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

**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:

\[
\begin{array}{c}
S \\
\searrow & \\
S & S \\
\searrow & \searrow \\
a & b \\
\searrow & \searrow \\
b & S \\
\searrow & \searrow \\
S & a \\
\searrow & \searrow \\
\varepsilon & \varepsilon \\
\end{array}
\]

\[
\begin{array}{c}
S \\
\searrow & \\
S & S \\
\searrow & \searrow \\
a & b \\
\searrow & \searrow \\
S & \varepsilon \\
\searrow & \searrow \\
\varepsilon & \varepsilon \\
\end{array}
\]

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:

\[
\begin{array}{c}
S \\
\searrow & \\
A & S \\
\searrow & \searrow \\
b & a \\
\searrow & \searrow \\
A & S \\
\searrow & \searrow \\
b & \varepsilon \\
\end{array}
\]

5. **Problem**: Design a recursive-descent (top-down) parser for the grammar from Problem 6/6.

**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.
- $\text{lex}()$ — read the next lexeme from the input, and store it in a global variable
  $\text{current}._\text{tok}$.
- $\text{expect}(\text{int token})$ — tries to read the lexeme $\text{token}$ from input. Gives an error
  message if it fails.
- $\text{consume}._\text{token}()$ — mark the current lexeme used. This is necessary because some-
times 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 $\text{lex}$ and $\text{yacc}$ tools.1
Of these, $\text{lex}$ generates a finite automaton-based lexical analyser from identifying lex-
emes that have been defined using regular expression, and $\text{yacc}$ constructs a pushdown
automaton-based parser for a given context-free grammar.

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

/* Define the alphabet */
enum TOKEN { $\text{DO}$, $\text{FOR}$, $\text{END}$, $\text{BEGIN}$, $\text{T\text{IMES}}$, $\text{OP}$, $\text{SC}$, $\text{NUMBER}$, $\text{ERROR}$ }; const char* tokens[] = { "$\text{do}$", "$\text{for}$", "$\text{end}$", "$\text{begin}$", "$\text{times}$", "$\text{a}$", "$\text{;}$", "$\text{NUMBER}$", 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 $S(\text{void})$;
void $C(\text{void})$;

int $\text{lex}(\text{void})$;
void $\text{consume}._\text{token}(\text{void})$;
void $\text{error}(\text{char }\ast\text{st})$;
void $\text{expect}(\text{int token})$;

void $C(\text{void})$
{
    $S(\text{void})$;
    $\text{lex}(\text{void})$;
    if ($\text{current}._\text{tok}$ == $\text{SC}$) {
        $\text{consume}._\text{token}(\text{void})$;
        $C(\text{void})$;
        printf("$C \rightarrow S ; C\n\text{\"});
    } else {

1Or some of their derivatives, like $\text{flex}$ or $\text{bison}$.
printf("C -> S\n");
} }

void S(void)
{
    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 != EDF && 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;
}