



$$S$$

$$\text{if } b \text{ then } S \text{ else } S$$

$$\text{if } b \text{ then } S \text{ } s$$

$$s$$

Yleensä ohjelmointikielissä halutaan **else**-lause liittää lähimpään mahdolliseen **if**-lauseeseen. Ylläolevista puista ensimmäinen vastaa tätä käytäntöä.

Määritellään kielioppi seuraavasti:

$$G = (V, \Sigma, P, S)$$

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

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

$$P = \{S \rightarrow B \mid U$$

$$B \rightarrow \text{if } b \text{ then } B \text{ else } B \mid s$$

$$U \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } B \text{ else } U\}$$

Tässä välikiehellä  $B$  saadaan johdettua vain ohjelmia, joissa kaikilla **if**-lauseilla on sekä **then**- että **else**-haarat. Välikiehellä  $U$  johdetaan sitten **if**-lauseet, joista puuttuu **else**-haara.

6. **Tehtävä:** Laadi rekursiivisesti etenevä jäsentäjä edellisten harjoitusten tehtävän 6 kieliopille.

**Vastaus:** Alla oleva C-ohjelma toteuttaa rekursiivisen jäsentäjän kieliopille:

$$C \rightarrow S \mid S; C$$

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

Tässä on yksinkertaistettu hieman edellisen laskuharjoituskerran 6. tehtävän kielioppia korvaamalla erilliset numerot terminaalilla  $n$ , joka tarkoittaa mitä tahansa numeroa.

Tärkeimmät ohjelmassa esiintyvät funktiot ovat:

- `C()`, `S()` — toteuttavat kieliopin varsinaiset säännöt
- `lex()` — lukee syötteestä seuraavan lekseemin ja tallettaa sen globaaliin muuttujaan `current_tok`.
- `expect(int token)` — yrittää lukea syötteestä lekseemin `token`. Mikäli lukeminen epäonnistuu annetaan virheilmoitus.
- `consume_token()` — merkitään tämänhetkinen lekseemi käytetyksi. Tämä (tai jokin muu vastaava funktio) tarvitaan siksi, että joissain tapauksissa täytyy syötettä lukea yksi lekseemi eteenpäin ennen kuin tiedetään, mitä sääntöä täytyy käyttää.

Käytännössä ohjelmointikielten jäsentäjät toteutetaan yleensä käyttäen *lex*- ja *yacc*-työkaluja<sup>1</sup>. Näistä *lex* muodostaa tilakonepohjaisen selaaajan, joka tunnistaa säännöllisillä lausekkeilla määritellyt lekseemit, ja *yacc* tekee pinoautomaattipohjaisen jäsentimen annetulle yhteydettömälle kieliopille.

<sup>1</sup>Tai niiden johdannaisia.

```

#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)
{
    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 = 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;
}
```