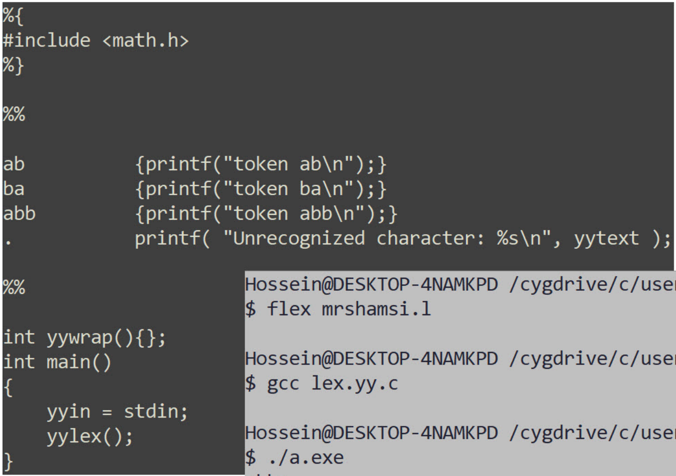
بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

پاسخ به پرسش آقای شمسی



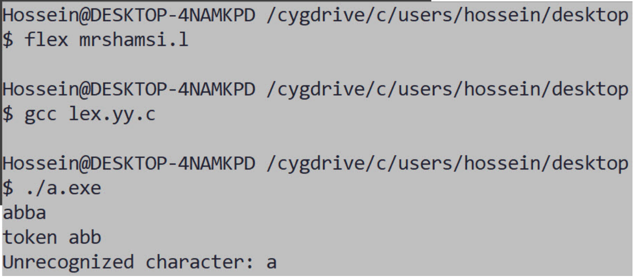
```
%{
#include <math.h>
%}

%%

ab          {printf("token ab\n");}
ba          {printf("token ba\n");}
abb         {printf("token abb\n");}
.           printf( "Unrecognized character: %s\n", yytext );

%%

int yywrap(){};
int main()
{
    yyin = stdin;
    yylex();
}
```



```
Hossein@DESKTOP-4NAMKPD /cygdrive/c/users/hossein/desktop
$ flex mrshamsi.l

Hossein@DESKTOP-4NAMKPD /cygdrive/c/users/hossein/desktop
$ gcc lex.yy.c

Hossein@DESKTOP-4NAMKPD /cygdrive/c/users/hossein/desktop
$ ./a.exe
abba
token abb
Unrecognized character: a
```

Where lexical analysis splits a text into tokens, the purpose of syntax analysis (also known as parsing) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called the syntax tree of the text. As the name indicates, this is a tree structure.

The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text.

Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree, and how the interior nodes of the tree are labeled. In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors.

The syntax of programming language constructs can be specified by CFGs or BNF (Backus-Naur Form) notation. Grammars offer significant benefits for both language designers and compiler writers.

☞ *A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.*

☞ *From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program. As a side benefit, the parser-construction process can reveal syntactic ambiguities and trouble spots that might have slipped through the initial design phase of a language.*

☞ *The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors.*

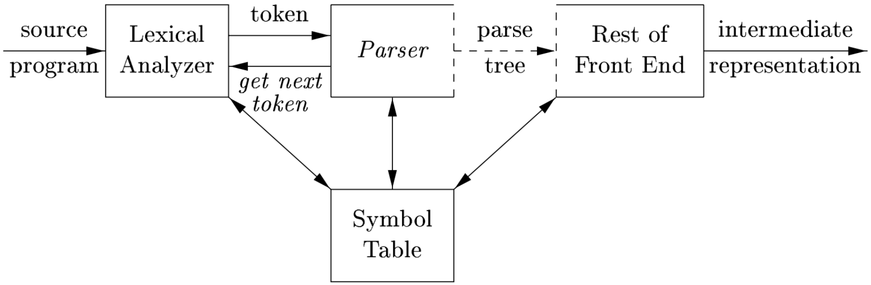
☞ *A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.*

The Role of the Parser

In our compiler model, the parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing. Thus, the parser and the rest of the front end could well be implemented by a single module.

Position of parser in compiler model



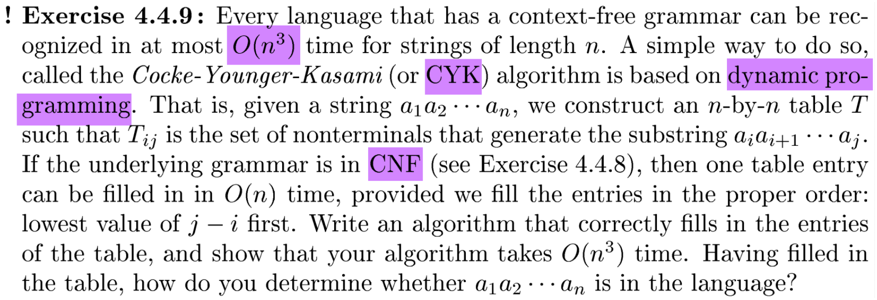
We assume that the output of the parser is some representation of the parse tree for the stream of tokens that comes from the lexical analyzer. In practice, there are a number of tasks that might be conducted during parsing, such as collecting information about various tokens into the symbol table, performing type checking and other kinds of semantic analysis, and generating intermediate code.

There are three general types of parsers for grammars:
universal, top-down, and bottom-up.

Universal parsing methods such as the Cocke-Younger-Kasami (CYK) algorithm and Earley's algorithm can parse any grammar. These general methods are, however, too inefficient to use in production compilers.

The methods commonly used in compilers can be classified as being either top-down or bottom-up. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

روش **CYK** معمولاً در درس نظریهٔ زبان‌ها و ماشین‌ها معرفی می‌شود، و مبتنی بر رویکرد ***Dynamic Programming*** است. این الگوریتم از مرتبهٔ $O(n^3)$، که n طول رشتهٔ ورودی می‌باشد.

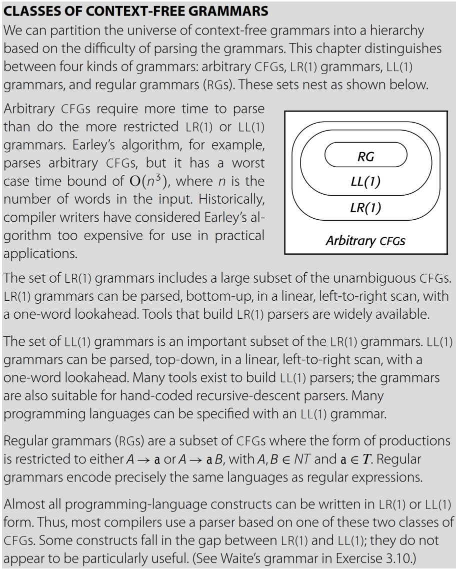


! Exercise 4.4.9: Every language that has a context-free grammar can be recognized in at most $O(n^3)$ time for strings of length n. A simple way to do so, called the *Cocke-Younger-Kasami* (or CYK) algorithm is based on dynamic programming. That is, given a string $a_1 a_2 \cdots a_n$, we construct an n-by-n table T such that T_{ij} is the set of nonterminals that generate the substring $a_i a_{i+1} \cdots a_j$. If the underlying grammar is in CNF (see Exercise 4.4.8), then one table entry can be filled in in $O(n)$ time, provided we fill the entries in the proper order: lowest value of $j - i$ first. Write an algorithm that correctly fills in the entries of the table, and show that your algorithm takes $O(n^3)$ time. Having filled in the table, how do you determine whether $a_1 a_2 \cdots a_n$ is in the language?

قرار است با گرامرهای ***LL*** و ***LR*** آشنا شویم

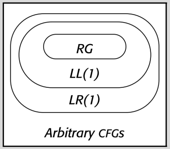
The most efficient top-down and bottom-up methods work only for subclasses of grammars, but several of these classes, particularly, LL and LR grammars, are expressive enough to describe most of the syntactic constructs in modern programming languages. Parsers implemented by hand often use LL grammars. Parsers for the larger class of LR grammars are usually constructed using automated tools.

از کتاب کوپر:



CLASSES OF CONTEXT-FREE GRAMMARS

We can partition the universe of context-free grammars into a hierarchy based on the difficulty of parsing the grammars. This chapter distinguishes between four kinds of grammars: arbitrary CFGs, LR(1) grammars, LL(1) grammars, and regular grammars (RGs). These sets nest as shown below.



Arbitrary CFGs require more time to parse than do the more restricted LR(1) or LL(1) grammars. Earley's algorithm, for example, parses arbitrary CFGs, but it has a worst case time bound of $O(n^3)$, where n is the number of words in the input. Historically, compiler writers have considered Earley's algorithm too expensive for use in practical applications.

The set of LR(1) grammars includes a large subset of the unambiguous CFGs. LR(1) grammars can be parsed, bottom-up, in a linear, left-to-right scan, with a one-word lookahead. Tools that build LR(1) parsers are widely available.

The set of LL(1) grammars is an important subset of the LR(1) grammars. LL(1) grammars can be parsed, top-down, in a linear, left-to-right scan, with a one-word lookahead. Many tools exist to build LL(1) parsers; the grammars are also suitable for hand-coded recursive-descent parsers. Many programming languages can be specified with an LL(1) grammar.

Regular grammars (RGs) are a subset of CFGs where the form of productions is restricted to either $A \rightarrow \mathtt{a}$ or $A \rightarrow \mathtt{a}\,B$, with $A, B \in NT$ and $\mathtt{a} \in T$. Regular grammars encode precisely the same languages as regular expressions.

Almost all programming-language constructs can be written in LR(1) or LL(1) form. Thus, most compilers use a parser based on one of these two classes of CFGs. Some constructs fall in the gap between LR(1) and LL(1); they do not appear to be particularly useful. (See Waite's grammar in Exercise 3.10.)

تعریف رسمی CFG (از کتاب سیپسر و کتاب مارتین)

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

A *context-free grammar* (CFG) is a 4-tuple $G = (V, \Sigma, S, P)$, where V and Σ are disjoint finite sets, $S \in V$, and P is a finite set of formulas of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

Elements of Σ are called *terminal symbols*, or *terminals*, and elements of V are *variables*, or *nonterminals*. S is the *start* variable, and elements of P are *grammar rules*, or *productions*.

4.2.2 Notational Conventions

1. These symbols are terminals:

   (a) Lowercase letters early in the alphabet, such as a, b, c.

   (b) Operator symbols such as $+$, $*$, and so on.

   (c) Punctuation symbols such as parentheses, comma, and so on.

   (d) The digits $0, 1, \ldots, 9$.

   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. These symbols are nonterminals:

   (a) Uppercase letters early in the alphabet, such as A, B, C.

   (b) The letter S, which, when it appears, is usually the start symbol.

   (c) Lowercase, italic names such as *expr* or *stmt*.

   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α the body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2, \ldots$, $A \rightarrow \alpha_k$ with a common head A (call them A-*productions*), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots , \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

4.2.3 Derivations

For a general definition of derivation, consider a nonterminal A in the middle of a sequence of grammar symbols, as in $\alpha A \beta$, where α and β are arbitrary strings of grammar symbols. Suppose $A \to \gamma$ is a production. Then, we write $\alpha A \beta \Rightarrow \alpha \gamma \beta$. The symbol $\Rightarrow$ means, "derives in one step." When a sequence of derivation steps $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ rewrites α_1 to α_n, we say α_1 *derives* α_n. Often, we wish to say, "derives in zero or more steps." For this purpose, we can use the symbol $\overset{*}{\Rightarrow}$. Thus,

1. $\alpha \overset{*}{\Rightarrow} \alpha$, for any string α, and

2. If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$.

Likewise, $\overset{+}{\Rightarrow}$ means, "derives in one or more steps."

If $S \overset{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, we say that α is a *sentential form* of G. Note that a sentential form may contain both terminals and nonterminals, and may be empty. A *sentence* of G is a sentential form with no nonterminals. The *language generated by* a grammar is its set of sentences. Thus, a string of terminals w is in $L(G)$, the language generated by G, if and only if w is a sentence of G (or $S \overset{*}{\Rightarrow} w$). A language that can be generated by a grammar is said to be a *context-free language*. If two grammars generate the same language, the grammars are said to be *equivalent*.

For example, consider the following grammar, with a single nonterminal E, which adds a production $E \rightarrow -\ E$ to the grammar (4.3):

$$E \quad \rightarrow \quad E\ +\ E \ \mid\ E\ *\ E \ \mid\ -\ E \ \mid\ (\ E\ ) \ \mid\ \mathbf{id} \tag{4.7}$$

The production $E \rightarrow -\ E$ signifies that if E denotes an expression, then $-\ E$ must also denote an expression. The replacement of a single E by $-\ E$ will be described by writing

$$E \Rightarrow -E$$

which is read, "E derives $-E$." The production $E \rightarrow (\ E\ )$ can be applied to replace any instance of E in any string of grammar symbols by (E), e.g., $E * E \Rightarrow (E) * E$ or $E * E \Rightarrow E * (E)$. We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. For example,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

We call such a sequence of replacements a *derivation* of $-(\mathbf{id})$ from E. This derivation provides a proof that the string $-(\mathbf{id})$ is one particular instance of an expression.

1. In leftmost derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \Rightarrow_{lm} \beta$.
2. In rightmost derivations, the rightmost nonterminal is always chosen; we write $\alpha \Rightarrow_{rm} \beta$ in this case.

A derivation in a context-free grammar is a *leftmost* derivation (LMD) if, at each step, a production is applied to the leftmost variable-occurrence in the current string. A rightmost derivation (RMD) is defined similarly.

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \textbf{id}$$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$

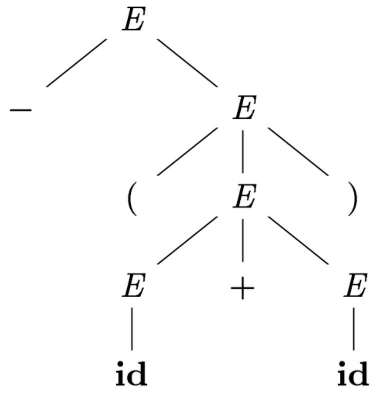
$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\textbf{id}) \underset{rm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$

4.2.4 Parse Trees and Derivations

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

Since a parse tree ignores variations in the order in which symbols in sentential forms are replaced, there is a many-to-one *relationship between derivations and parse trees.*

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$



Parse tree for $-(\mathbf{id} + \mathbf{id})$

Ambiguity

Both leftmost and rightmost derivations pick a particular order for replacing symbols in sentential forms, so they too filter out variations in the order. It is not hard to show that every parse tree has associated with it a unique leftmost and a unique rightmost derivation.

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

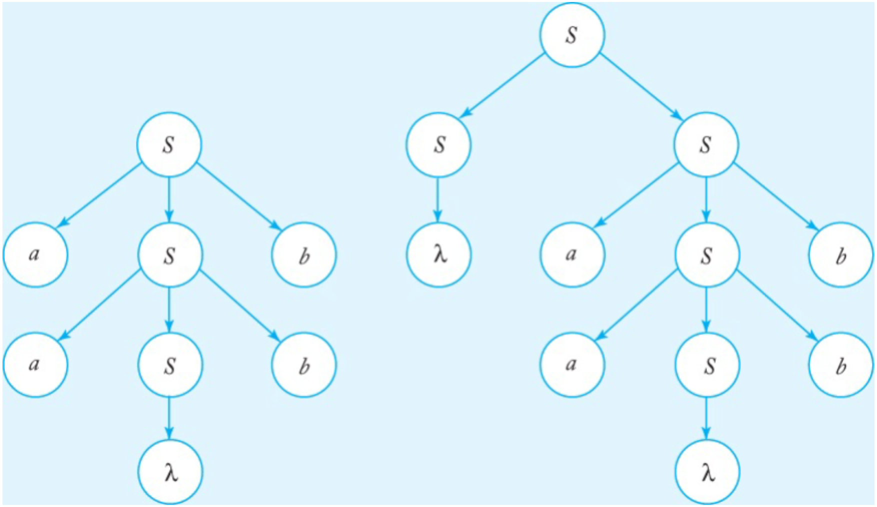
If G is a context-free grammar, then for every $x \in L(G)$, these three statements are equivalent:
1. x has more than one derivation tree.
2. x has more than one leftmost derivation.
3. x has more than one rightmost derivation.

A context-free grammar G is ambiguous if for at least one $x \in L(G)$, x has more than one derivation tree (or, equivalently, more than one leftmost derivation). In other words, a context-free grammar G is said to be ambiguous if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

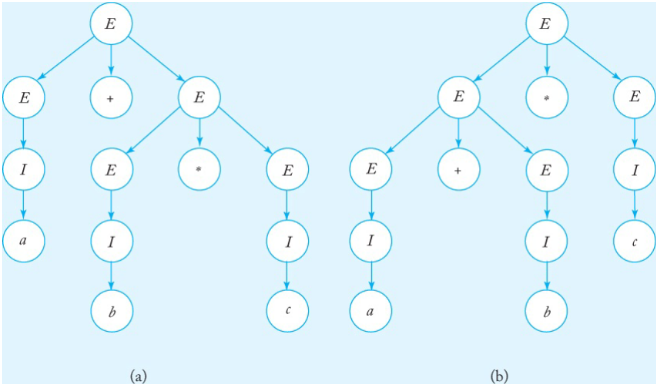
Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence. In other cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that “throw away” undesirable parse trees, leaving only one tree for each sentence.

$$S \to aSb|SS|\lambda$$



$$\begin{aligned} E &\rightarrow I, \\ E &\rightarrow E + E, \\ E &\rightarrow E * E, \\ E &\rightarrow (E)\,, \\ I &\rightarrow a|b|c. \end{aligned}$$



The Dangling *else*

In the C programming language, an if-statement can be defined by these grammar rules:

$$\begin{aligned} S \;\rightarrow\;& \text{if}\ \ (\ E\ )\ S \ \ | \\ & \text{if}\ \ (\ E\ )\ S\ \ \text{else}\ S\ \ | \\ & OS \end{aligned}$$

(In our notation, E is short for <expression>, S for <statement>, and OS for <otherstatement>.) A statement in C that illustrates the ambiguity of these rules is

```
if (e1) if (e2) f(); else g();
```

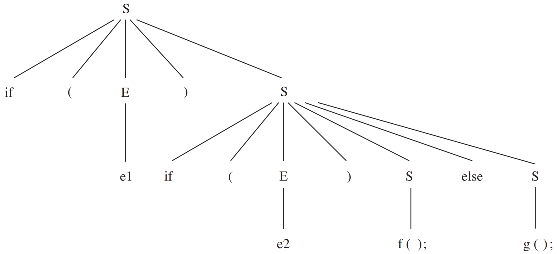
The problem is that although in C the *else* should be associated with the second *if*, as in

```
if (e1) { if (e2) f(); else g(); }
```

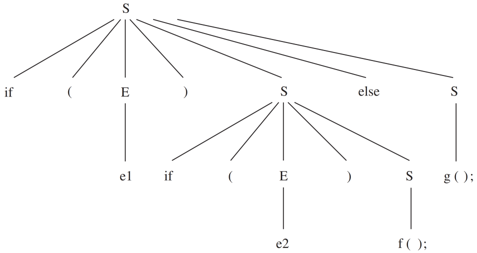
there is nothing in these grammar rules to rule out the other interpretation:

```
if (e1) { if (e2) f(); } else g();
```

The two derivation trees in Figure 4.21 show the two possible interpretations of the statement; the correct one is in Figure 4.21a.



(*a*)



(*b*)

There are equivalent grammar rules that allow only the correct interpretation. One possibility is

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow \text{if } (\, E \,) \; S_1 \;\; \text{else } S_1 \mid OS \\ S_2 &\rightarrow \text{if } (\, E \,) \; S \mid \\ & \;\; \text{if } (\, E \,) \; S_1 \;\; \text{else } S_2 \end{aligned}$$

These rules generate the same strings as the original ones and are unambiguous. We will not prove either fact, but you can see how the second might be true. The variable S_1 represents a statement in which every *if* is matched by a corresponding *else*, and every statement derived from S_2 contains at least one unmatched *if*. The only variable appearing before *else* in these rules is S_1; because the *else* cannot match any of the *if* s in the statement derived from S_1, it must match the *if* that appeared at the same time it did.