

Sonderklausur Programmierparadigmen — Beispiellösung

SS20, 29. Juni 2020, 15:00 – 17:00 Uhr

Zugelassene Hilfsmittel: Papierbasierte Quellen (Vorlesungsfolien, Übungsblätter, eigene Aufzeichnungen, Bücher, ...)

Die Verwendung von elektronischen Geräten ist verboten.

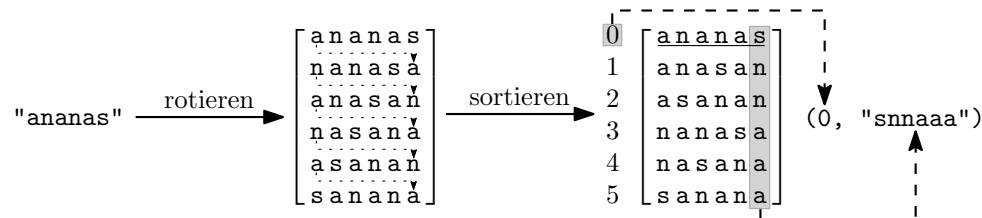
Bearbeitungszeit: 120 min

Aufgabe 1 (Haskell: Burrows-Wheeler-Transformation)

[12 Punkte]

Die *Burrows-Wheeler-Transformation* ordnet einen String mit ähnlichen Wörtern umkehrbar so um, dass gleiche Zeichen häufiger aufeinander folgen. Das Resultat lässt sich effizienter als das Original mit z.B. Lauflängenkodierung kodieren.

Dazu wird zu einem gegebenen String die Matrix all seiner Rotationen aufgestellt. Diese werden dann lexikographisch sortiert. Der Index des ursprünglichen Strings in der sortierten Matrix wird schließlich zusammen mit der letzten Spalte der Matrix weiter kodiert. Am Beispiel des Strings "ananas":



(a) Implementieren Sie die Funktion

[3 Punkte]

```
indexOf :: String -> [String] -> Int
```

die den Index eines Strings in einer Liste von Strings zurück gibt. Sie dürfen als Vorbedingung annehmen, dass der String in der übergebenen Liste auftaucht.

(b) Implementieren Sie die Funktion

[4 Punkte]

```
rotations :: [a] -> [[a]]
```

die zu einer nicht-leeren Liste der Länge n die Liste aller ihrer n Rotationen bestimmt.

Hinweis: Nutzen Sie `iterate` mit einer geeigneten Iterationsvorschrift und `take`, um von der resultierenden unendlichen Liste die ersten n Elemente zurück zu geben.

```
> rotations "abc"
["abc", "bca", "cab"]
```

(c) Implementieren Sie die Funktion

[5 Punkte]

```
transform :: String -> (Int, String)
```

die für einen nicht-leeren String die letzte Spalte seiner Burrows-Wheeler-Matrix und den Zeilenindex des gegebenen Strings in der Matrix bestimmt.

Hinweis: Auf `[String]` sortiert `Data.List.sort` (das Sie in dieser Aufgabe benutzen dürfen) lexikographisch. Nutzen Sie auch die Funktionen aus den vorherigen Teilaufgaben.

```
> transform "ananas"
(0, "snnaaa")
```

Beispiellösung:

```
indexOf :: String -> [String] -> Int
indexOf s (s':ss)
  | s == s'    = 0
  | otherwise  = 1 + indexOf s ss
```

```
indexOf2 :: String -> [String] -> Int
```

```
indexOf2 s ss = length (takeWhile (/= s) ss)

rotations :: [a] -> [[a]]
rotations xs = take (length xs) (iterate rot xs)
  where
    rot (x:xs) = xs ++ [x]

transform :: String -> (Int, String)
transform s = (k, col)
  where
    k      = indexOf s matrix
    matrix = sort (rotations s)
    col    = map last matrix
```

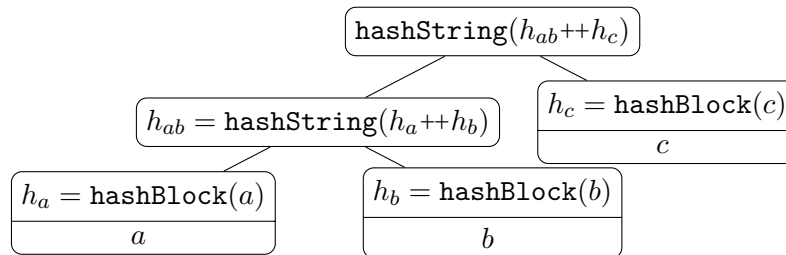
Aufgabe 2 (Haskell, Hashbäume)

[18 Punkte]

Ein *Hashbaum* ist ein Binärbaum, in dessen Knoten je ein Hashwert gespeichert ist, und dessen Blätter zusätzlich noch einen Datenblock besitzen. Der Hashwert eines Elternknotens berechnet sich als der Hash der konkatenierten Hashes seiner Kinder. Der Hashwert eines Blattknotens ist der Hashwert seines Blocks.

Im Folgenden dürfen Sie annehmen, dass Hashwerte vom Typ **String** und Blöcke vom Typ **Block** sind, und dass es die Hashfunktionen **hashString :: String -> String** und **hashBlock :: Block -> String** gibt.

Beispiel: Ein Hashbaum für die Blockliste `[a, b, c]` kann wie folgt aussehen:



- (a) Definieren Sie einen Datentyp **Tree** für Hashbäume, nennen Sie die inneren Knoten **Node** und die Blattknoten **Leaf**. [2 Punkte]

Beispiellösung:

```
data Tree = Node String Tree Tree | Leaf String Block
```

Im Folgenden dürfen Sie die Funktion **mkNode :: Tree -> Tree -> Tree** verwenden, welche für zwei Knoten den Elternknoten erzeugt.

- (b) Definieren Sie die Funktion **getHash :: Tree -> String**, die für einen Knoten des Baumes den darin gespeicherten Hashwert zurückgibt. [1 Punkt]

Beispiellösung:

```
getHash (Node s _ _) = s
getHash (Leaf s _ ) = s
```

- (c) Definieren Sie die Funktion **mkLeaf :: Block -> Tree** welche für einen Block einen Blattknoten erzeugt. [1 Punkt]

Beispiellösung:

```
mkLeaf b = Leaf (hashBlock b) b
```

- (d) Definieren Sie die Funktion **fromBlocks :: [Block] -> Tree** die für eine nicht-leere Liste von Blöcken einen Baum erzeugt, der diese in seinen Blättern enthält. Der Baum muss nicht balanciert sein. [3,5 Punkte]

Hinweis: Die Funktion **mkNode** könnte nützlich sein.

Beispiellösung:

```
fromBlocks [x]      = mkLeaf x
fromBlocks (x:xs) = mkNode (mkLeaf x) (fromBlocks xs)
— Alternativlösung
fromBlocks bs      = foldr mkNode 1 ls
  where (l:ls) = map mkLeaf bs
```

— *Alternativlösung*

```
fromBlocks bs = foldl mkNode 1 ls
  where (l:ls) = map mkLeaf bs
```

- (e) Definieren Sie die Funktion **check :: Tree -> Bool** die für einen gegebenen Baum zurückgibt, ob er ein valider Hashbaum ist, d.h. die Hashwerte aller Knoten stimmen. [3,5 Punkte]

Beispiellösung:

```
check (Node s l r) = check l && check r && getHash (mkNode l r) == s
check (Leaf s b ) = hashBlock b == s
```

Im Folgenden geht es um die Erzeugung eines vollständig *balancierten* Baums für eine Liste von Knoten, deren Länge einer Zweierpotenz und damit nicht leer ist. Ein vollständig balancierter Baum ist hier ein Baum, bei dem für jeden Elternknoten gilt, dass seine beiden Unterbäume gleich groß sind.

- (f) Definieren Sie die Funktion **condense :: [Tree] -> [Tree]**, welche in einer Liste von Bäumen benachbarte Listenelement zusammenfügt. Die übergebene Liste hat gerade Länge, die resultierende Liste soll den Elternknoten für den 1. und den 2. Knoten, für den 3. und den 4., ... enthalten. [3 Punkte]

Beispiellösung:

```
condense [] = []
condense (x:y:xs) = (mkNode x y):(condense xs)
```

- (g) Definieren Sie die Funktion **merge :: [Tree] -> Tree**, die eine Liste von Teilbäumen zu einem balancierten Hashbaum zusammenfügt. Die Eingabe besteht aus 2^n vielen balancierten Bäumen, welche alle die gleiche Höhe haben. [2 Punkte]

Beispiellösung:

```
merge [x] = x
merge xs = merge (condense xs)
```

- (h) Definieren Sie die Funktion **fromBlocks2 :: [Block] -> Tree**, die für eine Liste von Blöcken einen balancierten Hashbaum erzeugt. Die Länge der Liste ist wieder eine Zweierpotenz. [2 Punkte]

Beispiellösung:

```
fromBlocks2 bs = merge (map mkLeaf bs)
```

Aufgabe 3 (Prolog, SAT-Solver)

[15 Punkte]

In dieser Aufgabe sollen Sie einen einfachen SAT-Solver in Prolog entwickeln. Das zu lösende SAT-Problem stellen wir in konjunktiver Normalform dar. Dabei codieren wir ...

- ... Literale als `pos(V)` (nicht-negiert) oder `neg(V)` (negiert), also z.B. a als `pos(a)` und $\neg b$ als `neg(b)`.
- ... Klauseln als Liste von Literalen, also z.B. $(a \vee \neg b)$ als `[pos(a), neg(b)]`.
- ... das SAT-Problem als Liste von Klauseln, also z.B. $(a \vee \neg b) \wedge (\neg c \vee d \vee e) \wedge (\neg f)$ als `[[pos(a), neg(b)], [neg(c), pos(d), pos(e)], [neg(f)]]`.

Das Prädikat `sat(P)` prüft, ob das SAT-Problem P eine Lösung besitzt:

`sat(P) :- solve(P, []).`

Das Prädikat `solve(P, A)` soll genau dann erfüllbar sein, wenn das SAT-Problem P eine Lösung besitzt, wobei die Variablenbelegung in A vorgegeben ist. A ist eine Liste von Literalen. Ein Vorkommen von `pos(V)` in A soll bedeuten, dass die Variable V mit „wahr“ belegt ist, entsprechend steht `neg(V)` für die Belegung mit „falsch“.

- (a) Implementieren Sie zunächst das Prädikat `contradict(A)`. Es soll genau dann [4 Punkte]
erfüllbar sein, wenn die Variablenbelegung A widersprüchlich ist, also eine Variable
sowohl mit „wahr“ als auch mit „falsch“ belegt ist.

- (b) Implementieren Sie nun das Prädikat `solve(P, A)`. [11 Punkte]

Gehen Sie dabei folgendermaßen vor:

- Das leere SAT-Problem ($P = []$) ist unabhängig von der vorgegebenen Variablenbelegung erfüllbar.
- Falls P nicht das leere SAT-Problem ist:
 - Rate ein Literal L aus der ersten Klausel von P .
 - Füge L zur vorgegebenen Variablenbelegung hinzu.
 - Prüfe, dass die neue vorgegebene Variablenbelegung nicht widersprüchlich ist.
Bei Widerspruch rate ein anderes Literal.
 - Löse die restlichen Klauseln von P mit der neuen vorgegebenen Variablenbelegung.
Sind sie unlösbar, rate ein anderes Literal.

Hinweis: Zum Raten eines anderen Literals können Sie sich auf Backtracking verlassen.

Beispiellösung:

(a) `contradict(A) :-
 member(pos(V), A),
 member(neg(V), A).`

/ Alternative Lösung */*

```
contradict([pos(V)|A]) :- member(neg(V), A).  
contradict([neg(V)|A]) :- member(pos(V), A).  
contradict(_|A) :- contradict(A).
```

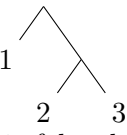
(b) `solve([], _).`

```
solve([Clause|Rest], Assignments) :-  
    member(Literal, Clause),  
    NewAssignments = [Literal|Assignments],  
    not(contradict(NewAssignments)),  
    solve(Rest, NewAssignments).
```

Aufgabe 4 (λ -Kalkül, Binärbäume)

[12 Punkte]

In dieser Aufgabe betrachten wir Binärbäume mit natürlichzahligen Blattinhalten. Wie auch andere Datenstrukturen können solche Bäume direkt als λ -Terme kodiert werden: So entspricht beispielsweise

der Baum  dem Baum-Term $\text{node } (\text{leaf } c_1) (\text{node } (\text{leaf } c_2) (\text{leaf } c_3))$, wobei leaf und node wie folgt definiert sind:

$\text{leaf} \equiv \lambda n. (\lambda b. \lambda r. b \ n)$
 $\text{node} \equiv \lambda t_l. \lambda t_r. (\lambda b. \lambda r. r \ (t_l \ b \ r) \ (t_r \ b \ r))$

leaf akzeptiert also zuerst den Blattinhalt, und node den linken und rechten Teilbaum. Danach akzeptieren beide Funktionen zwei Parameter b und r . b wird auf alle Blattinhalte angewendet, und r rekursiv auf die Ergebnisse von Teilbäumen. Durch geschickte Wahl von b und r lassen sich viele Operationen auf solchen Bäumen direkt ausdrücken.

- (a) Geben Sie einen vollständig reduzierten Term für den Baum-Term [4 Punkte]
 $\text{node } (\text{leaf } c_1) (\text{node } (\text{leaf } c_2) (\text{leaf } c_3))$ von oben an.

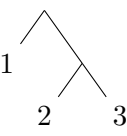
Hinweis: Sie müssen keine Zwischenschritte angeben. Die Church-Zahlen c_n brauchen Sie nicht zu reduzieren.

Beispiellösung:

$\lambda b. \lambda r. r \ (b \ c_1) \ (r \ (b \ c_2) \ (b \ c_3))$

- (b) Definieren Sie einen λ -Term sum , sodass $\text{sum } t$ die Summe aller Blattinhalte des Baum-Terms t berechnet. [3 Punkte]

Hinweis: Machen Sie sich die Struktur eines Baum-Terms zu Nutze. Sie benötigen keinen Fixpunktkombinator.

Beispiel: Die Summe des Baums  ist 6.

Beispiellösung:

$\text{sum} \equiv \lambda t. t \ (\lambda n. n) \ \text{plus}$

- (c) Definieren Sie einen λ -Term rev , sodass $\text{rev } t$ einen Baum-Term t in einen umgekehrten Baum-Term umformt, bei dem also die Kinder jedes Knotens vertauscht sind. [5 Punkte]

Hinweis: Auch hier benötigen Sie keinen Fixpunktkombinator. leaf und node könnten nützlich sein.

Beispiel: Die Umkehrung des Baums  ist .

Beispiellösung:

$\text{rev} \equiv \lambda t. t \ \text{leaf} \ (\lambda t_l. \lambda t_r. \text{node } t_r \ t_l)$

Alternativ:

$\text{rev} \equiv \lambda t. \lambda b. \lambda r. t \ (\lambda n. b \ n) \ (\lambda t_l. \lambda t_r. r \ t_r \ t_l)$

Name:

Matrikelnummer:

Aufgabe 5 (Typinferenz)

[11 Punkte]

Diese Aufgabe beschäftigt sich mit der Typherleitung für den Lambda-Term

$$\mathbf{let} \ b = \lambda f. \lambda g. \lambda x. \ f \ (g \ x) \ \mathbf{in} \ \lambda f. \lambda x. \ b \ f \ (b \ x)$$

Es sei $\Gamma' = \Gamma, f : \alpha_2, x : \alpha_4$.

Gegeben ist das folgende Skelett des Herleitungsbaums:

$$\begin{array}{c}
 \text{Var} \frac{\textcircled{1}}{\Gamma' \vdash b : \alpha_8} \quad \text{Var} \frac{\textcircled{2}}{\Gamma' \vdash f : \alpha_9} \quad \text{Var} \frac{\textcircled{3}}{\Gamma' \vdash b : \alpha_{10}} \quad \text{Var} \frac{\textcircled{4}}{\Gamma' \vdash x : \alpha_{11}} \\
 \text{App} \frac{\quad}{\Gamma' \vdash b \ f : \alpha_6} \quad \text{App} \frac{\quad}{\Gamma' \vdash b \ x : \alpha_7} \\
 \text{App} \frac{\quad}{\Gamma' \vdash b \ f \ (b \ x) : \alpha_5} \\
 \text{Abs} \frac{\quad}{\Gamma, f : \alpha_2 \vdash \lambda x. \ b \ f \ (b \ x) : \alpha_3} \\
 \text{Abs} \frac{\vdots}{\vdash \dots : \beta_1} \\
 \text{Let} \frac{\quad}{\vdash \mathbf{let} \ b = \lambda f. \lambda g. \lambda x. \ f \ (g \ x) \ \mathbf{in} \ \lambda f. \lambda x. \ b \ f \ (b \ x) : \alpha}
 \end{array}$$

Aus dem linken (nicht dargestellten) Teilbaum ergibt sich der allgemeinste Unifikator σ_{let} und es gilt $\sigma_{let}(\beta_1) = (\beta_9 \rightarrow \beta_7) \rightarrow (\beta_6 \rightarrow \beta_9) \rightarrow \beta_6 \rightarrow \beta_7$.

(a) Berechnen Sie Γ .

[2 Punkte]

Beispiellösung:

$$\begin{aligned}
 \Gamma &= b : ta(\sigma_{let}(\beta_1), \emptyset) \\
 &= b : ta((\beta_9 \rightarrow \beta_7) \rightarrow (\beta_6 \rightarrow \beta_9) \rightarrow \beta_6 \rightarrow \beta_7, \emptyset) \\
 &= b : \forall \beta_6. \forall \beta_7. \forall \beta_9. (\beta_9 \rightarrow \beta_7) \rightarrow (\beta_6 \rightarrow \beta_9) \rightarrow \beta_6 \rightarrow \beta_7
 \end{aligned}$$

(b) Ergänzen Sie, was an den mit ① bis ④ markierten Stellen einzutragen ist.

[4 Punkte]

Beispiellösung:

① $\Gamma'(b) = \dots,$

$\Gamma'(b) \succeq (\alpha_{12} \rightarrow \alpha_{13}) \rightarrow (\alpha_{14} \rightarrow \alpha_{12}) \rightarrow \alpha_{14} \rightarrow \alpha_{13}$

② $\Gamma'(f) = \alpha_2, \alpha_2 \succeq \alpha_2$

③ $\Gamma'(b) = \dots,$

$\Gamma'(b) \succeq (\alpha_{15} \rightarrow \alpha_{16}) \rightarrow (\alpha_{17} \rightarrow \alpha_{15}) \rightarrow \alpha_{17} \rightarrow \alpha_{16}$

④ $\Gamma'(x) = \alpha_4, \alpha_4 \succeq \alpha_4$

(c) Geben Sie nun die Constraintmenge an, die aus den *Abs*- und *App*-Regeln im rechten Teilbaum der *Let*-Regel entsteht. [5 Punkte]

Beispiellösung:

$$C = \left\{ \begin{array}{l} \alpha_1 = \alpha_2 \rightarrow \alpha_3, \\ \alpha_3 = \alpha_4 \rightarrow \alpha_5, \\ \alpha_6 = \alpha_7 \rightarrow \alpha_5, \\ \alpha_8 = \alpha_9 \rightarrow \alpha_6, \\ \alpha_{10} = \alpha_{11} \rightarrow \alpha_7, \\ \end{array} \right\}$$

Aufgabe 6 (MPI)

[14 Punkte]

Gegeben ist untenstehende Methode `countDigitOccurrences`. Übergeben wird eine Sequenz von `int`-Ganzzahlen mittels des Parameters `digitSequence` und deren Anzahl an Elementen mittels des Parameters `sequenceLength`. Der Wert einer Zahl in `digitSequence` kann 0, 1 oder 2 sein. Es steht die Methode **`int* countDigits(int sequence[], int length)`** zur Verfügung. Diese Methode ermittelt die Anzahl der Vorkommen der Zahlen 0, 1 und 2 in dem übergebenen Array `sequence`, welches `length` Elemente beinhaltet. Das Ergebnis von `countDigits` ist ein Array bei dem die Anzahl der Vorkommen der jeweiligen Zahl an ihrem korrespondierenden Index steht. Beispielsweise ist die Rückgabe für **`countDigits({0,2,2,2,0,1}, 6)`** das Array `{2,1,3}`.

Gehen Sie davon aus, dass der Root-Prozess mit dem rank 0 die zu bearbeitende `int`-Sequenz durch den Parameter `digitSequence` erhält und der Parameter `sequenceLength` die korrekte Anzahl an Elementen in `digitSequence` beinhaltet. Gehen Sie weiterhin davon aus, dass das MPI-Programm vom Aufrufer initialisiert wird, das Programm mit *exakt* 3 Prozessen ausgeführt wird und die Länge der Eingabesequenz `digitSequence` ein Vielfaches der Anzahl der gestarteten Prozesse (3) ist.

```
1  int* countDigitOccurrences(int digitSequence[], int sequenceLength) {
2      int size, rank;
3      int sum = 0;
4      int sortedCounts[3];
5      int total[3];
6
7      MPI_Comm_size(MPI_COMM_WORLD, &size);
8      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9
10     int digitsPerProcess = sequenceLength / size;
11     int localSequence[digitsPerProcess];
12
13     MPI_Scatter(digitSequence, digitsPerProcess, MPI_INT, localSequence,
14                digitsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);
15
16     int* localCounts = countDigits(localSequence, digitsPerProcess);
17
18     MPI_Alltoall(localCounts, 1, MPI_INT, sortedCounts, 1, MPI_INT,
19                 MPI_COMM_WORLD);
20
21     for (int index = 0; index < size; index++){
22         sum += sortedCounts[index];
23     }
24
25     MPI_Gather(&sum, 1, MPI_INT, total, 1, MPI_INT, 0, MPI_COMM_WORLD);
26
27     //Ignore that we return a pointer to stack memory
28     return total;
29 }
```

- (a) Gegeben sind für jede verwendete kollektive MPI-Operation des zuvor gegebenen Quellcodes jeweils zwei Tabellen. Die linken Tabellen stellen den Inhalt des Sendebuffers und die rechten Tabellen den Inhalt des Empfangs-Buffers *nach* Ausführung der kollektiven MPI-Operationen dar. [5 Punkte]

In der zweiten Spalte der Kopfzeile jeder Tabelle ist der Name des als Buffer verwendeten Arrays oder Variablen angegeben. Die zweite Spalte einer Zeile der Tabellen beschreiben den Inhalt des Buffers auf dem Prozess mit dem zu *Prozess Nr.* korrespondierenden *rank*. In der ersten Tabelle (Sende Buffer der Operation `MPI_Scatter`) ist der Inhalt des Arrays `digitSequence` für den Prozess mit dem *rank* 0 gegeben.

Ergänzen Sie den Inhalt der restlichen Tabellen auf Basis der ersten Tabelle. Falls ein Prozess bei einer kollektiven MPI-Operation keine Elemente von dem Sendebuffer versendet oder in den Empfangs-Buffer empfängt, kann die Zeile der entsprechenden Tabelle leer bleiben.

MPI_Scatter (Zeile 13)			
Prozess Nr.	digitSequence (Lösung)	Prozess Nr.	localSequence (Lösung)
0	0, 2, 0, 2, 2, 2, 1, 1, 0	0	0, 2, 0
1	—	1	2, 2, 2
2	—	2	1, 1, 0

MPI_Alltoall (Zeile 17)			
Prozess Nr.	localCounts (Lösung)	Prozess Nr.	sortedCounts (Lösung)
0	2, 0, 1	0	2, 0, 1
1	0, 0, 3	1	0, 0, 2
2	1, 2, 0	2	1, 3, 0

MPI_Gather (Zeile 23)			
Prozess Nr.	sum (Lösung)	Prozess Nr.	total (Lösung)
0	3	0	3, 2, 4
1	2	1	—
2	4	2	—

Beispiellösung: *Siehe Tabellen*

- (b) Geben Sie einen Ersatz der Zeilen 17 bis einschließlich 23 an, welcher aus dem Aufruf einer kollektiven MPI-Operation besteht und nach deren Ausführung das gleiche Ergebnis in `total` enthalten ist. [5 Punkte]

Beispiellösung:

```
MPI_Reduce(localCounts, total, 3, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

-
- (c) In der gegebenen Implementierung besteht die Einschränkung, dass die Anzahl der Elemente in `digitSequence` ein Vielfaches der Anzahl von gestarteten Prozessen sein muss. Erklären Sie kurz, wodurch sich die Notwendigkeit für diese Beschränkung im Bezug auf die gegebene Implementierung ergibt. Beschreiben Sie stichpunktartig, wie Sie die Implementierung ändern müssen, damit das Programm auch ohne diese Einschränkung korrekte Ergebnisse liefert. Gehen Sie jedoch weiterhin davon aus, dass das Programm mit 3 Prozessen gestartet wird. [4 Punkte]

Beispiellösung: Die Einschränkung ergibt sich nur durch die Anwendung von `MPI_Scatter` in Zeile 13, die die restlichen kollektiven MPI-Operationen immer auf Arrays fixer Länge (3) arbeiten. `MPI_Scatter` kann nur Blöcke gleicher Größe an alle Prozesse verteilen. Hierdurch ergibt sich die Einschränkung an die Größe von `input` auf ein Vielfaches der Anzahl an Prozessen, da nur so alle `int`-Zahlen in `input` korrekt auf die Prozesse verteilt werden können. Um diese Einschränkung zu beheben, kann die Vektor-Variante `MPI_Scatterv` verwendet werden. `MPI_Scatterv` ermöglicht es für jeden Empfänger-Prozess eine unterschiedliche Anzahl an Elementen zu definieren welche dieser zugesendet bekommen soll. Die Parametrisierung von `MPI_Scatterv` unterscheidet sich hierbei, unter anderem, dadurch von `MPI_Scatter`, dass anstatt einer festen Zahl der zu versendenden Elemente für alle Prozesse, ein Array mit einer Anzahl an zu versendenden Elementen für jeden Prozess übergeben werden kann. Somit muss für jeden Prozess zunächst eine Anzahl an Elementen berechnet werden, so dass die Summe der Anzahlen die Länge von `input` (`inputLength`) ergibt.

Name:

Matrikelnummer:

Aufgabe 7 (Java und Design by Contract)

[16 Punkte]

Gegeben sei folgende sequenzielle Implementierung der Methode `calculateMax`, welche den größten Wert einer Sequenz von `Integer`-Zahlen berechnet. In einem Vorverarbeitungsschritt wird die Sequenz in disjunkte Blöcke aufgeteilt und über den Parameter `blocksOfNumbers` der Methode übergeben. Für jeden Block wird der größte Wert berechnet und folgend der größte Wert über die Ergebnisse aller Blöcke bestimmt. Der größte Wert einer Folge von `Integer`-Zahlen wird mittels der ebenfalls bereitgestellten Methode `findMax` berechnet.

Gehen Sie in den folgenden Aufgaben davon aus, dass alle Datenstrukturen und `Integer`-Objekte in dem Parameter `blocksOfNumbers` instanziiert sind, weswegen Sie das Auftreten und Behandeln von `null`-Werten nicht berücksichtigen müssen.

```
1 public class MaxOfMax {
2
3     public int calculateMax(Collection<List<Integer>> blocksOfNumbers) {
4
5         if(blocksOfNumbers.size() == 0){
6             return Integer.MIN_VALUE;
7         }
8
9         List<Integer> results = new ArrayList<>();
10        for(List<Integer> block : blocksOfNumbers){
11            results.add(findMax(block))
12        }
13
14        return findMax(results);
15    }
16
17    /*
18     * @ requires numbers.size() > 0;
19     * @ ensures \result > Integer.MIN_VALUE;
20     */
21    private Integer findMax(Collection<Integer> numbers){
22
23        Integer maxValue = Integer.MIN_VALUE;
24        for(Integer number : numbers) {
25            if (number > maxValue) {
26                maxValue = number;
27            }
28        }
29
30        return maxValue;
31    }
32 }
```


Parallelverarbeitung in Java (10 Punkte)

- (a) Ergänzen Sie untenstehenden Quellcode, sodass er als parallelisierter asynchroner Ersatz für Zeile 10 bis einschließlich 12 in der Methode `calculateMax` dienen kann. Hierbei soll die Berechnung des größten Wertes jedes Blocks in `blocksOfNumbers` mittels paralleler asynchroner Berechnung unter Verwendung von `findMax` erfolgen und die Ergebnisse in `results` geschrieben werden. Verwenden Sie für die Erfüllung der Aufgabe einen `ExecutorService` und `Futures`. Das Auftreten von `Exceptions` muss nicht beachtet werden.
Hinweis: Mittels der Methode `Executors::newCachedThreadPool` erhalten Sie einen `ExecutorService` bei dem die Erzeugung und Anzahl der Threads vom `ExecutorService` automatisch vorgenommen wird. [6 Punkte]

```
//... List<Integer> results = new ArrayList<>();

List<Future<Integer>> futures = new ArrayList<>();
ExecutorService executor = Executors.newCachedThreadPool();
for(List<Integer> numberBlock : blocksOfNumbers){
    futures.add(executor.submit(() -> {return findMax(numberBlock);}));
}

for(Future<Integer> future : futures){
    results.add(future.get());
}

executor.shutdown();

//... return findMax(results);
```

Beispiellösung: siehe Code

- (b) Wird für obige parallele Implementierung (Teilaufgabe (a)) für `calculateMax` Task- oder Daten-Parallelismus verwendet? Begründen Sie Ihre Antwort kurz. [2 Punkte]

Beispiellösung: Es wird Datenparallelismus verwendet. Die Daten werden in der Vorverarbeitung in unabhängige Blöcke aufgeteilt. Jedem Task der in den `ExecutorService` per `submit` hinzugefügt wird, erhält einen Block und führt darauf die gleiche Methode (`findMax`) parallel aus.

- (c) Geben Sie für die Methode `findMax` eine Ersatzimplementierung mittels Java Streams an, welche die bestehende Implementierung gleichwertig ersetzt. Füllen Sie hierfür den unten gegebenen Methodenrumpf aus. [3 Punkte]
Hinweis: Die Methode `IntStream::max` gibt das im Wert größte Element des Streams über ein Objekt des Typs `OptionalInt` zurück. `OptionalInt` kann einen oder kann keinen `int`-Wert beinhalten. Die Methode `OptionalInt::orElse` mit dem Parameter `int other` gibt den gekapselten `int`-Wert oder, falls kein `int`-Wert vorhanden ist, den Wert des Parameters `other` zurück. Die Methode `Integer::intValue` gibt den Wert eines `Integer`-Objekts als primitiven `int`-Datentyp zurück.

```
public Integer findMax(Collection<Integer> numbers){  
  
    return numbers.parallelStream().  
    mapToInt(x -> x.intValue()).max().orElse(Integer.MIN_VALUE);  
}
```

Beispiellösung: *siehe Code*

Design by Contract (5 Punkte)

Beachten Sie nun den JML-Vertrag der Methode `findMax` der Klasse `MaxOfMax` der zuvor gegebenen sequentiellen Implementierung in Aufgabe 7.

- (d) Wird der Vertrag von `findMax` vom *Aufrufer* in Zeile 14 erfüllt? Begründen Sie [2,5 Punkte] Ihre Antwort.

Beispiellösung: Ja, der Vertrag wird vom Aufrufer erfüllt. Die Methode `calculateMax` überprüft zunächst, dass `blocksOfNumbers` nicht leer ist (Zeile 5 bis einschließlich 7). Somit wird `findMax` in Zeile 14 nur aufgerufen, wenn mindestens ein Block vorhanden ist, wodurch mindestens ein Wert in Zeile 11 zu `results` hinzugefügt wird.

- (e) Wird der Vertrag von `findMax` vom *Aufgerufenen* erfüllt? Begründen Sie Ihre [2,5 Punkte] Antwort.

Beispiellösung: Nein, der Vertrag wird nicht erfüllt. Selbst wenn die Vorbedingung erfüllt ist, kann in der übergebenen `Collection numbers` nur der Wert `Integer.MIN_VALUE` enthalten sein. Hierdurch kann das Resultat `Integer.MIN_VALUE` sein, obwohl die übergebene `Collection numbers` mindestens ein Element beinhaltet. Da die Nachbedingung in Zeile 19 fordert, dass das Ergebnis echt-größer ($>$) `Integer.MIN_VALUE` ist, ist diese nicht erfüllt.

Name:

Matrikelnummer:

Aufgabe 8 (~~λ -Kalkül~~ Compiler)

[22 Punkte]

Textuell dargestellte λ -Ausdrücke können grob angesehen werden, als werden sie von der folgenden Grammatik generiert:

$$\begin{aligned} L &\rightarrow (L) \mid LL \mid \lambda \text{ id} . L \mid \text{Var} \\ \text{Var} &\rightarrow \text{id} \end{aligned}$$

Leider ist diese Grammatik zum Parsen ungeeignet: Sie ist mehrdeutig, und die meisten generierten Ableitungsbäume sind strukturell falsch! Diese Mängel können aber ähnlich wie bei der Expression-Grammatik behoben werden.

- (a) Stellen Sie eine bessere Grammatik für das λ -Kalkül auf! Ihre Grammatik soll die gleiche Sprache akzeptieren, eindeutig sein und Assoziativität und Operatorprioritäten respektieren. Das Startsymbol Ihrer Grammatik soll L heißen. [6 Punkte]

Hinweise:

- λ -Abstraktion ist ein unärer Operator auf λ -Ausdrücken mit Operatorsymbol “ $\lambda \text{ id} .$ ”. Geschachtelte Abstraktionen müssen nicht mit Klammern getrennt werden.
- Abstraktion hat niedrigste Priorität, gefolgt von Applikation, gefolgt von Atomen/Klammerung. (D.h. die Ableitung von “ $\lambda x . x \ x$ ” produziert zuerst eine Abstraktion!)
- Applikation ist linksassoziativ! D.h. bei “ $x \ x \ x$ ” ist eine Applikation linkes Kind einer anderen Applikation.

Beispiellösung: Zunächst führen wir drei Nichtterminale *Abstr*, *App* und *Atom* für die Prioritätsstufen ein. Jedes dieser Nichtterminale kann entweder zu Syntaxelementen dieser Priorität abgeleitet werden (für Operatoren: zu beliebig vielen, via rekursiven Ableitungen), oder zum nächst-höher-priorisierten Nichtterminalen.

$$\begin{aligned} L &\rightarrow \text{Abstr} \\ \text{Abstr} &\rightarrow \lambda \text{ id} . \text{Abstr} \mid \text{App} \\ \text{App} &\rightarrow \text{App} \text{ Atom} \mid \text{Atom} \\ \text{Atom} &\rightarrow \text{Var} \mid (\text{Abstr}) \end{aligned}$$

Die Wahl sprechender Nichtterminal-Namen empfiehlt sich.

Im folgenden nun etwas ausführlicher, wie man aus der Aufgabenbeschreibung auf die Grammatik kommt:

Das Startsymbol der Grammatik soll L heißen, entweder nutzen wir diesen Namen für das niedrigstpriorierte Syntaxelement oder lassen es wie hier auf das Element für Abstraktion ableiten:

$$L \rightarrow \text{Abstr}$$

Die Abstraktion hat die niedrigste Priorität, damit kommt die dazugehörige Regel in der Grammatik am Anfang.

Ganz allgemein gilt: In der Regel für das Syntaxelement der Priorität n nutzen wir nur die Nichtterminale der Syntaxelemente mit Priorität n oder $n + 1$. Konkret hat die λ -Abstraktion (*Abstr*) die Priorität 0, Applikation (*App*) die Priorität 1 und Atome (*Atom*) die Priorität 2.

Damit ist nun klar, dass die Regeln zu *Abstr* nur *Abstr* oder *App* nutzen können (neben den Terminalen).

Laut Aufgabentext ist die λ -Abstraktion ein unärer Operator mit klammerfreier Schachtelung, also gibt es die Regel:

$$Abstr \rightarrow \lambda \text{ id} . Abstr$$

Da reicht für Abstraktionen aber noch nicht aus, da von *Abstr* auch noch *App* abgeleitet werden können muss, da sonst nur Ketten von $\lambda \text{ id} .$ abgeleitet werden können.

$$Abstr \rightarrow App$$

Nun folgt in den Prioritätsstufen die Applikation, da die Applikation linksassoziativ ist, muss die dazugehörige Regel linksrekursiv sein. Damit die Grammatik eindeutig ist, darf sie nicht *auch noch* rechtsrekursiv sein:

$$App \rightarrow App \text{ Atom}$$

Eine rechtsrekursive Regel wäre $App \rightarrow \text{Atom} App$. Da Applikationen aus mindestens einem Atom bestehen müssen (ersichtlich aus der Ursprungsgrammatik oder der Definition des λ -Kalküls aus der Vorlesung) muss *App* noch nach *Atom* abbilden:

$$App \rightarrow \text{Atom}$$

Wären als Applikationen auch leere Ketten von Atomen erlaubt, müsste die Regel $App \rightarrow \epsilon$ sein. Ein Atom ist entweder ein wirkliches Atom (eine Variable) oder eine Klammerung eines validen λ -Ausdrucks. Deshalb führen wir die folgenden Regeln ein:

$$\text{Atom} \rightarrow \text{Var} \quad | \quad (Abstr)$$

Natürlich kann man auch statt *Atom* die Nichtterminale *Var* und *Klammerung* ($Klammerung \rightarrow (Abstr)$) nutzen. Dann muss man aber die Regeln für *App* anpassen:

$$App \rightarrow App \text{ Var} \quad | \quad App \text{ Klammerung} \quad | \quad \text{Var} \quad | \quad \text{Klammerung}$$

- (b) Folgendes ist eine Grammatik einer *Variante* des λ -Kalküls mit abweichenden Operatorprioritäten. Startsymbol ist *App*. [4 Punkte]

$$\begin{aligned} App &\rightarrow Abstr \text{ AppList} \quad \text{egal} / \{ \text{id}, (, \lambda \} \\ AppList &\rightarrow \epsilon \quad \{ \}, \# \} \quad | \quad Abstr \text{ AppList} \quad \{ \text{id}, (, \lambda \} \\ Abstr &\rightarrow \text{Atom} \quad \{ \text{id}, (\} \quad | \quad \lambda \text{ id} . Abstr \quad \{ \lambda \} \\ Atom &\rightarrow \text{Var} \quad \{ \text{id} \} \quad | \quad (App) \quad \{ (\} \\ Var &\rightarrow \text{id} \quad \text{egal} / \{ \text{id} \} \end{aligned}$$

Diese Grammatik ist SLL(1). Rechnen Sie nach, indem Sie die nötigen Indizmengen explizit aufstellen und in die dafür vorgesehenen Kästchen eintragen.

- (c) Vervollständigen Sie den unten angegebenen rekursiven Abstiegsparser so, dass er [12 Punkte] die Variante des λ -Kalküls aus (b) in folgende Datenstruktur parst:

```

1 abstract class Expr {} // expression
2
3 class Var extends Expr {
4   public Var(String name) ...
5 }
6 class Abstr extends Expr { // abstraction
7   public Abstr(String varname, Expr body) ...
8 }
9 class App extends Expr { // application
10  public App(Expr left, Expr right) ...
11 }
```

Brechen Sie bei Parserfehlern mit der globalen `error()`-Funktion ab.

Hinweis: Sie können davon ausgehen, dass die Lexer-Schnittstelle wie folgt aussieht:

- Die Token sind Werte eines enum namens `Token` und heißen `LAMBDA`, `LPAREN`, `RPAREN`, `DOT`, `IDENT` und `EOF` für λ , `(`, `)`, `.`, Identifier sowie das Ende der Eingabe. Token können bspw. mit `==` verglichen werden.
- `lexer.current` gibt nichtkonsumierenden Zugriff auf das aktuelle Token.
- `lexer.string_value` gibt nichtkonsumierenden Zugriff auf den String-Wert des aktuellen Token, falls dieses ein `IDENT` ist. Ansonsten enthält das Feld `null`.
- `lexer.lex()` konsumiert das aktuelle Token.

```
1 Expr parseAtom() {
2     // gegeben
3     ...
4 }
5
6 String expect(Token tok) {
7     if (lexer.current != tok)
8         error();
9     lexer.lex();
10    return lexer.string_value;
11 }
12
13 Expr parseApp() {
14     Expr l = parseAbstr();
15     return parseAppList(l);
16 }
```

Beispiellösung:

```
1 Expr parseAbstr() {
2     switch (lexer.current) {
3     case IDENT: case LPAREN:
4         return parseAtom();
5     case LAMBDA:
6         expect(LAMBDA);
7         String varname = expect(IDENT);
8         expect(DOT);
9         Expr body = parseAbstr();
10        return new Abstr(varname, body);
11    default:
12        error();
13    }
14 }
15
16 // Hier müssen die linken Hälften der Applikationen durchgefädelt werden
17 Expr parseAppList(Expr left) {
18     switch(lexer.current) {
19     case RPAREN: case EOF: // epsilon, da war gar keine Applikation
20         return left;
21     case IDENT: case LAMBDA: case LPAREN:
22         Expr right = parseAbstr();
23         left = new App(left, right);
24     }
```

```
24     return parseAppList(left);  
25 default:  
26     error();  
27 }  
28 }
```