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

WS18/19, 04. April 2019, 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	max. Punkte	err. Punkte
1	12	
2	18	
3	16	
4	15	
5	10	
6	15	
7	15	
8	19	
Σ	**120**	
$\lceil\Sigma\rceil$	**120**	

Jeder Punkt entspricht ca. 1 min Bearbeitungszeit. Es ist garantiert, dass die Klausur mit 60 Punkten bestanden ist, und dass 115 Punkte für die Note 1,0 ausreichen.

Name: _____

Matrikelnummer: _____

Studiengang: _____

Schreiben Sie Ihre Lösungen direkt in die Klausur. Beschriften Sie **alle verwendeten Blätter** mit Ihrem Namen und Ihrer Matrikelnummer. Trennen Sie die geklammerten Blätter **nicht** auf. Weitere Blätter erhalten Sie bei Bedarf.

Aufgabe 1 (Haskell: Listen) [12 Punkte]

(a) Implementieren Sie die Haskell-Funktion [6 Punkte]

```
substr :: Eq a => [a] -> [a] -> Bool
```

die angewendet auf zwei endliche Listen `True` zurückgibt, falls die erste Liste zusammenhängend an beliebiger Position in der zweiten vorkommt, und ansonsten `False`.

Hinweis: Eine Hilfsfunktion könnte nützlich sein!

Beispiele:

```
> substr "bc" "abc"
True
> substr "ac" "abc"
False
> substr "aa" "abc"
False
```

(b) Analysieren Sie das Verhalten Ihrer Implementierung von `substr` auf unendlichen Listen: [6 Punkte]
Geben Sie in jeder Tabellenzelle an, zu welchen der Werte `True`, `False` oder $\perp$ (Nichttermination) `substr xs ys` unter den entsprechenden Einschränkungen auswerten kann, und geben Sie für jeden Wert einen Beispielaufruf an. Die Zelle für den endlichen Fall ist als Beispiel schon ausgefüllt.

Hinweis: Benutzen Sie bspw. die unendliche Liste `as = repeat 'a'` $\equiv$ `"aaaa...`

	`xs` endlich	`xs` unendlich
`ys` endlich	`True` (`substr "bc" "abc"`), `False` (`substr "ac" "abc"`)	
`ys` unendlich		

Aufgabe 2 (Haskell: Warteschlangen) [18 Punkte]

Haskells einfach verkettete Listen eignen sich gut, um einen Stack zu modellieren. Weniger effizient ist es, eine Warteschlange als Haskell-Liste zu implementieren, denn die `enqueue`-Operation muss dazu die gesamte Liste durchqueren. Abhilfe schafft folgende Definition für Warteschlangen:

```
data Queue a = Q [a] [a]
```

`Q front back` stellt die Liste dar, die durch Konkatenation der *Vorderseite* `front` und der Umkehrung der *Rückseite* `back` entsteht. So stellen diese drei Haskell-Werte die gleiche Liste dar:

```
Q [1,2,3,4,5] []
Q [1,2,3] [5,4]
Q [] [5,4,3,2,1]
```

(a) Implementieren Sie die Funktionen [3 Punkte]

```
fromList :: [a] -> Queue a
toList :: Queue a -> [a]
```

die zwischen einer Liste und ihrer Darstellung als `Queue` konvertieren.

(b) Implementieren Sie die Funktion [2 Punkte]

```
enqueue :: a -> Queue a -> Queue a
```

`enqueue x q` hängt `q` das Element `x` an. Dabei soll nur konstant viel Speicher berührt werden.

Beispiel:

```
> toList (enqueue 4 (fromList [1, 2, 3]))
[1, 2, 3, 4]
```

(c) Implementieren Sie die Funktion [5 Punkte]

```
dequeue :: Queue a -> Maybe (a, Queue a)
```

Ist die von `q` dargestellte Liste leer, gilt `dequeue q == Nothing`. Ansonsten teilt sich `q` in das vorderste Element `x` und den Rest `q'`, und `dequeue q == Just (x,q')`.

Achten Sie darauf, dass sich die Rückseiten von `q` und `q'` nur dann unterscheiden, wenn die Vorderseite von `q` leer ist.

(d) Implementieren Sie die Funktion [8 Punkte]

```
bfs :: Tree a -> [a]
```

die einen Binärbaum in Form des aus der Vorlesung bekannten Datentyps entgegennimmt:

```
data Tree t
  = Leaf
  | Node (Tree t) t (Tree t)
```

`bfs` gibt die Knotenlabels des übergebenen Baums in Breitenordnung zurück.

Hinweise: Definieren Sie hierzu zum Beispiel eine Hilfsfunktion `go :: Queue (Tree a) -> [a]`, die die Warteschlange von Knoten abarbeitet. Diese ist initial nur mit der Wurzel befüllt und wird bei der Behandlung von inneren Knoten mit den linken und rechten Kindern erweitert.

Beispiel (`N` für `Node`, `L` für `Leaf`):

```
> bfs (N (N (N L 4 L) 2 L) 1 (N L 3 L))
[1, 2, 3, 4]
```

Aufgabe 3 (Prolog: Buchstabenrätsel) [16 Punkte]

Ein beliebtes Buchstabenrätsel funktioniert wie folgt: Ein Start- und ein Zielwort werden vorgegeben. Das Startwort soll nun schrittweise zum Zielwort transformiert werden. Dabei gibt es 3 mögliche Arten von Schritten:

- Ein einzelner Buchstabe des momentanen Worts wird ersetzt.
- Ein einzelner Buchstabe des momentanen Worts wird entfernt.
- Ein einzelner Buchstabe wird dem momentanen Wort hinzugefügt.

Nach jedem Schritt muss wieder ein gültiges Wort entstehen.

Beispiel:
Rast → Rat → Rad → Rand

In Prolog lassen sich Buchstaben als Atome und Wörter als Listen von Atomen darstellen (z.B. "Rad" als `[r,a,d]`). Zudem seien folgende Prolog-Prädikate bereits vordefiniert: Der Generator `buchstabe(X)`, der bei Reerfüllung für `X` alle gültigen Buchstaben generiert, sowie der Tester `erlaubt(W)`, der testet, ob das Wort `W` gültig ist.

(a) Definieren Sie einen zweistelligen *Generator* [8 Punkte]

```
schritt(Wort1, Wort2),
```

welcher für ein gegebenes `Wort1` bei Reerfüllung alle Wörter `Wort2` generiert, die aus `Wort1` durch einen Schritt entstehen. Sie müssen hier noch nicht prüfen, ob das Wort gültig ist oder ob sich `Wort2` von `Wort1` unterscheidet.

Beispiel:

```
? schritt([r,a,d], W2).
W2 = [a,a,d] ; W2 = [b,a,d] ; ... ; W2 = [z,a,d] ;
W2 = [r,a,d] ; W2 = [r,b,d] ; ... ; W2 = [r,z,d] ;
... ; W2 = [r,a,z] ;
W2 = [a,d] ; W2 = [r,d] ; W2 = [r,a] ;
W2 = [a,r,a,d] ; ... ; W2 = [z,r,a,d] ;
... ;
W2 = [r,a,d,a] ; ... ; W2 = [r,a,d,z] ;
false.
```

(b) Das Prädikat `lösung(Woerter)` generiert alle Lösungen des Problems: [8 Punkte]

```
lösung(Woerter) :- start(S), ziel(Z), erreichbar(S,[S],Woerter,Z).
start([r,a,s,t]).
ziel([r,a,n,d]).
```

Definieren Sie das hierzu benötigte vierstellige Prädikat

```
erreichbar(S,Besucht,Woerter,Z)
```

welches für Start-Wort `S` und Ziel-Wort `Z` erfüllt ist, falls `Z` von `S` durch eine Folge von Zwischen-**`Woerter`**n erreichbar ist. Dabei dürfen nur **`erlaubt`**e Zwischenwörter entstehen. Um Endlosschleifen zu vermeiden, darf dabei weiterhin keine der in der Liste `Besucht` enhaltenen Wörter

nochmals vorkommen. Die Liste `Woerter` soll bei Erfüllung die einzelnen Zwischen-**`Woerter`** in richtiger Reihenfolge enthalten.

Hinweis: Verwenden Sie die Prädikate `member` und `not` aus der Vorlesung.

Beispiel:

```
? lösung(Wörter).
Wörter = [[r,a,s,t], [r,a,t], [r,a,d], [r,a,n,d]] ;
...
Wörter = [[r,a,s,t], [r,o,s,t], [r,o,t], [r,a,t], [r,a,d], [r,a,n,d] ;
false.
```

Aufgabe 4 (Typsysteme) [15 Punkte]

In der Vorlesung haben Sie bereits Kodierungen im λ-Kalkül für boolsche Werte und die natürlichen Zahlen gesehen.

Erinnerung: $\mathsf{c}_2 = (\lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}))$ $\mathsf{c}_{true} = (\lambda \mathtt{t}.\ \lambda \mathtt{f}.\ \mathtt{t})$

Der Ausdruck $\mathsf{c}_2\ \mathsf{c}_{true} = (\lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}))\ (\lambda \mathtt{t}.\ \lambda \mathtt{f}.\ \mathtt{t})$ ist nicht typisierbar.

Es seien:

$$\Gamma_{sz} = \mathtt{s} : \alpha_4, \mathtt{z} : \alpha_6$$
$$\Gamma_{tf} = \mathtt{t} : \alpha_{12}, \mathtt{f} : \alpha_{14}$$

Im Folgenden ist der Typherleitungsbaum für diesen Ausdruck abgebildet:

$$\textit{App}\ \frac{\textit{Abs}\ \frac{\textit{App}\ \frac{\textit{Abs}\ \frac{\textit{App}\ \frac{\textit{Var}\ \frac{\Gamma_{sz}(\mathtt{s}) = \alpha_4}{\Gamma_{sz} \vdash \mathtt{s} : \alpha_8}\ \textit{App}\ \frac{\textit{Var}\ \frac{\Gamma_{sz}(\mathtt{s}) = \alpha_4}{\Gamma_{sz} \vdash \mathtt{s} : \alpha_{10}}\ \textit{Var}\ \frac{\Gamma_{sz}(\mathtt{z}) = \alpha_6}{\Gamma_{sz} \vdash \mathtt{z} : \alpha_{11}}}{\Gamma_{sz} \vdash \mathtt{s}\ \mathtt{z} : \alpha_9}}{\Gamma_{sz} \vdash \mathtt{s}\ (\mathtt{s}\ \mathtt{z}) : \alpha_7}}{\mathtt{s} : \alpha_4 \vdash (\lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})) : \alpha_5}}{\vdash (\lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})) : \alpha_2}\ \textit{Abs}\ \frac{\textit{Var}\ \frac{\Gamma_{tf}(\mathtt{t}) = \alpha_{12}}{\Gamma_{tf} \vdash \mathtt{t} : \alpha_{15}}}{\mathtt{t} : \alpha_{12} \vdash (\lambda \mathtt{f}.\ \mathtt{t}) : \alpha_{13}}}{\vdash (\lambda \mathtt{t}.\ \lambda \mathtt{f}.\ \mathtt{t}) : \alpha_3}}{\vdash (\lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}))\ (\lambda \mathtt{t}.\ \lambda \mathtt{f}.\ \mathtt{t}) : \alpha_1}$$

(a) Geben Sie das Constraint-System für diesen Herleitungsbaum an. [9 Punkte]

(b) Auch wenn $\mathsf{c}_2\ \mathsf{c}_{true}$ nicht typisierbar ist, gibt es bei der β-Reduktion dieses Terms keine Probleme. Zeigen Sie: [3 Punkte]

$$\mathsf{c}_2\ \mathsf{c}_{true}\ \mathtt{a}\ \mathtt{b}\ \mathtt{c} \Rightarrow^* \mathtt{a}$$

(c) Im Gegensatz zu [3 Punkte]

$$\mathsf{c}_2\ \mathsf{c}_{true} = (\lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}))\ (\lambda \mathtt{t}.\ \lambda \mathtt{f}.\ \mathtt{t})$$

ist der Ausdruck

$$\textbf{let}\ \mathtt{s} = (\lambda \mathtt{t}.\ \lambda \mathtt{f}.\ \mathtt{t})\ \textbf{in}\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$$

typisierbar. Für die rechte Seite des Let-Ausdrucks (also $\lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$) wird bei der Typinferenz dabei die Typumgebung

$$\Gamma = \mathtt{s} : ta\,(\alpha \to \beta \to \alpha, \emptyset)\,.$$

verwendet.

Berechnen Sie $ta\,(\alpha \to \beta \to \alpha, \emptyset)$ und beschreiben Sie *kurz*, was dieser Typ in der Typumgebung für die Typinferenz bedeutet.

Aufgabe 5 (Listen im λ-Kalkül) [10 Punkte]

Neben natürlichen Zahlen und Booleans lassen sich auch Listen im λ-Kalkül definieren. Seien also:

```
nil  =  λn. λc. n
cons =  λx. λxs. λn. λc. c x xs
```

Hierbei ist `nil` die Darstellung der leeren Liste, und `cons x xs` die Darstellung der Liste mit erstem Element `x` und Listenrest `xs` (vgl. Haskell `x:xs`).

(a) Geben Sie Definitionen für λ-Ausdrücke `head` und `tail` an, so dass für beliebige A, B gilt: [4 Punkte]

$$\texttt{head (cons } A\ B\texttt{)} \Rightarrow^* A$$
$$\texttt{tail (cons } A\ B\texttt{)} \Rightarrow^* B$$

Das Verhalten bei Übergabe von `nil` als Argument ist Ihnen überlassen.

(b) Geben Sie einen λ-Ausdruck `replicate` an, für den für beliebiges A gilt: [4 Punkte]

$$\texttt{replicate } c_n\ A \Rightarrow^* \underbrace{\texttt{cons } A\ \texttt{(cons } A\ \texttt{...(}\ \texttt{cons } A\ \texttt{nil)...)}}_{n\ \texttt{mal}}$$

Hinweis: Rufen Sie sich in Erinnerung, welche Struktur c_n hat.

(c) Zeigen Sie für Ihre Definition von `replicate` per β-Reduktion: [2 Punkte]

$$\texttt{replicate } c_3\ \texttt{A} \Rightarrow^* \texttt{cons A (cons A (cons A nil))}$$

Aufgabe 6 (MPI: Ziffern-Zählung mit MapReduce) [15 Punkte]

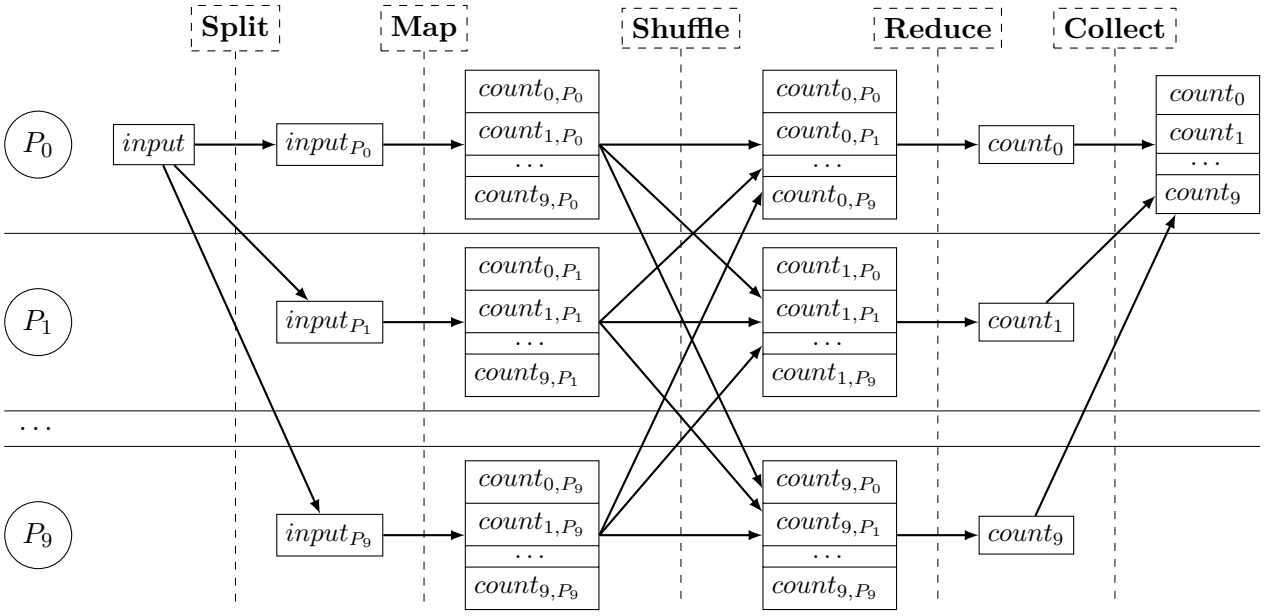
(a) In der untenstehenden Tabelle ist der Inhalt eines Arrays `sendbuf` auf 3 Prozessen dargestellt. Füllen Sie in der leeren Tabelle den Inhalt von `recvbuf` nach einem Aufruf der folgenden kollektiven MPI-Operation aus: [3 Punkte]

```
MPI_Alltoall(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT, MPI_COMM_WORLD);
```

Prozess Nr.	**`sendbuf`**
0	3, 6, 4
1	1, 3, 2
2	9, 8, 1

Prozess Nr.	**`recvbuf`**
0	
1	
2	

Die unten stehende Grafik zeigt ein *MapReduce*-Schema zur Bestimmung der Anzahl jeder Ziffer in einer Ziffernsequenz *input*. Die Ziffernsequenz wird zunächst vom Root-Prozess P_0 in 10 Teile zerlegt und auf die Prozesse P_0 bis P_9 verteilt (*Split*). Jeder Prozess zählt in seiner Teilsequenz die Vorkommen jeder Ziffer (*Map*). Für die Ziffer i in Prozess j ist dies der Wert $count_{i,P_j}$. Daraufhin werden die Daten zwischen den Prozessen neu verteilt, sodass auf jedem der 10 Prozesse alle Zählwerte für genau eine der Ziffern liegen (*Shuffle*). Diese Zählwerte werden in jedem Prozess summiert (*Reduce*), sodass anschließend Prozess P_i die Anzahl der Vorkommen $count_i$ von i in der Eingabesequenz kennt. Abschließend werden die Werte in ein Array auf dem Root-Prozess P_0 zusammengeführt (*Collect*).



(b) Geben Sie in der folgenden Tabelle an, mit welchen MPI-Operationen die Teilschritte *Split*, *Shuffle* und *Collect* implementiert werden können. [4 Punkte]

Schritt	**MPI-Operation**
Split	
Shuffle	
Collect	

(c) Füllen Sie im folgenden Quellcode die Lücken so aus, dass das zuvor dargestellte und beschriebene Schema implementiert wird. Gehen Sie davon aus, dass der Root-Prozess mit der ID 0 die zu bearbeitende `char`-Sequenz erhält, welche nur die Ziffern 0 bis 9 enthält. Es steht die Methode `int* countDigits(char*)` zur Verfügung, welche in der übergebenen `char`-Sequenz die Anzahl der Vorkommen jeder Ziffer ermittelt und in einem Array, welches mit der jeweiligen Ziffer indiziert ist, zurückgibt. Beispielsweise ist die Rückgabe für `countDigits("12341")` das Array $(0, 2, 1, 1, 1, 0, 0, 0, 0, 0)$. Gehen Sie weiterhin davon aus, dass MPI vom Aufrufer initialisiert wird, das Programm mit exakt 10 Prozessen ausgeführt wird und die Länge der Eingabesequenz ein Vielfaches von 10 ist. [8 Punkte]

```
1  int* countDigitsParallel(char *input) {
2      int size, rank;
3      MPI_Comm_size(MPI_COMM_WORLD, &size);
4      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5
6      // Split
7      int digitsPerProcess = strlen(input) / size;
8      char localSequence[digitsPerProcess];
9      [          ]([          ], [          ],
           [          ], [          ], [          ],
           [          ], [          ], MPI_COMM_WORLD);
10
11     // Map
12     int* digitsCounts = countDigits(localSequence);
13
14     // Shuffle
15     int localDigitCounts[10];
16     [          ]([          ], [          ],
           [          ], [          ], [          ],
           [          ], MPI_COMM_WORLD);
17
18     // Reduce
19     int summedCount = 0;
20     [                                                    ]{
21         [                                                ]1
22     }
23
24     // Collect
25     int totalDigitCounts[10];
26     [          ]([          ], [          ],
           [          ], [          ], [          ],
           [          ], [          ], MPI_COMM_WORLD);
27
28     // Please ignore that we return a pointer to stack memory
29     return totalDigitsCounts;
30 }
```

Aufgabe 7 (Java und Design-by-Contract: Warteschlagen-Synchronisation) [15 Punkte]

Gegeben sei folgende Implementierung einer Warteschlange. Es werden die Methode `enqueue` und `dequeue` für das Hinzufügen und Entfernen von Elementen entsprechend einer FIFO-Strategie angeboten. Die Elemente `enqueueLock` und `dequeueLock` dienen als feingranulare Lock-Objekte, damit gleichzeitig Elemente zur Warteschlange hinzugefügt und aus dieser entfernt werden können.

```
1  public class SynchronizedQueue {
2     private static class Entry {
3        final Object content;
4        Entry next;
5
6        Entry(Object content) {
7           this.content = content;
8        }
9     }
10
11    private Entry first = null;
12    private Entry last = null;
13    private final Object enqueueLock = new Object();
14    private final Object dequeueLock = new Object();
15
16    public void enqueue(Object object) {
17       synchronized(enqueueLock) {
18          Entry newEntry = new Entry(object);
19          if (this.last == null) {
20             this.first = newEntry;
21          } else {
22             this.last.next = newEntry;
23          }
24          this.last = newEntry;
25       }
26    }
27
28    /*@ private behavior
29      @ ensures \result == null ==> \old(first) == null;
30      @ ensures \old(first) == null ==> first == null;
31     */
32    public Object dequeue() {
33       synchronized(dequeueLock) {
34          Object returnedObject = null;
35          if (this.first != null) {
36             returnedObject = this.first.content;
37             this.first = this.first.next;
38             if (this.first == null) this.last = null;
39          }
40          return returnedObject;
41       }
42    }
43 }
```

(a) Kann bei der Verwendung des angegebenen Codes ein Deadlock auftreten? Begründen Sie Ihre Antwort. [2,5 Punkte]

(b) Nehmen Sie an, die Methode `enqueue` wird auf einer Instanz der Klasse `SynchronizedQueue` von mehreren, nebenläufigen Programmfäden aufgerufen. Kann es dabei zu einer Wettlaufsituation kommen? Begründen Sie Ihre Antwort. [2,5 Punkte]

(c) Nehmen Sie an, die Methoden `enqueue` und `dequeue` werden auf einer Instanz der Klasse `SynchronizedQueue` von mehreren, nebenläufigen Programmfäden in unbekannter Reihenfolge aufgerufen. Erläutern Sie kurz, warum dabei eine Wettlaufsituation vorliegt. Geben Sie eine Aufruf- und Ausführungsreihenfolge an, bei der dies zu unerwünschtem Verhalten führt. [5 Punkte]

(d) Betrachten Sie die folgenden Aufrufsequenz. Wird der Vertrag hier *vom Aufrufer* erfüllt? Begründen Sie Ihre Antwort kurz. [2 Punkte]

```
1 SynchronizedQueue queue = new SynchronizedQueue();
2 queue.enqueue(null);
3 queue.dequeue();
```

(e) Der Vertrag der Methode `dequeue` wird *vom Aufgerufenen* verletzt. Begründen Sie dies. [3 Punkte]
Hinweis: Sie müssen hierfür keine Wettlaufsituationen berücksichtigen.

Aufgabe 8 (Syntaktische Analyse) [19 Punkte]

Gegeben sei die folgende Grammatik, die eine Untermenge von SGML beschreibt:

$$\begin{aligned} \textit{SGML} &\rightarrow \texttt{<}\ \texttt{ident}\ \texttt{>}\ \textit{Children}\ \texttt{<}\ \texttt{/}\ \texttt{>} \\ \textit{Children} &\rightarrow \varepsilon \mid \textit{SGML}\ \textit{Children} \end{aligned}$$

Das Startsymbol dieser Grammatik ist *SGML*.

(a) Begründen Sie formal, warum die obige Grammatik nicht in SLL(1)-Form ist. [3 Punkte]

(b) Entwickeln Sie für die folgende, linksfaktorisierte SGML-Grammatik einen rekursiven Abstiegsparser mit AST-Aufbau in Pseudocode. [16 Punkte]

$$\begin{aligned} \textit{SGML} &\rightarrow \texttt{<}\ \texttt{ident}\ \texttt{>}\ \textit{ChildrenAndEnd} \\ \textit{ChildrenAndEnd} &\rightarrow \texttt{<}\ \textit{OpenOrClose} \\ \textit{OpenOrClose} &\rightarrow \texttt{/}\ \texttt{>} \\ &\mid \texttt{ident}\ \texttt{>}\ \textit{ChildrenAndEnd}\ \textit{ChildrenAndEnd} \end{aligned}$$

Der Parser von *SGML* soll ein Objekt der Klasse `SGML` zurückgeben. Diese ist folgendermaßen definiert:

```
1  class SGML {
2    public String tag;
3    public List<SGML> children;
4    public SGML(String tag, List<SGML> children) { ... }
5  }
```

Hinweis zum AST-Aufbau:
In der Produktion *OpenOrClose* → **ident >** *ChildrenAndEnd ChildrenAndEnd* parst das erste Vorkommen von *ChildrenAndEnd* alle Kinder des gerade geöffneten Tags, das zweite Vorkommen parst die restlichen Kinder des umgebenden Tags.

Lexer-Schnittstelle:
Die globale Variable `token` enthält immer das aktuelle Token. Tokens besitzen die folgenden Methoden:

- `getType()` gibt den Token-Typ zurück
- `getIdent()` gibt einen String zurück, der den Namen eines **ident**-Tokens enthält.

Folgende Token-Typen sind definiert:

`IDENT`	für einen Bezeichner **ident**
`LT`	für das Kleiner-Zeichen **<**
`GT`	für das Größer-Zeichen **>**
`SLASH`	für den Schrägstrich **/**

Die globale Methode `nextToken()` setzt `token` auf das nächste Token. Brechen Sie bei Parsefehlern durch Aufruf der globalen `error()`-Methode ohne Fehlermeldung ab.