Grammar 3

CFGs that are limited to productions of the form $A \rightarrow aB$ and $C \rightarrow \lambda$ from the class of *regular grammars*. As their name suggests, regular grammars define (exactly) the class of regular sets (see Exercise 6). We observed in Chapter 3 that the language $\{ [^i] \, ^i \mid i \geq 1\}$ is not regular.

$S \rightarrow [\ T\ ]$
$T \rightarrow [\ T\ ] \mid \lambda$

This grammar establishes that the languages definable by regular grammars(regular sets) are a proper subset of the context-free languages.

Although CFGs are widely used to define the syntax of programming languages, not all syntactic rules are expressible using CFGs. For example, the rule that variables must be declared before they are used cannot be expressed in a CFG – there is no way to transmit the exact set of variables that has been declared to the body of a program. In practice, syntactic details that cannot be represented in a CFG are considered part of the static semantics and are checked by semantic routines (along with scope and type rules).

CFG can be generalized to create richer definitional mechanisms. *Context-sensitive grammars* require that nonterminals be written only when they appear in a particular context (for example, $\alpha A \beta \delta \beta$). Type-0 grammars are still more general and allow arbitrary patterns to be rewritten (for example, $\alpha \rightarrow \beta$). Although context-sensitive and type-0 grammars are more powerful than CFGs, they are also far less useful. The problem is that efficient parsers for these extended grammar classes do not exist, and without a parser there is no way to use a grammar definition to drive a compiler. Efficient parsers for many classes of CFGs do exist, however; hence, CFGs represent a nice balance between generality and practicality. Throughout this text we will focus on CFGs. Whenever we mention a grammar (without saying which kind), the grammar will be assumed context-free.

Errors in Context-Free Grammars

CFGs are a definitional mechanism. They may, however, have errors, just as programs may. Some errors are easy to detect and fix; others are far more subtle.

The basic notion of CFGs is that we start with the start symbol and apply productions until a terminal string is produced. Some CFGs are flawed, however, in that they contain “useless” nonterminals. Consider the following grammar (G_1):

S→A | B
A→a
B→B b
C→c

In G_1, nonterminal C cannot be reached from S (the start symbol), and nonterminal B derives no terminal string. Nonterminals that are unreachable or derive no terminal string are termed *useless*. Useless nonterminals (and productions that involve them) can be safely removed from a grammar without changing the language defined by the grammar. A grammar containing useless nonterminals is said to be *nonreduced*. After useless nonterminals are removed, the grammar is *reduced*. G_1 is nonreduced. After B and C are removed, we obtain an equivalent grammar, G_2, which is reduced:

$S \rightarrow A$

$A \rightarrow a$

Algorithms that detect useless nonterminals are easy to write (see Exercise 7). Many parser generators check to see if a grammar is reduced. If it is not, the grammar probably contains errors (often caused by mistyping the grammar specification).

A more serious grammar flaw is that sometimes a grammar allows a program to have two or more different parse trees (and thus a nonunique structure). Consider, for example, the following grammar, which generates expressions using just infix:

<expression>→<expression> - <expression>

<expression>→ ID

This grammar allow two different parse trees for ID-ID-ID, as illustrated in Figures 4.2 and 4.3.

Grammars that allow different parse trees for the same terminal string are termed *ambiguous*. They are rarely used because a unique structure (that is, parse free) cannot be guaranteed for all inputs, and hence a unique translation, guided by the parse tree structure, may not be obtained. We normally restrict ourselves to unambiguous grammars in order to guarantee unique structure.

Naturally, we would like an algorithm that checks to see if a grammar is ambiguous. However, it is impossible to decide whether a given CFG is ambiguous (Hopcroft and Ullman 1969), so such an algorithm is impossible to create. Fortunately for certain grammar classes, including those for which we can generate parses, we can prove that constituent grammars are unambiguous.

4.3 Capabilities of Context-Free Grammars

Context-free grammars are capable of describing most, but not all, of the syntax of programming language. In this section we shall try to indicate what programming language constructs can, and cannot, be described by context-free grammars.

Regular Expressions vs. Context-Free Grammars

Regular expression, as we have seen, are capable of describing the syntax of tokens. Any syntactic construct that can be described by a regular expression can also be described by a context-free grammar.

For example, the regular expression (a | b)(a | b | 0 | 1)* and the context-free grammar

$$S \rightarrow aA \mid bA$$
$$A \rightarrow aA \mid bA \mid 0A \mid 1A \mid \varepsilon$$

describe the same language. This grammar was constructed from the obvious NFA for the regular expression using the following construction: For each state there is a nonterminal symbol. If state A has a transition to state B on symbol a, introduce production $A \rightarrow aB$. If A goes to B on input ε, introduce $A \rightarrow B$. If A is an accepting state, introduce $A \rightarrow \varepsilon$. Make the start state of the NFA be the start symbol of the grammar.

Since every regular set can be described by a context-free grammar, we may reasonably ask, “Why bother with regular expression?” There are several reasons. First, the lexical rules are usually quite simple and we don’t need a notation as powerful as context-free grammar. With the regular expression notation it is a bit easier to understand what set of strings is being defined than it is to grasp the language defined by a collection of productions. Second, it is easier to construct efficient recognizes from regular expression than from context-free grammar. Third, separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords and so forth. Context-free grammars, on the other hand, are most useful in describing nested structures such as balanced parentheses, matching begin-end's corresponding if-then-else's and so on. These nested structures cannot be described by regular expressions.

Examples of Context-Free Grammars

Let us consider some examples of grammar fragments for common programming language constructs.

Example 4.7. Consider the grammar (4.10)

$$S \rightarrow (S)\ S \mid \varepsilon \qquad (4.10)$$

This simple grammar generates all strings of balanced parentheses, and only those. To see this, we shall show that every sentence derivable from S is balanced, and that every balanced string is derivable from S. To show that every sentence derivable from S is balanced, a simple inductive proof on the number of steps in a derivation suffices. The only string of terminals derivable from S in one step is the empty string, which surely is balanced.

Now if we assume that all derivations of fewer than n steps produce balanced sentences, consider a leftmost derivation of exactly n steps. Such a derivation must be of the form:

$$S \Rightarrow (S)\ S \overset{*}{\Rightarrow} (x)\ S \overset{*}{\Rightarrow} (x)\ y$$

The derivation of x and y from S take fewer than n steps so, by the inductive hypothesis, x and y are balanced. Therefore the string (x)y must be balanced.

We have thus shown that any string derivable from S is balanced. We must next show that every balanced string is derivable from S. We now use induction on the length of a string. The empty string is derivable from S. Assume that every balanced string of length less than 2n is derivable from S, and consider a balanced string w of length 2n, n $\geq$1. Surely w begins with a left parenthesis. Let (x) be the shortest prefix of w having an equal number of left and right parentheses. Then w can be written as (x)y where both x and y are balanced. Since x and y of length less than 2n, they are derivable from S by the inductive hypothesis. Thus, we can find a derivation of the form

$$S \Rightarrow (S)\ S \overset{*}{\Rightarrow} (x)\ S \overset{*}{\Rightarrow} (x)\ y$$

proving that w= (x)y is also derivable from S.

Example 4.9. The language $L_2=\{a^nb^mc^nd^m \mid n\geq 1 \text{ and } m\geq 1\}$ is not context-free. That is, L_2 consists of a words in a*b*c*d* such that the number of a's and c's are equal and the number of b's and d's are equal. (Recall an means a written n times.) L_2 is embedded in languages which require that procedures be declared with the same number of formal parameters as there are actual parameters in their use. That is, a^n and b^m could represent the formal parameter lists in two procedures declared to have n and m arguments, respectively. Then c^n and d^m represent the actual parameter lists in calls to these two procedures.

Again note that the typical syntax of procedure definitions and uses does not concern itself with counting the number of parameters. For example, the CALL statement in a FORTRAN-like language might be described

statement → CALL id (expression list)
expression list → expression list, expression
|expression

with suitable productions for expression. Checking that the number of actual parameters in the call is correct is usually done during semantic analysis.

Example 4.10. The language $L_3 = \{ a^n b^n c^n \mid n \geq 0 \}$, that is, strings in a*b*c* with equal numbers of a's, b's and c's, is not context-free. An example of a problem, which embeds L_3 is the following. Typeset text uses italics where ordinary typed text uses underlining. In converting a file of text destined to be printed on a line printer to text suitable for a phototypesetter, one has to replace underlined words by italics. An underlined word is a string of letters followed by an equal number of backspaces and an equal number of underscores. If we regard a as any letter, b as backspace, and c as underscore, the language L_3 represents underlined words.

The conclusion is that we cannot use a grammar to describe underlined words, and more importantly, we cannot use a parser-generating tool based solely on context-free grammars to create a program to convert underlined words to italics. This situation is unusual, in that most simple text-processing programs can be written easily with the aid of a scanner generator like LEX of Chapter 3, which is even less powerful than a parser generator.

It is interesting to note that languages very similar to L_1, L_2 and L_3 are context-free. For example,

L_1 '= { wcw^R | w is in (a | b) * }, where w^R stands for w reversed, is context-free. It is generated by the grammar

$S \rightarrow aSa \mid bSb \mid c$

L_2'= {$a^n b^m c^m d^n$ | n≥1 and m≥ 1} is context-free, generated by

$S \rightarrow aSd \mid aAd$

$A \rightarrow bAc \mid bc$

Also, L_2'= {$a^n b^m c^m d^n$ | n≥1 and m≥ 1} is context-free, with grammar

$S \rightarrow AB$

$A \rightarrow aAb \mid ab$

$B \rightarrow cBd \mid cd$

Finally, L_3'= { $a^n b^n$ | n≥1 } is context-free, with grammar

$S \rightarrow aSb \mid ab$

It is worth noting that L_3' is an example of language not definable by any regular expression. To see this, suppose L_3' were the language of regular expression R. Then we could construct a DFA. A accepting L_3', A must be some finite number of states, say k. Consider the sequence of states, s_0, s_1, s_2, …, s_k entered by A given inputs ε, a, aa, … . In general, it is the state entered by A having read i a's. Then as there are only k different states, two states among s_0, s_1, …, s_k must be the same, say $s_i = s_j$. Then an additional sequence of i b's takes s_i to an accepting state, since a^ib^i is in L_3'. But then there is also a path from the initial state s_0 to s_1 to f labeled a^jb^i, as shown in Fig.4.6.

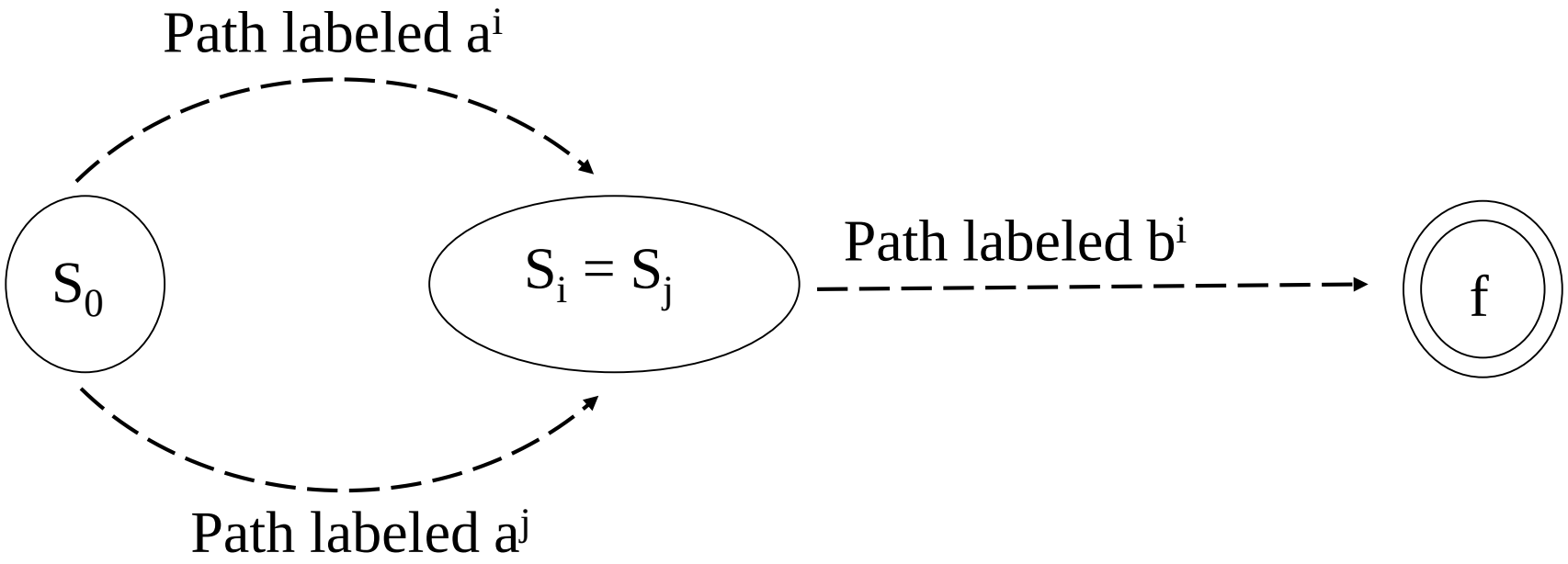

Fig. 4.6. The DFA A.

Thus, A also accepts a^jb^i, which is not in L_3', contradicting the assumption that L_3' is the language accepted by A.

Colloquially, we say that “finite automata cannot count,” meaning they cannot accept a language like L_3’ which requires that they count the number of a’s exactly. Similarly, we say “grammars can count two things but not three” since with a context-free grammar we can define L_3’ but not L_3.

Example 4.11. Consider the abstract language L1 = { wcw | w is in (a | b) * }. L_1 consists of all words composed of a repeated string of a's and b's separated by a c, such as aabcaab. It can be proven this language is not context free. This language abstracts the problem of checking that identifiers are declared before their use in a program. That is, the first w in wcw represents the declaration of an identifier w. The second w represents its use. While it is beyond the scope of this book to prove it, the non-context-freedom of L_1 directly implies the non-context-freedom of programming languages like Algol and Pascal, which require declaration of identifiers before their use, and which allow identifiers of arbitrary length.

For this reason, a grammar for the syntax of Algol or Pascal does not specify the characters in an identifier. Instead, all identifiers are represented by a token such as **id** in the grammar. In a compiler for such a language, the semantic analysis phase checks that identifiers have been declared before their use.

Example 4.12.

The language $L_2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$ is not context free. That is, L_2 consists of strings in the language generated by the regular expression a*b*c*d* such that the number of a's and c's are equal and the number of b's and d's are equal. (Recall an means a written n times.) L_2 abstracts the problem of checking that the number of formal parameters in the declaration of a procedure agrees with the number of actual parameters in a use of the procedure. That is, an and bm could represent the formal parameter lists in two procedures declared to have n and m arguments, respectively. Then cn and dm represent the actual parameter lists in calls to these two procedures.

Again note that the typical syntax of procedure definitions and uses does not concern itself with counting the number of parameters. For example, the CALL statement in a Fortran-like language might be described

stmt → **call id** (*expr_list*)

expr_list → *expr_list*, *expr* | *expr*

with suitable productions for *expr*. Checking that the number of actual parameters in the call is correct is usually done during the semantic analysis phase.

Example 4.13. The language $L_3 = \{ a^n b^n c^n \mid n \geq 0 \}$, that is, strings in L(a*b*c*) with equal numbers of a's, b's and c's, is not context free. An example of a problem that embeds L_3 is the following.

Typeset text uses italics where ordinary typed text uses underlining. In converting a file of text destined to be printed on a line printer to text suitable for a phototypesetter, one has to replace underlined words by italics. An underlined word is a string of letters followed by an equal number of backspaces and an equal number of underscores. If we regard a as any letter, b as backspace, and c as underscore, the language L_3 represents underlined words. The conclusion is that we cannot use a grammar to describe underlined words in this fashion. On the other hand, if we represent an underlined word as a sequence of letter-backspace-underscore triples then we can represent underlined words with the regular expression (abc)*.

It is interesting to note that language very similar to L_1, L_2 and L_3 are context free. For example,

L'_1= { wcw^R | w is in (a | b)* }, where w^R stands for w reversed, is context free. It is generated by the grammar

S → aSa | bSb | c

The language L_2'= { $a^n b^m c^n d^m$ | n≥1 and m≥1 } is context free, with grammar

S → aSd | aAd

A → bAc | bc

Also, L'_2'= { $a^n b^n c^m d^m$ | n≥1 and m≥1 } is context free, with grammar

S → AB

A → aAb | ab

B → cBd | cd

Finally, $L'_3 = \{ a^n b^n \mid n \geq 0 \}$ is context free, with grammar

$S \rightarrow aSb \mid ab$

It is worth noting that L'_3 is the prototypical example of a language not definable by any regular expression.