T-79.1001 **Syksy 2005**
Tietojenkäsittelyteorian perusteet T
Harjoitus 6
Demonstraatiotehtävien ratkaisut

4. **Tehtävä**:Modernissa WWW-sivujen kuvaamiseen käytetyssä XML-kielessä on sivujen suunnittelijan mahdollista laatia omia ns. dokumenttityyppimäärityksiä (engl. *Document Type Definition*, lyh. DTD), jotka ovat oleellisesti sivulla esitettävän tekstin tai muun datan rakennetta kuvaavia yhteydettömiä kielioppeja. Tutustu tämän XML/DTD-kuvauskielen notaatioon (esim. WWW-sivulta `http://www.rpbourret.com/xml/xmldtd.htm`), ja laadi seuraavaa XML/DTD-kuvausta vastaava yhteydetön kielioppi:

```
<!DOCTYPE Book [
   <!ELEMENT Book (Title, Chapter+)>
   <!ATTLIST Book Author CDATA #REQUIRED>
   <!ELEMENT Title (#PCDATA)>
   <!ELEMENT Chapter (#PCDATA)>
   <!ATTLIST Chapter id ID #REQUIRED>
]>
```

Vastaus: Yllä oleva DTD-kuvaus määrittelee rakenteen kirjalle. Määrittelyssä esiintyy kahdenlaisia asioita: *osia* (element) ja *attribuutteja* (attlist). Perusajatuksena on, että kirja itsessään koostuu osista, ja attribuutit puolestaan liittävät kirjan osiin ylimääräistä tietoa.

Yleisesti ottaen attribuutteja ei voida esittää puhtailla kieliopeilla, vaan niitä varten tarvitaan *attribuuttikieliopit* (ks. opetusmoniste s. 60–68). Näin ollen DTD-kuvauksesta mallinnetaan ensin vain osat, eli käytännössä ainoastaan rivit, jotka alkavat merkinnällä `!ELEMENT`.

Näistä riveistä ensimmäinen:

```
<!ELEMENT Book (Title, Chapter+)>
```

kertoo, että kirja (Book) sisältää otsikon (Title) ja listan lukuja (Chapter). Lukuja täytyy olla vähintään yksi. Seuraava rivi:

```
<!ELEMENT Title (#PCDATA)>
```

puolestaan määrittelee otsikon merkkijoukoksi (#PCDATA). Merkkijoukkoon voi kuulua periaattessa mitä tahansa tietokoneen merkkijärjestelmään kuuluvia symboleita.

Lopuksi, rivi:

```
<!ELEMENT Chapter (#PCDATA)>
```

kertoo luvun olevan taas joukko merkkejä.

Kirjan rakenteen kuvaa siis kielioppi:[1]

$$\begin{aligned} \textit{Book} &\rightarrow \textit{Title Chapters} \\ \textit{Title} &\rightarrow \textbf{data} \\ \textit{Chapters} &\rightarrow \textit{Chapter Chapters} \mid \textit{Chapter} \\ \textit{Chapter} &\rightarrow \textbf{data} \end{aligned}$$

[1] *Kursiivilla* kirjoitetut sanat ovat välikkeitä, **vahvennetut** päätemerkkejä.

XML-kielessä erotetaan dokumentin osat toisistaan käyttäen apuna `<A>` ja `</A>` koodeja. Kun nämä koodit lisätään yllä olevaan kielioppiin, saadaan tulokseksi seuraavanlainen yhteydetön kielioppi:

$$\begin{aligned} \mathit{Book} &\rightarrow \langle \mathbf{Book} \rangle\ \mathit{Title}\ \mathit{Chapters}\ \langle /\mathbf{Book} \rangle \\ \mathit{Title} &\rightarrow \langle \mathbf{Title} \rangle\ \mathbf{data}\ \langle /\mathbf{Title} \rangle \\ \mathit{Chapters} &\rightarrow \mathit{Chapter}\ \mathit{Chapters} \mid \mathit{Chapter} \\ \mathit{Chapter} &\rightarrow \langle \mathbf{Chapter} \rangle\ \mathbf{data}\ \langle /\mathbf{Chapter} \rangle \end{aligned}$$

Vaikka attribuutteja ei voidakaan täysin esittää yhteydettömällä kieliopilla, niiden syntaksi voidaan kuitenkin kuvata. XML-kielessä osan attribuutit kirjoitetaan osan aloittavan koodin sisään. Lisäämällä nämä ylläolevaan kielioppiin saadaan tulokseksi:

$$\begin{aligned} \mathit{Book} &\rightarrow \langle\ \mathbf{Book}\ \mathit{BookAttributes}\ \rangle\ \mathit{Title}\ \mathit{Chapters}\ \langle /\mathbf{Book} \rangle \\ \mathit{Title} &\rightarrow \langle \mathbf{Title} \rangle\ \mathbf{data}\ \langle /\mathbf{Title} \rangle \\ \mathit{Chapters} &\rightarrow \mathit{Chapter}\ \mathit{Chapters} \mid \mathit{Chapter} \\ \mathit{Chapter} &\rightarrow \langle\ \mathbf{Chapter}\ \mathit{ChapterAttributes}\ \rangle\ \mathbf{data}\ \langle /\mathbf{Chapter} \rangle \\ \mathit{BookAttributes} &\rightarrow \mathbf{author} = \mathbf{data} \\ \mathit{ChapterAttributes} &\rightarrow \mathbf{id} = \mathbf{data} \end{aligned}$$

Tässä on huomattava, että yhteydettömällä kieliopilla ei voida määritellä ehtoa, jonka mukaan kaikilla luvuilla on eri tunnus. Tällaisten ehtojen toteutuminen täytyy tarkistaa jollain erillisellä ohjelmakoodilla.

6. **Tehtävä**:Laadi rekursiivisesti etenevä jäsentäjä edellisten harjoitusten tehtävän 5 kieliopille.

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

$$\begin{aligned} C &\rightarrow S \mid S; C \\ S &\rightarrow a \mid \mathbf{begin}\ C\ \mathbf{end} \mid \mathbf{for}\ n\ \mathbf{times\ do}\ S \end{aligned}$$

   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[2]. Näistä *lex* muodostaa tilakonepohjaisen selaajan, joka tunnistaa säännöllisillä lausekkeilla määritellyt lekseemit, ja *yacc* tekee pinoautomaattipohjaisen jäsentimen annetulle yhteydettömälle kieliopille.

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

[2]Tai niiden johdannaisia.

```
#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;
}
```

Liite: oikealle lineaariset kieliopit

Yhteydetön kielioppi G on oikealle lineaarinen, mikäli sen kaikki säännöt ovat muotoa:

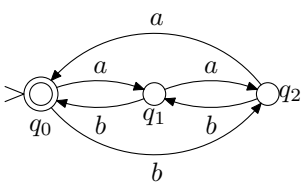
$$A \to \alpha B,$$

missä $\alpha \in \Sigma^*$ ja $B \in V \cup \{\varepsilon\}$. Toisin sanoen, säännön oikealla puolella saa esiintyä korkeintaan yksi välike, ja sen täytyy olla säännön lopussa. Esimerkiksi säännöt $A \to abC$ ja $A \to \varepsilon$ ovat oikealle lineaarisia, mutta $A \to Ba$ ja $A \to abCD$ eivät ole. Vastaavasti vasemmalle lineaarisen kieliopin kaikki säännöt ovat muotoa $A \to B\alpha$.

Lineaariset kieliopit ovat ilmaisuvoimaltaan samalla tasolla kuin äärelliset automaatit, eli niillä voidaan ilmaista kaikki säännölliset kielet.

Säännöllisen kielen tuottava oikealle lineaarinen kielioppi voidaan lukea suoraan kielen tunnistavasta automaatista. Kieliopin välikkeiksi otetaan automaatin tilat, ja kustakin automaatin siirtymästä $q_i \xrightarrow{a} q_j$ saadaan sääntö $Q_i \to aQ_j$. Lisäksi kielioppiin lisätään säännöt $Q_f \to \varepsilon$ kaikille hyväksyville lopputiloille $q_f \in F$.

Esimerkiksi automaattia:



vastaava kielioppi on:

$$
\begin{aligned}
Q_0 &\to aQ_1 \mid bQ_2 \mid \varepsilon \\
Q_1 &\to aQ_2 \mid bQ_0 \\
Q_2 &\to aQ_0 \mid bQ_1 \ .
\end{aligned}
$$