UNIVERSITÄT HEIDELBERG
Zukunft. Seit 1386.
IWR Interdisciplinary Center for Scientific Computing

IPI: Einführung in die Praktische Informatik
Wintersemester 2018
Prof. Björn Ommer
ommer@uni-heidelberg.de

Übungsblatt 3: Funktionale Programmierung

Abgabe am *14.11.2018*, 13:00.

Anmerkungen:

- Bei der Lösung der Programmieraufgaben dürfen ausschliesslich die C++-Konstrukte verwendet werden, die bisher in der Vorlesung behandelt wurden, d. h. keine Schleifen, Variablenzuweisungen, usw. sondern ausschliesslich Funktionsdefinitionen, rekursive Aufrufe und Funktionen des auf Moodle (im Verzeichnis 'C++ Quellcode') zur Verfügung gestellten Headers `fcpp.hh`, wie z. B. `cond`.

- Die Lösungen von Programmieraufgaben müssen so abgegeben werden, dass sie *ohne* das weitere Hinzufügen von Dateien oder das Einstellen von Pfaden *lauffähig* sind.

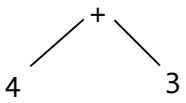
- Zur Lösung der Aufgaben 3 und 4 müssen Sie überprüfen können, ob eine Zahl b Teiler von a ist. Der einfachste Test ist, zu überprüfen, ob der Rest der ganzzahligen Division a/b gleich Null ist. Diesen Rest erhält man mittels der *modulo*-Funktion. So ist z. B.:

$$15 \bmod 13 = 2$$
$$30 \bmod 15 = 0$$
$$345 \bmod 13 = 7$$

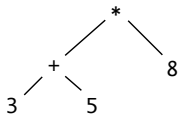
  In C/C++ schreibt man die Modulo-Funktion als '`%`' also z. B.

```
bool istteilbar(int a, int b) {
  return cond( a % b == 0, true, false );
}
```

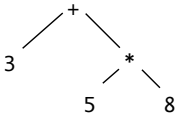
Aufgabe 1: Darstellung von binären Bäumen **(5P)**



Baum 1



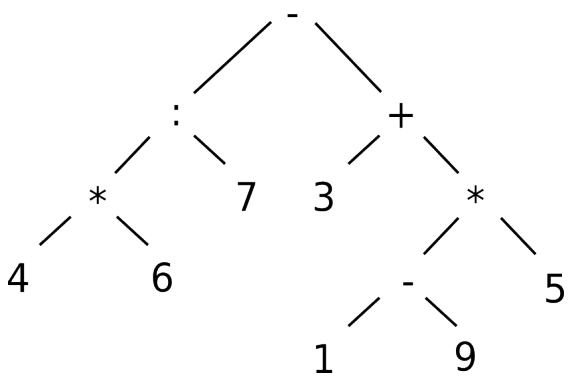
Baum 2



Baum 3

Der arithmetische Ausdruck $4 + 3$ kann wie Baum 1 als binärer Baum dargestellt werden; umgekehrt entspricht jedem binären Baum ein Ausdruck. Für den Ausdruck können unterschiedliche Darstellungsformen gewählt werden, wie zum Beispiel die aus der Vorlesung bekannte Schreibweise $+(4, 3)$, die man auch zu + 4 3 abkürzen kann. Die Reihenfolge ist bei der Auswertung eines Ausdrucks wichtig und entspricht unterschiedlichen Bäumen: Baum 2 entspricht (3+5)∗8 und Baum 3 enspricht $3 + 5 * 8$.
Um von einem binären Baum zu dem ihm entsprechenden Ausdruck zu kommen, muss man den Baum durchlaufen und dabei jeden Knoten genau einmal besuchen. Die Ordnung dieses Durchlaufs wird (wie der Baum) rekursiv definiert. In der Praxis sind vor allem drei Durchlaufordnungen wichtig:

Preorder	**Inorder**	**Postorder**
Stelle die Wurzel dar	Mache eine Klammer auf	Mache eine Klammer auf
Mache eine Klammer auf	Stelle den linken Teilbaum dar (in Inorder-Reihenfolge)	Stelle den linken Teilbaum dar (in Postorder-Reihenfolge)
Stelle den linken Teilbaum dar (in Preorder-Reihenfolge)	Stelle die Wurzel dar	Stelle den rechten Teilbaum dar (in Postorder-Reihenfolge)
Stelle den rechten Teilbaum dar (in Preorder-Reihenfolge)	Stelle den rechten Teilbaum dar (in Inorder-Reihenfolge)	Mache eine Klammer zu
Mache eine Klammer zu	Mache eine Klammer zu	Stelle die Wurzel dar
+(4 3)	(4 + 3)	(4 3) +



Baum 4

(a) Geben Sie die Darstellung von Baum 4 in Preorder-, Postorder- und Inorder-Reihenfolge an. Die Preorder- und Postorder-Reihenfolgen sind auch als *polnische Notation* bzw. *umgekehrte polnische Notation* bekannt, nach dem polnischen Logiker Łukasiewicz.

(b) Welche der drei beschriebenen Darstellungen sind auch ohne Klammerung oder Operator-Rangfolgen („Punkt vor Strich“) eindeutig?

(c) Eine weitere Möglichkeit zur Darstellung ist die Reihenfolge *Wurzel – Rechts – Links*. Steht diese Reihenfolge in einem einfachen Zusammenhang mit einer der obigen Reihenfolgen?

Aufgabe 2: Lineare Gleichungssysteme in EBNF **(5P)**

Aus der Schule oder aus der Linearen Algebra kennen Sie lineare Gleichungssysteme. Ein einfaches Beispiel eines linearen Gleichungssystems ist zum Beispiel:

$$\begin{aligned} 3x_0 + 3x_1 &= 8 \\ 5x_0 + 7x_1 - 3x_2 &= 5 \\ 4x_1 - 3x_2 &= 11 \end{aligned}$$

In der Vorlesung haben Sie die erweiterte Backus-Naur-Form (EBNF) kennengelernt, um formale Sprachen zu beschreiben. Beschreiben Sie die Menge der linearen Gleichungssysteme mit beliebiger Anzahl von Variablen $x_0, x_1, \ldots$ und einer beliebigen Anzahl von Gleichungen in EBNF. Verwenden sie hierzu die Menge

$$T = \{0, \ldots, 9, {}_0, \ldots, {}_9, +, -, =, x, \backslash n\}$$

als Menge von Terminalsymbolen. Dabei bezeichnet $\backslash n$ in Anlehnung an die Programmiersprache C einen Zeilenumbruch. Beachten Sie außerdem, dass die Addition kommutativ ist und die Summanden dementsprechend nicht wie in obigem Beispiel nach Variablen sortiert auftreten müssen. Sie können sich sowohl bei den Koeffizienten als auch bei der rechten Seite auf ganze Zahlen beschränken.
Überlegen Sie sich, ob es möglich ist, mit Ihrer EBNF-Grammatik sicherzustellen, dass die Anzahl der Variablen und die Anzahl der Gleichungen gleich sind.

Aufgabe 3: Vollkommene Zahlen **(5P)**

Eine Zahl wird *vollkommen* genannt, wenn sie gleich der Summe ihrer Teiler, sich selbst ausgenommen, ist. Zum Beispiel sind die Teiler von 6: 1, 2, 3 und 6. Die Summe der Teiler ausgenommen 6 selbst ist $1 + 2 + 3 = 6$. Also ist 6 eine vollkommene Zahl (genau genommen die kleinste). Weitere vollkommenen Zahlen sind 28 ($1 + 2 + 4 + 7 + 14 = 28$) , 496, 8128 und 33550336.
Schreiben Sie eine Funktion `bool vollkommen(int zahl)`, die `true` zurückgibt, wenn `zahl` vollkommen ist, sonst `false`.

Aufgabe 4: Schnelle Potenzberechnung **(5P)**

Bei der Berechnung von Potenzen kann man sich einige Multiplikationen sparen. Zum Beispiel kann man

$$x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$$

in mehreren Schritten rechnen:

$$\begin{aligned} x^2 &= x \cdot x \\ x^4 &= x^2 \cdot x^2 \\ x^8 &= x^4 \cdot x^4 \end{aligned}$$

und benötigt dann nur 3 statt 7 Multiplikationen. Allgemein lässt sich das folgendermaßen rekursiv ausdrücken:

$$x^n = \begin{cases} (x^{n/2})^2 & \text{falls } n \text{ gerade} \\ x \cdot x^{n-1} & \text{falls } n \text{ ungerade} \\ x & \text{falls } n = 1 \end{cases}$$

(a) Schreiben Sie eine Funktion `int potenz(int x, int exp)`, die auf diese Weise schnell Potenzen mit `exp` ≥ 1 berechnet.

Hinweis: Benutzen Sie die in der Vorlesung definierte Funktion `int quadrat(int x)`!

(b) Bei der Auswertung von Funktionen in C++ werden zuerst die Argumente der Funktion ausgewertet (falls diese einen zusammengesetzten Ausdruck bilden), bevor der Rumpf der Funktion abgearbeitet wird. Wenn zum Beispiel `funktion(2*3+5)` aufgerufen wird, so wird zuerst $2 * 3 + 5$ ausgewertet und dann 11 an die Funktion übergeben. Dies ist die *applikative Reihenfolge*.

Bei der *normalen Reihenfolge* werden Argumente einer Funktion erst ausgewertet, wenn sie gebraucht werden. Das bedeutet, dass im Funktionenrumpf jeder formale Parameter durch den Ausdruck des aktuellen Parameters ersetzt wird. Wenn im obigen Beispiel `funktion(2*3+5)` aufgerufen wird und die Deklaration der Funktion `funktion(int x)` ist, so wird mit der Auswertung des Rumpfes begonnen und jedes Vorkommen von `x` durch $(2 * 3 + 5)$ ersetzt, bevor $(2 * 3 + 5)$ dann ausgewertet wird.

Welcher Ausdruck entsteht, wenn `potenz(5,4)` applikativ, welcher, wenn die Funktion normal ausgewertet wird? Gibt es einen Unterschied?