

10

CHAPTER

Grammars

Grammar is another of those common words for which the study of formal language introduces a precise technical definition. For us, a grammar is a certain kind of collection of rules for building strings. Like DFAs, NFAs, and regular expressions, grammars are mechanisms for defining languages rigorously.

A simple restriction on the form of these grammars yields the special class of right-linear grammars. The languages that can be defined by right-linear grammars are exactly the regular languages. There it is again!

10.1 A Grammar Example for English

Let's start with an example of a grammar for a familiar language: English. An *article* can be the word *a* or *the*. We use the symbol *A* for *article* and express our definition this way:

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow the \end{aligned}$$

A *noun* can be the word *dog*, *cat*, or *rat*:

$$\begin{aligned} N &\rightarrow dog \\ N &\rightarrow cat \\ N &\rightarrow rat \end{aligned}$$

A *noun phrase* is an article followed by a noun:

$$P \rightarrow AN$$

A *verb* can be the word *loves*, *hates*, or *eats*:

$$\begin{aligned} V &\rightarrow loves \\ V &\rightarrow hates \\ V &\rightarrow eats \end{aligned}$$

A *sentence* is a noun phrase, followed by a verb, followed by another noun phrase:

$$S \rightarrow PVP$$

Put them all together and you have a grammar that defines a small subset of unpunctuated English. Call this grammar G_1 :

$$\begin{aligned} S &\rightarrow PVP \\ P &\rightarrow AN \\ V &\rightarrow loves \\ V &\rightarrow hates \\ V &\rightarrow eats \\ A &\rightarrow a \\ A &\rightarrow the \\ N &\rightarrow dog \\ N &\rightarrow cat \\ N &\rightarrow rat \end{aligned}$$

How does G_1 define a language? Think of each arrow as a rule that says how to modify strings by substitution. These rules are called *productions*. A production of the form $x \rightarrow y$ says that wherever you see the substring x , you may substitute y . For example, if we are

given the string *thedog eatsP*, the production $P \rightarrow AN$ permits us to replace the P with AN , producing *thedog eatsAN*. By starting from S and following the productions of G_1 repeatedly, you can derive a variety of unpunctuated English sentences. For example:

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow theNVP \Rightarrow thecatVP \Rightarrow thecateatsP \Rightarrow thecateatsAN \\ &\Rightarrow thecateatsaN \Rightarrow thecateatsarat \end{aligned}$$

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow aNVP \Rightarrow adogVP \Rightarrow adoglovesP \Rightarrow adoglovesAN \\ &\Rightarrow adoglovestheN \Rightarrow adoglovesthecat \end{aligned}$$

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow theNVP \Rightarrow thecatVP \Rightarrow thecathatesP \Rightarrow thecathatesAN \\ &\Rightarrow thecathatestheN \Rightarrow thecathatesthedog \end{aligned}$$

The language defined by G_1 is the language of all the lowercase strings you can get by applying the productions of G_1 , starting from S . The three examples above show that the language of G_1 includes the strings *thecateatstherat*, *adoglovesthecat*, and *thecathatesthedog*.

Often there is more than one place in a string where a production could be applied. For example, if we are given the string *PlovesP*, there are two places where the production $P \rightarrow AN$ can be used. Replacing the leftmost P gives *PlovesP* $\Rightarrow$ *ANlovesP*; replacing the rightmost one gives *PlovesP* $\Rightarrow$ *PlovesAN*. Because of this, there is often more than one way to produce the same final string:

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow ANVP \Rightarrow aNVP \Rightarrow adogVP \Rightarrow adoglovesP \Rightarrow adoglovesAN \\ &\Rightarrow adoglovestheN \Rightarrow adoglovesthecat \end{aligned}$$

$$\begin{aligned} S &\Rightarrow PVP \Rightarrow PVAN \Rightarrow PVAcat \Rightarrow PVthecat \Rightarrow Plovesthecat \Rightarrow ANlovesthecat \\ &\Rightarrow Adoglovesthecat \Rightarrow adoglovesthecat \end{aligned}$$

These two different paths both produce the same string, *adoglovesthecat*.

When a grammar contains more than one production with the same left-hand side, those productions can be written in a compressed form, with the common left-hand side first, then the arrow, then the list of alternatives for the right-hand side, separated by vertical lines. Using this notation, our G_1 looks like this:

$$\begin{aligned} S &\rightarrow PVP \\ P &\rightarrow AN \\ V &\rightarrow loves \mid hates \mid eats \\ A &\rightarrow a \mid the \\ N &\rightarrow dog \mid cat \mid rat \end{aligned}$$

This makes the grammar easier to read, but it is merely shorthand. G_1 is not changed by being written this way and still has a total of ten separate productions.

For a more abstract example, consider this grammar, which we'll call G_2 :

$$S \rightarrow aS$$

$$S \rightarrow X$$

$$X \rightarrow bX$$

$$X \rightarrow \varepsilon$$

The final production for X says that an X may be replaced by the empty string, so that for example the string $abbX$ can become the string abb . Written in the more compact way, G_2 is

$$S \rightarrow aS \mid X$$

$$X \rightarrow bX \mid \varepsilon$$

Here are some derivations using G_2 :

$$S \Rightarrow aS \Rightarrow aX \Rightarrow a$$

$$S \Rightarrow X \Rightarrow bX \Rightarrow b$$

$$S \Rightarrow aS \Rightarrow aX \Rightarrow abX \Rightarrow abbX \Rightarrow abb$$

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaaX \Rightarrow aaabX \Rightarrow aaabbX \Rightarrow aaabb$$

All derivations for G_2 follow a simple pattern: first they use $S \rightarrow aS$ zero or more times, then they use $S \rightarrow X$ once, then they use $X \rightarrow bX$ zero or more times, and then they use $X \rightarrow \varepsilon$ once. So the string that is generated always consists of zero or more a s followed by zero or more b s. Thus, the language generated by G_2 is a regular language: $L(a^*b^*)$.

In all our examples, we have used productions with a single uppercase letter on the left-hand side. Not all grammars are this simple. Productions can have any nonempty string on the left-hand side. Exactly the same mechanism of substitution still applies; for example, the production $Sb \rightarrow bS$ says that wherever you have the substring Sb , you can substitute bS . Productions of this kind can be very powerful, and we'll explore them briefly in a later chapter. But for now, all our examples and exercises will continue to use productions with a single nonterminal symbol on the left-hand side.

10.2 The 4-Tuple

Up to this point, we have followed the convention of using S as the start symbol, with derivations from there ending only when the resulting string is fully lowercase. This special treatment for the Latin alphabet with its uppercase and lowercase letters is not sufficiently general for a formal treatment of grammars. To switch to a formal treatment, we have to include definitions of two separate sets of symbols: the *terminal symbols* (corresponding to our lowercase letters) and the *nonterminal symbols* (corresponding to our uppercase letters). Counting these symbol sets, a grammar has four key parts, so we describe it mathematically as a 4-tuple.

A grammar G is a 4-tuple $G = (V, \Sigma, S, P)$, where

V is an alphabet, the *nonterminal alphabet*

Σ is another alphabet, the *terminal alphabet*, disjoint from V

$S \in V$ is the *start symbol*

P is a finite set of productions, each of the form $x \rightarrow y$, where
 x and y are strings over $\Sigma \cup V$ and $x \neq \varepsilon$.

The first part of the grammar is an alphabet, the nonterminal alphabet V . The nonterminal alphabet contains the symbols we have been writing as uppercase letters—the symbols that are used in derivations but do not occur in the string that is finally derived. The second part of a grammar is the terminal alphabet. We use the same symbol Σ that we have used throughout this book for the alphabet of current interest; these are the symbols that we have been writing as lowercase letters. The sets Σ and V are *disjoint*, meaning that no symbol occurs in both. The third part of the grammar is a special nonterminal symbol S , the start symbol. The final part of a grammar is the set of productions. Each production is of the form $x \rightarrow y$, where x and y are both strings over $\Sigma \cup V$ and x is not permitted to be the empty string.

For example, consider our previous grammar for the language $L(a^*b^*)$:

$$S \rightarrow aS \mid X$$

$$X \rightarrow bX \mid \varepsilon$$

Formally, the grammar is $G = (V, \Sigma, S, P)$, where $V = \{S, X\}$, $\Sigma = \{a, b\}$, and the set P of productions is

$$\{S \rightarrow aS, S \rightarrow X, X \rightarrow bX, X \rightarrow \varepsilon\}$$

A grammar is a 4-tuple: it is a mathematical structure with four parts given in order. The names given to those four parts in the definition above— V , Σ , S , and P —are conventional, but the grammar G could just as well have been defined as $G = (\{S, X\}, \{a, b\}, S, \{S \rightarrow aS, S \rightarrow X, X \rightarrow bX, X \rightarrow \varepsilon\})$, without naming all the parts. The important thing is to specify the four parts in the required order.

10.3 The Language Generated by a Grammar

For DFAs, we defined a one-step transition function δ and then used it to define the zero-or-more-step extended transition function δ^* ; that let us define the language of a DFA. For NFAs, we defined a one-step relation $\mapsto$, then extended it to a zero-or-more-step relation $\mapsto^*$; that let us define the language of an NFA. For grammars we will do something similar: we will first define a one-step derivation relation $\Rightarrow$ and then use it to define a zero-or-more-step extended derivation relation $\Rightarrow^*$.

The following four definitions are all made with respect to some fixed grammar $G = (V, \Sigma, S, P)$. Each production in the grammar specifies a possible substring substitution;

the production $x \rightarrow y$ says that wherever you see the substring x , you may substitute y . When a string w can be transformed into a string z in this way, we write $w \Rightarrow z$. (This is read as “ w derives z .”) Formally:

$\Rightarrow$ is a relation on strings, with $w \Rightarrow z$ if and only if there exist strings u, x, y , and v over $\Sigma \cup V$, with $w = uxv$, $z = uyv$, and $(x \rightarrow y) \in P$.

A sequence of $\Rightarrow$ -related strings, $x_0 \Rightarrow x_1 \Rightarrow \dots \Rightarrow x_n$, is called a *derivation*:

An *n-step derivation*, for any $n \geq 0$, is a sequence of strings x_0 through x_n such that for all $0 \leq i < n$, $x_i \Rightarrow x_{i+1}$.

Using n -step derivations, we can define the extended derivation relation $\Rightarrow^*$:

$\Rightarrow^*$ is a relation on strings, with $w \Rightarrow^* z$ if and only if there is a derivation of zero or more steps that starts with w and ends with z .

Notice here that $\Rightarrow^*$ is reflexive: for any string x , $x \Rightarrow^* x$ by a zero-step derivation. Using the $\Rightarrow^*$ relation, we can define the language generated by G :

The language generated by the grammar G is $L(G) = \{x \in \Sigma^* \mid S \Rightarrow^* x\}$

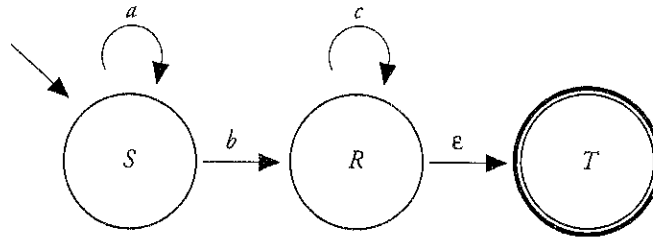
Notice here the restriction that $x \in \Sigma^*$. The intermediate strings in a derivation can involve both terminal and nonterminal symbols, but only the fully terminal strings derivable from the start symbol are in the language generated by the grammar.

All four of the preceding definitions were made with respect to some fixed grammar $G = (V, \Sigma, S, P)$. Clearly, different grammars produce different $\Rightarrow$ and $\Rightarrow^*$ relations. In those rare cases when we need to work with two or more grammars at once, we use subscripts to identify the relations of each: $\Rightarrow_G$, $\Rightarrow_H$, and so on.

10.4 Every Regular Language Has a Grammar

In exercises and examples we have seen many regular languages generated by grammars. Is there a grammar for every regular language? The answer is yes, and the proof is by construction. We will show how to take any NFA M and build a grammar G for which $L(M) = L(G)$.

First, an example of this construction. Consider the language $L(a^*bc^*)$. This is an NFA for the language:



We will construct a grammar that generates the same language. Not only will it generate the same language, but its derivations will exactly mimic the behavior of the NFA. To make the construction clearer, the states in the NFA are named S , R , and T , instead of the usual q_0 , q_1 , and q_2 . That's because in our construction, each state of the NFA will become a nonterminal symbol in the grammar, and the start state of the NFA will become the start symbol in the grammar. That settles the two alphabets and the start state for the new grammar; all that remains is to construct the set of productions.

For each possible transition $Y \in \delta(X, z)$ in the NFA (where z is any single symbol from Σ or $z = \varepsilon$), we add a production $X \rightarrow zY$ to the grammar. For our example this is

Transition of M	Production in G
$\delta(S, a) = \{S\}$	$S \rightarrow aS$
$\delta(S, b) = \{R\}$	$S \rightarrow bR$
$\delta(R, c) = \{R\}$	$R \rightarrow cR$
$\delta(R, \varepsilon) = \{T\}$	$R \rightarrow T$

In addition, for each accepting state in the NFA, the grammar contains an ε -production for that nonterminal. For our example this is

Accepting State of M	Production in G
T	$T \rightarrow \varepsilon$

So the complete grammar constructed for this example is

$S \rightarrow aS$
 $S \rightarrow bR$
 $R \rightarrow cR$
 $R \rightarrow T$
 $T \rightarrow \varepsilon$

To see how this G simulates M , let's compare the behavior of M as it accepts the string abc with the behavior of G as it generates the same string. Here are the accepting sequence of instantaneous descriptions of M and the derivation of G , side by side:

$$\begin{array}{ccccccc}
(S, abc) & \mapsto & (S, bc) & \mapsto & (R, c) & \mapsto & (R, \varepsilon) & \mapsto & (T, \varepsilon) \\
S & \Rightarrow & aS & \Rightarrow & abR & \Rightarrow & abcR & \Rightarrow & abcT & \Rightarrow & abc
\end{array}$$

M starts in the state S ; G starts with the start symbol S . After reading the first symbol a , M is in the state S ; after the first derivation step G has produced the string aS . After reading the first two symbols, ab , M is in the state R ; after the first two derivation steps G has produced the string abR . After the first three symbols, abc , M is in the state R ; after the first three derivation steps G has produced the string $abcR$. As its fourth move, M makes an ε -transition to state T ; after the first four derivation steps G has produced the string $abcT$. Now M has finished reading the string abc and ends in an accepting state; G applies a final derivation step using $T \rightarrow \varepsilon$, deriving the fully terminal string abc . Except for this final derivation step, there is a simple relation between the NFA and the grammar: every time the NFA reads a symbol, the grammar generates that symbol. The NFA can be in the state Y after reading a string x if and only if the grammar can derive the string xY .

That's how the general construction will work. Every derivation from S of a fully terminal string has the same general form:

$$S \Rightarrow z_1 R_1 \Rightarrow z_1 z_2 R_2 \Rightarrow z_1 z_2 z_3 R_3 \Rightarrow \dots \Rightarrow z_1 \dots z_n R_n \Rightarrow z_1 \dots z_n$$

At every step but the last, z_i (which is a single symbol or the empty string) is added to the right-hand side of the growing terminal string, along with a rightmost nonterminal symbol. In the last step, one of the ε -productions is used to eliminate the nonterminal symbol. This exactly traces a possible behavior of the NFA when it reads the same string.

Theorem 10.1: Every regular language is generated by some grammar.

Proof: By construction. Every regular language is accepted by some NFA, so let $M = (Q, \Sigma, \delta, S, F)$ be any NFA. (For convenience, we have named the start state S , instead of the more common q_0 .) Let G be the grammar $G = (Q, \Sigma, S, P)$, using M 's set of states as the nonterminal alphabet, M 's alphabet as the terminal alphabet, and M 's start state S as the start symbol. The set P of productions is defined as follows: wherever M has $Y \in \delta(X, \omega)$, P contains the production $X \rightarrow \omega Y$, and for each $X \in F$, P contains $X \rightarrow \varepsilon$.

By construction, G has $X \rightarrow \omega Y$ whenever M has $(X, \omega) \mapsto (Y, \varepsilon)$. We can extend this property to all strings $z \in \Sigma^*$: G has a derivation $X \Rightarrow^* zY$ if and only if M has $(X, z) \mapsto^* (Y, \varepsilon)$. (This can be proved by induction on the length of the derivation.) By construction, the grammar has $Y \rightarrow \varepsilon$ whenever the NFA has $Y \in F$. So for all strings $z \in \Sigma^*$, $\delta^*(S, z)$ contains at least one element of F if and only if $S \Rightarrow^* z$. Thus $L(M) = L(G)$.

10.5 Right-Linear Grammars

The construction of Theorem 10.1 generates productions of a particularly restricted form: *single-step form*.

A grammar $G = (V, \Sigma, S, P)$ is *single step* if and only if every production in P is in one of these two forms:

$$X \rightarrow \omega Y$$

$$X \rightarrow \varepsilon$$

where $X \in V$, $Y \in V$, and $\omega \in \Sigma \cup \{\varepsilon\}$.

If you're given a grammar that happens to be single step, you can easily reverse the construction and build a corresponding NFA. For instance, here's a single-step grammar that generates the language $L(ab^*a)$:

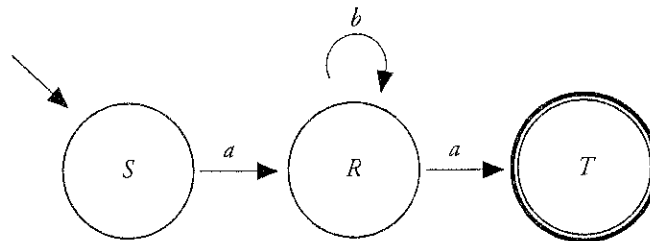
$$S \rightarrow aR$$

$$R \rightarrow bR$$

$$R \rightarrow aT$$

$$T \rightarrow \varepsilon$$

We can make an NFA with three states, S , R , and T . The first three productions correspond to three transitions in the NFA diagram, and the final production tells us to make T an accepting state. The resulting NFA is



So it is easy enough to convert a single-step grammar to an NFA. Even if the productions are not of the right form, it is sometimes possible to massage them until they are. Here's a grammar that does not follow our restrictions on production forms—a grammar for the language $\{aba\}$:

$$S \rightarrow abR$$

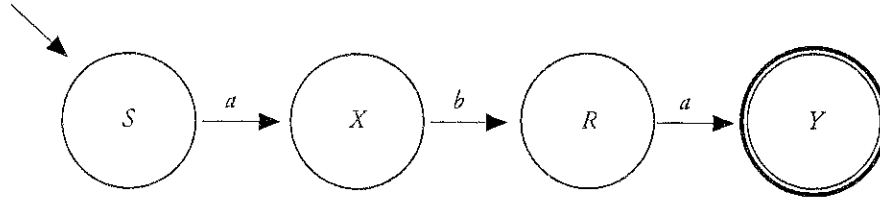
$$R \rightarrow a$$

The production $S \rightarrow abR$ does not qualify because it has the string ab instead of just a single symbol before the final nonterminal R . But this is easy to correct; we can just substitute the two productions $S \rightarrow aX$ and $X \rightarrow bR$, both of which are single step. The production $R \rightarrow a$ does not qualify either, but again, it is easy to correct; we can just substitute the two

productions $R \rightarrow aY$ and $Y \rightarrow \varepsilon$, both of which are single step. The resulting single-step grammar is

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow bR \\ R &\rightarrow aY \\ Y &\rightarrow \varepsilon \end{aligned}$$

This can then be converted into an NFA using our construction:



The grammar we started with this time, although not quite in the form we wanted, was still fairly close; every production had a single nonterminal as its left-hand side and a string containing at most one nonterminal as its right-hand side. Moreover, that nonterminal on the right-hand side was always the rightmost symbol in the string. A grammar that follows this restriction is called a *right-linear grammar*.

A grammar $G = (V, \Sigma, S, P)$ is *right linear* if and only if every production in P is in one of these two forms:

$$X \rightarrow zY$$

$$X \rightarrow z$$

where $X \in V$, $Y \in V$, and $z \in \Sigma^*$.

Every right-linear grammar generates a regular language and can be converted fairly easily into an NFA that accepts that language. As in the example above, the trick is to first rewrite the grammar to make it single step. This is always possible.

Lemma 10.1: Every right-linear grammar G is equivalent to some single-step grammar G' .

Proof: By construction. Let $G = (V, \Sigma, S, P)$ be any right-linear grammar. Each production in P is of the form $X \rightarrow z_1 \dots z_n \omega$, where $\omega \in V$ or $\omega = \varepsilon$. For each such production let P' contain these $n + 1$ productions:

$$\begin{aligned} X &\rightarrow z_1 K_1 \\ K_1 &\rightarrow z_2 K_2 \\ &\dots \end{aligned}$$

$$\begin{aligned} K_{n-1} &\rightarrow z_n K_n \\ K_n &\rightarrow \omega \end{aligned}$$

where each K_i is a new nonterminal symbol. Now let $G' = (V', \Sigma, S, P')$, where V' is the set of nonterminals used in the new productions. The new grammar G' is single step; it remains to be proven that $L(G) = L(G')$. This is straightforward. Any derivation of a terminal string in G can be converted to a derivation of the same terminal string in G' simply by replacing each step that used a rule $X \rightarrow z_1 \dots z_n \omega$ with the $n + 1$ corresponding steps in G' and vice versa.

The construction given above makes more new productions than strictly necessary. For example, it converts a production $A \rightarrow bC$ into the two productions $A \rightarrow bK_1$ and $K_1 \rightarrow C$. Conversion is unnecessary in that case, of course, since the original production was already in the required form. But it doesn't hurt, and it makes the proof simpler to use a single, general transformation for every production in the original grammar.

10.6 Every Right-Linear Grammar Generates a Regular Language

Using the techniques just introduced, we can take any right-linear grammar and convert it to an NFA. This shows that the language generated by a right-linear grammar is always a regular language.

Theorem 10.2: For every right-linear grammar G , $L(G)$ is regular.

Proof: By construction, using Lemma 10.1 and the reverse of the construction of Theorem 10.1.

Incidentally, there is parallel definition for *left-linear grammars*:

A grammar $G = (V, \Sigma, S, P)$ is *left linear* if and only if all productions in P are of one of these two forms:

$$X \rightarrow Yz \text{ or}$$

$$X \rightarrow z,$$

where $X \in V$, $Y \in V$, and $z \in \Sigma^*$.

With a little more work, one can show that the language generated by a left-linear grammar is also always a regular language. A grammar that is either left linear or right linear is sometimes called a *regular grammar*.

A grammar does not have to be regular to generate a regular language. Consider this grammar:

$$S \rightarrow aSaa \mid \varepsilon$$

This is neither right linear nor left linear, yet the language it generates is a simple regular language: $L((aaa)^*)$. If a grammar is regular, you can conclude that the language it generates is regular. But if a grammar is not regular, you can't conclude anything about the language it generates; the language may still be regular.

All of this leads to the next important question: are there grammars that generate languages that are *not* regular? This is answered in the next chapter.

Exercises

EXERCISE 1

These questions concern grammar G_1 from section 10.1:

- Show a derivation from S for the string *thedogeatsthecat*.
- List all the lowercase strings that can be derived from P .
- How many lowercase strings are in the whole language defined by G_1 —the language of all lowercase strings that can be derived from S ? Show your computation.

EXERCISE 2

Modify grammar G_1 from Section 10.1 to make it allow at most one of the adjectives *red*, *little*, *black*, or *cute*, as part of any noun phrase. For example, your grammar should generate the sentence *thereddoglovestheblackcat*. Make sure it also generates all the original sentences, without adjectives.

EXERCISE 3

These questions concern grammar G_2 from Section 10.1:

- Show a derivation from S for the string *bbb*.
- Show a derivation from S for the string *aabb*.
- Show a derivation from S for the empty string.

EXERCISE 4

Give a regular expression for the language generated by each of these grammars.

- $S \rightarrow abS \mid \varepsilon$
- $S \rightarrow aS \mid aA$ (Hint: This one is tricky—be careful!)
 $A \rightarrow aS \mid aA$
- $S \rightarrow smellA \mid fishA$
 $A \rightarrow y \mid \varepsilon$
- $S \rightarrow aaSa \mid \varepsilon$

EXERCISE 5

Give a grammar for each of the following languages. In each case, use S as the start symbol.

- $L(a^*)$
- $L(aa^*)$
- $L(a^*b^*c^*)$