CS-C2160 Theory of Computation, Spring 2022
Problem Set 6 (Lecture weeks 6 and 7)

Solutions to Supplementary Problems

S6.1

(a) Prove that the following context-free grammar is ambiguous:

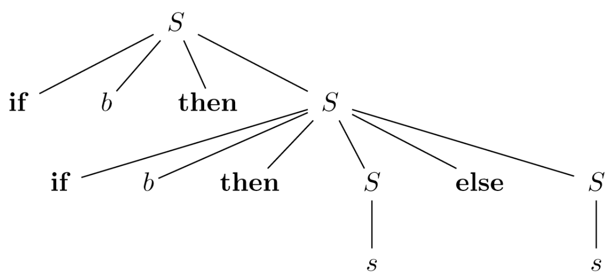
$$
\begin{aligned}
S &\rightarrow \textbf{if } b \textbf{ then } S \\
S &\rightarrow \textbf{if } b \textbf{ then } S \textbf{ else } S \\
S &\rightarrow s.
\end{aligned}
$$

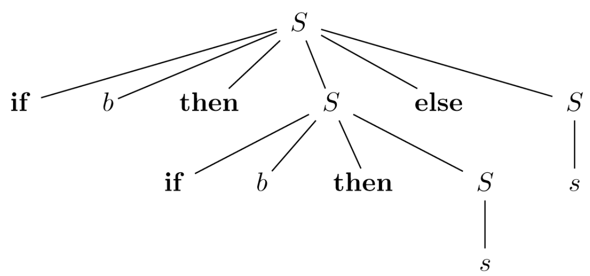
(b) Design an unambiguous grammar that is equivalent to the grammar in item (a), i.e. that generates the same language. (*Hint:* Introduce new variables B and U that generate, respectively, only "balanced" and "unbalanced" **if-then-else-**sequences.)

Solution A context-free grammar is ambiguous if there exists a word $w \in L(G)$ such that w has at least two different parse trees. The simplest word for the given grammar that has this property is:

$$\textbf{if } b \textbf{ then if } b \textbf{ then } s \textbf{ else } s.$$

Its two parse trees are:





Usually one wants to associate an **else**-branch to the closest preceding **if**-statement. In the present case the first tree above corresponds to this practice.

An unambiguous grammar for the same language can be designed follows:

$$
\begin{aligned}
&G = (V, \Sigma, P, S) \\
&V = \{S, B, U\} \\
&\Sigma = \{s, b, \textbf{if}, \textbf{then}, \textbf{else}\} \\
&P = \{S \rightarrow B \mid U \\
&\qquad B \rightarrow \textbf{if } b \textbf{ then } B \textbf{ else } B \ \mid s \\
&\qquad U \rightarrow \textbf{if } b \textbf{ then } S \mid \textbf{if } b \textbf{ then } B \textbf{ else } U\}
\end{aligned}
$$

Here the variable B is used to derive balanced **if**-statements where each **if**-clause has both **then**- and **else**-branches. The variable U derives those **if**-statements that do not have an **else**-branch.

S6.2 Design a recursive-descent (top-down) parser for the grammar of the "programming language" discussed in Supplementary Problem S5.2 of Problem Set 5.

Solution. The C-program presented below implements a top-down parser for the following grammar:

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

This grammar is a simplified version of the one in Problem S5.2. The difference is that all different (single-digit) numbers are replaced by a single new terminal symbol n.

The most important functions of the program are:

- `C()`, `S()` — implement the rules of the grammar.
- `lex()` — reads the next lexeme from the input, and stores its token type in global variable `current_tok`.
- `expect(int token)` — tries to match the lexeme pattern for *token* to the input. Gives an error message if this fails.
- `consume_token()` — marks the current lexeme used. This is necessary because sometimes we need a one-token lookahead before we know what rule of the grammar must be applied.

In practice, programming language parsers are implemented using tools such as *lex* and *yacc*.[1] Of these, *lex* generates a finite automaton -based lexical analyser for identifying lexemes that have been defined using regular expressions, and *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  { 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 lexeme corresponding to a token */
#define TOKEN_LEN 128

/* Declare parsing functions corresponding to the variables */
void S(void);
void C(void);

int lex(void);
void consume_token(void);
```

[1] Or some of their derivatives, like *flex* or *bison*.

```c
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;
```

```
}
```