5

Regular Grammars and Finite-State Automata

Regular grammars, Type 3 grammars, are the simplest form of grammars that still have generative power. They can describe concatenation (joining two strings together) and repetition, and can specify alternatives, but they cannot express nesting. Regular grammars are probably the best-understood part of formal linguistics and almost all questions about them can be answered.

5.1 Applications of Regular Grammars

In spite of their simplicity there are many applications of regular grammars, of which we will briefly mention the most important ones.

5.1.1 Regular Languages in CF Parsing

In some parsers for CF grammars, a subparser can be discerned which handles a regular grammar. Such a subparser is based implicitly or explicitly on the following surprising phenomenon. Consider the sentential forms in leftmost or rightmost derivations. Such sentential forms consist of a closed (finished) part, which contains terminal symbols only and an open (unfinished) part which contains non-terminals as well. In leftmost derivations the open part starts at the leftmost non-terminal and extends to the right; in rightmost derivations the open part starts at the rightmost non-terminal and extends to the left. See Figure 5.1 which uses sample sentential forms from Section 2.4.3.

Fig. 5.1. Open parts in leftmost and rightmost productions

It can easily be shown that these open parts of the sentential form, which play an important role in some CF parsing methods, can be described by a regular grammar, and that that grammar follows from the CF grammar.

To explain this clearly we first have to solve a notational problem. It is conventional to use upper case letters for non-terminals and lower case for terminals, but here we will be writing grammars that produce parts of sentential forms, and since these sentential forms can contain non-terminals, our grammars will have to produce non-terminals. To distinguish these "dead" non-terminals from the "live" ones which do the production, we shall print them barred: $\bar{\mathtt{X}}$.

With that out of the way we can construct a regular grammar *G* with start symbol R for the open parts in leftmost productions of the grammar *C* used in Section 2.4.3, which we repeat here:

```
S_S  →  L & N
 S   →  N
 L   →  N , L
 L   →  N
 N   →  t | d | h
```

The first possibility for the start symbol R of *G* is to produce the start symbol of *C*; so we have R→$\bar{\mathtt{S}}$, where $\bar{\mathtt{S}}$ is just a token. The next step is that this token, being the leftmost non-terminal in the sentential form, is turned into a "live" non-terminal, from which we are going to produce more of the sentential form: R→S. Here S is a non-terminal in *G*, and describes open parts of sentential forms deriving from S in *C*. The first possibility for S in *G* is to produce the right-hand side of S in *C* as tokens: S→$\bar{\mathtt{L}}$&$\bar{\mathtt{N}}$. But it is also possible that $\bar{\mathtt{L}}$, being the leftmost non-terminal in the sentential form, is already alive: S→L&$\bar{\mathtt{N}}$, and it may even have finished producing, so that all its tokens have already become part of the closed part of the sentential form; this leaves &$\bar{\mathtt{N}}$ for the open part: S→&$\bar{\mathtt{N}}$. Next we can move the & from the open part to the closed part: S→$\bar{\mathtt{N}}$. Again this $\bar{\mathtt{N}}$ can become productive: S→N, and, like the L above, can eventually disappear entirely: S→ε. We see how the original S→$\bar{\mathtt{L}}$&$\bar{\mathtt{N}}$ gets gradually worked down to S→ε. The second alternative of S in *C*, S→N, yields the rules S→$\bar{\mathtt{N}}$, S→N, and S→ε, but we had obtained these already.

The above procedure introduces the non-terminals L and N of *G*. Rules for them can be derived in the same way as for S; and so on. The result is the left-regular grammar *G*, shown in Figure 5.2. We have already seen that the process can create

```
R → S̄          L → N̄ , L̄
R → S          L → N , L̄
S → L̄ & N̄      L → , L̄
S → L & N̄      L → L̄
S → & N̄        L → L ✘
S → N̄          L → ε
S → N          L → N̄
S → ε          L → N
N → t | d | h
N → ε
```

Fig. 5.2. A (left-)regular grammar for the open parts in leftmost derivations

duplicate copies of the same rule; we now see that it can also produce loops, for example the rule **L→L**, marked ✘ in the figure. Since such rules contribute nothing, they can be ignored.

In a similar way a right-regular grammar can be constructed for open parts of sentential forms in a rightmost derivation. These grammars are useful for a better understanding of top-down and bottom-up parsing (Chapters 6 and 7) and are essential to the functioning of some parsers (Sections 9.13.2 and 10.2.3).

5.1.2 Systems with Finite Memory

CF (or stronger) grammars allow nesting. Since nesting can, in principle, be arbitrarily deep, the generation of correct CF (or stronger) sentences can require an arbitrary amount of memory to temporarily hold the unprocessed nesting information. Mechanical systems do not possess an arbitrary amount of memory and consequently cannot exhibit CF behavior and are restricted to regular behavior. This is immediately clear for simple mechanical systems like vending machines, traffic lights and DVD recorders: they all behave according to a regular grammar. It is also in principle true for more complicated mechanical systems, like a country's train system or a computer. However, here the argument gets rather vacuous since nesting information can be represented very efficiently and a little memory can take care of a lot of nesting. Consequently, although these systems in principle exhibit regular behavior, it is often easier to describe them with CF or stronger means, even though that incorrectly ascribes infinite memory to them.

Conversely, the global behavior of many systems that do have a lot of memory can still be described by a regular grammar, and many CF grammars are already for a large part regular. This is because regular grammars already take adequate care of concatenation, repetition and choice; context-freeness is only required for nesting. If we call a rule that produces a regular (sub)language (and which consequently could be replaced by a regular rule) "quasi-regular", we can observe the following. If all alternatives of a rule contain terminals only, that rule is quasi-regular (choice). If all alternatives of a rule contain only terminals and non-terminals with quasi-regular and non-recursive rules, then that rule is quasi-regular (concatenation). And if a rule is recursive but recursion occurs only at the end of an alternative and involves only quasi-regular rules, then that rule is again quasi-regular (repetition). This often covers large parts of a CF grammar. See Krzemień and Łukasiewicz [142] for an algorithm to identify all quasi-regular rules in a grammar.

Natural languages are a case in point. Although CF or stronger grammars seem necessary to delineate the set of correct sentences (and they may very well be, to catch many subtleties), quite a good rough description can be obtained through regular languages. Consider the stylized grammar for the main clause in a Subject-Verb-Object (SVO) language in Figure 5.3. This grammar is quasi-regular: **Verb**, **Adjective** and **Noun** are regular by themselves, **Subject** and **Object** are concatenations of repetitions of regular forms (regular non-terminals and choices) and are therefore quasi-regular, and so is **MainClause**. It takes some work to bring this grammar into standard regular form, but it can be done, as shown in Figure 5.4,

```
MainClause_S  ->  Subject Verb Object
    Subject   ->  [ a | the ] Adjective* Noun
     Object   ->  [ a | the ] Adjective* Noun
       Verb   ->  verb1 | verb2 | ···
  Adjective   ->  adj1 | adj2 | ···
       Noun   ->  noun1 | noun2 | ···
```

Fig. 5.3. A not obviously quasi-regular grammar

in which the lists for verbs, adjectives and nouns have been abbreviated to **verb**, **adjective** and **noun**, to save space.

```
            MainClause_S  ->  a SubjAdjNoun_verb_Object
            MainClause_S  ->  the SubjAdjNoun_verb_Object

SubjAdjNoun_verb_Object   ->  noun verb_Object
SubjAdjNoun_verb_Object   ->  adjective SubjAdjNoun_verb_Object

           verb_Object    ->  verb Object

                Object    ->  a ObjAdjNoun
                Object    ->  the ObjAdjNoun

             ObjAdjNoun   ->  noun
             ObjAdjNoun   ->  adjective ObjAdjNoun

                  verb    ->  verb1 | verb2 | ···
             adjective    ->  adj1 | adj2 | ···
                  noun    ->  noun1 | noun2 | ···
```

Fig. 5.4. A regular grammar in standard form for that of Figure 5.3

Even (finite) context-dependency can be incorporated: for languages that require the verb to agree in number with the subject, we duplicate the first rule:

```
MainClause  ->  SubjectSingular VerbSingular Object
            |   SubjectPlural VerbPlural Object
```

and duplicate the rest of the grammar accordingly. The result is still regular. Nested subordinate clauses may seem a problem, but in practical usage the depth of nesting is severely limited. In English, a sentence containing a subclause containing a subclause containing a subclause will baffle the reader, and even in German and Dutch nestings over say five deep are frowned upon. We replicate the grammar the desired number of times and remove the possibility of further recursion from the deepest level. Then the deepest level is regular, which makes the other levels regular in turn. The resulting grammar will be huge but regular and will be able to profit from all simple and efficient techniques known for regular grammars. The required duplications

and modifications are mechanical and can be done by a program. Dewar, Bratley and Thorne [376] describe an early example of this approach, Blank [382] a more recent one.

5.1.3 Pattern Searching

Many linear patterns, especially text patterns, have a structure that is easily expressed by a (quasi-)regular grammar. Notations that indicate amounts of money in various currencies, for example, have the structure given by the grammar of Figure 5.5, where ␣ has been used to indicate a space symbol. Examples are `$␣19.95` and `¥␣1600`. Such notations, however, do not occur in isolation but are usually embedded in long stretches of text that themselves do not conform to the grammar of Figure 5.5. To

```
        Amount_S  →  CurrencySymbol Space* Digit+ Cents?
CurrencySymbol    →  € | $ | ¥ | £ | ···
         Space    →  ␣
         Digit    →  [0123456789]
         Cents    →  . Digit Digit | .--
```

Fig. 5.5. A quasi-regular grammar for currency notations

isolate the notations, a recognizer (rather than a parser) is derived from the grammar that will accept arbitrary text and will indicate where sequences of symbols are found that conform to the grammar. Parsing (or another form of analysis) is deferred to a later stage. A technique for constructing such a recognizer is given in Section 5.10.

5.1.4 SGML and XML Validation

Finite-state automata also play an important role in the analysis of SGML and XML documents. For the details see Brüggemann-Klein and Wood [150] and Sperberg-McQueen [359], respectively.

5.2 Producing from a Regular Grammar

When producing from a regular grammar, the producer needs to remember only one thing: which non-terminal is next. We shall illustrate this and further concepts using the simple regular grammar of Figure 5.6. This grammar produces sentences consisting of an `a` followed by an alternating sequence of `b`s and `c`s followed by a terminating `a`. For the moment we shall restrict ourselves to regular grammars in standard notation; further on we shall extend our methods to more convenient forms.

The one non-terminal the producer remembers is called its *state* and the producer is said to be *in* that state. When a producer is in a given state, for example `A`, it chooses one of the rules belonging to that state, for example `A→bC`, produces the `b`

```
S_S  →  a A
 S   →  a B
 A   →  b B
 A   →  b C
 B   →  c A
 B   →  c C
 C   →  a
```

Fig. 5.6. Sample regular grammar

and moves to state **C**. Such a move is called a *state transition*, and for a rule $P \rightarrow tQ$ is written $P \xrightarrow{t} Q$. A rule without a non-terminal in the right-hand side, for example **C→a**, corresponds to a state transition to the accepting state; for a rule $P \rightarrow t$ it is written $P \xrightarrow{t} \Diamond$, where $\Diamond$ is the accepting state.

It is customary to combine the states and the possible transitions of a producer in a *transition diagram*. Figure 5.7 shows the transition diagram for the regular grammar of Figure 5.6; we see that, for example, the state transition $\mathbf{A} \xrightarrow{\mathbf{b}} \mathbf{C}$ is represented

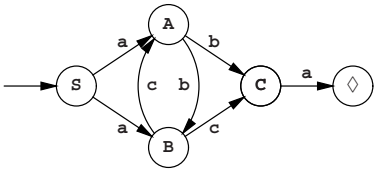


Fig. 5.7. Transition diagram for the regular grammar of Figure 5.6

by the arc marked **b** from **A** to **C**. **S** is the initial state and the accepting state is marked with a $\Diamond$. [1] The symbols on the arcs are those produced by the corresponding move. The producer can stop when it is in an accepting state.

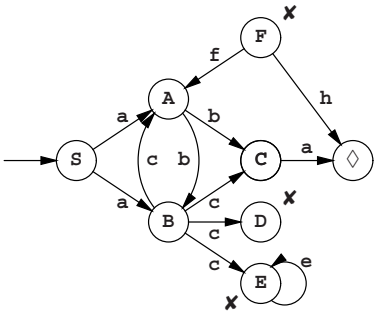
Like the non-deterministic automaton we saw in Section 3.3, the producer is an automaton, or to be more precise, a non-deterministic finite automaton, *NFA* or *finite-state automaton*, *FSA*. It is called "finite" because it can only be in a finite number of states (5 in this case; 3 bits of internal memory would suffice) and "non-deterministic" because, for example, in state **S** it has more than one way to produce an **a**.

Regular grammars can suffer from undefined, unproductive and unreachable non-terminals just like context-free grammars, and the effects are even easier to visualize. If the grammar of Figure 5.6 is extended with the rules

[1] Another convention to mark an accepting state is by drawing an extra circle around it; since we will occasionally want to explicitly mark a non-accepting state, we do not use that convention.

```
B  →  c D   undefined
B  →  c E
E  →  e E   unproductive
F  →  f A   unreachable
F  →  h
```

we obtain the transition diagram



where we can see that no further transitions are defined from **D**, which is the actual meaning of saying that **D** is undefined; that **E**, although being defined, literally has no issue; and that **F** has no incoming arrows.

The same algorithm used for cleaning CF grammars (Section 2.9.5) can be used to clean a regular grammar. Unlike CF grammars, regular grammars and finite-state automata can be minimized: for a given FS automaton *A*, a FS automaton can be constructed that has the least possible number of states and still recognizes the same language as *A*. An algorithm for doing so is given in Section 5.7.

5.3 Parsing with a Regular Grammar

The above automaton for producing a sentence can in principle also be used for parsing. If we have a sentence, for example, **abcba**, and want to check and parse it, we can view the above transition diagram as a maze and the (tokens in the) sentence as a guide. If we manage to follow a path through the maze, matching symbols from our sentence to those on the walls of the corridors as we go, and end up in ◊ exactly at the end of the sentence, we have checked the sentence. See Figure 5.8, where the path is shown as a dotted line. The names of the rooms we have visited form the backbone of the parse tree, which is shown in Figure 5.9.

But finding the correct path is easier said than done. How did we know, for example, to turn left in room **S** rather than right? Of course we could employ general maze-solving techniques (and they would give us our answer in exponential time) but a much simpler and much more efficient answer is available here: we split ourselves in two and head both ways. After the first **a** of **abcba** we are in the set of rooms {**A**, **B**}. Now we have a **b** to follow; from **B** there are no exits marked **b**, but from **A**

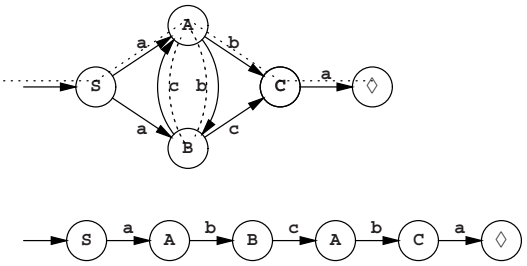


Fig. 5.8. Actual and linearized passage through the maze

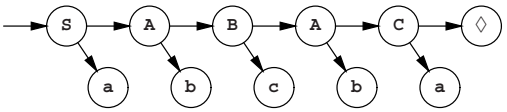


Fig. 5.9. Parse tree from the passage through the maze

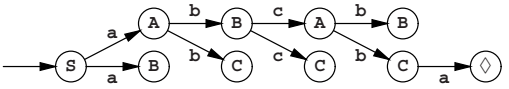


Fig. 5.10. Linearized set-based passage through the maze

there are two, which lead to **B** and **C**. So we are now in rooms {**B C**}. Our path is now more difficult to depict but still easy to linearize, as shown in Figure 5.10.

We can find the parsing by starting at the end and following the pointers backwards: ◊ ⇐ **C** ⇐ **A** ⇐ **B** ⇐ **A** ⇐ **S**. If the grammar is ambiguous the backward pointers may bring us to a fork in the road: an ambiguity has been found and both paths have to be followed separately to find both parsings. With regular grammars, however, one is often not interested in the parse, but only in the recognition: the fact that the input is correct and it ends here suffices.

5.3.1 Replacing Sets by States

Although the process described above is linear in the length of the input (each next token takes an amount of work that is independent of the length of the input), still a lot of work has to be done for each token. What is worse, the grammar has to be consulted repeatedly and so we expect the speed of the process to depend adversely on the size of the grammar. In short, we have designed an interpreter for the non-deterministic automaton, which is convenient and easy to understand, but inefficient.

Fortunately there is a surprising and fundamental improvement possible: from the NFA in Figure 5.7 we construct a new automaton with a new set of states, where each new state is equivalent to a set of old states. Where the original — non-deterministic — automaton was in doubt after the first **a**, a situation we represented as {**A**, **B**}, the new — deterministic — automaton firmly knows that after the first **a** it is in state **AB**.

The states of the new automaton can be constructed systematically as follows. We start with the initial state of the old automaton, which is also the initial state of the new one. For each new state we create, we examine its contents in terms of the old states, and for each token in the language we determine to which set of old states the given set leads. These sets of old states are then considered states of the new automaton. If we create the same state a second time, we do not analyse it again. This process is called the *subset construction* and results initially in a (deterministic) state tree. The state tree for the grammar of Figure 5.6 is depicted in Figure 5.11. To stress

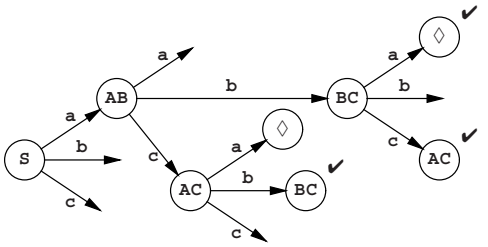


Fig. 5.11. Deterministic state tree for the grammar of Figure 5.6

that it systematically checks all new states for all symbols, outgoing arcs leading nowhere are also shown. Newly generated states that have already been generated before are marked with a ✔.

The state tree of Figure 5.11 is turned into a transition diagram by leading the arrows to states marked ✔ to their first-time representatives and removing the dead ends. The new automaton is shown in Figure 5.12. It is deterministic, and is therefore

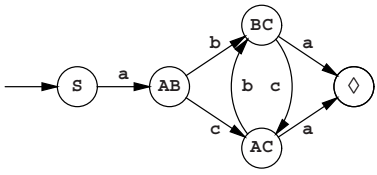


Fig. 5.12. Deterministic automaton for the grammar of Figure 5.6

called a *deterministic finite-state automaton*, or a *DFA* for short.

When we now use the sentence **abcba** as a guide for traversing this transition diagram, we find that we are never in doubt and that we safely arrive at the accepting state. All outgoing arcs from a state bear different symbols, so when following a list of symbols, we are always pointed to at most one direction. If in a given state there is no outgoing arc for a given symbol, then that symbol may not occur in that position. If it does, the input is in error.

There are two things to be noted here. The first is that we see that most of the possible states of the new automaton do not actually materialize: the old automaton had 5 states, so there were $2^5 = 32$ possible states for the new automaton while in fact it has only 5; states like **SB** or **ABC** do not occur. This is usual; although there are non-deterministic finite-state automata with n states that turn into a DFA with 2^n states, these are rare and have to be constructed on purpose. The average garden variety NFA with n states typically results in a DFA with less than or around $10 \times n$ states.

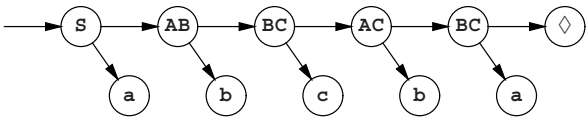
The second is that consulting the grammar is no longer required; the state of the automaton together with the input token fully determine the next state. To allow efficient look-up the next state can be stored in a table indexed by the old state and the input token. The table for our DFA is given in Figure 5.13. Using such a table, an

	input symbol			
	a	**b**	**c**
S	**AB**		
old state **AB**		**BC**	**AC**
AC		◊	**BC**
BC		◊	**AC**

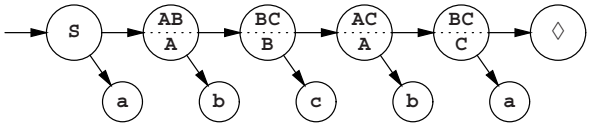
Fig. 5.13. Transition table for the automaton of Figure 5.12

input string can be checked at the cost of only a few machine instructions per token. For the average DFA, most of the entries in the table are empty (cannot be reached by correct input and refer to error states). Since the table can be of considerable size (300 states times 100 tokens is normal), several techniques exist to exploit the empty space by compressing the table. Dencker, Dürre and Heuft [338] give a survey of some techniques.

The parse tree obtained looks as follows:



which is not the original parse tree. If the automaton is used only to recognize the input string this is no drawback. If the parse tree is required, it can be reconstructed in the following fairly obvious bottom-up way. Starting from the last state ◊ and the last token **a**, we conclude that the last right-hand side (the "handle segment" in bottom-up parsing) was **a**. Since the state was **BC**, a combination of **B** and **C**, we look through the rules for **B** and **C**. We find that **a** derived from **C**→**a**, which narrows down **BC** to **C**. The rightmost **b** and the **C** combine into the handle **bC** which in the set {**A**, **C**} must derive from **A**. Working our way backwards we find the parsing



This method again requires the grammar to be consulted repeatedly; moreover, the way back will not always be so straight as in the above example and we will have problems with ambiguous grammars.

Efficient full parsing of regular grammars has received relatively little attention; substantial information can be found in papers by Ostrand, Paull and Weyuker [144] and by Laurikari [151].

5.3.2 ε-Transitions and Non-Standard Notation

A regular grammar in standard form can only have rules of the form $A \rightarrow a$ and $A \rightarrow aB$. We shall now first extend our notation with two other types of rules, $A \rightarrow B$ and $A \rightarrow \varepsilon$, and show how to construct NFAs and DFAs for them. We shall then turn to regular expressions and rules that have regular expressions as right-hand sides (for example, $P \rightarrow a^*bQ$) and show how to convert them into rules in the extended notation.

The grammar in Figure 5.14 contains examples of both new types of rules; Figure

```
S_S  ->  A
 S   ->  a B
 A   ->  a A
 A   ->  ε
 B   ->  b B
 B   ->  b
```

Fig. 5.14. Sample regular grammar with ε-rules

5.15 presents the usual trio of NFA, state tree and DFA for this grammar. First consider the NFA. When we are in state **S** we see the expected transition to state **B** on the token **a**, resulting in the standard rule **S→aB**. The non-standard rule **S→A** indicates that we can get from state **S** to state **A** without reading (or producing) a symbol; we then say that we read the zero-length string ε and that we make an ε-*transition* (or ε-*move*): $\mathbf{S} \xrightarrow{\varepsilon} \mathbf{A}$. The non-standard rule **A→**ε creates an ε-transition to the accepting state: $\mathbf{A} \xrightarrow{\varepsilon} \Diamond$. ε-transitions should not be confused with ε-rules: unit rules create ε-transitions to non-accepting states and ε-rules create ε-transitions to accepting states.

Now that we have constructed an NFA with ε-moves, the question arises how we can process the ε-moves to obtain a DFA. To answer this question we use the same reasoning as before; in Figure 5.7, after having seen an **a** we did not know if we were in state **A** or state **B** and we represented that as {**A**, **B**}. Here, when we enter state **S**, even before having processed a single symbol, we already do not know if we are in

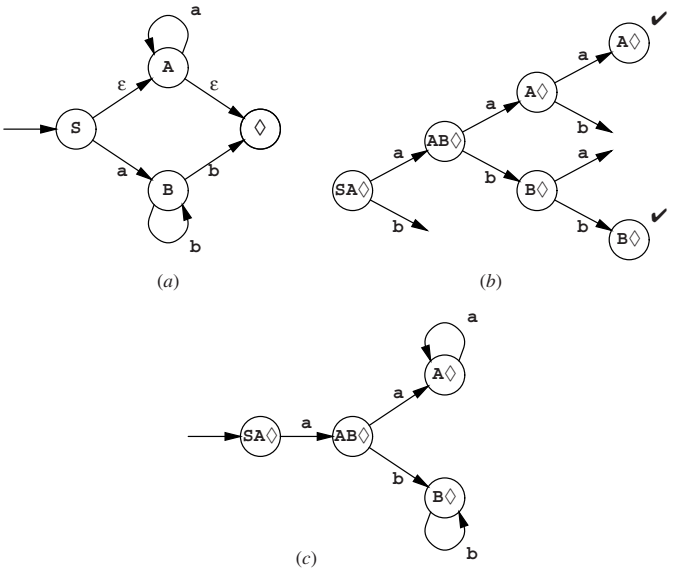


Fig. 5.15. NFA (*a*), state tree (*b*) and DFA (*c*) for the grammar of Figure 5.14

states **S**, **A** or ◊, since the latter two are reachable from **S** through ε-moves. So the initial state of the DFA is already compound: **SA**◊. We now have to consider where this state leads to for the symbols **a** and **b**. If we are in **S** then **a** will bring us to **B** and if we are in **A**, **a** will bring us to **A**. So the new state includes **A** and **B**, and since ◊ is reachable from **A** through ε-moves, it also includes ◊ and its name is **AB**◊. Continuing in this vein we can construct the complete state tree (Figure 5.15(*b*)) and collapse it into a DFA (*c*). Note that all states of the DFA contain the NFA state ◊, so the input may end in all of them.

The set of NFA states reachable from a given state through ε-moves is called the ε-*closure* of that state. The ε-closure of, for example, **S** is {**S**, **A**, ◊}.

For a completely different way of obtaining a DFA from a regular grammar that has recently found application in the field of XML validation, see Brzozowski [139].

5.4 Manipulating Regular Grammars and Regular Expressions

As mentioned in Section 2.3.3, regular languages are often specified by regular expressions rather than by regular grammars. Examples of regular expressions are `[0-9]+(.[0-9]+)?` which should be read as "one or more symbols from the set 0 through 9, possibly followed by a dot which must then be followed by one or more symbols from 0 through 9" (and which represents numbers with possibly a dot in them) and `(ab)*(p|q)+`, which should be read as "zero or more strings **ab** followed by one or more **p**s or **q**s" (and which is not directly meaningful). The usual forms occurring in regular expressions are recalled in the table in Figure 5.16, where

R, R_1, and R_2 are arbitrary regular expressions; some systems provide more forms,

Form	Meaning	Name
R_1R_2	R_1 followed by R_2	concatenation
$R_1 \mid R_2$	R_1 or R_2	alternative
R^*	zero or more Rs	optional sequence (Kleene star)
R^+	one or more Rs	(proper) sequence
$R^?$	zero or one R	optional
(R)	R	grouping
$[abc\cdots]$	any symbol from the set $abc\cdots$	
a	the symbol a itself	

Fig. 5.16. Some usual elements of regular expressions

some provide fewer.

In computer input, no difference is generally made between the metasymbol * and the symbol `*`, etc. Special notations will be necessary if the language to be described contains any of the symbols | `* + ? ( ) [` or `]`.

5.4.1 Regular Grammars from Regular Expressions

A regular expression can be converted into a regular grammar by using the transformations given in Figure 5.17. The T in the transformations stands for an intermediate non-terminal, to be chosen fresh for each application of a transformation; α stands for any regular expression not involving non-terminals, possibly followed by a non-terminal. If α is empty, it should be replaced by ε when it appears alone in a right-hand side.

The expansion from regular expression to regular grammar is useful for obtaining a DFA from a regular expression, as is for example required in lexical analysers like *lex*. The resulting regular grammar corresponds directly to an NFA, which can be used to produce a DFA as described above. There is another method to create an NFA from the regular expression, which requires, however, some preprocessing on the regular expression; see Thompson [140].

We shall illustrate the method using the expression `(ab)*(p|q)+`. Our method will also work for regular grammars that contain regular expressions (like $A \to ab^*cB$) and we shall in fact immediately turn our regular expression into such a grammar:

$$\texttt{S}_\texttt{s} \quad \to \quad \texttt{(ab)*(p|q)+}$$

Although the table in Figure 5.17 uses T for generated non-terminals, we use `A`, `B`, `C`, ... in the example since that is less confusing than $T_1, T_2, T_3, \ldots$. The transformations are to be applied until all rules are in (extended) standard form.

The first transformation that applies is $P \to R^*\alpha$, which replaces $\texttt{S}_\texttt{s} \to \texttt{(ab)*(p|q)+}$ by

Rule pattern	Replace by
$P \to a$	(standard)
$P \to aQ$	(standard)
$P \to Q$	(extended standard)
$P \to \varepsilon$	(extended standard)
$P \to a\alpha$	$P \to aT$
	$T \to \alpha$
$P \to (R_1 \vert R_2 \vert \cdots)\alpha$	$P \to R_1\alpha$
	$P \to R_2\alpha$
	$\cdots$
$P \to (R)\alpha$	$P \to R\alpha$
$P \to R^*\alpha$	$P \to T$
	$T \to RT$
	$T \to \alpha$
$P \to R^+\alpha$	$P \to RT$
	$T \to RT$
	$T \to \alpha$
$P \to R^?\alpha$	$P \to R\alpha$
	$P \to \alpha$
$P \to [abc\cdots]\alpha$	$P \to (a\vert b\vert c\vert \cdots)\alpha$

Fig. 5.17. Transformations on extended regular grammars

```
S_S  ->  A                ✔
 A   ->  (ab) A
 A   ->  (p | q)+
```

The first rule is already in the desired form and has been marked ✔. The transformations $P \to (R)\alpha$ and $P \to a\alpha$ work on `A -> (ab) A` and result in

```
A  ->  a B                ✔
B  ->  b A                ✔
```

Now the transformation $P \to R^+\alpha$ must be applied to `A -> (p | q)+`, yielding

```
A  ->  (p | q)  C
C  ->  (p | q)  C
C  ->  ε                  ✔
```

The ε originated from the fact that `(p | q)+` in `A -> (p | q)+` is not followed by anything (of which ε is a faithful representation). Now `A -> (p | q) C` and `C -> (p | q) C` are easily decomposed into

```
A  ->  p C                ✔
A  ->  q C                ✔
C  ->  p C                ✔
C  ->  q C                ✔
```

```
S_S  →  A
A    →  a B
B    →  b A
A    →  p C
A    →  q C
C    →  p C
C    →  q C
C    →  ε
```

Fig. 5.18. Extended-standard regular grammar for `(ab)*(p|q)+`

The complete extended-standard version can be found in Figure 5.18; an NFA and DFA can now be derived using the methods of Section 5.3.1 (not shown).

5.4.2 Regular Expressions from Regular Grammars

Occasionally, for example in Section 9.12, it is useful to condense a regular grammar into a regular expression. The transformation can be performed by alternatingly substituting a rule and applying the transformation patterns from Figure 5.19. The first

Rule pattern	Replace by
$P \to R_1Q_1$ $P \to R_2Q_2$ $\cdots$	$P \to R_1Q_1 \mid R_2Q_2 \cdots$
$P \to R_1Q \mid R_2Q \mid \cdots Q \mid \alpha$	$P \to (R_1 \mid R_2 \mid \cdots)Q \mid \alpha$
$P \to (R)P \mid R_1Q_1 \mid R_2Q_2 \mid \alpha$	$P \to (R)^*R_1Q_1 \mid (R)^*R_2Q_2 \mid \beta$

Fig. 5.19. Condensing transformations on regular grammars

pattern combines all rules for the same non-terminal. The second pattern combines all regular expressions that precede the same non-terminal in a right-hand side; α is a list of alternatives that do not end in Q (but see next paragraph). The third pattern removes right recursion: if the repetitive part is (R), it prepends $(R)^*$ to all non-recursive alternatives; here β consists of all the alternatives in α, with $(R)^*$ prepended to each of them. $Q_1, Q_2, \cdots$ should not be equal to P (but see next paragraph). When α is ε it can be left out when it is concatenated with a non-empty regular expression.

The substitutions and transformations may be applied in any order and will always lead to a correct regular expression, but the result depends heavily on the application order; to obtain a "nice" regular expression, human guidance is needed. Also, the two conditions in the previous paragraph may be violated without endangering the correctness, but the result will be a more "ugly" regular expression.

We will now apply the transformation to the regular grammar of Figure 5.18, and will not hesitate to supply the human guidance. We first combine the rules by their left-hand sides (transformation 1):

```
S  →  A
A  →  a B | p C | q C
B  →  b A
C  →  p C | q C | ε
```

Next we substitute **B**:

```
A  →  a b A | p C | q C
C  →  p C | q C | ε
```

followed by scooping up prefixes (transformation 2):

```
A  →  (ab) A | (p|q) C
C  →  (p|q) C | ε
```

Note that we have also packed the **ab** that prefixes **A**, to prepare it for the next transformation, which involves turning recursion into repetition:

```
S  →  A
A  →  (ab)* (p|q) C
C  →  (p|q)*
```

Now **C** can be substituted in **A** and **A** in **S**, resulting in

```
S  →  (ab)* (p|q) (p|q)*
```

This is equivalent but not identical to the **(ab)*(p|q)+** we started with.

5.5 Manipulating Regular Languages

In Section 2.10 we discussed the set operations "union", "intersection", and "negation" on CF languages, and saw that the latter two do not always yield CF languages. For regular languages the situation is simpler: these set operations on regular languages always yield regular languages.

Creating a FS automaton for the union of two regular languages defined by the FS automata A_1 and A_2 is trivial: just create a new start state and add ε-transitions from that state to the start states of A_1 and A_2. If need be the ε-transitions can then be removed as described in Section 5.3.1.

There is an interesting way to get the negation (complement) of a regular language L defined by a FS automaton, provided the automaton is ε-free. When an automaton is ε-free, each state t in it shows directly the set of tokens C_t with which an input string that brings the automaton in state t can continue: C_t is exactly the set of tokens for which t has an outgoing transition. This means that if the string continues with a token which is not in C_t, the string is not in L, and so we may conclude it is in $\neg L$. Now we can "complete" state t by adding outgoing arrows on all tokens not in C_t and lead these to a non-accepting state, which we will call s_{-1}. If we perform this completion for all states in the automaton, including s_{-1}, we obtain a so-called *complete automaton*, an automaton in which all transitions are defined.

The complete version of the automaton of Figure 5.7 is shown in Figure 5.20, where the non-accepting state is marked with a ✘.

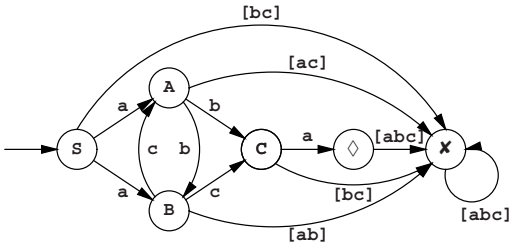


Fig. 5.20. The automaton of Figure 5.7 completed

The importance of a complete automaton lies in the fact that it never gets stuck on any (finite) input string. For those strings that belong to the language L of the automaton, it ends in an accepting state; for those that do not it ends in a non-accepting state. And this immediately suggests how to get an automaton for the complement (negative) of L: swap the status of accepting and non-accepting states, by making the accepting states non-accepting and the non-accepting states accepting!

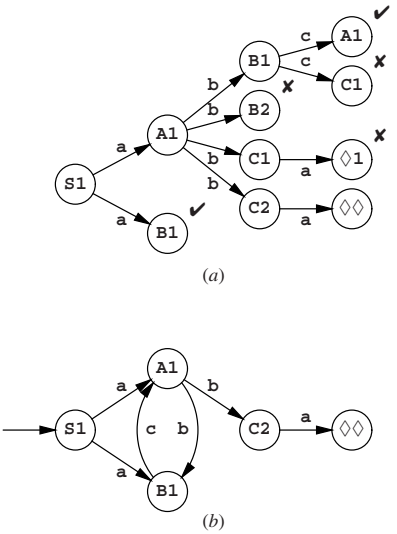
Note that completing the automaton has damaged its error detection properties, in that it will not reject an input string at the first offending character but will process the entire string and only then give its verdict.

The completion process requires the automaton to be ε-free. This is easily achieved by making it deterministic, as described on page 145, but that may be overkill. See Problem 5.4 for a way to remove the ε-transitions only.

Now that we have negation of FSAs, constructing the intersection of two FSAs seems easy: just negate both automata, take the union, and negate the result, in an application of De Morgan's Law $p \cap q = \neg((\neg p) \cup (\neg q))$. But there is a hitch here. Constructing the negation of an FSA is easy only if the automaton is ε-free, and the union in the process causes two ε-transitions in awkward positions, making this "easy" approach quite unattractive.

Fortunately there is a simple trick to construct the intersection of two FS automata that avoids these problems: run both automata simultaneously, keeping track of their two states in one single new state. As an example we will intersect automaton A_1, the automaton of Figure 5.7, with an FSA A_2 which requires the input to contain the sequence **ba**. A_2 is represented by the regular expression $\texttt{.}^*\texttt{ba.}^*$. It needs 3 states, which we will call **1** (start state), **2** and ◊ (accepting state); it has the following transitions: $\texttt{1} \xrightarrow{\texttt{[abc]}} \texttt{1}$, $\texttt{1} \xrightarrow{\texttt{b}} \texttt{2}$, $\texttt{2} \xrightarrow{\texttt{a}} \lozenge$, $\lozenge \xrightarrow{\texttt{[abc]}} \lozenge$.

We start the intersection automaton $A_1 \cap A_2$ in the combined state **S1**, which is composed of the start state **S** of A_1 and the start state **1** of A_2. For each transition $P_1 \xrightarrow{t} Q_1$ in A_1 and for each transition $P_2 \xrightarrow{t} Q_2$ in A_2 we create a transition $(P_1P_2) \xrightarrow{t} (Q_1Q_2)$ in $A_1 \cap A_2$. This leads to the state tree in Figure 5.21(*a*); the corresponding FSA is in (*b*). We see that it is similar to that in Figure 5.7, except that the transition $\texttt{B} \xrightarrow{\texttt{C}} \texttt{C}$ is missing: the requirement that the string should contain the sequence **ba** removed it.



(*a*)

(*b*)

Fig. 5.21. State tree (*a*) and FSA (*b*) of the intersection of Figure 5.7 and `.*ba.*`

In principle, the intersection of an FSA with n states and one with m states can require $n \times m$ states, but in practice something like $c \times (n+m)$ for some small value of c is more usual.

Conversely, sometimes a complex FSA can be decomposed into the intersection of two much simpler FSAs, with great gain in memory requirements, and sometimes it cannot. There is unfortunately little theory on how to do this, though there are some heuristics; see Problem 5.7. The process is also called "factorization", but that is an unfortunate term, since it suggests the same uniqueness of factorization we find in integers, and the decomposition of FSAs is not unique.

5.6 Left-Regular Grammars

In a left-regular grammar, all rules are of the form $A \rightarrow a$ or $A \rightarrow Ba$ where a is a terminal and A and B are non-terminals. Figure 5.22 gives a left-regular grammar equivalent to that of Figure 5.6.

Left-regular grammars are often brushed aside as just a variant of right-regular grammars, but their look and feel is completely different. Take the process of producing a string from this grammar, for example. Suppose we want to produce the sentence **abcba** used in Section 5.3. To do so we have to first decide all the states we are going to visit, and only when the last one has been decided upon can the first token be produced:

```
S_S  →  C a
C    →  B c
C    →  A b
B    →  A b
B    →  a
A    →  B c
A    →  a
```

Fig. 5.22. A left-regular grammar equivalent to that of Figure 5.6

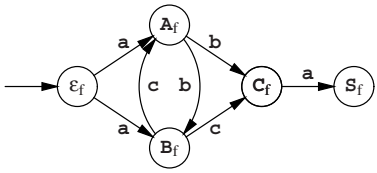
```
S
C a
A b a
B c b a
A b c b a
a b c b a
```

And once the first token is available, all of them are, and we do not have any choice any more; this is vastly different from producing from a right-regular grammar.

Parsing with a left-regular grammar is equally weird. It is easy to see that initially we are in a union of all states {**S**,**A**,**B**,**C**}, but if we now see an **a** in the input, we can move over this **a** in two rules, **B→a**, and **A→a**. Suppose we use rule **A→a**; what state are we in now? The rule specifies no state except **A**; so what does the move mean?

The easy way out is to convert the grammar to a right-regular one (see below in this section), but it is more interesting to try to answer the question what a move over **a** in **A→a** means. The only thing we know after such a move is that we have just completed a production of **A**, so the state we are in can justifiably be described as "**A** finished"; we will write such a state as $\mathbf{A}_\mathrm{f}$. And in the same manner the first rule in Figure 5.22 means that when we are in a state $\mathbf{C}_\mathrm{f}$ and we move over an **a** we are in a state $\mathbf{S}_\mathrm{f}$; this corresponds to a transition $\mathbf{C}_\mathrm{f} \xrightarrow{\mathbf{a}} \mathbf{S}_\mathrm{f}$. Then we realize that "S finished" means that we have parsed a complete terminal production of **S**; so the state $\mathbf{S}_\mathrm{f}$ is the accepting state $\Diamond$ and we see the rightmost transition in Figure 5.7 appear.

Now that we have seen that the rule $A \rightarrow Bt$ corresponds to the transition $B_f \xrightarrow{t} A_f$, and that the rule $S_S \rightarrow Bt$ corresponds to $B_f \xrightarrow{t} \Diamond$, what about rules of the form $A \rightarrow t$? After the transition over t we are certainly in the state A_f, but where did we start from? The answer is that we have not seen any terminal production yet, so we are in a state ε_f, the start state! So the rules **A→a** and **B→a** correspond to transitions $\varepsilon_\mathrm{f} \xrightarrow{\mathbf{a}} \mathbf{A}_\mathrm{f}$ and $\varepsilon_\mathrm{f} \xrightarrow{\mathbf{a}} \mathbf{B}_\mathrm{f}$, two more components of Figure 5.7. Continuing this way we quickly reconstruct the transition diagram of Figure 5.7, with modified state names:



This exposes an awkward asymmetry between start state and accepting state, in that unlike the start state the accepting state corresponds to a symbol in the grammar. This asymmetry can be partially removed by representing the start state by a more neutral symbol, for example □. We then obtain the following correspondence between our right-regular and left-regular grammar:

```
□ → a A        A → □ a
□ → a B        B → □ a
A → b B        B → A b
A → b C        C → A b
B → c A        A → B c
B → c C        C → B c
C → a ◊        ◊ → C a

□: start state  □: ε
◊: ε            ◊: start state
```

Obtaining a regular expression from a left-regular grammar is simple: most of the algorithm in Section 5.4.2 can be taken over with minimal change. Only the transformation that converts recursion into repetition

Rule pattern	Replace by
$P \rightarrow (R)P \mid R_1Q_1 \mid R_2Q_2 \mid \alpha$	$P \rightarrow (R)^*R_1Q_1 \mid (R)^*R_2Q_2 \mid \beta$

must be replaced by

$$P \rightarrow P(R) \mid Q_1R_1 \mid Q_2R_2 \mid \alpha \qquad P \rightarrow Q_1R_1(R)^* \mid Q_2R_2(R)^* \mid \beta'$$

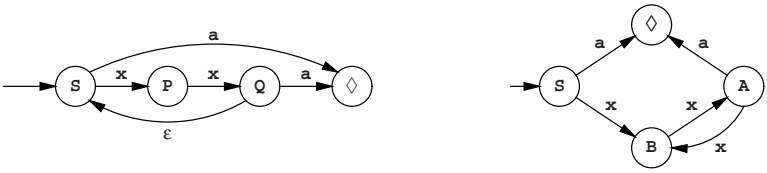
where β' consists of all the alternatives in α, with $(R)^*$ appended to each of them. This is because **A→aA|b** yields $\mathtt{a}^*\mathtt{b}$ but **A→Aa|b** yields $\mathtt{ba}^*$.

5.7 Minimizing Finite-State Automata

Turning an NFA into a DFA usually increases the size of the automaton by a moderate factor, perhaps 10 or so, and may occasionally grossly inflate the automaton. Considering that for a large automaton a size increase of a factor of say 10 can pose a major problem; that even for a small table any increase in size is undesirable if the table has to be stored in a small electronic device; and that large inflation factors may occur unexpectedly, it is often worthwhile to try to reduce the number of states in the DFA.

The key idea of the *DFA minimization algorithm* presented here is that we consider states to be equivalent until we can see a difference. To this end the algorithm keeps the DFA states in a number of mutually disjoint subsets, a "partition." A *partition* of a set S is a collection of subsets of S such that each member of S is in exactly one of those subsets; that is, the subsets have no elements in common and their union is the set S. The algorithm iteratively splits each subset in the partition as long as it can see a difference between states in it.

We will use the DFA from Figure 5.23(b) as an example; it can be derived from the NFA in Figure 5.23(a) through the subset algorithm with **A** = **SQ** and **B** = **P**, and is not minimal, as we shall see.



(a) (b)

Fig. 5.23. A non-deterministic FSA and the resulting deterministic but not minimal FSA

Initially we partition the set of states into two subsets: one containing all the accepting states, the other containing all the other states; these are certainly different. In our example this results in one subset containing states **S**, **B** and **A**, and one subset containing the accepting state ◊.

Next, we process each subset S_i in turn. If there exist two states q_1 and q_2 in S_i that on some symbol a have transitions to members of different subsets in the current partition, we have found a difference and S_i must be split. Suppose we have $q_1 \xrightarrow{a} r_1$ and $q_2 \xrightarrow{a} r_2$, and r_1 is in subset X_1 and r_2 is in a different subset X_2, then S_i must be split into one subset containing q_1 and all other states q_j in S_i which have $q_j \xrightarrow{a} r_j$ with r_j in X_1, and a second subset containing the other states from S_i. If q_1 has no transition on a but q_2 does, or vice versa, we have also found a difference and S_i must be split as well.

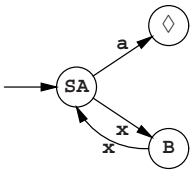
In our example, states **S** and **A** have transitions on **a** (to the same state, ◊), but state **B** does not, so this step results in two subsets, one containing the states **S** and **A**, and the other containing state **B**.

We repeat applying this step to all subsets in the partition, until no subset can be split any more. This will eventually happen, because the total number of subsets is bounded: there can be no more subsets in a partition than there are states in the original DFA, and during the process subsets are never merged. (This is another example of a closure algorithm.)

When this process is completed, all states in a subset S_i of the resulting partition have the property that for any alphabet symbol a their transition on a ends up in the same subset $S_i(a)$ of the partition. Therefore, we can consider each subset to be a sin-

gle state in the *minimized DFA*. The start state of the minimized DFA is represented by the subset containing the start state of the original DFA, and the accepting states of the minimized DFA are represented by the subsets containing accepting states of the original DFA. The resulting DFA is, in fact, the smallest DFA that recognizes the language specified by the DFA that we started with. See, for example, Hopcroft and Ullman [391].

In our example we find no further splits, and the resulting DFA is depicted below.



5.8 Top-Down Regular Expression Recognition

The Type 3 recognition technique of Section 5.3 is a bottom-up method collecting hypotheses about the reconstruction of the production process, with a top-down component making sure that the recognized string derives from the start symbol. In fact, the subset algorithm can be derived quite easily from a specific bottom-up parser, the Earley parser, which we will meet in Section 7.2 (Problem 5.9). Somewhat surprisingly, much software featuring regular expressions uses the straightforward backtracking top-down parser from Section 6.6, adapted to regular expressions. The main advantage is that this method does not require preprocessing of the regular expression; the disadvantage is that it may require much more than linear time. We will first explain the technique briefly (backtracking top-down parsing is more fully discussed in Section 6.6), and then return to the advantages and disadvantages.

5.8.1 The Recognizer

The top-down recognizer follows the grammar of regular expressions, which we summarize here:

```
regular_expression_S  ->  compound_re*
        compound_re   ->  repeat_re | simple_re
          repeat_re   ->  simple_re ['*'|'+'|'?']
          simple_re   ->  token | '(' regular_expression ')'
```

The recognizer keeps two pointers, one in the regular expression and one in the input, and tries to move both in unison: when a token is matched both pointers move one position forward, but when a **`simple_re`** must be repeated, the regular expression pointer jumps backwards, and the input pointer stays in place. When the regular expression pointer points to the end of the regular expression, the recognizer registers a match, based on how far the input pointer got.

When the recognizer tries to recognize a **compound_re**, it first finds out whether it is a **repeat_re**. If so, it checks the mark. If that is a **+** indicating a mandatory **simple_re**, the recognizer just continues searching for a **simple_re**, but if the **simple_re** is optional (*****, **?**), the search splits in two: one for a **simple_re**, and one for the rest of the regular expression, after this **repeat_re**. When the recognizer comes to the end of a **repeat_re**, it again checks the mark. If it is a **?**, it just continues, but if it was a real repeater (*****, **+**), the search again splits in two: one jumping back to the beginning of the **repeat_re**, and one continuing with the rest of the regular expression.

When the recognizer finds that the **simple_re** is a **token**, it compares the token with the token at the input pointer. If they match, both pointers are advanced; otherwise this search is abandoned.

Two questions remain: how do we implement the splitting of searches, and what do we do with the recorded matches. We implement the search splitting by doing them sequentially: we first do the entire first search up to the end or failure, including all its subsearches; then, regardless of the result, we do the second search. This sounds bothersome, both in coding and in efficiency, but it isn't. The skeleton code for the optional **repeat_re** is just

```
procedure try_optional_repeat_re(rp, ip: int):
begin
    try_simple_re(rp, ip);
    try_regular_expression(after_subexpression(rp), ip);
end;
```

where `rp` and `ip` are the regular expression pointer and the input pointer. And the algorithm is usually quite efficient, since almost all searches fail immediately because a **token** search compares two non-matching tokens.

The processing of the recorded matches depends on the application. If we want to know if the regular expression matches the entire string, as for example in file name matching, we check if we have simultaneously reached the end of the input, and if so, we abandon all further searches and return success; if not, we just continue searching. But if, for example, we want the longest match, we keep a high-water mark and continue until all searches have been exhausted.

5.8.2 Evaluation

Some advantages of top-down regular expression matching are obvious: the algorithm is very easy to program and involves no or hardly any preprocessing of the regular expression, depending on the implementation of structuring routines like **after_subexpression()**. Other advantages are less directly visible. For example, the technique allows naming a part of the regular expression and checking its repeated presence somewhere else in the input; this is an unexpectedly powerful feature. A simple example is the pattern **(.*)=x\x**, which says: match an arbitrary segment of the input, call it **x**, and then match the rest of the input to whatever has been recognized for **x**; **\x** is called a *backreference*. (A more usual but less clear

notation for the same regular expression is `\(.*\)\1`, in which `\1` means: match the first subexpression enclosed in `\(` and `\)`.)

Faced with the input **abab**, the recognizer sets **x** to the values ε, **a**, **ab**, **aba**, and **abab** in any order, and then tries to match the tail left over in each case to the present value of **x**. This succeeds only for **x**=ε and **x**=**ab**, and only in the last case is the whole input recognized. So the above expression recognizes the language of all strings that consist of two identical parts: *ww*, where *w* is any string over the given alphabet. Since this is a context-sensitive language, we see to our amazement that, skipping the entire Type 2 languages, the Type 3 regular expressions with backreferences recognize a Type 1 language! A system which uses this feature extensively is the §-calculus (Jackson [285, 291]), discussed further in Section 15.8.3.

The main disadvantage of top-down regular expression recognition is its time requirements. Although they are usually linear with a very modest multiplication constant, they can occasionally be disturbingly high, especially at unexpected moments. $O(n^k)$ time requirement occur with patterns like $a^*a^*\cdots a^*$, where the a^* is repeated k times, so in principle the cost can be any polynomial in the length of the input, but behavior worse than quadratic is unusual. Finding all 10000 occurrences of lines matching the expression `.*)` in this book took 36 sec.; finding all 11000 occurrences of just the `)` took no measurable time.

5.9 Semantics in FS Systems

In FS systems, semantic actions can be attached to states or to transitions. If the semantics is attached to the states, it is available all the time and is static. It could control an indicator on some panel of some equipment, or keep the motor of an elevator running. Semantics associated with the states is also called *Moore semantics* (Moore [136]).

If the semantics is attached to the transitions, it is available only at the moment the transition is made, in the form of a signal or procedure call; it is dynamic and transitory. Such a signal could cause a plastic cup to drop in a coffee machine or shift railroad points; the stability, staticness, is then provided by the physical construction of the equipment. And a procedure call could tell the lexical analyser in a compiler that a token **begin** has been found. Semantics associated with transitions is also called *Mealy semantics* (Mealy [134]).

There are many variants of transition-associated semantics. The signal can come when specific transition $s_i \xrightarrow{t} s_j$ occurs (Mealy [134]); when a specific token causes a specific state to be entered ($* \xrightarrow{t} s_j$, where $*$ is any state); when a specific state is entered ($* \xrightarrow{*} s_j$, McNaughton and Yamada [137]); when a specific state is left ($s_j \xrightarrow{*} *$); etc. Not much has been written about these differences. Upon reading a paper it is essential to find out which convention the author(s) use. In practical situations it is usually self-evident which variant is the most appropriate.

5.10 Fast Text Search Using Finite-State Automata

Suppose we are looking for the occurrence of a short piece of text, for example, a word or a name (the "search string") in a large piece of text, for example, a dictionary or an encyclopedia. One naive way of finding a search string of length n in a text would be to try to match it to the characters 1 to n; if that fails, shift the pattern one position and try to match against characters 2 to $n+1$, etc., until we find the search string or reach the end of the text. This process is, however, costly, since we may have to look at each character n times.

Finite automata offer a much more efficient way to do text search. We derive a DFA from the string, let it run down the text and when it reaches an accepting state, it has found the string. Assume for example that the search string is **ababc** and that the text will contain only **a**s, **b**s and **c**s. The NFA that searches for this string is shown in Figure 5.24(a); it was derived as follows. At each character in the text there are two

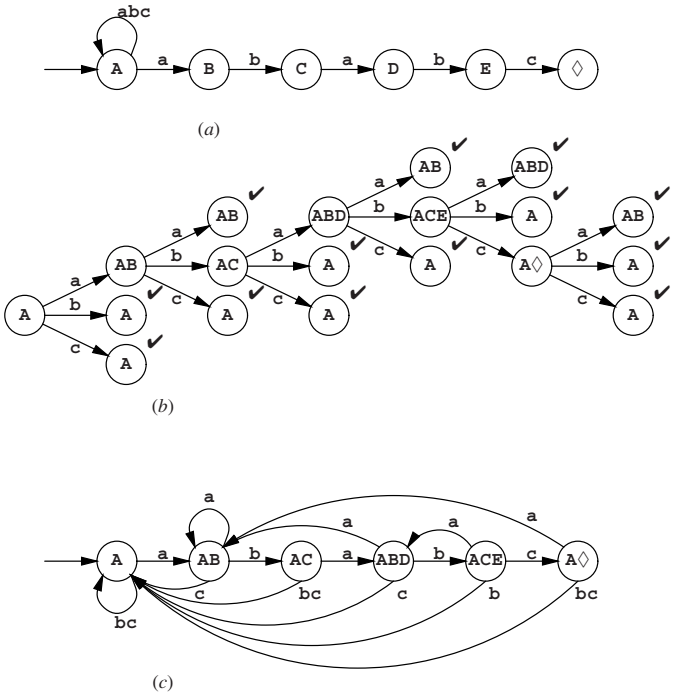


Fig. 5.24. NFA (a), state tree (b) and DFA (c) to search for **ababc**

possibilities: either the search string starts there, which is represented by the chain of states going to the right, or it does not start there, in which case we have to skip the present character and return to the initial state. The automaton is non-deterministic, since when we see an **a** in state A, we have two options: to believe that it is the start of an occurrence of **ababc** or not to believe it.

Using the traditional techniques, this NFA can be used to produce a state tree (*b*) and then a DFA (*c*). Figure 5.25 shows the states the DFA goes through when fed the text **aababababca**. We see that we have implemented *superstring recognition*, in

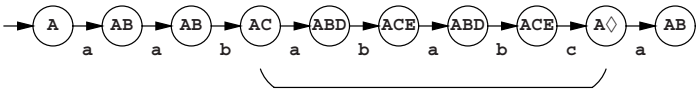


Fig. 5.25. State transitions of the DFA of Figure 5.24(*c*) on **aababababca**

which a substring of the input is recognized as matching the grammar rather than the entire input. This makes the input a superstring of a string in the language, hence the name.

This application of finite-state automata is known as the *Aho and Corasick bibliographic search algorithm* (Aho and Corasick [141]). Like any DFA, it requires only a few machine instructions per character. As an additional bonus it will search for several strings for the price of one. The DFA corresponding to the NFA of Figure 5.26 will search simultaneously for **Kawabata**, **Mishima** and **Tanizaki**. Note

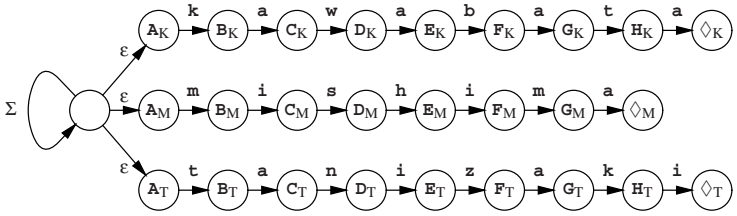


Fig. 5.26. Example of an NFA for searching multiple strings

that three different accepting states result, $\Diamond_K$, $\Diamond_M$ and $\Diamond_T$.

The Aho and Corasick algorithm is not the last word in string search. It faces stiff competition from the Rabin-Karp algorithm (Karp and Rabin [145]) and the Boyer-Moore algorithm (Boyer and Moore [143]). An excellent overview of fast string search algorithms is given by Aho [147]. Watson [149] extends the Boyer-Moore technique, which searches for a single word, so it can search for a regular expression. However fascinating all these algorithms are, they are outside the scope of this book and will not be treated here.

5.11 Conclusion

Regular grammars are characterized by the fact that no nesting is involved. Switching from one grammar rule or transition network to another is a memory-less move.

Consequently the production process is determined by a single position in the grammar and the recognition process is determined by a finite number of positions in the grammar.

Regular grammars correspond to regular expression, and vice versa, although the conversion algorithms tend to produce results that are more complicated than would be possible.

Strings in a regular set can be recognized bottom-up, using finite-state automata created by the "subset algorithm", or top-down, using recursive descent routines derived from the regular expression. The first has the advantage that it is very efficient; the second allows easy addition of useful semantic actions and recognition restrictions.

Finite-state automata are extremely important in all kinds of text searches, from bibliographical and Web searches through data mining to virus scanning.

Problems

Problem 5.1: Construct the regular grammar for open parts of sentential forms in rightmost derivations for the grammar C in Section 5.1.1.

Problem 5.2: The FS automata in Figures 5.7 and 5.12 have only one accepting state, but the automaton in Figure 5.15(c) has several. Are multiple accepting states necessary? In particular: 1. Can any FS automaton A be transformed into an equivalent single accepting state FS automaton B? 2. So that in addition B has no ε-transitions? 3. So that in addition B is deterministic?

Problem 5.3: Show that the grammar cleaning operations of removing non-productive rules and removing unreachable non-terminals can be performed in either order when cleaning a regular grammar.

Problem 5.4: Design an algorithm for removing ε-transitions from a FS automaton.

Problem 5.5: Design a way to perform the completion and negation of a regular automaton (Section 5.5) on the regular grammar rather than on the automaton.

Problem 5.6: *For readers with a background in logic:* Taking the complement of the complement of an FSA does not always yield the original automaton, but taking the complement of the complement of an already complemented FSA does, which shows that complemented automata are in some way different. Analyse this phenomenon and draw parallels with intuitionistic logic.

Problem 5.7: *Project:* Study the factorization/decomposition of FSAs; see, for example, Roche, [148].

Problem 5.8: When we assign *two* states to each non-terminal A, A_s for "A start" and A_f for "A finished, a rule $A \rightarrow XY$ results in 3 ε-transitions, $A_s \xrightarrow{\varepsilon} X_s$, $X_f \xrightarrow{\varepsilon} Y_s$ and $Y_f \xrightarrow{\varepsilon} A_f$, and a non-ε-transition $X_s \xrightarrow{X} X_f$ or $Y_s \xrightarrow{Y} Y_f$, depending on whether X or Y is a terminal. Use this view to write a more symmetrical and esthetic account of left- and right-regular grammars than given in Section 5.6.

Problem 5.9: Derive the subset algorithm from the Earley parser (Section 7.2) working on a left-regular grammar.

Problem 5.10: Derive a regular expression for `S` from the grammar of Figure 5.22.

Problem 5.11: *Project:* Section 5.7 shows how to minimize a FS automaton/grammar by initially assuming all non-terminal are equal. Can a CF grammar be subjected to a similar process and what will happen?

Problem 5.12: *History:* Trace the origin of the use of the Kleene star, the raised star meaning "the set of an unbounded number of occurrences". (See [135].)