4 Tehtävä: Laadi yhteydetön kieliloppu kielelle
\[ \{w \in \{a, b\}^* \mid w\text{ssä on yhtä monta }a\text{:ta kuin }b\text{tä}\}. \]

Vastaus: Kielilopin voi laatia parillakin eri tapaa. Yksinkertaisinta on laatia moniselitteinen kieliloppi:
\[ S \rightarrow aSbS \mid bSaS \mid \varepsilon. \]

Kielilopin ensimmäinen sääntö esittää ehdon: ”Jos sana alkaa a:lla, niin jossain kohtaa sanassa esiintyy myös vastaava b. Tätä ennen ja jälkeen voi esiintyä mitä tahansa tasa-painoisia merkkijonoja.”

Esimerkiksi merkkijonolle \(abab\) on kaksi eri jäsennyspuuta:

\[
\begin{align*}
S & \rightarrow aSbS \mid bSaS \mid \varepsilon \\
A & \rightarrow aAb \mid b \\
B & \rightarrow bBa \mid a
\end{align*}
\]

Mikäli kielelle halutaan laatia yksiselitteinen kieliloppu, täytyy varmistaa, että sanan alussa oleva \(a\) liitetään ensimmäiseen mahdolliseen \(b\)-hen:
\[ S \rightarrow aAS \mid bBS \mid \varepsilon \]
\[ A \rightarrow aAb \mid b \\
B \rightarrow bBa \mid a \]

Nyt jonolla \(abab\) on vain yksi jäsennyspuu:

5. Tehtävä:

(a) Osoita, että seuraava yhteydetön kieliloppi on moniselitteinen:
\[
\begin{align*}
S & \rightarrow \text{if } b \text{ then } S \\
S & \rightarrow \text{if } b \text{ then } S \text{ else } S \\
S & \rightarrow s.
\end{align*}
\]
(b) Muodosta (a)-kohdan kielipin kanssa ekvivalentti, so. saman kielen tuottava yksiselitteinen kielipiti.

Vastaus: Yhteydetön kielipiti $G$ on moniselliteinen, mikäli on olemassa sana $w \in L(G)$ siten, että $w$:llä on ainakin kaksi erilaista jäsennyspuuta. Tehtävän kielipipalle yksinkertaisin tällainen sana on:

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

joka voidaan jäsentää lahdella tapaa:

![Diagram](attachment:image.png)

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

Määritellään kielipiti seuraavasti:

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

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

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

$$P = \{ S \to B \mid U \}$$

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

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

Tässä välilellä $B$ saadaan johdettua vain ohjelma, joissa kaikkilla if-lauseilla on selä then- että else-haarat. Välilellä $U$ johdetaan siten if-lauseet, joista puuttuu else-haara.

6. Tehtävä: Laadi rekursiivisesti etenevä jäsentäjä edellisten harjoituksen tehtävän 5 kielipipalle.

Vastaus: Alla oleva C-ohjelma toteuttaa rekursiivisen jäsentäjän kielipipalle:

$$C \to S \mid S; C$$

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

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

Tärkeimmät ohjelmaa esintyvät funktiot ovat:
• C(), S() — toteuttavat kieliopin varsinaiset säännöt
• lex() — lukee syötteestä seuraavan lekkeemin ja tallettaa sen globaaliin muuttujaan current_tok.
• expect(int token) — yrittää lukea syötteestä lekkeemin token. Mikäli lukeminen epäonnistuu annetaan virheilmoitus.
• consume_token() — merkitään tämänhetkinen lekkeemi käytetyksi. Tämä (tai jokin muu vastaava funktio) tarvitaan siksi, että joissakin tapauksissa täytyy syötettä lukea yksi lekkeemi eteenpän ennen kuin tiedetään, mitä sääntöä täytyy käyttää.

Käyttännössä ohjelmointikielten jäsentäjät toteutetaan yleensä käyttäen lex- ja yacc-työkaluja\(^1\). Näistä lex muodostaa tilakonepohjaisen selajan, joka tunnistaa säännöllisillä lausekkeilla määritellyt lekkeemit, ja yacc tekee pinoautomaattipohjaisen jäsentimen annetulle yhteydettömälle kieliopille.

#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", "\n", "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)

\(^1\)Tai niiden johdannaisia.
{  
    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 = DG; 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;
}