Informatik I für D-ITET Musterlösung 8 WS 04/05

URL: http://www.ti.inf.ethz.ch/ew/courses/inf1-ITET/

Aufgabe 1

Programm: calc.C

```
// Programm: calc.C
// Parser fuer arithmetische Ausdruecke.

// benutzt folgende kontextfreie Grammatik:
//   add_expr:        mult_expr add_expr_aux
//   add_expr_aux:    '+' mult_expr add_expr_aux
//                    '-' mult_expr add_expr_aux
//                    epsilon
//
//   mult_expr:       pm_expr mult_expr_aux
//   mult_expr_aux:   '*' pm_expr mult_expr_aux
//                    '/' pm_expr mult_expr_aux
//                    epsilon
//
//   pm_expr:         '(' add_expr ')'
//                    number
//
//   number:          sign digit digits
//
//   digits:          digit digits
//                    epsilon
//
//   sign:            '+'|'-'|epsilon
//
//   digit:           '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

#include <iostream>
#include <string>
#include <cctype>
#include <stdexcept>

// Parse-Functionen fuer die Nichtterminale:
// -----------------------------------------

// Zu jedem Nichtterminal N der Grammatik gibt es genau eine Funktion
// desselben Namens, die folgende Pre- und Postconditions hat:
//
// PRE: [b,e) ist ein gueltiger range.
// POST: Wenn sich ein Praefix P=[b,f) der durch den range [b,e)
// beschriebenen Zeichenfolge aus dem Nichtterminal N ableiten laesst,
// so ist b==f und Rueckgabewert ist der numerische Wert des durch P
// beschriebenen arithmetischen Ausdrucks. Andernfalls wirft die
// Funktion eine exception vom Typ std::invalid_argument.

// Zahlentyp fuer alle Berechnungen
typedef int NT;

// Iteratortyp fuer alle Funktionen
typedef std::string::const_iterator SSCI;
```

```
NT digit(SSCI& b, SSCI e)
// PRE: b ist dereferenzierbar.
{
  return *(b++) - '0';
}

NT digits(SSCI& b, SSCI e, const NT& left_op)
{
  if (b == e || !std::isdigit(*b))
    return left_op;
  NT val = left_op * 10 + digit(b, e);
  return digits(b, e, val);
}

NT sign(SSCI& b, SSCI e)
// POST: Rueckgabewert -1, falls Vorzeichen negativ;
//   +1, falls Vorzeichen positiv oder kein Vorzeichen.
{
  if (b == e) return 1;
  if (*b == '-') {
    ++b;
    return -1;
  }
  if (*b == '+') ++b;
  return 1;
}

NT number(SSCI& b, SSCI e)
{
  NT sgn = sign(b, e);
  if (b == e || !std::isdigit(*b))
    throw std::invalid_argument("digit expected.");
  NT val = digit(b, e);
  return digits(b, e, sgn * val);
}

// Muss hier deklariert werden, da in pm_expr verwendet.
NT add_expr(SSCI& b, SSCI e);

NT pm_expr(SSCI& b, SSCI e)
{
  if (b != e && *b == '(') {
    NT value = add_expr(++b, e);
    if (b == e || *b != ')')
      throw std::invalid_argument("')' expected.");
    ++b;
    return value;
  }
  return number(b, e);
}

NT mult_expr_aux(SSCI& b, SSCI e, const NT& left_op)
{
  if (b == e) return left_op;
  if (*b == '*') {
    NT right_op = pm_expr(++b, e);
    return mult_expr_aux(b, e, left_op * right_op);
```

```
  }
  if (*b == '/') {
    NT right_op = pm_expr(++b, e);
    return mult_expr_aux(b, e, left_op / right_op);
  }
  return left_op;
}

NT mult_expr(SSCI& b, SSCI e)
{
  NT value = pm_expr(b, e);
  return mult_expr_aux(b, e, value);
}

NT add_expr_aux(SSCI& b, SSCI e, const NT& left_op)
{
  if (b == e) return left_op;
  if (*b == '+') {
    NT right_op = mult_expr(++b, e);
    return add_expr_aux(b, e, left_op + right_op);
  }
  if (*b == '-') {
    NT right_op = mult_expr(++b, e);
    return add_expr_aux(b, e, left_op - right_op);
  }
  return left_op;
}

NT add_expr(SSCI& b, SSCI e)
{
  NT value = mult_expr(b, e);
  return add_expr_aux(b, e, value);
}

int main()
{
  std::cout << "Eingabe: ";
  std::string input;
  std::cin >> input;

  SSCI beg = input.begin();
  SSCI end = input.end();
  SSCI cur = beg;
  try {
    std::cout << add_expr(cur, end) << std::endl;
    if (cur != end)
      throw std::invalid_argument("Unexpected character.");
  } catch (std::invalid_argument err) {
    std::cerr << "Parse error: " << err.what() << "\n"
      << std::string(beg, cur)
      << "|-here->"
      << std::string(cur, end) << std::endl;
    return 1;
  }
  return 0;
}
```

Aufgabe 2

a) $\underline{S}$: Tbb

T : aTbbb | ε

b) $\underline{S}$: aSc | T

T : bTc | ε

c) $\underline{S}$: aSbS | bSaS | cS | ε

Da in jeder Produktion die Anzahl von a oder b Symbolen gleich ist, besteht offensichtlich jedes aus der Grammatik abgeleitete Word aus gleich vielen a wie b.

Umgekehrt betrachte ein Wort $w \in \Sigma^*$, das gleich viele a wie b enthält. Wir zeigen per Induktion nach $|w|$, das w aus der Grammatik abgeleitet werden kann.

Für $|w| = 0$, d.h. $w = \varepsilon$, ist die Behauptung aufgrund der Produktion $S \to \varepsilon$ wahr.

Wenn $w = cw'$, so verwende die Produktion $S \to cS$ und leite w' induktiv ab. (Offenbar enthält auch w' gleich viele a wie b.)

Falls $w = aw'$, so verwende die Produktion $S \to aSbS$. Da w' genau ein b mehr enthält als a, muss es eine Zerlegung $w' = ubv$ geben, so dass sowohl u als auch v gleich viele a wie b enthalten. (Durchlaufe w' von links nach rechts und zähle die Anzahl von a und b im durchlaufenen Präfix. Am Anfang sind beide Zahlen gleich. Da aber mehr b als a in w' sind, muss an irgendeinem Punkt ein b auftreten, so dass an diesem Punkt genau ein b mehr als a im gelesenen Präfix auftritt. Dieses b definiert die behauptete Zerlegung $w' = ubv$.) Leite u und v induktiv ab.

Der Fall $w = bw'$ ist symmetrisch zum Fall $w = aw'$.

d) $\underline{S}$: D | AE

A : aA | ε

B : bB | ε

C : cC | ε

D : aDc | aAB | BcC

E : bEc | bB | cC

Es gilt nicht $i = j = k$ genau dann, wenn $i \neq k$ oder $j \neq k$. Der Fall $i \neq k$ entspricht genau der Produktion $S \to D$, denn $L(D) = a^{n+1}a^*b^*c^n \mid a^nb^*c^*c^{n+1}$. Der Fall $j \neq k$ entspricht genau der Produktion $S \to AE$, denn $L(E) = b^{n+1}b^*c^n \mid b^nc^*c^{n+1}$.