Tik-79.148 **Spring 2001**
Introduction to Theoretical Computer Science
Tutorial 6
Solutions to Demonstration Exercises

Demonstration Exercises:

4. The allowed regular expressions of the alphabet $\Sigma = \{a, b, (,), \cup, ^*, \emptyset\}$ can be generated using the grammar $G = (V, \Sigma, R, S)$,

$$
\begin{aligned}
V =& \Sigma \cup \{S, T, F\} \\
R =& \{S \to T \cup S, S \to T \\
& T \to F, T \to F^*, \\
& T \to FT, T \to F^*T \\
& F \to \emptyset, F \to a, \\
& F \to b, F \to (S)\}
\end{aligned}
$$

E.g. the regular expression $(a \cup bb)^*b$ is derived as $S \to T \to F^*T \to (S)^*T \to (T \cup S)^*T \to (F \cup S)^*T \to (a \cup S)^*T \to (a \cup T)^*T \to (a \cup FT)^*T \to (a \cup bT)^*T \to (a \cup bF)^*T \to (a \cup bb)^*T \to (a \cup bb)^*F \to (a \cup bb)^*b$

Nontrivial formal languages are typically infinite and it is thus not possible to represent them by exhaustive enumeration. Each family in the formal hierarchy of languages is characterised by the way of how its members can be concisely represented. For instance, every regular language can be described using a regular expression and a similar connection applies between context-free languages and context-free grammars.

The point in the example above is that the representation language for regular languages, i.e. regular expressions, is actually a context-free language and can thus be generated by creating the appropriate context-free grammar.

5. a) The context-free grammar corresponding to the language $L = \{a^m b^n \mid m \geq n\}$ is:

$$
\begin{aligned}
\Sigma =& \{a, b\} \\
V =& \{a, b, S, A, B\} \\
R =& \{S \to ASB, S \to e \\
& A \to Aa, A \to a, \\
& B \to b, B \to e\}
\end{aligned}
$$

b) The context-free grammar corresponding to the language $L = \{uawb \mid u, w \in \{a, b\}^*, |u| = |w|\}$ is:

$$
\begin{aligned}
\Sigma =& \{a, b\} \\
V =& \{a, b, S, T\} \\
R =& \{S \to Tb, T \to aTa, \\
& T \to aTb, T \to bTb \\
& T \to bTa, T \to a\}
\end{aligned}
$$

6. (*applying*)

Almost every modern programming language is based on a context-free grammar. However, the grammar describes only syntactically correct programs, since many programming concepts require information about the context. Among these are, e.g., checking whether a variable was correctly declared before its use and type checking. In practice, a program is parsed according to a grammar to a parse tree from which the aforementioned concepts are verified.

The addition and subtraction in the exercise can be implemented using the following simple grammar:

$$
\begin{aligned}
&G = (V, \Sigma, R, S)\\
&\Sigma = \{number, +, -, (,)\}\\
&R = \{S \to number, S \to S + S, S \to S - S, S \to (S)\}
\end{aligned}
$$

Above, each integer is abstracted to a terminal symbol *number*. The simplest way of implementing the parser is to make it recursive so that each nonterminal has a corresponding function. In the grammar above, only one function is needed, let us call it `expr`. The body of the function reads the input until it is clear, which rule to apply. As pseudo code the function looks like this:

```
integer expr()
  integer value

  If the rule is  S -> number,
     value = number()

  If the rule is S -> S + S,
     value = expr() + expr()

  If the rule is S -> S - S
     value = expr() - expr()

  If the rule is S -> ( S )
     read '('
     value = expr();
     read ')'

  return value
```

Below is a C implementation of a simple parser for the grammar.

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

```c
/* Read the next input symbol */
int get_token(char expect);

int number(void);
int expr(void);

int current_token;

void error(char *st)
{
  printf("%s\n", st);
  exit(0);
}

int number(void)
{
  int num;
  ungetc(current_token, stdin);
  scanf("%d", &num);
  return num;
}

int get_token(char expect)
{
  int ch;
  do {
    ch = getchar();
  } while (ch == ' '); /* skip spaces in input */
  return ch;
}

int expr(void)
{
  int left_value = 0;
  int right_value = 0;
  int result = 0;

  current_token = get_token(NONE);

  if (isdigit(current_token)) {
    left_value = number();
    current_token = get_token(NONE);
  } else if (current_token == '(') {
    left_value = expr();
    if (current_token != ')') {
      error("parenthesis error");
    }
  } else {
    error("expression has to begin with a number or '('");
  }
```

```
  switch (current_token) {
  case '\n':
    result = left_value;
    break;
  case '+':
    right_value = expr();
    result = left_value + right_value ;
    break;
  case '-':
    right_value = expr();
    result = left_value - right_value ;
    break;
  case ')':
    result = left_value;
    break;
  default:
    error("invalid character in an expression");
  }
  return result;
}

int main(void)
{
  int value = 0;

  value = expr();
  printf("Result: %d\n", value);
}
```