Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation

Prof. Dr.-Ing. Gregor Snelting | Prof. Dr. Ralf H. Reussner

gregor.snelting@kit.edu | reussner@kit.edu

Klausur Programmierparadigmen — Beispiellösung

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 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
```

Beispiellösung:

```
prefix []     _      = True
prefix _      []     = False
prefix (x:xs) (y:ys) = x == y && prefix xs ys

substr [] _      = True
substr _  []     = False
substr xs (y:ys) = prefix xs (y:ys) || substr xs ys
```

Alternativ:

```
substr [] _  = True
substr _  [] = False
substr xs ys = take (length xs) ys == xs || substr xs (tail ys)
```

(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 ⊥ (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'` ≡ `"aaaa...`

	`xs` endlich	`xs` unendlich
`ys` endlich	`True` (`substr "bc" "abc"`), `False` (`substr "ac" "abc"`)	`False` (`substr as "a"`)
`ys` unendlich	`True` (`substr "a" as`), ⊥ (`substr "b" as`)	⊥ (`substr as as`)

Für die Alternativlösung: (Benutzung von `length` erhöht die Striktheit)

	`xs` endlich	`xs` unendlich
`ys` endlich	`True` (`substr "bc" "abc"`), `False` (`substr "ac" "abc"`)	$\perp$ (`substr as "a"`)
`ys` unendlich	`True` (`substr "a" as`), $\perp$ (`substr "b" as`)	$\perp$ (`substr as as`)

Für andere, prinzipiell korrekte Lösungen: entsprechend der Implementierung

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:

```haskell
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:

```haskell
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]

```haskell
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]

```haskell
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]

```haskell
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]

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

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

```haskell
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]
```

Beispiellösung:

```
fromList :: [a] -> Queue a
fromList xs = Q xs []

toList :: Queue a -> [a]
toList (Q front back) = front ++ reverse back

enqueue :: a -> Queue a -> Queue a
enqueue x (Q front back) = Q front (x:back)

dequeue :: Queue a -> Maybe (a, Queue a)
dequeue (Q [] []) = Nothing
dequeue (Q [] back) = dequeue (Q (reverse back) [])
dequeue (Q (x:front) back) = Just (x, Q front back)

dequeue2 :: Queue a -> Maybe (a, Queue a)
dequeue2 (Q [] []) = Nothing
dequeue2 (Q [] back) = Just (x, Q front [])
  where
    (x:front) = reverse back
dequeue2 (Q (x:front) back) = Just (x, Q front back)

dequeue3 :: Queue a -> Maybe (a, Queue a)
dequeue3 (Q [] []) = Nothing
dequeue3 (Q [] back) = Just (last back, Q [] (init back))
dequeue3 (Q (x:front) back) = Just (x, Q front back)

bfs :: Tree a -> [a]
bfs t = go (fromList [t])
  where
    go q = go2 (dequeue q)
    go2 Nothing = []
    go2 (Just (Leaf, q')) = go q'
    go2 (Just (Node l x r, q')) = x : go (enqueue r (enqueue l q'))

bfs2 :: Tree a -> [a]
bfs2 t = go (fromList [t])
  where
    go q = case dequeue q of
      Nothing -> []
      Just (Leaf, q') -> go q'
      Just (Node l x r, q') -> x : go (enqueue r (enqueue l q'))

bfs3 :: Tree a -> [a]
bfs3 t = go1 [t] []
  where
    go1 (x:xs) result = go1 (go2 x xs) (res x result)
    go1 [] result = result
```

```
go2 Leaf xs = xs
go2 (Node a x b) xs = xs ++ [a,b]
res Leaf xs = xs
res (Node a x b) xs = xs ++ [x]
```

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.
```

Beispiellösung:

```
schritt(Wort1, Wort2) :-
        append(Xs, [_|Ys], Wort1),
        buchstabe(Y),
        append(Xs, [Y|Ys], Wort2).
schritt(Wort1, Wort2) :-
        append(Xs, [_|Ys], Wort1),
        append(Xs, Ys, Wort2).
schritt(Wort1, Wort2) :-
        append(Xs, Ys, Wort1),
        buchstabe(Y),
        append(Xs, [Y|Ys], Wort2).
```

Alternativlösung:

```
schritt2(Wort, [X|Wort]) :- buchstabe(X).
schritt2([_|Wort], Wort).
schritt2([_|Wort], [X|Wort]) :- buchstabe(X).
schritt2([X|Wort1], [X|Wort2]) :- schritt2(Wort1, Wort2).

erreichbar(S,_,[S],S).
erreichbar(S,Besucht,[S|Woerter],Z) :-
        schritt(S,W), erlaubt(W), not(member(W,Besucht)),
        erreichbar(W,[W|Besucht],Woerter,Z).
```

Aufgabe 4 (Typsysteme) [15 Punkte]

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

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

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

Es seien:

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

Im Folgenden ist der Typherleitungsbaum für diesen Ausdruck abgebildet:

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

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

Beispiellösung:

$$\begin{aligned} \alpha_2 &= \alpha_3 \to \alpha_1 & \alpha_4 &= \alpha_{10} \\ \alpha_2 &= \alpha_4 \to \alpha_5 & \alpha_6 &= \alpha_{11} \\ \alpha_5 &= \alpha_6 \to \alpha_7 & \alpha_3 &= \alpha_{12} \to \alpha_{13} \\ \alpha_8 &= \alpha_9 \to \alpha_7 & \alpha_{13} &= \alpha_{14} \to \alpha_{15} \\ \alpha_4 &= \alpha_8 & \alpha_{12} &= \alpha_{15} \\ \alpha_{10} &= \alpha_{11} \to \alpha_9 \end{aligned}$$

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

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

Beispiellösung:

$$\begin{aligned} & \mathsf{c}_2\ \mathsf{c}_{\mathit{true}}\ \texttt{a b c} \\ = \quad & (\lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s (s z)})\ \mathsf{c}_{\mathit{true}}\ \texttt{a b c} \\ \Rightarrow^2 \quad & \mathsf{c}_{\mathit{true}}\ (\mathsf{c}_{\mathit{true}}\ \texttt{a})\ \texttt{b c} \\ = \quad & (\lambda\texttt{t}.\ \lambda\texttt{f}.\ \texttt{t})\ (\mathsf{c}_{\mathit{true}}\ \texttt{a})\ \texttt{b c} \\ \Rightarrow^2 \quad & \mathsf{c}_{\mathit{true}}\ \texttt{a c} \\ = \quad & (\lambda\texttt{t}.\ \lambda\texttt{f}.\ \texttt{t})\ \texttt{a c} \\ \Rightarrow^2 \quad & \texttt{a} \end{aligned}$$

(c) Im Gegensatz zu [3 Punkte]

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

ist der Ausdruck

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

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

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

verwendet.

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

Beispiellösung: $ta\,(\alpha \to \beta \to \alpha, \emptyset) = \forall\alpha.\ \forall\beta.\ \alpha \to \beta \to \alpha$
Auf der rechten Seite des Let-Ausdrucks können für beide Vorkommen von s unterschiedliche Instanziierungen des Typschemas verwendet werden.

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

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

$$\begin{aligned} \texttt{nil} &= \lambda\texttt{n}.\ \lambda\texttt{c}.\ \texttt{n} \\ \texttt{cons} &= \lambda\texttt{x}.\ \lambda\texttt{xs}.\ \lambda\texttt{n}.\ \lambda\texttt{c}.\ \texttt{c x xs} \end{aligned}$$

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]

$$\begin{aligned} \texttt{head (cons } A\ B\texttt{)} &\Rightarrow^* A \\ \texttt{tail (cons } A\ B\texttt{)} &\Rightarrow^* B \end{aligned}$$

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

Beispiellösung:
$\texttt{head} = \lambda\texttt{l}.\ \texttt{l}\ c_{false}\ (\lambda\texttt{x}.\ \lambda\texttt{xs}.\ \texttt{x})$
$\texttt{tail} = \lambda\texttt{l}.\ \texttt{l}\ c_{false}\ (\lambda\texttt{x}.\ \lambda\texttt{xs}.\ \texttt{xs})$

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

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

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

Beispiellösung: c_n hat Struktur $\underbrace{\lambda\texttt{s}.\lambda\texttt{z}.\ \texttt{s}\ (\ldots(\ \texttt{s z})\ldots)}_{n\ mal}$, sieht also schon *fast* aus wie der Ausdruck, den wir haben wollen. Es ergibt sich also:
$\texttt{replicate} = \lambda\texttt{n}.\lambda\texttt{x}.\ \texttt{n (cons x) nil}$

Alternativlösung: Natürlich kann man den Hinweis auch einfach ignorieren, und `replicate` ausprogrammieren:
$\texttt{replicate}' = \lambda\texttt{rep}.\lambda\texttt{n}.\lambda\texttt{x}.\ \texttt{isZero n nil (cons x (rep (sub n}\ c_1\texttt{) x))}$
$\texttt{replicate} \ = \texttt{Y replicate}'$

Das hat allerdings den Nachteil, dass Teilaufgabe (c) mehr Schreibarbeit wird. Unter Benutzung einer Abkürzung für $(\lambda\texttt{x}.\ \texttt{replicate}'\ \texttt{(x x)})\ (\lambda\texttt{x}.\ \texttt{replicate}'\ \texttt{(x x)})$ bleibt die Größe der involvierten Terme aber auch noch überschaubar.

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

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

Beispiellösung:
$\texttt{replicate}\ c_3\ \texttt{A}$
$= (\lambda\texttt{n}.\lambda\texttt{x}.\ \texttt{n (cons x) nil})\ c_3\ \texttt{A}$
$\Rightarrow^2\ c_3\ \texttt{(cons A) nil}$
$= (\lambda\texttt{s}.\lambda\texttt{z}.\ \texttt{s (s (s z))})\ \texttt{(cons A) nil}$
$\Rightarrow^2\ \texttt{(cons A) ((cons A) ((cons A) nil))}$
Lässt man die unnötigen Klammern weg, erhält man genau
$= \texttt{cons A (cons A (cons A nil))}$

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

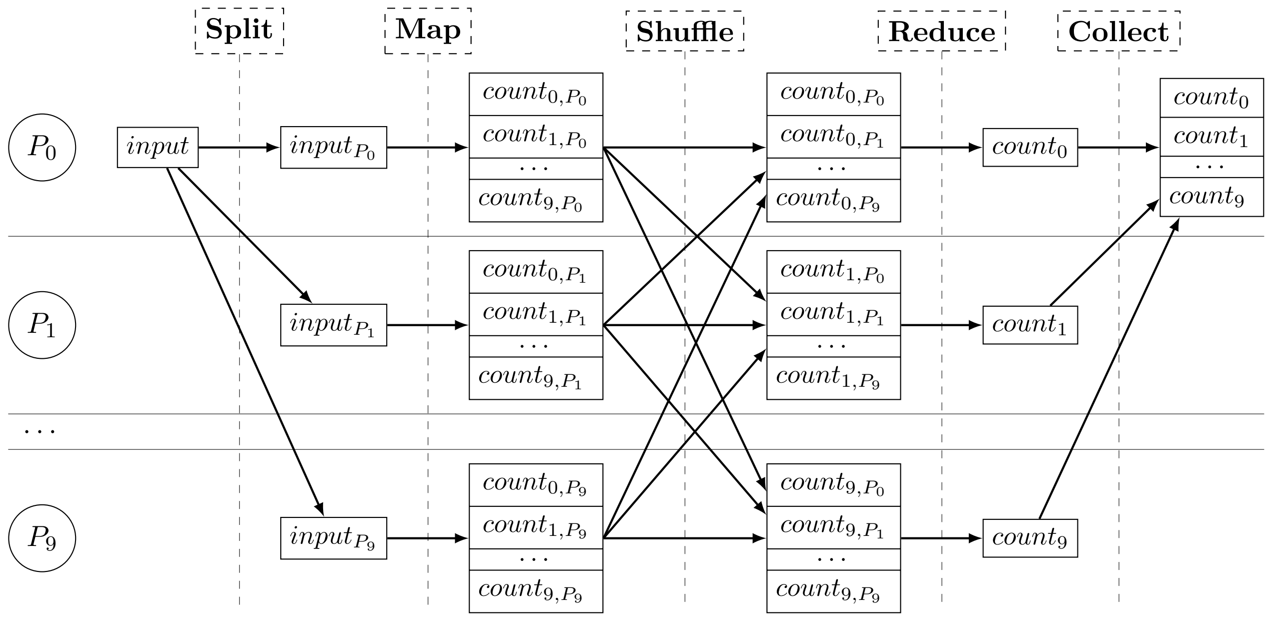
(a) In der untenstehenden Tabelle ist der Inhalt eines Arrays `sendbuf` auf 3 Prozessen dargestellt. Füllen Sie in der leeren Tabelle den Inhalt von `recvbuf` nach einem Aufruf der folgenden kollektiven MPI-Operation aus: [3 Punkte]

```
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 (Lösung)**
0	3, 1, 9
1	6, 3, 8
2	4, 2, 1

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	`MPI_Scatter`
Shuffle	`MPI_Alltoall`
Collect	`MPI_Gather`

(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]

```
int* countDigitsParallel(char *input) {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Split
    int digitsPerProcess = strlen(input) / size;
    char localSequence[digitsPerProcess];
    MPI_Scatter(input, digitsPerProcess, MPI_CHAR,
        localSequence, digitsPerProcess, MPI_CHAR, 0,
        MPI_COMM_WORLD);

    // Map
    int* digitsCounts = countDigits(localSequence);

    // Shuffle
    int localDigitCounts[10];
    MPI_Alltoall(digitsCounts, 1, MPI_INT, localDigitCounts, 1,
        MPI_INT, MPI_COMM_WORLD);

    // Reduce
    int summedCount = 0;
    for (int index = 0; index < size; index++) {
        summedCount += localDigitCounts[index]; 1
    }

    // Collect
    int totalDigitCounts[10];
    MPI_Gather(&summedCount, 1, MPI_INT, totalDigitCounts, 1,
        MPI_INT, 0, MPI_COMM_WORLD);

    // Please ignore that we return a pointer to stack memory
    return totalDigitsCounts;
}
```

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.

```java
public class SynchronizedQueue {
   private static class Entry {
      final Object content;
      Entry next;

      Entry(Object content) {
         this.content = content;
      }
   }

   private Entry first = null;
   private Entry last = null;
   private final Object enqueueLock = new Object();
   private final Object dequeueLock = new Object();

   public void enqueue(Object object) {
      synchronized(enqueueLock) {
         Entry newEntry = new Entry(object);
         if (this.last == null) {
            this.first = newEntry;
         } else {
            this.last.next = newEntry;
         }
         this.last = newEntry;
      }
   }

   /*@ private behavior
     @ ensures \result == null ==> \old(first) == null;
     @ ensures \old(first) == null ==> first == null;
    */
   public Object dequeue() {
      synchronized(dequeueLock) {
         Object returnedObject = null;
         if (this.first != null) {
            returnedObject = this.first.content;
            this.first = this.first.next;
            if (this.first == null) this.last = null;
         }
         return returnedObject;
      }
   }
}
```

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

Beispiellösung:

Es kann kein Deadlock auftreten, da immer nur auf einem Objekt synchronisiert wird und somit die notwendige Bedingung *hold-and-wait* nicht erfüllt ist.
Hinweis: Daraus resultiert, dass auch die Bedingung *circular wait* nicht erfüllt ist.

(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]

Beispiellösung:

Es kann zu keiner Wettlaufsituation kommen, da durch die Synchronisation auf `enqueueLock` immer nur ein Programmfaden den Methodenrumpf ausführen kann.

(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]

Beispiellösung:

Die Implementierungen der Methoden `enqueue` und `dequeue` synchronisieren auf verschiedenen Elementen, `enqueueLock` bzw. `dequeueLock`. Dies stellt sicher, dass nur jeweils eine `enqueue`- und nur eine `dequeue`-Operation zur gleichen Zeit ausgeführt werden können. Es ist jedoch möglich, dass gleichzeitig ein Aufruf von `enqueue` und `dequeue` ausgeführt wird, was beispielsweise problematisch ist, wenn sich vor den Aufrufen genau ein Objekt in der Warteschlage befand.

Beispielablauf: Es befindet sich ein Element in der Warteschlange. `dequeue` wird bis Mitte Zeile 38 (nach der Bedingungsprüfung) ausgeführt, `enqueue` wird vollständig ausgeführt, also bis Zeile 24. Schließlich wird der Rest von `dequeue` bis Zeile 40 ausgeführt. Danach zeigt `last` auf `null` statt auf das neue Element, wodurch beim nächsten Aufruf von `enqueue` das zuvor der Warteschlange hinzugefügte Element überschrieben wird.

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

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

Beispiellösung:

Ja, der Vertrag wird erfüllt. Die Methode spezifiziert keinerlei Vorbedingungen, die die Verwendung einschränken.
Hinweis: Keine Vorbedingungen sind gleichbedeutend mit der Angabe `@requires` **`true`**.

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

Beispiellösung:

Die erste Nachbedingung wird nicht erfüllt, da der Rückgabewert auch `null` sein kann, falls der Inhalt eines Eintrags `null` ist, da `enqueue` das nicht ausschließt. Somit ist die Implikation, dass vor dem Methodenaufruf `first == null` gegolten haben muss, nicht korrekt.

Aufgabe 8 (Syntaktische Analyse) [19 Punkte]

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

$$\begin{aligned} \mathit{SGML} &\to \texttt{\textbf{< ident >}}\ \mathit{Children}\ \texttt{\textbf{< / >}} \\ \mathit{Children} &\to \varepsilon \mid \mathit{SGML}\ \mathit{Children} \end{aligned}$$

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]

$$\begin{aligned} \mathit{SGML} &\to \texttt{\textbf{< ident >}}\ \mathit{ChildrenAndEnd} \\ \mathit{ChildrenAndEnd} &\to \texttt{\textbf{<}}\ \mathit{OpenOrClose} \\ \mathit{OpenOrClose} &\to \texttt{\textbf{/ >}} \\ &\mid\ \texttt{\textbf{ident >}}\ \mathit{ChildrenAndEnd}\ \mathit{ChildrenAndEnd} \end{aligned}$$

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

```
class SGML {
  public String tag;
  public List<SGML> children;
  public SGML(String tag, List<SGML> children) { ... }
}
```

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.

Beispiellösung:

(a) Es gilt $\text{Follow}_1(\mathit{Children}) = \{\texttt{<}\}$. Wir prüfen die SLL(1)-Bedingung anhand der Produktionen von *Children*:

$$\begin{aligned}
& \text{First}_1(\varepsilon \cdot \text{Follow}_1(\mathit{Children})) \cap \text{First}_1(\mathit{SGML} \cdot \mathit{Children} \cdot \text{Follow}_1(\mathit{Children})) \\
= \; & \text{Follow}_1(\mathit{Children}) \cap \text{First}_1(\mathit{SGML}) \\
= \; & \{\texttt{<}\} \cap \{\texttt{<}\} \\
\neq \; & \emptyset
\end{aligned}$$

Die Grammatik ist also nicht in SLL(1).

(b)

```
    void expect(TokenType tt) {
       if (token.getType() != tt) {
          error();
       }
       nextToken();
    }

    SGML parseSGML() {
       expect(LT);

       if (token.getType() != IDENT) {
          error();
       }
       String tag = token.getIdent();
       nextToken();

       expect(GT);

       List<SGML> children = parseChildrenAndEnd();

       return new SGML(tag, children);
    }

    List<SGML> parseChildrenAndEnd() {
       expect(LT);

       return parseOpenOrClose();
    }

    List<SGML> parseOpenOrClose() {
       switch (token.getType()) {
       case SLASH:
          nextToken();

          expect(GT);

          return new ArrayList<SGML>();

```

```
      case IDENT:
        String tag = token.getIdent();
        nextToken();

        expect(GT);

        List<SGML> children = parseChildrenAndEnd();

        SGML elementHere = new SGML(tag, children);

        List<SGML> siblings = parseChildrenAndEnd();
        siblings.add(0, elementHere);

        return siblings;

      default:
        error();
        return null;
      }
  }
```