4. **Problem:** Construct a context-free grammar for the language \( \{w \in \{a, b\}^* \mid \text{w has as many } a\text{s as } b\text{s}\} \).

**Solution:** There are several different ways of designing a grammar for this language. The simplest answer is the ambiguous grammar:

\[
S \rightarrow aSbS \mid bSaS \mid \varepsilon .
\]

The first rule of the grammar expresses the condition: “If the string starts with an \( a \), then at some point of the string there has to be a corresponding \( b \). Between these two symbols there may be arbitrary balanced strings.”

For example, the string \( abab \) has two parse trees:

If we want to have an unambiguous grammar for the language, we have to ensure that the first \( a \) is associated with the first possible \( b \):

\[
S \rightarrow aAS \mid bBS \mid \varepsilon
\]

\[
A \rightarrow aAb \mid b
\]

\[
B \rightarrow bBb \mid a
\]

Now \( abab \) has only one parse tree:

5. **Problem:** Prove that the following context-free grammar is ambiguous:

\[
\begin{align*}
S & \rightarrow \text{if } b \text{ then } S \\
S & \rightarrow \text{if } b \text{ then } S \text{ else } S \\
S & \rightarrow s.
\end{align*}
\]

Design an unambiguous grammar that is equivalent to the grammar, i.e. one that generates the same language.
Solution: A context-free grammar is ambiguous if there exists a word \( w \in L(G) \) such that \( w \) has at least two different parse trees. The simplest word for the given grammar that has this property is:

\[
\text{if } b \text{ then } \text{if } b \text{ then } s \text{ else } s.
\]

Its two parse trees are:

\[
\begin{align*}
S & \rightarrow \text{if } b \text{ then } S \\
S & \rightarrow \text{if } b \text{ then } S \text{ else } S
\end{align*}
\]

Usually we want to associate an else-branch to the closest preceding if-statement. In this case the former tree corresponds to this practice.

We define a grammar \( G \) as follows:

\[
G = (V, \Sigma, P, S)
\]

\[
V = \{S, B, U, s, b, \text{if, then, else}\}
\]

\[
\Sigma = \{s, b, \text{if, then, else}\}
\]

\[
P = \{S \rightarrow B | U
\]

\[
| \quad B \rightarrow \text{if } b \text{ then } B \text{ else } B \mid s
\]

\[
| \quad U \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } B \text{ else } U
\]

Here the nonterminal \( B \) is used to derive balanced programs where each if-statement has both then- and else-branches. The nonterminal \( U \) derives those if-statements that do not have an else-branch.


Solution: The following C-program implements a top-down parser for the following grammar:

\[
C \rightarrow S \mid S; C
\]

\[
S \rightarrow a \mid \text{begin } C \text{ end} \mid \text{for } n \text{ times } do \ S
\]

This grammar is a simplified form of the one in problem 6.6. The difference is that all different numbers are replaced by a new terminal symbol \( n \) that denotes a number.

The most important functions of the program are:

- \( C() \), \( S() \) — implement the rules of the program.
• `lex()` — read the next lexeme from the input, and store it in a global variable `current_tok`.
• `expect(int token)` — tries to read the lexeme `token` from input. Gives an error message if it fails.
• `consume_token()` — mark the current lexeme used. This is necessary because sometimes we have to have a one-token lookahead before we know what rule we must apply.

In practice, the programming language parsers are implemented using `lex` and `yacc` tools\(^1\). Of these, `lex` generates a finite automaton-based lexical analyser from identifying lexemes that have been defined using regular expression, and `yacc` constructs a pushdown automaton-based parser for a given context-free grammar.

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

/* Define the alphabet */
enum TOKEN { DO, FOR, END, BEGIN, TIMES, OP, SC, NUMBER, ERROR };
const char* tokens[] = { "do", "for", "end", "begin", "times", "a", ";", "NUMBER", NULL };

/* A global variable holding the current token */
int current_tok = ERROR;

/* Maximum length of a token */
#define TOKEN_LEN 128

/* declare functions corresponding to nonterminals */
void S(void);
void C(void);

int lex(void);
void consume_token(void);
void error(char *st);
void expect(int token);

void C(void)
{
    S();
    lex();
    if (current_tok == SC) {
        consume_token();
        C();
        printf("C -> S ; C\n");
    } else {
        printf("C -> S\n");
    }
}

void S(void)
{
\(^1\)Or some of their derivatives, like `flex` or `bison`
lex();
switch (current_tok) {
    case OP:
        consume_token();
        printf("S -> a\n");
        break;
    case BEGIN:
        consume_token();
        C();
        expect(END);
        printf("S -> begin C end\n");
        break;
    case FOR:
        consume_token();
        expect(NUMBER);
        expect(TIMES);
        expect(DO);
        S();
        printf("S -> for N times do S\n");
        break;
    default:
        error("Parse error");
        } } /* int lex(void) returns the next token of the input. */
int lex(void)
{ static char token_text[TOKEN_LEN];
    int pos = 0, c, i, next_token = ERROR;

    /* Is there an existing token already? */
    if (current_tok != ERROR)
        return current_tok;

    /* skip whitespace */
    do {
        c = getchar();
    } while (c != EOF && isspace(c));
    if (c != EOF) ungetc(c, stdin);

    /* read token */
    c = getchar();
    while (c != EOF && c != ';' && !isspace(c) && pos < TOKEN_LEN) {
        token_text[pos++] = c;
        c = getchar();
    }
    if (c == ';') {
        if (pos == 0) /* semicolon as token */
            next_token = SC;
        else /* trailing semicolon, leave it for future */
            ungetc(';', stdin);
    }
}
token_text[pos] = '\0'; /* trailing zero */

/* identify token */
if (isdigit(token_text[0])) { /* number */
    next_token = NUMBER;
} else { /* not a number */
    for (i = 0; i < NUMBER; i++) {
        if (!strcmp(tokens[i], token_text)) {
            next_token = i;
            break;
        }
    }
    current_tok = next_token;
return next_token;
}

void consume_token(void)
{
    current_tok = ERROR;
}

void error(char *st)
{
    printf(st);
    exit(1);
}

/* try to read a 'token' from input */
void expect(int token)
{
    int next_tok = lex();
    if (next_tok == token) {
        consume_token();
        return;
    } else
        error("Parse error");
}

int main(void)
{
    int i;
    C();
    return 0;
}