T-79.148 Autumn 2003

Introduction to Theoretical Computer Science Tutorial 7

Solutions to the demonstration problems

4. **Problem**: Prove that the class of context-free languages is closed under unions, concatenations, and the Kleene star operation, i.e. if the languages  $L_1, L_2 \subseteq \Sigma^*$  are context-free, then so are the languages  $L_1 \cup L_2$ ,  $L_1L_2$  and  $L_1^*$ .

**Solution**: Let  $L_1$  and  $L_2$  be context-free languages that are defined by grammars  $G_1 = (V_1, \Sigma_1, R_1, S_1)$  and  $G_2 = (V_2, \Sigma_2, R_2, S_2)$ . In addition we require that  $(V_1 - \Sigma_1) \cap (V_2 - \Sigma_2) = \emptyset$ . That is, the grammars may not have any common nonterminals. Since the nonterminals may be renamed if necessary, this is not an essential limitation.

Union: Let S be a new nonterminal and  $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 \mid S_2\}, S\}$ . Now  $L(G) = L(G_1) \cup L(G_2) = L_1 \cup L_2$ . This holds, since the initial symbol S may derive only  $S_1$  or  $S_2$ , and they in turn may derive only strings that belong to the respective languages. (If the sets of nonterminals were not disjoint, this would not hold).

Concatenation: The language  $L_1L_2$  is defined by the following grammar: $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1S_2\}, S\}$ 

Kleene star: The language  $L_1^*$  is defined by the following grammar:  $G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \rightarrow \epsilon | SS_1\}, S)$ 

5. **Problem**: Prove that the following context-free grammar is ambiguous:

$$S \rightarrow \text{if } b \text{ then } S$$
  
 $S \rightarrow \text{if } b \text{ then } S \text{ else } S$   
 $S \rightarrow s$ .

Design an unambiguous grammar that is equivalent to the grammar, i.e. one that generates the same language.

**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:

if b then if b then s else s.

Its two parse trees are:

Sif then if bthen Selse SSif bthen Selse Sif bSthen

Usually we want to associate an **else**-branch to the closest preceding **if**-statement. In this case the former tree corresponds to this practice.

We define a grammar G as follows:

```
\begin{split} G &= (V, \Sigma, P, S) \\ V &= \{S, B, U, s, b, \mathbf{if}, \mathbf{then}, \mathbf{else} \} \\ \Sigma &= \{s, b, \mathbf{if}, \mathbf{then}, \mathbf{else} \} \\ P &= \{S \rightarrow B \mid U \\ B \rightarrow \mathbf{if} \ b \ \mathbf{then} \ B \ \mathbf{else} \ B \ \mid s \\ U \rightarrow \mathbf{if} \ b \ \mathbf{then} \ S \mid \mathbf{if} \ b \ \mathbf{then} \ B \ \mathbf{else} \ U \} \end{split}
```

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

6. **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:

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

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.
- lex() read the next lexeme from the input, and store it in a global variable current\_tok.
- expect(int token) tries to read the lexeme token from input. Gives an error message if it fails.
- 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 lex and yacc tools<sup>1</sup>. Of these, lex generates a finite automaton-based lexical analyser from identifying lexemes that have been defined using regular expression, and yacc constructs a pushdown automaton-based parser for a given context-free grammar.

<sup>&</sup>lt;sup>1</sup>Or some of their derivatives, like *flex* or *bison*.

```
/* 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 \Rightarrow 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(D0);
    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++) {</pre>
      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;
```