

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

Aufgabe 1 (8 Punkte)

Gegeben ist folgende kontextfreie Grammatik für ganzzahlige arithmetische Ausdrücke.

TERMINALE.

$+, -, *, /, (,),$
 $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 .$

NICHTTERMINALE.

add-expr, *add-expr-aux*, *mult-expr*,
mult-expr-aux, *pm-expr*, *digits*, *digit*.

PRODUKTIONEN.

add-expr:
mult-expr add-expr-aux

add-expr-aux:
 $+ \text{ mult-expr add-expr-aux}$
 $- \text{ mult-expr add-expr-aux}$
 ϵ

mult-expr:
pm-expr mult-expr-aux

mult-expr-aux:
 $* \text{ pm-expr mult-expr-aux}$
 $/ \text{ pm-expr mult-expr-aux}$
 ϵ

pm-expr:
number
 (add-expr)

number:
sign digit digits

digits:
digit digits
 ϵ

sign: ONE OF
 $+ - \epsilon$

digit: ONE OF
 $0 1 2 3 4 5 6 7 8 9$

Schreiben Sie ein Programm `calc.C`, das einen Taschenrechner simuliert. Erstellen Sie für jedes Nichtterminal N der Grammatik eine Funktion `int N(SSCI& b, SSCI e)`, welche versucht, ein Präfix der Zeichenfolge im Range $[b, e)$ aus N abzuleiten. (Hierbei steht SSCI für `std::string::const_iterator`.)

Das heisst, die Funktion N hat folgende Postcondition: Kann ein Präfix $[b, c)$ von $[b, e)$ aus N abgeleitet werden, so ist $b = c$ und Rückgabewert ist der Wert des durch $[b, c)$ beschriebenen arithmetischen Ausdrucks. Andernfalls wirft N eine exception vom Typ `std::invalid_argument`.

In den Funktionen von *add-expr-aux*, *mult-expr-aux* und *digits* werden die arithmetischen Operationen tatsächlich ausgerechnet. Deshalb erhalten die entsprechenden Funktionen einen dritten Parameter vom Typ `int`, der den linken Operanden angibt.

Nehmen Sie das Programm `calc-simple.C` aus der Vorlesung als Ausgangspunkt.

Abgabe: Aufgabe 1: bis 20. Dezember 2004, 12.00 Uhr, per Email.

Aufgabe 2: am 20. Dezember 2004, vor den Übungen, schriftlich.

Aufgabe 2 (4 Punkte)

Geben Sie kontextfreie Grammatiken für folgende Sprachen über dem Alphabet $\Sigma = \{a, b, c\}$ an. In den Aufgabenteilen c) und d) begründen Sie, warum Ihre Grammatik korrekt ist.

- Alle Wörter der Form $a^i b^{3i+2}$, $i \geq 0$.
- Alle Wörter der Form $a^i b^j c^{i+j}$, $i, j \geq 0$.
- Alle Wörter, die gleich viele a wie b enthalten.
- Alle Wörter der Form $a^i b^j c^k$, $i, j, k \geq 0$, für die nicht $i = j = k$ gilt.

Informatik I:**Material aus der Vorlesung****Programm: calc-simple.C**

```
// Programm: calc-simple.C
// Parser fuer einfache arithmetische Ausdruecke.

// Benutzt folgende kontextfreie Grammatik:
// add_expr:   number add_expr_aux
// add_expr_aux: '+' number add_expr_aux
//            '-' number add_expr_aux
//            epsilon
//            number:   digit digits
//            digits:  digit digits
//            epsilon
//            digit:   '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

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

// Parse-Funktionen 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;

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

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

NT add_expr(SSCI& b, SSCI e)
{
    NT value = number(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;
}
```