Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting
gregor.snelting@kit.edu
Prof. Dr. Ralf H. Reussner
reussner@kit.edu

# Klausur Programmierparadigmen — Beispiellösung

WS2016/17, 06. April 2017, 14:00 – 16:00 Uhr

**Zugelassene Hilfsmittel:** Papierbasierte Quellen (Vorlesungsfolien, Übungsblätter, eigene Aufzeichnungen, Bücher, ...)
Die Verwendung von elektronischen Geräten ist verboten.

**Bearbeitungszeit:** 120 min

**Aufgabe 1** (Haskell: Huffman-Codierung) [22 Punkte]

Huffman-Codes sind ein einfaches Verfahren zur Datenkompression. Sie encodieren einen Eingabe-Zeichenstrom in einen Strom von Bits. Ein Huffman-Code wird durch einen binären Baum (den Huffman-Baum) folgendermaßen definiert:

Der linken (ersten) ausgehenden Kante eines Knotens wird das Bit 0 zugeordnet, der rechten Kante das Bit 1. In den Blättern des Baumes stehen die möglichen Eingabezeichen. Die Codierung eines Zeichens ergibt sich aus den Bits entlang des Pfades von der Wurzel zum zugehörigen Blatt (siehe Beispiel). Diese Pfade können unterschiedlich lang sein.

Für diese Aufgabe nehmen wir **`Char`** als Typen für Eingabezeichen an. Außerdem sind die folgenden Datentypen gegeben:

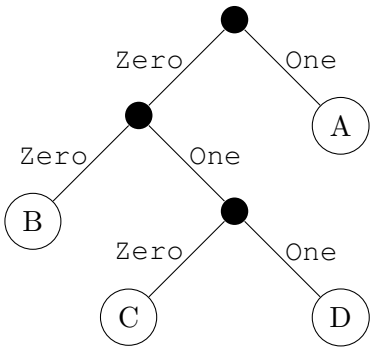
```
data Bit = Zero | One

data HuffmanTree = Node HuffmanTree HuffmanTree
                 | Leaf Char
```

**Beispiel:**

Der nebenstehende Huffman-Baum repräsentiert folgende Codierung:

```
'A'  ↔  [One]
'B'  ↔  [Zero, Zero]
'C'  ↔  [Zero, One, Zero]
'D'  ↔  [Zero, One, One]
```




Als Haskell-Datum sieht der Baum so aus:

```
Node (Node (Leaf 'B')
           (Node (Leaf 'C')
                 (Leaf 'D')))
     (Leaf 'A')
```

(a) Implementieren Sie eine Funktion [7 Punkte]

```
decode :: HuffmanTree -> [Bit] -> [Char]
```

die eine potenziell unendliche Liste von Bits einliest, entsprechend des gegebenen Huffman-Baums decodiert und die Liste der decodierten Zeichen zurückgibt. Wenn die Eingabe mitten in einem Zeichen endet, ignorieren Sie die überzähligen Bits.

In den folgenden Teilaufgaben dürfen Sie den Fall ignorieren, dass ein Eingabezeichen nicht oder mehrfach im Huffman-Baum vorkommt.

(b) Nehmen Sie an, dass Sie eine Funktion [3 Punkte]

```
encodeOne :: HuffmanTree -> Char -> [Bit]
```

gegeben haben, die ein einzelnes Zeichen entsprechend des gegebenen Huffman-Baums codiert. Implementieren Sie mit ihrer Hilfe eine Funktion

```
encode :: HuffmanTree -> [Char] -> [Bit]
```

die eine potenziell unendliche Liste von Zeichen im Huffman-Code codiert.

(c) Implementieren Sie nun die Funktion `encodeOne` wie oben beschrieben. [12 Punkte]

**Hinweise:**

- Zwei mögliche Ansätze um die Funktion `encodeOne` zu implementieren sind:
  - Konvertieren Sie den Huffman-Baum in eine Liste von Paaren `(`**`Char,`** `[Bit])` und finden Sie in dieser Liste das zu encodierende Zeichen.
  - Verwenden Sie den **`Maybe`**-Typen, um anzugeben, ob in einem Unterbaum das gesuchte Zeichen gefunden wurde, und benutzen Sie den `choose`-Kombinator um die Ergebnisse zusammenzuführen. Dieser ist wie folgt definiert:

```
choose Nothing  y = y
choose (Just x) _ = Just x
```

- Sie können einen Akkumulator benutzen um während der Baum-Traversierung das Codewort aufzubauen.

**Beispiellösung**:

(a)
```
decode :: HuffmanTree -> [Bit] -> [Char]
decode tree bits = iter tree bits
    where
      iter (Leaf c)     bs            = c : iter tree bs
      iter _            []            = []
      iter (Node t1 _ ) (Zero : rest) = iter t1 rest
      iter (Node _  t2) (One  : rest) = iter t2 rest
```

(b)
```
encode :: HuffmanTree -> [Char] -> [Bit]
encode tree = concatMap (encodeOne tree)
```

(c)
```
encodeOne :: HuffmanTree -> Char -> [Bit]
encodeOne tree char = findChar (treeToList tree [])
    where
      treeToList (Leaf c)     prefix = [(c, prefix)]
      treeToList (Node t1 t2) prefix = (treeToList t1 (prefix ++ [Zero])) ++
                                       (treeToList t2 (prefix ++ [One]))

      findChar = snd . head . filter (\(c, _) -> c == char)

choose :: Maybe a -> Maybe a -> Maybe a
choose Nothing  x = x
choose (Just x) _ = Just x

encodeOne' :: HuffmanTree -> Char -> [Bit]
encodeOne' tree char = fromJust (iter tree [])
    where
      iter (Leaf c) prefix | c == char = Just prefix
                           | otherwise = Nothing
      iter (Node t1 t2) prefix = (iter t1 (prefix ++ [Zero])) `choose`
                                 (iter t2 (prefix ++ [One]))
```

**Aufgabe 2** (Prolog, Nullableitungen) [18 Punkte]

Gegeben sei eine kontextfreie Grammatik $G = (\Sigma, V, P, S)$. Eine Sequenz $\alpha \in (\Sigma \cup V)^*$ von Terminalen und Nichtterminalen heißt *nullable*, wenn sich aus $\alpha$ die leere Sequenz $\epsilon$ ableiten lässt ($\alpha \Rightarrow^* \epsilon$).

In Prolog notieren wir: Nichtterminale A als Ausdrücke `nonterm(A)`, Terminale x als Ausdrücke `term(x)`, Sequenzen $\alpha$ als Liste solcher Ausdrücke, und Mengen $P$ von Produktionsregeln $\mathsf{A} \to \alpha$ als Liste von Paaren `(A,`$\alpha$`)`. Z.B. notieren wir die Produktionsregeln $P$ der Grammatik

$$
\begin{array}{lcl}
S & \to & T\ S' \\
S' & \to & \underline{+}\ S \mid \varepsilon \\
T & \to & \underline{\texttt{value}} \mid \underline{(}\ S\ \underline{)}
\end{array}
$$

als

```
[  (nonterm(s),  [nonterm(t), nonterm(s_)]),
   (nonterm(s_), [term(+), nonterm(s)]),
   (nonterm(s_), []),
   (nonterm(t), [term(value)]),
   (nonterm(t), [term(lp), nonterm(s), term(rp)])
]
```

(a) Definieren Sie ein Prädikat `derive(P, `$\alpha$`, `$\beta$`)` das genau dann erfüllt ist, wenn $\beta$ in *einem* Schritt aus $\alpha$ ableitbar ist ($\alpha \Rightarrow \beta$). Insbesondere sollen alle solche $\beta$ bei Reerfüllung generiert werden. [7 Punkte]

**Hinweis:** Verwenden Sie `member`, `append`.

(b) Definieren Sie ein Prädikat `nullable(P, `$\alpha$`)` das genau dann erfüllt ist, wenn $\alpha$ *nullable* ist. [11 Punkte]

**Hinweis:** Die Aufgabe lässt sich *nicht* lösen, indem lediglich `derive` wiederholt angewandt wird. Um nachzuweisen, dass eine Sequenz *nullable* ist, merken wir uns (um nicht unendlichlange Beweise zu suchen), welche Nonterminale nicht mehr untersucht werden dürfen. Es gilt:

- Die leere Sequenz $\epsilon$ ist *nullable*.
- Eine Sequenz $\mathsf{A}\beta$ ist *nullable* falls
  - Nichtterminal A noch untersucht werden darf,
  - $\mathsf{A} \to \alpha$ eine Produktion in $P$ ist,
  - ohne nochmals A zu untersuchen nachweisbar ist, dass $\alpha$ *nullable* ist, und
  - nachweisbar ist, dass $\beta$ *nullable* ist.

Definieren Sie also ein passendes drei-stelliges Hilfsprädikat. Verwenden Sie **`not`**, `member`.

**Beispiellösung**:

(a)
```
derive(P, [nonterm(A)|Beta], AlphaBeta) :-
    member((nonterm(A),Alpha), P),
    append(Alpha, Beta, AlphaBeta).
derive(P, [Y|Alpha], [Y|AlphaNew]) :-
    derive(P, Alpha, AlphaNew).
```

(b)
```
nullable(P, Alpha) :- nullableIgnoring(P, Alpha, []).

nullableIgnoring(_, [], _).
nullableIgnoring(P, [nonterm(A)|Beta], N) :-
    not(member(A,N)),
    member((nonterm(A),Alpha), P),
    nullableIgnoring(P,Alpha, [A|N]),
    nullableIgnoring(P,Beta,     N ).
```

**Aufgabe 3** (Typsysteme) [20 Punkte]

Für diese Aufgabe nehmen wir an, dass die Konstante `17` den Typ **`int`** hat und die Konstante `true` den Typ **`bool`**.

(a) Gegeben sind die folgenden Typgleichungssysteme: [6 Punkte]

$$C_1 = \{\ \alpha_9 = \alpha_{10} \to \alpha_8, \alpha_9 = \alpha_4, \alpha_{10} = \textbf{bool}\ \}$$
$$C_2 = \{\ \alpha_{12} = \alpha_{13} \to \alpha_{11}, \alpha_{12} = \alpha_4, \alpha_{13} = \textbf{int}\ \}$$

i. Geben Sie allgemeinste Unifikatoren $\sigma_1$ für $C_1$ und $\sigma_2$ für $C_2$ an.

ii. Ist auch $C_1 \cup C_2$ unifizierbar? Falls ja: Geben Sie einen allgemeinsten Unifikator an. Falls nein: Begründen Sie Ihre Antwort *kurz*!

iii. Ist der Ausdruck

$$\lambda\texttt{a}.\ \lambda\texttt{f}.\ \texttt{f}\ (\texttt{a}\ \texttt{true})\ (\texttt{a}\ \texttt{17})$$

typisierbar? Begründen Sie Ihre Antwort *kurz*.

(b) Betrachten Sie den Ausdruck [14 Punkte]

$$\textbf{let}\ \texttt{a} = \lambda\texttt{x}.\ \lambda\texttt{y}.\ \texttt{true}\ \textbf{in}\ \lambda\texttt{f}.\ \texttt{f}\ (\texttt{a}\ \texttt{true})\ (\texttt{a}\ \texttt{17})$$

i. Wie lautet der polymorphe Typ $\tau_{\texttt{a}}^{poly}$ von `a`?

ii. Unten sehen Sie einen Herleitungsbaum für einen allgemeinsten Typen unter der Typannahme $\Gamma = \texttt{a} : \tau_{\texttt{a}}^{poly}$. Geben Sie das zugehörige Typgleichungssystem an und ergänzen Sie außerdem, was an den mit Ⓐ bzw. Ⓑ markierten Stellen einzutragen ist.

$$\textit{Abs}\ \frac{\textit{App}\ \dfrac{\textit{App}\ \dfrac{\textit{Var}\ \dfrac{(\Gamma, \texttt{f} : \alpha_2)\,(\texttt{f}) = \alpha_5}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{f} : \alpha_5} \quad \textit{App}\ \dfrac{\textit{Var}\ \dfrac{\text{Ⓐ}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{a} : \alpha_7} \quad \textit{C}\ \dfrac{\texttt{true} \in \textit{Const}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{true} : \alpha_8}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{a}\ \texttt{true} : \alpha_6}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{f}\ (\texttt{a}\ \texttt{true}) : \alpha_4} \quad \textit{App}\ \dfrac{\textit{Var}\ \dfrac{\text{Ⓑ}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{a} : \alpha_{10}} \quad \textit{C}\ \dfrac{\texttt{17} \in \textit{Const}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{17} : \alpha_{11}}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{a}\ \texttt{17} : \alpha_9}}{\Gamma, \texttt{f} : \alpha_2 \vdash \texttt{f}\ (\texttt{a}\ \texttt{true})\ (\texttt{a}\ \texttt{17}) : \alpha_3}}{\Gamma \vdash \lambda\texttt{f}.\ \texttt{f}\ (\texttt{a}\ \texttt{true})\ (\texttt{a}\ \texttt{17}) : \alpha_1}$$

**Beispiellösung**:

(a) i.

$$\sigma_1 = [\alpha_9 \Rightarrow \textbf{bool} \to \alpha_8, \alpha_4 \Rightarrow \textbf{bool} \to \alpha_8, \alpha_{10} \Rightarrow \textbf{bool}]$$
$$\sigma_2 = [\alpha_{12} \Rightarrow \textbf{int} \to \alpha_{11}, \alpha_4 \Rightarrow \textbf{int} \to \alpha_{11}, \alpha_{13} \Rightarrow \textbf{int}]$$

ii. $C_1 \cup C_2$ ist nicht unifizierbar, weil sich die Gleichungssysteme widersprechen: $C_1$ fordert $\alpha_4 = \textbf{bool} \to \alpha_8$, $C_2$ fordert $\alpha_4 = \textbf{int} \to \alpha_{11}$, was sich nicht unifizieren lässt.

iii. Der Ausdruck ist nicht typisierbar, da `a` in ihm einmal als Funktion $\textbf{bool} \to \alpha$ und einmal als Funktion $\textbf{int} \to \beta$ verwendet wird, was die Typregeln ohne let-Polymorphismus nicht erlauben.

(b) i. $\tau_{\texttt{a}}^{poly} = \forall\alpha.\forall\beta.\alpha \to \beta \to \textbf{bool}$

ii.

$$
\begin{aligned}
\alpha_1 &= \alpha_2 \to \alpha_3 \\
\alpha_4 &= \alpha_9 \to \alpha_3 \\
\alpha_5 &= \alpha_6 \to \alpha_4 \\
\alpha_2 &= \alpha_5 \\
\alpha_7 &= \alpha_8 \to \alpha_6 \\
\alpha_{10} &= \alpha_{11} \to \alpha_9 \\
\alpha_8 &= \texttt{bool} \\
\alpha_{11} &= \texttt{int} \\
\alpha_7 &= \alpha' \to \beta' \to \texttt{bool} \\
\alpha_{10} &= \alpha'' \to \beta'' \to \texttt{bool} \\
\text{(A)} : (\Gamma, \texttt{f} : \alpha_2)\,(a) &= \forall\alpha.\forall\beta.\alpha \to \beta \to \texttt{bool} \succeq \alpha' \to \beta' \to \texttt{bool} \\
\text{(B)} : (\Gamma, \texttt{f} : \alpha_2)\,(a) &= \forall\alpha.\forall\beta.\alpha \to \beta \to \texttt{bool} \succeq \alpha'' \to \beta'' \to \texttt{bool}
\end{aligned}
$$

**Aufgabe 4** (Lambdakalkül, Maybe-Datentyp) [10 Punkte]

Aus der Vorlesung kennen Sie die Modellierung von Church-Zahlen und Church-Booleans. Auch der **`Maybe`**-Datentyp aus Haskell lässt sich im Lambdakalkül darstellen. In Haskell ist dieser wie folgt definiert:

```
data Maybe a = Nothing | Just a
```

Im λ-Kalkül können die Konstruktoren definiert werden durch:

$$\begin{aligned} \mathsf{nothing} &= \lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \mathtt{n} \\ \mathsf{just} &= \lambda \mathtt{x}.\ \lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \mathtt{j}\ \mathtt{x} \end{aligned}$$

Außerdem sei der Term maybe definiert als:

$$\mathsf{maybe} = \lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \lambda \mathtt{m}.\ \mathtt{m}\ \mathtt{n}\ \mathtt{j}$$

(a) Zeigen Sie, dass für beliebige $n, j, x$ gilt: [4 Punkte]

$$\begin{aligned} \mathsf{maybe}\ n\ j\ \mathsf{nothing} &\Rightarrow^* n \\ \mathsf{maybe}\ n\ j\ (\mathsf{just}\ x) &\Rightarrow^* j\ x \end{aligned}$$

**Hinweis:** Für die nachfolgenden Teilaufgaben dürfen Sie maybe sowie diese Gleichungen verwenden.

(b) Der Lambdaterm isJust sei wie folgt definiert: [3 Punkte]

$$\mathsf{isJust} = \mathsf{maybe}\ c_{\mathsf{false}}\ (\lambda \mathtt{a}.\ c_{\mathsf{true}})$$

Zeigen Sie, dass isJust folgende Eigenschaften erfüllt:

$$\begin{aligned} \mathsf{isJust}\ \mathsf{nothing} &\Rightarrow^* c_{\mathsf{false}} \\ \mathsf{isJust}\ (\mathsf{just}\ x) &\Rightarrow^* c_{\mathsf{true}} \end{aligned}$$

(c) Definieren Sie einen Lambdaterm choose mit folgenden Eigenschaften: [3 Punkte]

$$\begin{aligned} \mathsf{choose}\ \mathsf{nothing}\ m &\Rightarrow^* m \\ \mathsf{choose}\ (\mathsf{just}\ x)\ m &\Rightarrow^* \mathsf{just}\ x \end{aligned}$$

**Beispiellösung**:

(a)

$$
\begin{array}{rcl}
\mathsf{maybe}\ n\ j\ \mathsf{nothing} & = & (\lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \lambda \mathtt{m}.\ \mathtt{m}\ \mathtt{n}\ \mathtt{j})\ n\ j\ \mathsf{nothing} \\
& \Rightarrow^3 & \mathsf{nothing}\ n\ j \\
& = & (\lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \mathtt{n})\ n\ j \\
& \Rightarrow^2 & n \\
\mathsf{maybe}\ n\ j\ (\mathsf{just}\ x) & = & (\lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \lambda \mathtt{m}.\ \mathtt{m}\ \mathtt{n}\ \mathtt{j})\ n\ j\ (\mathsf{just}\ x) \\
& \Rightarrow^3 & \mathsf{just}\ x\ n\ j \\
& = & (\lambda \mathtt{x}.\ \lambda \mathtt{n}.\ \lambda \mathtt{j}.\ \mathtt{j}\ \mathtt{x})\ x\ n\ j \\
& \Rightarrow^3 & j\ x
\end{array}
$$

(b)

$$
\begin{array}{rcl}
\mathsf{isJust\ nothing} & = & \mathsf{maybe}\ c_{\mathsf{false}}\ (\lambda \mathtt{a}.\ c_{\mathsf{true}})\ \mathsf{nothing} \\
& \overset{(a)}{\Longrightarrow}^* & c_{\mathsf{false}} \\
\mathsf{isJust}\ (\mathsf{just}\ x) & = & \mathsf{maybe}\ c_{\mathsf{false}}\ (\lambda \mathtt{a}.\ c_{\mathsf{true}})\ (\mathsf{just}\ x) \\
& \overset{(a)}{\Longrightarrow}^* & (\lambda \mathtt{a}.\ c_{\mathsf{true}})\ x \\
& \Rightarrow & c_{\mathsf{true}}
\end{array}
$$

(c)

$$\mathsf{choose} = \lambda \mathtt{m}.\ \lambda \mathtt{d}.\ (\mathsf{isJust}\ \mathtt{m})\ \mathtt{m}\ \mathtt{d}$$

Alternativlösung:

$$\mathsf{choose} = \lambda \mathtt{m}.\ \lambda \mathtt{d}.\ \mathsf{maybe}\ \mathtt{d}\ \mathsf{just}\ \mathtt{m}$$

**Aufgabe 5** (MPI) [11 Punkte]

Gegeben sei die unten stehende Methode, welche in einem MPI-Programm mit einer beliebigen Anzahl an Prozessen aufgerufen wird. Sie können davon ausgehen, dass die Initialisierung und Finalisierung von MPI vom Aufrufer übernommen wird. Weiterhin können Sie davon ausgehen, dass `elementCount` immer ein Vielfaches der Anzahl verfügbarer Prozesse ist.

```
1  void operation(int elementCount, int elements[elementCount]) {
2      int rank;
3      int processCount;
4      MPI_Status status;
5      MPI_Request request;
6
7      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8      MPI_Comm_size(MPI_COMM_WORLD, &processCount);
9
10     int elementsPerProcess = elementCount / processCount;
11     int local[elementsPerProcess];
12     if (rank == 0) {
13         for (int i = 0; i < processCount; i++) {
14             MPI_Isend(&elements[i*elementsPerProcess], elementsPerProcess,
15                 MPI_INT, i, 0, MPI_COMM_WORLD, &request);
16         }
17     }
18     MPI_Recv(local, elementsPerProcess, MPI_INT, 0, 0,
19         MPI_COMM_WORLD, &status);
20
21     int localSum = 0;
22     for (int i = 0; i < elementsPerProcess; i++) {
23         localSum += local[i];
24     }
25
26     int sum = 0;
27     MPI_Isend(&localSum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
28     if (rank == 0) {
29         for (int i = 0; i < processCount; i++) {
30             int value;
31             MPI_Recv(&value, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
32             sum += value;
33         }
34     }
35
36     if (rank == 0) {
37         double result = (double)sum / elementCount;
38         printf("Result: \"%f\"\n", result);
39     }
40 }
```

(a) Erklären Sie, welche Funktion das Programm erfüllt und geben Sie seine Ausgabe für folgende Eingabe an: `(4, [0, 2, 9, 5])` [3 Punkte]

**Beispiellösung**: Das Programm berechnet für das Eingabe-Array `elements` den Durchschnittswert der Elemente.

Ausgabe: "Result: 4"

(b) Erklären Sie den Unterschied zwischen den MPI-Operationen `MPI_Send` und `MPI_Isend`. Was passiert, wenn in Zeile 14 `MPI_Isend` durch die Operation `MPI_Send` ersetzt wird? [2,5 Punkte]

**Beispiellösung**: `MPI_Send` ist eine blockierende Operation. Das Programm führt erst weitere Operationen aus, wenn die Nachricht durch ein zugehöriges `MPI_Recv` entgegen genommen wurde. `MPI_Isend` ist die zugehörige nicht-blockierende (*immediate*) Operation, bei der nicht auf das Entgegennehmen der Nachricht gewartet wird.

Würde man hier `MPI_Send` verwenden, so würde das Programm nicht terminieren, da diese Operation auch für den Versand von Daten von Prozess 0 zu sich selbst aufgerufen wird.

(c) Schreiben Sie einen Ersatz für die Zeilen 12 bis 19, der ausschließlich aus dem Aufruf einer MPI-Operation besteht. [2,5 Punkte]

**Beispiellösung**:

```
MPI_Scatter(elements, elementsPerProcess, MPI_INT, local,
      elementsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);
```

(d) Schreiben Sie einen Ersatz für die Zeilen 27 bis 34, der ausschließlich aus dem Aufruf einer MPI-Operation besteht. [3 Punkte]

**Beispiellösung**:

```
MPI_Reduce(&localSum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

**Aufgabe 6** (Parallelprogrammierung mit Java) [12 Punkte]

Gegeben sei folgende Implementierung einer Warteschlange, angelehnt an die Implementierung der `LinkedList` aus der Java-API. Es werden die Methode `add` für das Hinzufügen eines Elementes und die Methode `poll` für das Entfernen des letzten Elementes, entsprechend einer FIFO-Strategie, angeboten. Die Elemente `firstheader` und `lastHeader` dienen als Wächterobjekte und zeigen auf das erste bzw. letzte Element der Warteschlange. Ein Beispiel für eine Warteschlange mit zwei Elementen ist in Abbildung 1 dargestellt.

```
1  public class Queue<E> {
2      private Entry<E> firstHeader = new Entry<E>(null, null, null);
3      private Entry<E> lastHeader = new Entry<E>(null, null, null);
4
5      public Queue() {
6          firstHeader.next = firstHeader.previous = lastHeader;
7          lastHeader.next = lastHeader.previous = firstHeader;
8      }
9
10     private static class Entry<E> {
11         E element;
12         Entry<E> next;
13         Entry<E> previous;
14
15         Entry(E element, Entry<E> next, Entry<E> previous) {
16             this.element = element;
17             this.next = next;
18             this.previous = previous;
19         }
20     }
21
22     public void add(E e) {
23         synchronized (lastHeader) {
24             Entry<E> newEntry =
25                 new Entry<E>(e, lastHeader, lastHeader.previous);
26             lastHeader.previous.next = newEntry;
27             lastHeader.previous = newEntry;
28         }
29     }
30
31     public E poll() {
32         synchronized (firstHeader) {
33             if (firstHeader.next == lastHeader) // Queue is empty
34                 return null;
35             Entry<E> removedEntry = firstHeader.next;
36             firstHeader.next = removedEntry.next;
37             removedEntry.next.previous = firstHeader;
38             return removedEntry.element;
39         }
40     }
41 }
```

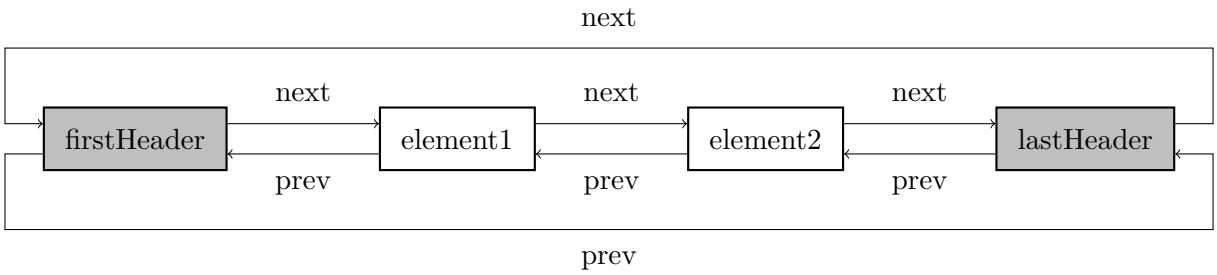



Abbildung 1: Beispiel einer `Queue` mit zwei Elementen

(a) Nehmen Sie an, eine Instanz der Klasse `Queue` wird von mehreren, nebenläufigen Programmfäden verwendet. Erläutern Sie kurz, warum dabei eine Wettlaufsituation vorliegt. Geben Sie eine Aufruf- und Ausführungsreihenfolge an, bei der dies zu unerwünschtem Verhalten führt. [3,5 Punkte]

**Beispiellösung**: Die Implementierungen der Methoden `add` und `poll` synchronisieren auf verschiedenen Elementen, `firstHeader` bzw. `lastHeader`. Dies stellt sicher, dass nur jeweils eine `add`- und nur eine `poll`-Operation zur gleichen Zeit ausgeführt werden können. Bei der gleichzeitigen Ausführung von `add` und `poll` kann jedoch das gleiche Element modifiziert werden, wodurch das Ergebnis von der Ausführungsreihenfolge abhängt. Dies nennt man eine *Wettlaufbedingung* (engl. *race condition*).

Beispielablauf: Es befindet sich ein Element in der Warteschlange. `poll` wird bis einschließlich Zeile 36 ausgeführt, `add` wird vollständig ausgeführt und schließlich wird der Rest von `poll` ausgeführt. Danach zeigt `lastHeader.previous` auf `firstHeader` statt auf das neue Element.

(b) Das Programm soll so ergänzt werden, dass keine Wettlaufsituation mehr vorliegt. [6 Punkte]

i. Geben Sie an, wo und durch welche Ausdrücke das Programm ergänzt werden muss, sodass keine Wettlaufsituation mehr vorliegt. Die Ergänzung soll eine möglichst feingranulare Synchronisation durchführen, keine Veränderungen am bestehenden Code durchführen und darf insbesondere nicht durch eine Synchronisation auf `this`, `lastHeader` oder `firstHeader` erreicht werden.

**Beispiellösung**: Zusätzlich zu der existierenden Synchronisation müssen noch weitere Elemente synchronisiert werden.
Für `add` ist folgende Synchronisation notwendig:

```
1 synchronized(lastHeader) {
2     synchronized(lastHeader.previous) {
3         ...
4     }
5 }
```

Für `poll` ist folgende Synchronisation notwendig:

```
1 synchronized(firstHeader) {
2     synchronized(firstHeader.next.next) {
3         ...
4     }
5 }
```

**Beispiellösung**: Obiger Lösungsvorschlag löst das Problem auf Basis des Kenntnisstandes aus der Vorlesung. Unter Beachtung von Speicherbarrieren und damit möglichen bzw. unmöglichen Befehlsumordnungen durch den Compiler ist folgende, umfangreichere Synchronisation notwendig.
Für `add` ist folgende Synchronisation notwendig:

```
1 synchronized(lastHeader.previous) {
2     synchronized(lastHeader) {
3         ...
4     }
5 }
```

Für `poll` ist folgende Synchronisation notwendig:

```
1 synchronized(firstHeader) {
2     synchronized(firstHeader.next) {
3         synchronized(firstHeader.next.next) {
4             ...
5         }
6     }
7 }
```

ii. Begründen Sie, warum ihre Ergänzung nicht zu einem Deadlock führen kann.

**Beispiellösung**:
*1. Beispiellösung:* Damit ein Deadlock auftreten kann, müssen alle Coffman-Bedingungen erfüllt sein. Die Coffman-Bedingungen *mutual exclusion*, *no preemption* und *hold-and-wait* sind im Beispiel erfüllt, die Bedingung *circular wait* jedoch nicht. Somit kann kein Deadlock auftreten. Es könnte lediglich zu einer zirkulären Warteabhängigkeit kommen, wenn

- `lastHeader.previous == firstHeader` und
- `lastHeader == firstHeader.next.next`

erfüllt wären. Dies ist jedoch nicht möglich, denn:

i. Falls `lastHeader.previous == firstHeader` gilt, so ist die Liste leer und damit gilt `firstHeader.next.next == firstHeader != lastHeader`.
ii. Falls `lastHeader == firstHeader.next.next`, so befindet sich ein Element in der Liste und somit ist `lastHeader.previous` das erste Element und damit nicht `lastHeader`.

Somit kann niemals eine zirkuläre Wartesituation eintreten.

*2. Beispiellösung:* Die Begründung funktioniert analog zur ersten Beispiellösung. Durch die Umordnung der Synchronisation in der `add`-Methode ist auch hier *circular wait* ausgeschlossen.

(c) Die sequentielle Implementierung der `LinkedList` in der Java-API verwendet nur ein `header`-Objekt. Erläutern Sie, was der Vorteil der Verwendung von `firstHeader` und `lastHeader` in der hier gezeigten Implementierung hinsichtlich der nebenläufigen Verwendung der Warteschlange gegenüber der Verwendung eines einzelnen `header`-Objektes ist. [2,5 Punkte]

**Beispiellösung**: Bei einer Implementierung mit nur einem Header könnten sowohl `add` als auch `poll` nur auf diesem Objekt synchronisieren. Es wäre niemals möglich, gleichzeitig eine `add`- und eine `poll`-Operation auszuführen, obwohl dabei bei einer Warteschlangenlänge größer 1 kein gleichzeitiger Zugriff auf eine gemeinsam genutzt Variable stattfindet und somit die Voraussetzung für einen kritischen Abschnitt nicht gegeben ist.

**Aufgabe 7** (Design by Contract) [7 Punkte]

Gegeben sei folgendes Code-Fragment mit einer (unvollständigen) Implementierung einer Collection-Klasse und einer Verwendung dieser Klasse. Für die Methode `add` der Collection-Klasse ist im Javadoc zusätzlich ein Vertrag angegeben, der Vor- und Nachbedingungen der Methode definiert. Die Vor- und Nachbedingungen sind als Java-Ausdrücke definiert. Zusätzlich wird mittels `\old(<expression>)` der Wert von `<expression>` vor dem Aufruf der Methode referenziert.
*Anmerkung:* Die vierte Nachbedingung ist syntaktisch inkorrekt. In Zeile 20 müsste es korrekterweise `elements[i] == null` heißen. Dies sollte bei der Lösung der Aufgaben nicht beachtet werden.

```
1  public class ObjectCollection {
2      Object[] elements;
3      int elementCount;
4
5      public ObjectCollection(int collectionSize) {
6          elements = new Object[collectionSize];
7          elementCount = 0;
8      }
9
10     /**
11     * Vorbedingungen:
12     *   object != null;
13     *
14     * Nachbedingungen:
15     *   elementCount == \old(elementCount) + 1;
16     *   elements[elementCount-1] == object;
17     *   for (int i = 0; i < elementCount - 1; i++)
18     *       elements[i] == \old(elements[i]);
19     *   for (int i = elementCount; i < elements.length; i++)
20     *       elements[i] == 0;
21     */
22     public void add(Object object) {
23         if (elementCount < elements.length) {
24             elements[elementCount++] = object;
25         }
26     }
27 }
28
29
30 public class ObjectCollectionUser {
31     public static void main(String[] args) {
32         Object object = new Object();
33         ObjectCollection collection = new ObjectCollection(10);
34         collection.add(object);
35     }
36 }
```

(a) Wird der spezifizierte Vertrag vom Aufrufer erfüllt? Begründen Sie Ihre Antwort. [1,5 Punkte]

**Beispiellösung**:

Der Vertrag wird vom Aufrufer erfüllt, da die Methode mit einem gerade erzeugten Objekt aufgerufen wird, welches somit nicht `null` ist und damit die einzige Vorbedingung erfüllt ist.

(b) Geben Sie an, welche Nachbedingungen des Vertrages vom Aufgerufenen nicht erfüllt werden. Begründen Sie dies, indem Sie eine Situation beschreiben, in der die Nachbedingungen nicht erfüllt werden. [2 Punkte]

**Beispiellösung**:

Die erste und zweite Zeile der Nachbedingung (Zeile 15 und 16) werden nicht erfüllt.

Falls das `elements`-Array voll ist, wird bei einem Aufruf der `add`-Methode nichts am Zustand der Collection verändert. Insbesondere bleibt `elementCount` gleich (Widerspruch zur ersten Bedingung) und das Methodenargument wird nicht in den `(elementCount-1)`-ten Eintrag von `elements` geschrieben (Widerspruch zur zweiten Bedingung).
Die anderen beiden Bedingungen gelten weiterhin, da sie eigentlich keine Nachbedingungen, sondern Invarianten der Collection darstellen.

(c) Der Vertrag soll so verändert werden, dass er vom Aufgerufenen erfüllt wird. [2,5 Punkte]

i. Geben Sie eine Änderung der Vorbedingungen an, mit der dies erreicht werden kann.

**Beispiellösung**:
Hinzufügen der Vorbedingung `elementCount < elements.length`.

ii. Geben Sie eine Änderung der Nachbedingungen an, mit der dies erreicht werden kann.

**Beispiellösung**:
Die erste und zweite Nachbedingung einschränken, sodass sie nur erfüllt sein müssen, falls `\old(elementCount) < elements.length` gilt. Nicht notwendig für die Beantwortung der Aufgabe, aber sinnvoll wäre es zu fordern, dass in allen anderen Fällen `\old(elementCount) == elementCount` gelten muss.

(d) Nehmen Sie an, es würde die folgende Prüfung des Eingabeparameters am Beginn der Method `add` ergänzt: [1 Punkt]

```
if (object == null) return;
```

Würde diese Ergänzung den Vertrag verletzen? Begründen Sie Ihre Antwort kurz.

**Beispiellösung**:

Nein, die Prüfung würde den Vertrag nicht verletzen. Die Vorbedingung schließt ein Objekt mit Wert `null` als Methodenargument bereits aus. Somit wird die Bedingung bei vertragsgemäßer Verwendung der Methode sowieso immer zu `false` ausgewertet.

**Aufgabe 8** (Compiler) [20 Punkte]

Gegeben sei die folgende Grammatik für Prolog-Fakten mit dem Startsymbol *Fact*:

$$
\begin{array}{rcl}
\mathit{Fact} & \rightarrow & \mathit{Term}\ \texttt{.} \\
\mathit{Term} & \rightarrow & \texttt{atom} \mid \texttt{atom (}\ \mathit{Termlist}\ \texttt{)} \mid \texttt{number} \mid \texttt{variable} \\
\mathit{Termlist} & \rightarrow & \mathit{Term}\ \mathit{Termlist}' \\
\mathit{Termlist}' & \rightarrow & \varepsilon \mid \texttt{,}\ \mathit{Term}\ \mathit{Termlist}'
\end{array}
$$

(a) Geben Sie die $\text{First}_1$- und $\text{Follow}_1$-Mengen für die Nichtterminale *Fact* und *Term* an. [4 Punkte]

(b) Geben Sie eine Linksfaktorisierung an, so dass die Grammatik SLL(1) ist. [2 Punkte]
Es genügt geänderte bzw. neue Zeilen der Grammatik anzugeben.

(c) Geben Sie für alle Nichtterminale der linksfaktorisierten Grammatik einen rekursiven Abstiegsparser in Pseudocode an. [14 Punkte]
Dabei stehen Ihnen folgende Token-Typen zur Verfügung:

| | |
|---|---|
| atom | für Prolog-Atome |
| number | für Prolog-Zahlen |
| var | für Prolog-Variablen |
| dot | für den Punkt `.` |
| comma | für das Komma `,` |
| lp, rp | für öffnende bzw. schließende Klammern (, ) |

Die globale Variable `token` enthält immer das aktuelle Token. Es besitzt die Methode `getType()`, die den zugehörigen Token-Typ (atom, number, var, dot, comma, lp, rp) zurückgibt. Die globale Methode `nextToken()` fordert das nächste Token an. Brechen Sie bei Parsefehlern durch Aufruf der globalen `error()`-Methode ohne Fehlermeldung ab.

**Beispiellösung**:

(a)

$$\begin{aligned} \text{First}_1(\textit{Fact}) &= \{\texttt{atom}, \texttt{number}, \texttt{variable}\} \\ \text{First}_1(\textit{Term}) &= \{\texttt{atom}, \texttt{number}, \texttt{variable}\} \\ \text{Follow}_1(\textit{Fact}) &= \{\texttt{\#}\} \\ \text{Follow}_1(\textit{Term}) &= \{\texttt{.}, \texttt{,}, \texttt{)}\} \end{aligned}$$

(b)

$$\begin{aligned} \textit{Term} &\rightarrow \texttt{atom}\ \textit{Atom}' \mid \texttt{number} \mid \texttt{variable} \\ \textit{Atom}' &\rightarrow \varepsilon \mid \texttt{(}\ \textit{Termlist}\ \texttt{)} \end{aligned}$$

(c)

```
1  void parseFact() {
2      parseTerm();
3      if (token.getType() != dot) {
4          error();
5      }
6      nextToken();
7  }
8
9  void parseTerm() {
10     switch (token.getType()) {
11     case atom:
12         nextToken();
13         parseAtom′();
14         break;
15     case number:
16     case variable:
17         nextToken();
18         break;
19     default:
20         error();
21     }
22 }
23
24 void parseAtom′() {
25     switch (token.getType()) {
26     case dot:
27     case comma:
28     case rp:
29         break;
30     case lp:
31         nextToken();
32         parseTermlist();
33         if (token.getType() != rp) {
34             error();
35         }
36         nextToken();
37         break;
```

```
38      default:
39          error();
40      }
41  }
42
43  void parseTermlist() {
44      parseTerm();
45      parseTermlist'();
46  }
47
48  void parseTermlist'() {
49      switch (token.getType()) {
50      case rp:
51          break;
52      case comma:
53          nextToken();
54          parseTerm();
55          parseTermlist'();
56          break;
57      default:
58          error();
59      }
60  }
```