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 \(\text{current\_tok}\).
- \(\text{expect}(\text{int token})\) — tries to read the lexeme \(\text{token}\) from input. Gives an error message if it fails.
- \(\text{consume\_token()}\) — 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 \(\text{lex}\) and \(\text{yacc}\) tools\(^1\). Of these, \(\text{lex}\) generates a finite automaton-based lexical analyser from identifying lexemes that have been defined using regular expression, and \(\text{yacc}\) 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);
```

\(^1\)Or some of their derivatives, like \texttt{flex} or \texttt{bison}.
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)
{
    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 = getc();
} while (c != EOF && isspace(c));
if (c != EOF) ungetc(c, stdin);

/* read token */
c = getc();
while (c != EOF && c != ';' && !isspace(c) && pos < TOKEN_LEN) {
    token_text[pos++] = c;
    c = getc();
}
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 = DO; 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;
}