set $A \subseteq Q$ we define the set of ε-successor states A as

$$\mathsf{SS}_\epsilon(S) = \{p \in Q \mid \exists\, q \in S.\, (q, \varepsilon) \vdash_M^* (p, \varepsilon)\}$$

This set consists of all states that can be reached from states in S by ε-paths in the transition diagram of M. This closure can be computed by the following function:

```
set ⟨state⟩ closure(set ⟨state⟩ S) {
    set ⟨state⟩ result ← ∅;
    list ⟨state⟩ W ← list_of(S);
    state q, q′;
    while (W ≠ []) {
        q ← hd(W); W ← tl(W);
        if (q ∉ result) {
            result ← result ∪ {q};
            forall (q′ : (q, ε, q′) ∈ Δ)
                W ← q′ :: W;
        }
    }
    return result;
}
```

The states of the resulting DFA that are reachable from A, are collected in the set *result*. The list W contains all elements in *result* whose ε-transitions have not yet been processed. As long as W is not empty, the first state q from W is selected. To do this, functions hd and tl are used that return the first element and the tail of a list, respectively. If q is already contained in *result*, nothing needs to be done. Otherwise, q is inserted into the set *result*. All transitions (q, ε, q') for q in Δ are considered, and the successor states q' are added to W. By applying the closure operator $\mathsf{SS}_\epsilon(\_)$, the initial state $\mathcal{P}(q_0)$ of the subset automaton can be computed:

$$\mathcal{P}(q_0) = S_\varepsilon = \mathsf{SS}_\epsilon(\{q_0\})$$

When constructing the set of all states $\mathcal{P}(M)$ together with the transition function $\mathcal{P}(\Delta)$ of $\mathcal{P}(M)$, bookkeeping is required of the set $Q' \subseteq \mathcal{P}(M)$ of already generated states and of the set $\Delta' \subseteq \mathcal{P}(\Delta)$ of already created transitions.

Initially, $Q' = \{\mathcal{P}(q_0)\}$ and $\Delta' = \emptyset$. For a state $S \in Q'$ and each $a \in \Sigma$, its *successor state* S' under a and Q' and the transition (S, a, S') are added to Δ. The successor state S' for S under a character $a \in \Sigma$ is obtained by collecting the successor states under a of all states $q \in S$ and adding all ε-successor states:

$$S' = \mathsf{SS}_\epsilon(\{p \in Q \mid \exists q \in S :\ (q, a, p) \in \Delta\})$$

The function nextState() serves to compute this set:

```
set ⟨state⟩ nextState(set ⟨state⟩ S, symbol x) {
    set ⟨state⟩ S' ← ∅;
    state q, q';
    forall (q' : q ∈ S, (q, x, q') ∈ Δ) S' ← S' ∪ {q'};
    return closure(S');
    }
```

Insertions into the sets Q' and Δ' are performed until all successor states of the states in Q' under transitions for characters from Σ are already contained in the set Q'. Technically, this means that the set of all states *states* and the set of all transitions *trans* of the subset automaton can be computed iteratively by the following loop:

```
list ⟨set ⟨state⟩⟩ W;
set ⟨state⟩ S_0 ← closure({q_0});
states ← {S_0}; W ← [S_0];
trans ← ∅;
set ⟨state⟩ S, S';
while (W ≠ []) {
    S ← hd(W); W ← tl(W);
    forall (x ∈ Σ) {
        S' ← nextState(S, x);
        trans ← trans ∪ {(S, x, S')};
        if (S' ∉ states) {
            states ← states ∪ {S'};
            W ← W ∪ {S'};
        }
    }
}
```

□

Example 2.2.5 The subset construction applied to the FA of Example 2.2.4 can be executed by the steps described in Fig. 2.5. The states of the DFA to be constructed are denoted by primed natural numbers $0', 1', \ldots$. The initial state $0'$ is the set $\{0\}$. The states in Q' whose successor states are already computed are underlined. The state $3'$ is the empty set of states, i.e., the *error state*. It can never be left. It is the successor state of a state S under a if there is no transition of the FA under a for any FA state in S. □

Minimization

The DFA generated from a regular expression in the given two steps is not necessarily the smallest possible for the specified language. There may be states that have the same *acceptance behavior*. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a DFA. We say states

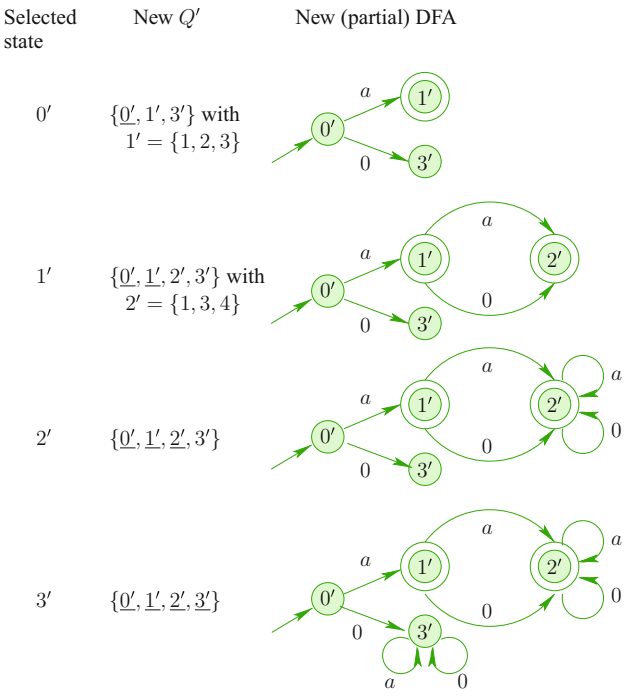


Fig. 2.5 The subset construction for the FA of Example 2.2.4

p and q of M have the same acceptance behavior if p and q are *indistinguishable* by means of words from Σ^*. In order to make this idea precise, we extend the transition function $\Delta : Q \times \Sigma \to Q$ of the DFA M to a function $\Delta^* : Q \times \Sigma^* \to Q$ that maps each pair $(q, w) \in Q \times \Sigma^*$ to the unique state which is reached by the w-path from q in the transition diagram of M. The function Δ^* is defined by induction over the length of words:

$$\Delta^*(q, \varepsilon) = q \quad \text{and} \quad \Delta^*(q, aw) = \Delta^*(\Delta(q, a), w)$$

for all $q \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$. Then states $p, q \in Q$ have the same acceptance behavior if for all $w \in \Sigma^*$,

$$\Delta^*(p, w) \in F \quad \text{if and only if} \quad \Delta^*(q, w) \in F$$

In this case we write $p \sim_M q$. The relation $\sim_M$ is an equivalence relation on Q. The DFA M is called *minimal* if there is no DFA with fewer states which accepts the same language as M. It turns out that for each DFA, a minimal DFA can be

constructed which accepts the same language, and this minimal DFA is unique up to isomorphism. This is the claim of the following theorem.

Theorem 2.2.3 For each DFA M, a minimal DFA M' can be constructed that accepts the same language as M. This minimal DFA is unique up to renaming of states.

Proof For a DFA $M = (Q, \Sigma, \Delta, q_0, F)$ we define a DFA $M' = (Q', \Sigma, \Delta', q_0', F')$ that is minimal. W.l.o.g., we assume that every state in Q is reachable from the initial state q_0 of M. As set of states of the DFA M' we choose the set of *equivalence classes* of states of the DFA M under $\sim_M$. For a state $q \in Q$ let $[q]_M$ denote the equivalence class of states q with respect to the relation $\sim_M$, i.e.,

$$[q]_M = \{p \in Q \mid q \sim_M p\}$$

The set of states of M' is given by:

$$Q' = \{[q]_M \mid q \in Q\}$$

Correspondingly, the initial state and the set of final states of M' are defined by

$$q_0' = [q_0]_M \qquad F' = \{[q]_M \mid q \in F\},$$

and the transition function of M for $q' \in Q'$ and $a \in \Sigma$ is defined by

$$\Delta'(q', a) = [\Delta(q, a)]_M \quad \text{for a } q \in Q \text{ such that } q' = [q]_M.$$

It can be verified that the new transition function Δ' is well-defined, i.e., that for $[q_1]_M = [q_2]_M$ it holds that $[\Delta(q_1, a)]_M = [\Delta(q_2, a)]_M$ for all $a \in \Sigma$. Furthermore,

$$\Delta^*(q, w) \in F \quad \text{if and only if} \quad (\Delta')^*([q]_M, a) \in F'$$

holds for all $q \in Q$ and $w \in \Sigma^*$. This implies that $L(M) = L(M')$. We now claim that the DFA M' is minimal. For a proof of this claim, consider another DFA $M'' = (Q'', \Sigma, \Delta'', q_0'', F'')$ with $L(M'') = L(M')$ whose states are all reachable from q_0''. Assume for a contradiction that there is a state $q \in Q''$ and words $u_1, u_2 \in \Sigma^*$ such that $(\Delta'')^*(q_0'', u_1) = (\Delta'')^*(q_0'', u_2) = q$, but $(\Delta')^*([q_0]_M, u_1) \neq (\Delta')^*([q_0]_M, u_2)$ holds. For $i = 1, 2$, let $p_i \in Q$ denote a state with $(\Delta')^*([q_0]_M, u_i) = [p_i]_M$. Since $[p_1]_M \neq [p_2]_M$ holds, the states p_1 and p_2 cannot be equivalent. On the other hand, we have for all words $w \in \Sigma^*$,

$$\begin{array}{lll} \Delta^*(p_1, w) \in F & \text{iff} & (\Delta')^*([p_1]_M, w) \in F' \\ & \text{iff} & (\Delta'')^*(q, w) \in F'' \\ & \text{iff} & (\Delta')^*([p_2]_M, w) \in F' \\ & \text{iff} & \Delta^*(p_2, w) \in F \end{array}$$

Therefore, the states p_1, p_2 are equivalent – in contradiction to our assumption. Since for every state $[p]_M$ of the DFA M', there is a word u such that $(\Delta')^*([q_0]_M, u) = [p]_M$, we conclude that there is a surjective mapping of the states of M'' onto the states of M'. Therefore, M'' must have at least as many states as M'. Therefore, the DFA M' is minimal. □

The practical construction of M' requires us to compute the equivalence classes $[q]_M$ of the relation $\sim_M$. If *each* state is a final state, i.e., $Q = F$, or none of the states is final, i.e., $F = \emptyset$, then all states are equivalent, and $Q = [q_0]_M$ is the only state of M'.

Let us assume in the following that there is at least one final and one nonfinal state, i.e., $Q \neq F \neq \emptyset$. The algorithm maintains a *partition* Π on the set Q of the states of the DFA M. A partition on the set Q is a set of nonempty subsets of Q, whose union is Q.

A partition Π is called *stable* under the transition relation Δ, if for all $q' \in \Pi$ and all $a \in \Sigma$ there is a $p' \in \Pi$ such that

$$\{\Delta(q, a) \mid q \in q'\} \subseteq p'$$

In a stable partition, all transitions from one set of the partition lead into exactly one set of the partition.

In the partition Π, those sets of states are maintained of which we assume that they have the same acceptance behavior. If it turns out that a set $q' \in \Pi$ contains states with different acceptance behavior, then the set q' is split up. Different acceptance behavior of two states q_1 and q_2 is recognized when the successor states $\Delta(q_1, a)$ and $\Delta(q_2, a)$ for some $a \in \Sigma$ lie in different sets of Π. Then the partition is apparently not stable. Such a split of a set in a partition is called *refinement* of Π. The successive refinement of the partition Π terminates if there is no need for further splitting of any set in the obtained partition. Then the partition is stable under the transition relation Δ.

In detail, the construction of the minimal DFA proceeds as follows. The partition Π is initialized with $\Pi = \{F, Q \backslash F\}$. Let us assume that the actual partition Π of the set Q of states of M' is not yet stable under Δ. Then there exists a set $q' \in \Pi$ and some $a \in \Sigma$ such that the set $\{\Delta(q, a) \mid q \in q'\}$ is not completely contained in any of the sets in $p' \in \Pi$. Such a set q' is then split to obtain a new partition Π' that consists of all nonempty elements of the set

$$\{\{q \in q' \mid \Delta(q, a) \in p'\} \mid p' \in \Pi\}$$

The partition Π' of q' consists of all nonempty subsets of states from q' that lead under a into the same sets in $p' \in \Pi$. The set q' in Π is replaced by the partition Π' of q', i.e., the partition Π is refined to the partition $(\Pi \backslash \{q'\}) \cup \Pi'$.

If a sequence of such refinement steps arrives at a stable partition in Π the set of states of M' has been computed.

$$\Pi = \{[q]_M \mid q \in Q\}$$

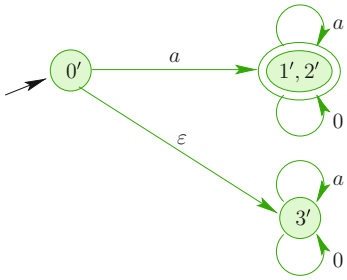


Fig. 2.6 The minimal DFA of Example 2.2.6

Each refinement step increases the number of sets in partition Π. A partition of the set Q may only have as many sets as Q has elements. Therefore, the algorithm terminates after finitely many steps. □

Example 2.2.6 We illustrate the presented method by minimizing the DFA of Example 2.2.5. At the beginning, partition Π is given by

$$\{\{0', 3'\}, \{1', 2'\}\}$$

This partition is not stable. The first set $\{0', 3'\}$ must be split into the partition $\Pi' = \{\{0'\}, \{3'\}\}$. The corresponding refinement of partition Π produces the partition

$$\{\{0'\}, \{3'\}, \{1', 2'\}\}$$

This partition is stable under Δ. It therefore delivers the states of the minimal DFA. The transition diagram of the resulting DFA is shown in Fig. 2.6. □

2.3 A Language for Specifying Lexical Analyzers

We have met regular expressions as specifcation mechanism for symbol classes in lexical analysis. For practical purposes, one often would like to have something more comfortable.

Example 2.3.1 The following regular expression describes the language of unsigned *int*-constants of Examples 2.2.2 and 2.2.3.

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$

A similar specification of *float*-constants would stretch over three lines. □

In the following, we present several extensions of the specification mechanism of regular expressions that increase the comfort, but not the expressive power of this mechanism. The class of languages that can be described remains the same.

2.3.1 Character Classes

In the specification of a lexical analyzer, one should be able to group sets of characters into *classes* if these characters can be exchanged against each other without changing the symbol class of symbols in which they appear. This is particularly helpful in the case of large alphabets, for instance the alphabet of all *Unicode*-characters. Examples of frequently occurring character classes are:

$$\begin{array}{lcl} \mathsf{alpha} & = & a-zA-Z \\ \mathsf{digit} & = & 0-9 \end{array}$$

The first two definitions of character classes define classes by using intervals in the underlying character code, e.g. the ASCII. Note that we need another metacharacter, '-', for the specification of intervals. Using this feature, we can nicely specify the symbol class of identifiers:

$$\mathsf{Id} = \mathsf{alpha}(\mathsf{alpha} \mid \mathsf{digit})^*$$

The specification of character classes uses three metacharacters, namely '=', '−', and the blank. For the *usage* of identifiers for character classes, though, the description formalism must provide another mechanism to distinguish them from ordinary character strings. In our example, we us a dedicated font. In practical systems, the defined names of character classes might be enclosed in dedicated brackets such as $\{\ldots\}$.

Example 2.3.2 The regular expression for unsigned *int*- and *float*-constants is simplified through the use of the character classes $\mathsf{digit} = 0-9$ to:

$$\begin{array}{l} \mathsf{digit}\ \mathsf{digit}^* \\ \mathsf{digit}\ \mathsf{digit}^* E(+ \mid -)?\mathsf{digit}\ \mathsf{digit}^* \mid \mathsf{digit}^*(.\mathsf{digit} \mid \mathsf{digit}.) \\ \quad \mathsf{digit}^*(E(+ \mid -)?\mathsf{digit}\ \mathsf{digit}^*)? \end{array}$$

□

2.3.2 Nonrecursive Parentheses

Programming languages have lexical units that are characterized by the enclosing parentheses. Examples are string constants and comments. Parentheses limiting comments can be composed of several characters: $(*$ and $*)$ or $/*$ and $*/$ or // and $\backslash n$ (newline). More or less arbitrary texts can be enclosed in the opening and the closing parentheses. This is not easily described. A comfortable abbreviation for this is:

$$r_1 \text{ until } r_2$$

Let L_1, L_2 be the languages specified by the expressions r_1 and r_2, respectively, where L_2 does not contain the empty word. The language specified by the *until*-expression then is given by:

$$L_1 \, \overline{\Sigma^* L_2 \Sigma^*} \, L_2$$

A comment starting with // and ending at the end of line, for example, can be described by:

$$// \text{ until } \backslash n$$

2.4 Scanner Generation

Section 2.2 presented methods for deriving FAs from regular expressions, for compiling FAs into DFAs and finally for minimizing DFAs. In what follows we present the extensions of these methods which are necessary for the implementation of scanners.

2.4.1 Character Classes

Character classes were introduced to simplify regular expressions. They may also lead to smaller automata. The character-class definition

$$\begin{array}{lcl} \mathsf{alpha} & = & a - z \\ \mathsf{digit} & = & 0 - 9 \end{array}$$

can be used to replace the 26 transitions between states under letters by one transition under bu. This may simplify the DFA for the expression

$$\mathsf{Id} = \mathsf{alpha}(\mathsf{alpha} \mid \mathsf{digit})^*$$

considerably. An implementation of character classes uses a map χ that associates each character a with its class. This map can be represented by means of an array that is indexed by characters. The array component indexed with a character, provides the character class of the character. In order for χ to be a function each character must be member of exactly one character class. Character classes are therefore implicitly introduced for characters that do not explicitly occur in a class and those that explicitly occur in the definition of a symbol class. The problem of non-disjoint character classes is resolved by refining the classes to become disjoint. Let us assume that the classes $z_1, \ldots, z_k$ have been specified. Then the generator introduces a new character class for each intersection $\tilde{z}_1 \cap \ldots \cap \tilde{z}_k$ that is nonempty where $\tilde{z}_i$ either denotes z_i or the complement of z_i. Let D be the set of these newly introduced character classes. Each character class z_i corresponds to one of the al-

ternatives $d_i = (d_{i1} \mid \ldots \mid d_{ir_i})$ of character classes d_{ij} in D. Each occurrence of the character class z_i in regular expressions is then replaced by d_i.

Example 2.4.1 Let us assume we have introduced the two classes

$$\begin{array}{lcl} \mathsf{alpha} & = & a-z \\ \mathsf{alphanum} & = & a-z0-9 \end{array}$$

to define the symbol class $\mathsf{Id} = \mathsf{alpha}\ \mathsf{alphanum}^*$. The generator would divide one of these character classes into

$$\begin{array}{lcl} \mathsf{digit}' & = & \mathsf{alphanum} \backslash \mathsf{alpha} \\ \mathsf{alpha}' & = & \mathsf{alpha} \cap \mathsf{alphanum} \quad = \quad \mathsf{alpha} \end{array}$$

The occurrence of alphanum in the regular expression will be replaced by (alpha′ | digit′). □

2.4.2 An Implementation of the *until*-Construct

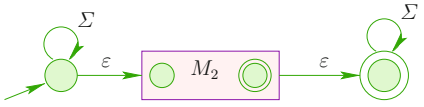
Let us assume the scanner should recognize symbols whose symbol class is specified by the expression $r = r_1$ until r_2. After recognizing a word of the language for r_1 it needs to find a word of the language for r_2 and then halt. This task is a generalization of the *pattern-matching* problem on strings. There exist algorithms that solve this problem for regular patterns in time, linear in the length of the input. These are, for example, used in the UNIX-program EGREP. They construct an FA for this task. Likewise, we now present a single construction of a DFA for the expression r.

Let L_1, L_2 be the languages described by the expressions r_1 and r_2. The language L defined by the expression r_1 until r_2 is:

$$L = L_1 \overline{\Sigma^* L_2 \Sigma^*} L_2$$

The process starts with automata for the languages L_1 and L_2, decomposes the regular expression describing the language, and applies standard constructions for automata. The process has the following seven steps: Fig. 2.7 shows all seven steps for an example.

1. The first step constructs FA M_1 and M_2 for the regular expressions r_1, r_2, where $L(M_1) = L_1$ and $L(M_2) = L_2$. A copy of the FA for M_2 is needed for step 2 and one more in step 6.
2. An FA M_3 is constructed for $\Sigma^* L_2 \Sigma^*$ using the first copy of M_2.



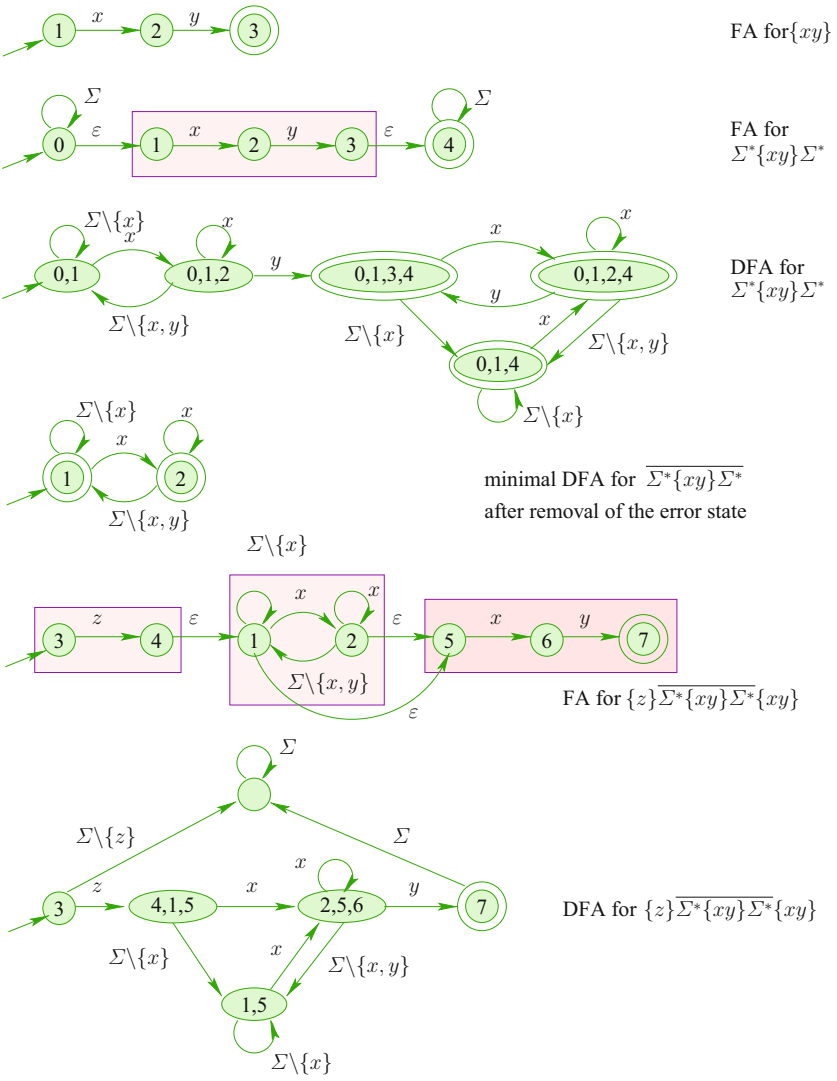
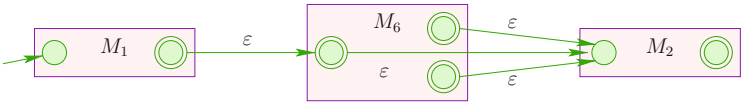


Fig. 2.7 The derivation of a DFA for z until xy with $x, y, z \in \Sigma$

The FA M_3 nondeterministically accepts all words over Σ that contain a subword from L_2.

3. The FA M_3 is transformed into a DFA M_4 by the subset construction.
4. A DFA M_5 is constructed that recognizes the language for $\overline{\Sigma^* L_2 \Sigma^*}$. To achieve this, the set of final states of M_4 is exchanged with the set of nonfinal states. In particular, M_5 accepts the empty word since according to our assumption $\varepsilon \notin L_2$. Therefore, the initial state of M_5 also is a final state.

5. The DFA M_5 is transformed into a minimal DFA M_6. All final states of M_4 are equivalent and dead in M_5 since it is not possible to reach a final state of M_5 from any final state of M_4.
6. Using the FA M_1, M_2 for L_1 and L_2 and M_6 an FA M_7 for the language $L_1\ \overline{\Sigma^* L_2 \Sigma^*}\ L_2$ is constructed.



   From each final state of M_6 including the initial state of M_6, there is an ε-transition to the initial state of M_2. From there paths under all words $w \in L_2$ lead into the final state of M_2, which is the only final state of M_7.
7. The FA M_7 is converted into a DFA M_8 and possibly minimized.

2.4.3 Sequences of Regular Expressions

Let a sequence

$$r_0, \ldots, r_{n-1}$$

of regular expression be given for the symbol classes to be recognized by the scanner. A scanner recognizing the symbols in these classes can be generated in the following steps:

1. In a first step, FAs $M_i = (Q_i, \Sigma, \Delta_i, q_{0,i}, F_i)$ for the regular expressions r_i are generated, where the Q_i should be pairwise disjoint.
2. The FAs M_i are combined into a single FA $M = (\Sigma, Q, \Delta, q_0, F)$ by adding a new initial state q_0 together with ε-transitions to the initial states $q_{0,i}$ of the FA M_i. The FA M therefore looks as follows:

$$\begin{array}{lcl} Q & = & \{q_0\} \cup Q_0 \cup \ldots \cup Q_{n-1} \quad \text{for some} \quad q_0 \notin Q_0 \cup \ldots \cup Q_{n-1} \\ F & = & F_0 \cup \ldots \cup F_{n-1} \\ \Delta & = & \{(q_0, \varepsilon, q_{0,i}) \mid 0 \leq i \leq n-1\} \cup \Delta_0 \cup \ldots \cup \Delta_{n-1}\ . \end{array}$$

   The FA M for the sequence accepts the *union* of the languages that are accepted by the FA M_i. The final state reached by a succesful run of the automaton indicates to which class the found symbol belongs.
3. The subset construction is applied to the FA M resulting in a DFA $\mathcal{P}(M)$. A word w is associated with the ith symbol class if it belongs to the language of r_i, but to no language of the other regular expressions $r_j, j < i$. Expressions with smaller indices are here preferred over expressions with larger indices.
   To which symbol class a word w belongs can be computed by the DFA $\mathcal{P}(M)$. The word w belongs to the ith symbol class if and only if it drives the DFA

$\mathcal{P}(M)$ into a state $q' \subseteq Q$ such that

$$q' \cap F_i \neq \emptyset \quad \text{and} \quad q' \cap F_j = \emptyset \quad \text{for all } j < i.$$

The set of all these states q' is denoted by F'_i.

4. Hereafter, the DFA $\mathcal{P}(M)$ may be minimized. During minimization, the sets of final states F'_i and F'_j for $i \neq j$ should be kept separate. The minimization algorithm should therefore start with the initial partition

$$\Pi = \{F'_0, F'_1, \ldots, F'_{n-1}, \mathcal{P}(Q) \setminus \bigcup_{i=0}^{n-1} F'_i\}$$

Example 2.4.2 Let the following sequence of character classes be given:

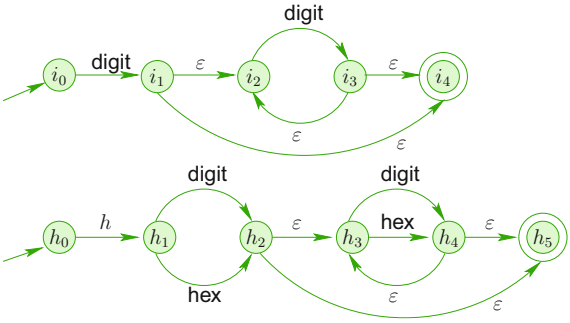
$$\begin{array}{lcl} \mathsf{digit} & = & 0-9 \\ \mathsf{hex} & = & A-F \end{array}$$

The sequence of regular definitions

$$\begin{array}{l} \mathsf{digit}\ \mathsf{digit}^* \\ h(\mathsf{digit} \mid \mathsf{hex})(\mathsf{digit} \mid \mathsf{hex})^* \end{array}$$

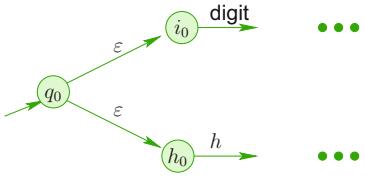
for the symbol classes Intconst and Hexconst are processed in the following steps:

- FA are generated for these regular expressions.

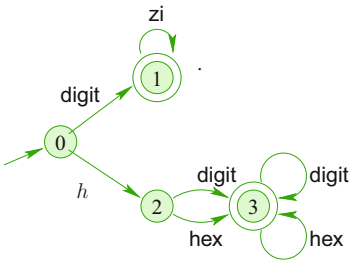


The final state i_4 stands for symbols of the class Intconst, while the final state h_5 stands for symbols of the class Hexconst.

- The two FA are combined with a new initial state q_0:



- The resulting FA is then made deterministic:



  An additional state 4 is needed, which is the error state corresponding to the empty set of original states. This state and all transitions into it are left out in the transition diagramm in order to keep readability.
- Minimzation of the DFA does not change it in this example.

The new final state of the generated DFA contains the old final state i_4 and therefore signals the recoginition of symbols of symbol class Intconst. Final state 3 contains h_5 and therefore signals the symbol class Hexconst.

Generated scanners always search for longest prefixes of the remaining input that lead into a final state. The scanner will therefore make a transition out of state 1 if this is possible, that is, if a digit follows. If the next input character is not a digit, the scanner should return to state 1 and reset its reading head. □

2.4.4 The Implementation of a Scanner

We have seen that the core of a scanner is a deterministic finte automaton. The transition function of this automaton can be represented by a two-dimensional array delta. This array is indexed by the actual state and the character class of the next input character. The selected array component contains the new state into which the DFA should go when reading this character in the actual state. While the access to delta$[q, a]$ is usually fast, the size of the array delta can be quite large. We observe, however, that a DFA often contains many transitions into the error state *error*. This state can therfore be chosen as the *default value* for the entries in delta. Representing transitions into only non-error states, may then lead to a sparsely populated array, which can be compressed using well-known methods. These save much space at the cost of slightly increased access times. The now empty entries represent transitions into the error state. Since they are still important for error detection of the scanner, the corresponding information must be preserved.

Let us consider one such compression method. Instead of using the original array delta to represent the transition function an array RowPtr is introduced, which is indexed by states and whose components are addresses of the original rows of delta, see Fig. 2.8.

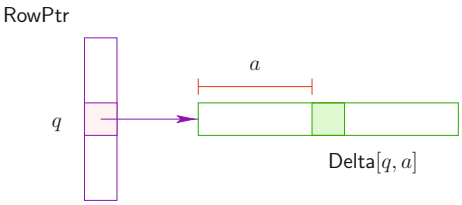


Fig. 2.8 Representation of the transition function of a DFA

We have not gained anything so far, and have even lost a bit of access efficiency. The rows of delta to which entries in RowPtr point are often almost empty. The rows are therefore overlaid into a single 1-dimensional array Delta in such a way that non-empty entries of delta do not collide. To find the starting position for the next row to be inserted into Delta, the *first-fit*-strategy can be used. The row is shifted over the array Delta starting at its beginning, until no non-empty entries of this row collide with non-empty entries already allocated in Delta.

The index in Delta at which the qth row of delta is allocated, is stored ina RowPtr$[q]$ (see Fig. 2.9). One problem is that the represented DFA now has lost its ability to identify errors, that is, undefined transitions. Even if $\Delta(q, a)$ is undefined (representing a transtion into the error state), the component Delta[RowPtr$[q] + a$] may contain a non-empty entry stemming from a shifted row of a state $p \neq q$. Therefore, another 1-dimensional array Valid is added, which has the same length as Delta. The array Valid contains the information to which states the entries in Delta belong. This means that Valid[RowPtr$[q] + a] = q$ if and only if $\Delta(q, a)$ is defined. The transition function of the DFA can then be implemented by a function

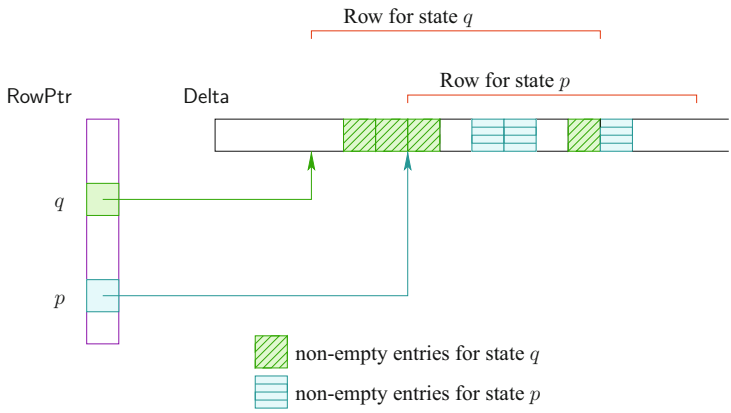


Fig. 2.9 Compressed representation of the transition function of a DFA

next() as follows:

```
State  next (State q, CharClass a) {
         if (Valid[RowPtr[q] + a] ≠ q) return error;
         return Delta[RowPtr[q] + a];
       }
```

2.5 The Screener

Scanners can be used in many applications, even beyond the pure splitting of a stream of characters according to a specification by means of a sequence of regular expressions. Scanners often provide the possibility of further processing the recognized elements.

To specify this extended functionality, each symbol class is associated with a corresponding semantic action. A screener can therefore be specified as a sequence of pairs of the form

$$\begin{array}{ll} r_0 & \{\mathsf{action}_0\} \\ \ldots & \\ r_{n-1} & \{\mathsf{action}_{n-1}\} \end{array}$$

where r_i is a (possibly extended) regular expression over character classes specifying the ith symbol class, and action_i denotes the semantic action to be executed when a symbol of this class is found. If the screener is to be implemented in a particular programming language, the semanic actions are typically specified as code in this language. Different languages offer different ways to return a representation of the found symbol. An implementation in C would, for instance, return an *int*-value to identify the symbol class, while all other relevant values have to be returned in suitable global values. Somewhat more comfort would be offered for an implementation of the screener in a modern object-oriented language such as JAVA. There a class Token can be introduced whose subclasses C_i correspond to the symbol classes. The last statement in action_i should then be a *return*-statement returning an object of class C_i whose attibutes store all properties of the identified symbol. In a functional language such as OCAML, a data type TOKEN can be supplied whose constructors C_i correspond to the different symbol classes. In this case, the semantic action action_i should be an expression of type token whose value $C_i(\ldots)$ represents the identified symbol of class C_i.

Semantic actions often need to access the text of the actual symbol. Some generated scanners offer access to it by means of a *global* variable *yytext*. Further global variables contain information such as the position of the actual symbol in the input. These are important for the generation of meaningful error messages. Some symbols should be ignored by the screener. Semantic actions therefore should also be able not to return a result but instead ask the scanner for another symbol from the input stream. A comment may, for example, be skipped or a compiler directive be realized without returning a symbol. In a corresponding action in a generator for C or JAVA, the *return*-statement simply would be omitted.

A function yylex() is generated from such a specification. It returns the next symbol every time it is called. Let us assume that a function scan() has been generated for the sequence $r_0, \ldots, r_{n-1}$ of regular expressions which stores the next symbol as a string in the global variable *yytext* and returns the number i of the class of ths symbol. The function yylex() then can be implemented by:

```
Token yylex() {
    while(true)
        switch scan() {
          case 0       :  action_0; break;
                          ...
          case n − 1   :  action_{n−1}; break;
          default      :  return error();
          }
}
```

The function error() is meant to handle the case that an error occurs while the scanner attempts to identify the next symbol. If an action action_i does not have a *return*-statement, the execution is resumed at the beginning of the *switch*-statement and reads the next symbol in the remaining input. If an action action_i terminates by executing a *return*-statement, the *switch*-statement together with the *while*-loop is terminated, and the corresponding value is returned as the return value of the actual call of the function yylex().

2.5.1 Scanner States

Sometimes it is useful to recognize different symbol classes depending on some context. Many scanner generators produce scanners with *scanner states*. The scanner may pass from one state to another one upon reading a symbol.

Example 2.5.1 Skipping comments can be elegantly implemented using scanner states. For this purpose, a distinction is made between a state normal and a state comment.

Symbols from symbol classes that are relevant for the semantics are processed in state normal. An additional symbol class CommentInit contains the start symbol of a comment, e.g., $/*$. The semantic action triggered by recognizing the symbol $/*$ switches to state comment. In state comment, only the end symbol for comments, $*/$, is recognized. All other input characters are skipped. The semantic action triggered upon finding the end-comment symbol switches back to state normal.

The actual scanner state can be kept in a global variable *yystate*. The assignment *yystate* $\leftarrow$ state changes the state to the new state state. The specification of a

scanner possessing scanner states has the form

$$\begin{array}{ll} A_0: & class\_list_0 \\ & \ldots \\ A_{r-1}: & class\_list_{r-1} \end{array}$$

where $class\_list_j$ is the sequence of regular expressions and semantic actions for state A_j. For the states normal and comment of Example 2.5.1 we get:

```
normal :
      /*   { yystate ← comment; }
           ...      //   further symbol classes
comment :
      */   { yystate ← normal; }
      .    {   }
```

The character . stands for an arbitrary input symbol. Since none of the actions for start, content, or end of comment has a *return*-statement, no symbol is returned for the whole comment. □

Scanner states determine the subsequence of symbol classes of which symbols are recognized. In order to support scanner states, the generation process of the function yylex() can still be applied to the concatenation of the sequence $class\_list_j$. The only function that needs to be modified is the function scan(). To identify the next symbol this function no longer has a single deterministic finite-state automaton but one automaton M_j for each subsequence $class\_list_j$. Depending on the actual scanner state A_j first the corresponding DFA M_j is selected and then applied for the identification of the next symbol.

2.5.2 Recognizing Reserved Words

The duties may be distributed between scanner and screener in many ways. Accordingly, there are also various choices for the functionality of the screener. The advantages and disadvantages are not easily determined. One example for two alternatives is the recognition of keywords. According to the distribution of duties given in the last chapter, the screener is in charge of recognizing reserved symbols (keywords). One possibility to do this is to form an extra symbol class for each reserved word. Figure 2.10 shows a finite-state automaton that recognizes several reserved words in its final states. Reserved keywords in C, JAVA, and OCAML, on the other hand, have the same form as identifiers. An alternative to recognizing them in the final states of a DFA therefore is to delegate the recognition of keywords to the screener while processing found identifiers.

The function scan() then signals that an identifier has been found. The semantic action associated with the symbol class identifier additionally checks whether, and

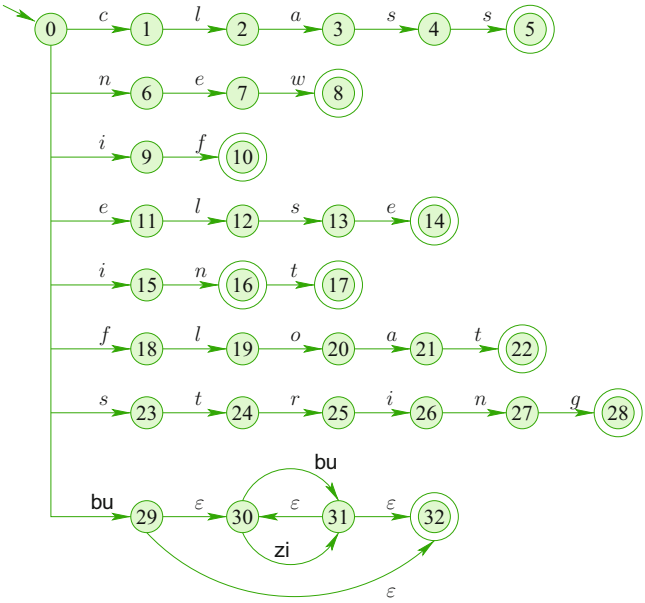


Fig. 2.10 Finite-state automaton for the recognition of identifiers and keywords class, new, if, else, in, int, float, string

if yes, which keyword was found. This distribution of work between scanner and screener keeps the size of the DFA small. A prerequisite, however, is that keywords can be quickly recognized.

Internally, identifiers are often represented by unique *int*-values, where the screener uses a hash table to compute this internal code. A hash table supports the efficient comparison of a newly found identifier with identifiers that have already been entered. If keywords have been entered into the table before lexical analysis starts, the screener thus can then identify their occurrences with approximately the same effort that is necessary for processing other identifiers.

2.6 Exercises

1. **Kleene star**
   Let Σ be an alphabet and $L, M \subseteq \Sigma^*$. Show:
   (a) $L \subseteq L^*$.
   (b) $\varepsilon \in L^*$.
   (c) $u, v \in L^*$ implies $uv \in L^*$.
   (d) L^* is the smallest set with properties (1) - (3), that is, if a set M satisfies: $L \subseteq M$, $\varepsilon \in M$ and $(u, v \in M \Rightarrow uv \in M)$ it follows $L^* \subseteq M$.

(e) $L \subseteq M$ implies $L^* \subseteq M^*$.
(f) $(L^*)^* = L^*$.

2. **Symbol classes**
FORTRAN provides the implicit declaration of identifiers according to their leading character. Identifiers beginning with one of the letters i, j, k, l, m, n are taken as *int*-variables or *int*-function results. All other identifiers denote *float*-variables.
Give a definition of the symbol classes Floatld and Intld.
3. **Extended regular expressions**
Extend the construction of finite automata for regular expressions from Fig. 2.3 in a way that it processes regular expressions r^+ and $r?$ directly. r^+ stands for rr^* and $r?$ for $(r \mid \varepsilon)$.
4. **Extended regular expressions (cont.)**
Extend the construction of finite automata for regular expressions by a treatment of *counting iteration*, that is, by regular expressions of the form:
$r\{u-o\}$ at least u and at most o consecutive instances of r
$r\{u-\}$ at least u consecutive instances of r
$r\{-o\}$ at most o consecutive instances of r
5. **Deterministic finite automata**
Convert the FA of Fig. 2.10 into a DFA.
6. **Character classes and symbol classes**
Consider the following definitions of character classes:

$$\begin{array}{lcl} \mathsf{bu} & = & a-z \\ \mathsf{zi} & = & 0-9 \\ \mathsf{bzi} & = & 0 \mid 1 \\ \mathsf{ozi} & = & 0-7 \\ \mathsf{hzi} & = & 0-9 \mid A-F \end{array}$$

and the definitions of symbol classes:

$$\begin{array}{l} b\ \mathsf{bzi}^+ \\ o\ \mathsf{ozi}^+ \\ h\ \mathsf{hzi}^+ \\ \mathsf{zi}^+ \\ \mathsf{bu}\ (\mathsf{bu} \mid \mathsf{zi})^* \end{array}$$

(a) Give the partitioning of the character classes that a scanner generator would compute.
(b) Describe the generated finite automaton using these character classes.
(c) Convert this finite automaton into a deterministic one.

7. **Reserved identifiers**
Construct a DFA for the FA of Fig. 2.10.
8. **Table compression**
Compress the table of the deterministic finite automaton using the method of Sect. 2.2.

9. **Processing of Roman numbers**
   (a) Give a regular expression for Roman numbers.
   (b) Generate a deterministic finite automaton from this regular expression.
   (c) Extend this finite automaton such that it computes the decimal value of a Roman number. The finite automaton can perform an assignment to *one* variable w with each state transition. The value is composed of the value of w and of constants. w is initialized with 0. Give an appropriate assignment for each state transition such that w contains the value of the recognized Roman number in each final state.
10. **Generation of a Scanner**
    Generate an OCAML-function yylex from a scanner specification in OCAML. Use functional constructs wherever possible.
    (a) Write a function skip that skips the recognized symbol.
    (b) Extend the generator by scanner states. Write a function next that receives the successor state as argument.

2.7 Bibliographic Notes

The conceptual separation of scanner and screener was proposed by F.L. DeRemer [15]. Many so-called compiler generators support the generation of scanners from regular expressions. Johnson et al. [29] describe such a system. The corresponding routine under UNIX, LEX, was realized by M. Lesk [42]. FLEX was implemented by Vern Paxson. The approach described in this chapter follows the scanner generator JFLEX for JAVA.

Compression methods for sparsely populated matrices as they are generated in scanner and parser generators are described and analyzed in [61] and [11].

http://www.springer.com/978-3-642-17539-8

Compiler Design
Syntactic and Semantic Analysis
Wilhelm, R.; Seidl, H.; Hack, S.
2013, X, 225 p., Hardcover
ISBN: 978-3-642-17539-8