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

SS19, 30. September 2019, 11:00 – 13:00 Uhr

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

Bearbeitungszeit: 120 min

Aufgabe 1 (Haskell: Sortierte Listensummen) [19 Punkte]

Ziel dieser Aufgabe ist es, $[a + b \mid a \in A, b \in B]$ zu berechnen, wobei A, B und das Resultat sortierte Haskell-Listen sein werden.

(a) Implementieren Sie die Funktion `splay :: [a] -> [[a]]`, welche jedes Element der Eingabeliste zu einer unendlichen Liste "auffächert". [1 Punkt]

```
> splay [1,2,5]
[[1,1,1,1..], [2,2,2,2..], [5,5,5,5..]]
```

Beispiellösung:

```
splay = map repeat
```

Versuchen Sie nach Möglichkeit, für die folgenden Teilaufgaben auch unendliche Eingabelisten korrekt zu behandeln. Für Lösungen, die nur auf endlichen Eingabelisten funktionieren, gibt es Teilpunkte.

(b) Implementieren Sie `zziipp :: Num a =>[[a]] -> [[a]] -> [[a]]`, welches die in den Eingabelisten enthaltenen Listen paarweise durch elementweise Addition kombiniert. Ist eine Liste kürzer als die andere, soll der Rest der längeren Liste ignoriert werden. [2 Punkte]

```
> zziipp [[1,3,2]]  [[5,7,100]]
[[6,10,102]]
> zziipp [[2,2], [3,3]]  [[0,7], [8,15], [69, 420]]
[[2,9], [11, 18]]
```

Beispiellösung:

```
zziipp = zipWith (zipWith (+))
```

Alternativlösung:

```
zziipp (a:as) (b:bs) = (zipWith (+) a b) : zziipp as bs
zziipp _ _ = []
```

(c) Implementieren Sie `distrib :: Num a =>[a] -> [a] -> [[a]]`, welches jedes Element der ersten übergebenen Liste jeweils auf die zweite übergebene Liste addiert. [4 Punkte]

```
> distrib [1,3,100] [5,7,11]
[[  6,  8, 12],
 [  8, 10, 14],
 [105,107,111]]
```

Beispiellösung:

```
distrib a b = zziipp (splay a) (repeat b)
```

Alternativlösung:

```
distrib a b = map (\x -> map (+x) b) a
```

(d) Implementieren Sie `mergeAll :: Ord a =>[[a]] -> [a]`, welches eine Liste von Listen zu einer einzelnen sortierten Liste zusammenfügt. Sie dürfen davon ausgehen, dass die Eingabeliste, als Matrix gesehen, sowohl spaltenweise als auch zeilenweise sortiert ist. [10 Punkte]
Benutzen Sie bei Bedarf `merge :: Ord a =>[a] -> [a] -> [a]`, welches zwei (potenziell unendliche) sortierte Listen zu einer sortierten Liste verschmelzen kann.

```
> mergeAll [ [  6,  8, 12],
             [  8, 10, 14],
             [105,107,111] ]
[6,8,8,10,12,14,105,107,111]
```

Hinweis: Ist nach teilweiser Berechnung das erste Element der ersten Liste einmal *nicht* das global kleinste, so können Sie diese Eigenschaft durch `mergen` der ersten beiden Listen wiederherstellen.

Beispiellösung: Der Vollständigkeit halber hier eine Implementierung von `merge`:

```
merge [] bs = bs
merge as [] = as
merge (a:as) (b:bs)
  | a <= b = a : merge as (b:bs)
  | otherwise = b : merge (a:as) bs
```

Leider haben die meisten verkannt, wie komplex die Problemstellung für unendliche Listen ist, es können nämlich nicht nur unendlich viele Listen in der Eingabe enthalten sein, jede in der Argument-Liste enthaltene Liste kann selbst unendlich sein.[1] Für endliche Listen funktioniert bereits folgendes:

```
mergeAll' as = foldl merge [] as
```

Will man auch doppelt unendliche Listen behandeln, muss man sicher stellen, dass stets nur endlich viele Funktionsaufrufe bis zum nächsten produzierten Element passieren können. Wir entnehmen dazu so lange Elemente aus der ersten "Zeile", bis die (spaltenweise) Sortiertheit gebrochen wird, und sortieren die restlichen Elemente der ersten Zeile weiter hinten in der zweiten Zeile wieder ein.

```
mergeAll ((f:fs):(g:gs):rest)
  | f <= g = f : mergeAll (fs:(g:gs):rest)
  | otherwise = g : mergeAll (merge (f:fs) gs : rest)
```

Ist die Sortierung gerade gebrochen, so ist das erste Element der zweiten Zeile das global kleinste, und kann zurückgegeben werden. Eine Definition, die `g` nicht sofort, sondern erst beim nächsten Aufruf produziert, wäre immer noch produktiv. Da dieser Pattern Match sehr spezifisch ist, ergeben sich noch ein paar Sonderfälle, an die man denken muss:

```
mergeAll (x:y:rest) = mergeAll (merge x y : rest)
mergeAll [rest] = rest
mergeAll [] = []
```

Damit hier die erste Gleichung ausgeführt wird, muss `x` oder `y` die leere Liste sein.
Alternativlösung für die Hauptgleichung: Lässt man es gar nicht erst soweit kommen, dass die spaltenweise Sortierung gebrochen wird, spart man sich die Fallunterscheidung. Dazu muss man bei jedem rekursivem Aufruf die ersten beiden Zeilen `mergen`. Langsamer, aber einfacher.

[1] `splay [1..]` ist z.B. eine zum Testen nützliche unendliche Liste unendlicher Listen

```
mergeAll ((f:fs):gs:rest) = f : mergeAll (merge fs gs : rest)
```

(e) Geben Sie eine Implementierung für `xadd :: (Ord a, Num a) => [a] -> [a] -> [a]` an, welches für zwei sortierte Eingabelisten A und B die sortierte Liste $[a + b \mid a \in A, b \in B]$ berechnet. [2 Punkte]

Beispiellösung:

```
xadd as bs = mergeAll (distrib as bs)
```

Aufgabe 2 (Haskell: Funktionen höherer Ordnung, `alter`) [16 Punkte]

Hin und wieder setzt man in Haskell auch Listen von Schlüssel-Wert-Paaren ("Assoziativlisten") als Map-Datenstruktur ein. Dabei kommt jeder Schlüssel höchstens einmal vor:

```
> Data.List.lookup 4 [(1, "a"), (4, "d"), (3, "c")]
Just "d"
> Data.List.lookup 2 [(1, "a"), (4, "d"), (3, "c")]
Nothing
```

Eine sehr vielseitige Operation auf Maps ist die Funktion `alter`.

```
alter :: Eq k => (Maybe v -> Maybe v) -> k -> [(k, v)] -> [(k, v)]
```

Bei Aufruf `alter f k xs` soll in `xs` nach dem Schlüssel `k` gesucht werden. Gibt es ein Paar `(k', v)` mit `k == k'` in `xs`, so soll `alter` die Funktion `f` mit `Just v` aufrufen, ansonsten mit `Nothing`.

`f` entscheidet dann, wie der aktuelle Wert für den Schlüssel zu verändern ist: Ist der Rückgabewert `Nothing`, so wird der Eintrag gelöscht. Ist er `Just v'`, so enthält das Ergebnis von `alter` nur den Wert `v'` für Schlüssel `k`.

(a) Gegeben sei folgende Hilfsfunktion `d`: [2 Punkte]

```
d Nothing  = Nothing
d (Just v) = Nothing
```

Zu welchen Assoziativlisten l_1, l_2 werten folgende beiden Aufrufe der `alter`-Funktion aus?
$l_1 =$ `alter d 4 [(1, "a"), (4, "b"), (3, "c")]`
$l_2 =$ `alter d 2 [(1, "a"), (4, "b"), (3, "c")]`

Beispiellösung: `d` realisiert das Löschen eines Eintrags zu gegebenem Schlüssel aus der Map
$l_1 =$ `[(1, "a"), (3, "c")]`
$l_2 =$ `[(1, "a"), (4, "b"), (3, "c")]`

(b) Implementieren Sie die Funktion [3 Punkte]

```
insert :: Eq k => k -> v -> [(k, v)] -> [(k, v)]
```

unter Nutzung von `alter`. `insert k v xs` fügt der Assoziativliste `xs` ein Schlüssel-Wert-Paar `(k, v)` hinzu. Ein vorher vorhandener Eintrag für `k` wird überschrieben.

```
> insert 2 "b" [(1, "a")]
[(1, "a"), (2, "b")]
> insert 1 "b" [(1, "a"), (4, "d")]
[(1, "b"), (4, "d")]
```

(c) Implementieren Sie die Funktion [3 Punkte]

```
update :: Eq k => (v -> v) -> k -> [(k, v)] -> [(k, v)]
```

unter Nutzung von `alter`. `update f k xs` ändert den zu `k` gehörigen Wert `v` in `xs` zu `f v`. Ist kein Eintrag für `k` vorhanden, wird `xs` unverändert zurückgegeben.

```
> update ("s"++) 1 [(1, "a"), (3, "c")]
[(1, "sa"), (3, "c")]
> update ("s"++) 2 [(1, "a"), (3, "c")]
[(1, "a"), (3, "c")]
```

(d) Implementieren Sie die Funktion `alter`. [8 Punkte]

Beispiellösung:

```
alter f k [] = kvp k (f Nothing)
alter f k ((k',v):xs)
  | k == k'   = kvp k (f (Just v)) ++ xs
  | otherwise = (k',v) : alter f k xs

kvp _ Nothing = []
kvp k (Just v) = [(k, v)]

insert k v = alter (\_ -> Just v) k

update f = alter fmapF
  where
    fmapF Nothing = Nothing
    fmapF (Just v) = Just (f v)
```

Aufgabe 3 (Prolog, Nichtdeterministische Turingmaschine) [19 Punkte]

Im Folgenden sollen Sie eine Turingmaschine in Prolog implementieren.

Ein Band wird als Tupel `(L, M, R)` repräsentiert, wobei `L` der linke Teil des Bandes (in umgekehrter Reihenfolge), `M` das Symbol unter dem Lesekopf und `R` der rechte Teil des Bands ist.

Beispiel:
Das Band `[1, 2, 3, 4, 5]` mit dem Lesekopf auf dem Symbol `3` wird als `([2, 1], 3, [4, 5])` dargestellt.

Der Generator `blank(X)` erzeugt ein Blanksymbol. Mit diesem Symbol kann das Band bei Bedarf links oder rechts erweitert werden.

(a) Entwickeln Sie nun ein Prädikat `band(A, B)` welches genau dann erfüllbar ist, wenn der Inhalt des Bandes `B` die Liste `A` ist, wobei der Lesekopf auf dem linkesten Symbol steht. Bedenken Sie insbesondere den Fall `A=[]`. [4 Punkte]

Beispiel:

```
band([1, 2, 3], ([], 1, [2, 3])).
```

(b) Das Prädikat `move(B, D, B2)` beschreibt die Bewegung des Lesekopfes auf dem Band. Dabei ist `B` das Band vor der Bewegung, `B2` das Band nach der Bewegung und `D` die Bewegungsrichtung. [5 Punkte]

Mögliche Werte für `D` sind:

`left`	Bewege den Lesekopf um ein Bandsymbol nach links
`none`	Bewege den Lesekopf nicht
`right`	Bewege den Lesekopf um ein Bandsymbol nach rechts

Implementieren Sie **nur** die Fälle `D = left` und `D = none`.

Beispiel:

```
move(([2, 1], 3, [4, 5]), left, ([1], 2, [3, 4, 5])).
move(([],     2, [3]),    left, ([],  B, [2, 3])).
```

Wobei `blank(B)` im obigen Ausdruck gilt.

(c) Schreiben Sie ein Prädikat `step(B, S, B2, S2)`, welches einen Schritt der Turingmaschine auf einem Band durchführt, wobei `B` und `S` jeweils Band und Zustand vor dem Schritt sind, und `B2` und `S2` jeweils Band und Zustand nach dem Schritt. [5 Punkte]

Verwenden Sie hierfür das vorgegebene Prädikat `transition(A, S, A2, D, S2)`, welches die möglichen Zustandsübergänge der Turingmaschine darstellt. Hierbei sind

`A`	das aktuelle Bandsymbol,
`S`	der aktuelle Zustand,
`A2`	das zurück aufs Band zu schreibende Symbol,
`D`	die resultierende Bewegungsrichtung und
`S2`	der Folgezustand der Turingmaschine.

Beispiel:

```
transition(3, s1, 4, left, s2).
?- step(([2], 3, []), s1, B2, S2).
   B2 = ([], 2, [4]), S2 = s2.
```

(d) Schreiben Sie ein Prädikat `turing(B, S, B2, S2)`, welches genau dann erfüllt ist, wenn eine Turingmaschine in Zustand `S` mit Bandinhalt `B` im Zustand `S2` mit Bandinhalt `B2` terminiert. Termination bedeutet, dass es keinen weiteren Zustandsübergang mehr gibt. Achten Sie darauf, dass auch nichtdeterministische Turingmaschinen simuliert werden können. [5 Punkte]

Beispiel:

```
turing(([2], 3, []), s1, ([], 2, [4]), s2).
```

Mit der beispielhaften Transition aus c)

Beispiellösung:

```
band([],    ([], E, [])) :- blank(E).
band([A], ([], A, [])).
band([A|As], ([], A, [B|Bs])) :- band(As, ([], B, Bs)).

move(B, none, B).
move(([],    M, R),     left,  ([],    E, [M|R])) :- blank(E).
move(([X|L], M, R),     left,  (L,     X, [M|R])).
/* War nicht gefordert */
move((L,     M, []),    right, ([M|L], E, []))    :- blank(E).
move((L,     M, [X|R]), right, ([M|L], X, R)).

step((L, M, R), S, B, S2) :-
        transition(M, S, A2, D, S2),
        move((L, A2, R), D, B).

turing(B, S, B2, S2) :-
        step(B, S, B3, S3),
        turing(B3, S3, B2, S2).
turing(B, S, B, S) :- not(step(B, S, _, _)).
```

Folgendes Prädikat funktioniert nur für deterministische Tu ringmaschinen:

```
turing(B, S, B2, S2) :- step(B, S, B3, S3), !, turing(B3, S3, B2, S2).
turing(B, S, B, S).
```

Aufgabe 4 (Ein universeller Kombinator) [13 Punkte]

In der Vorlesung waren die Kombinatoren S, K, I definiert als:

$S = \lambda\mathrm{x}.\ \lambda\mathrm{y}.\ \lambda\mathrm{z}.\ \mathrm{x}\ \mathrm{z}\ (\mathrm{y}\ \mathrm{z}) \qquad K = \lambda\mathrm{x}.\ \lambda\mathrm{y}.\ \mathrm{x} \qquad I = \lambda\mathrm{x}.\ \mathrm{x}$

Es gilt also zum Beispiel: $K\ x\ y \Rightarrow^2 x$ für λ-Terme x, y

Mit S und K lassen sich alle Kombinatoren darstellen. In dieser Aufgabe werden Sie nun zeigen, dass alle Kombinatoren allein durch den Kombinator U darstellbar sind, der wie folgt definiert ist:

$U = \lambda\mathrm{x}.\ \mathrm{x}\ S\ K$

Durch U lassen sich nämlich I, K und S gewinnen, weil (bis auf β, η-Konversion) gilt:

$U\ U = I \qquad U\ (U\ (U\ U)) = U\ (U\ I) = K \qquad U\ (U\ (U\ (U\ U))) = U\ K = S$

(a) Zeigen Sie dazu: [11 Punkte]

$$\begin{array}{lll} U\ K & \Rightarrow^* & S \\ U\ (U\ I) & \Rightarrow^* & K \\ U\ U\ x & \Rightarrow^* & x \end{array}$$

Beispiellösung:

$$\begin{array}{lll} U\ K & \Rightarrow & K\ S\ K \\ & \Rightarrow^2 & S \end{array}$$

$$\begin{array}{lll} U\ (U\ I) & \Rightarrow & U\ I\ S\ K \\ & \Rightarrow & I\ S\ K\ S\ K \\ & \Rightarrow & S\ K\ S\ K \\ & \Rightarrow^3 & K\ K\ (S\ K) \\ & \Rightarrow^2 & K \end{array}$$

$$\begin{array}{lll} U\ U\ x & \Rightarrow & U\ S\ K\ x \\ & \Rightarrow & S\ S\ K\ K\ x \\ & \Rightarrow^3 & S\ K\ (K\ K)\ x \\ & \Rightarrow^3 & K\ x\ (K\ K\ x) \\ & \Rightarrow^2 & x \end{array}$$

(b) Begründen Sie kurz, warum aus der Aussage $U\ U\ x \Rightarrow^* x$ die ursprüngliche Behauptung ($U\ U = I$ bis aus β, η-Konversion) folgt. [2 Punkte]

Beispiellösung:

$$U\ U \overset{\eta}{=} \lambda\mathrm{x}.\ U\ U\ \mathrm{x} \underset{\text{a)}}{\overset{\beta}{=}} \lambda\mathrm{x}.\ \mathrm{x} = I$$

Also wie gefordert $U\ U \overset{\beta,\eta}{=} I$.

Aufgabe 5 (MPI: Manuelles Alltoall) [10 Punkte]

Das folgende Programm führt die MPI-Operation `MPI_Alltoall` aus. Gehen Sie davon aus, dass das Programm mit beliebig vielen Prozessen ausgeführt wird und die Initialisierung von MPI mittels `MPI_Init` bereits korrekt vorgenommen wurde.

```
1  int size, rank;
2  MPI_Comm_size(MPI_COMM_WORLD, &size);
3  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4
5  int *sendBuffer = malloc(size * sizeof(int));
6  int *recvBuffer = malloc(size * sizeof(int));
7
8  for (int i = 0; i < size; i++) {
9      sendBuffer[i] = i + size * rank;
10 }
11
12 MPI_Alltoall(sendBuffer, 1, MPI_INT, recvBuffer, 1, MPI_INT,
       MPI_COMM_WORLD);
```

(a) Nehmen Sie an, dass das angegebene Programm von 3 Prozessen ausgeführt wird. Geben Sie den Inhalt von `sendBuffer` und `recvBuffer` in allen Prozessen nach der Ausführung des Programms an. [4 Punkte]

Prozess Nr.	**`sendBuffer` (Lösung)**	**`recvBuffer` (Lösung)**
0	0, 1, 2	0, 3, 6
1	3, 4, 5	1, 4, 7
2	6, 7, 8	2, 5, 8

(b) Geben Sie einen Ersatz für Zeile 12 an, der entweder eine oder mehrere andere kollektive MPI-Operationen als `MPI_Alltoall` verwendet oder nur aus Sende- und Empfangsoperationen von MPI besteht. Es dürfen selbstverständlich weitere C-Kontrollstrukturen und -Befehle verwendet werden. Stellen Sie sicher, dass Ihre Lösung mit beliebig vielen Prozessen funktioniert. [6 Punkte]

Beispiellösung:

Implementierung mit `MPI_Scatter`:

```
1 for (int sender = 0; sender < size; sender++) {
2     MPI_Scatter(sendBuffer, 1, MPI_INT, recvBuffer + sender, 1,
3         MPI_INT, sender, MPI_COMM_WORLD);
4 }
```

Implementierung mit `MPI_Gather`:

```
1 for (int receiver = 0; receiver < size; receiver++) {
2     MPI_Gather(sendBuffer + receiver, 1, MPI_INT, recvBuffer, 1,
3         MPI_INT, receiver, MPI_COMM_WORLD);
4 }
```

Implementierung mit Sende- und Empfangsoperationen:

```
1  MPI_Request request;
2  for (int sender = 0; sender < size; sender++) {
3      if (sender == rank) {
4          for (int receiver = 0; receiver < size; receiver++) {
5              MPI_ISend(sendBuffer + receiver, 1, MPI_INT, receiver, 0,
6                  MPI_COMM_WORLD, &request);
7          }
8      }
9      MPI_Recv(recvBuffer + sender, 1, MPI_INT, sender, 0,
10         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
11 }
```

Diese Implementierung rollt im Prinzip die Lösung mit der Scatter-Operation noch weiter aus, indem es diese kollektive Operation durch Sende- und Empfangsoperationen ersetzt. Anstatt einer nicht-blockierenden Sende-Operation kann eine normale Sende-Operation verwendet werden und stattdessen der Wert, der innerhalb des gleichen Prozesses übertragen wird, direkt in das Array geschrieben werden, ohne eine Sende- und Empfangsoperation zu verwenden.

Aufgabe 6 (Java und Design by Contract: Reduktion mit Fork-Join) [20 Punkte]

Gegeben sei folgendes Java-Codegerüst:

```
1  public interface ValueCombinator {
2     public int combine(int left, int right); // assoziativ und kommutativ
3  }
4
5  public class GenericOperator {
6     public static int execute(ValueCombinator combinator, Integer[] array) {
7        ForkJoinPool fjPool = new ForkJoinPool();
8        Action action = new Action(combinator, array, 0, array.length - 1);
9        return fjPool.invoke(action);
10    }
11 }
12
13 public class Action
14       extends java.util.concurrent.RecursiveTask<Integer> {
15
16    private Integer[] array;
17    private int low, high;
18    private ValueCombinator combinator;
19
20    public Action(ValueCombinator combinator,
21          Integer[] array, int low, int high) {
22       this.array = array;
23       this.low = low;
24       this.high = high;
25       this.combinator = combinator;
26    }
27
28    public Integer compute() {
29       if (low >= high) {
30          return array[low];
31       }
32
33       int middle = (low + high) / 2;
34       // handle the left sub-list using a child thread
35       Action leftAction = new Action(combinator, array, low, middle);
36       leftAction.fork();
37       // handle the right sub-list in the current thread
38       Action rightAction = new Action(combinator, array, middle + 1, high);
39       int rightResult = rightAction.compute();
40       int leftResult = leftAction.join();
41       return combinator.combine(leftResult, rightResult);
42
43    }
44 }
```

Parallelverarbeitung in Java (15 Punkte)

Gegeben seien das Interface `ValueCombinator`, die Klasse `GenericOperator`, sowie das Gerüst der Klasse `Action`. Die Klasse `GenericOperator` stellt die Methode `execute` bereit, die die Elemente des gegebenen `Integer`-Arrays durch Ausführung der `combine`-Methode des gegebenen `ValueCombinator` zu einem einzelnen Wert zusammenfassen soll. Nehmen Sie an, dass die Methode `combine` assoziativ und kommutativ ist, und somit die Reihenfolge der Ausführung auf den Werten des Arrays nicht relevant ist. Das Ergebnis für die Werte $(1, 2, 3, 4)$ soll beispielsweise dem Ergebnis der Berechnung `combine(combine(1, 2), combine(3, 4))` entsprechen. Die Methode könnte beispielsweise die Summe oder das Maximum der Elemente berechnen.

Die `execute`-Methode des `GenericOperator` erstellt für die Ausführung einen `ForkJoinPool` und startet auf diesem eine initiale `Action`. Das Gerüst der Klasse `Action` soll um die Implementierung der `compute`-Methode ergänzt werden. Diese soll das Array fiktiv, d.h. ohne Kopieren der Array-Inhalte durch Berechnung von Start- und Ende-Index, in zwei möglichst gleich große Teile aufteilen. Diese beiden Teile sollen dann von verschiedenen `Actions` weiterverarbeitet und das Ergebnis mit dem gegebenen `ValueCombinator` kombiniert werden. Der Teil des Arrays, der von einer `Action` verarbeitet wird, wird durch die Indizes `low` und `high` im Konstruktor angegeben. Wenn der zu verarbeitende Array-Teil nur noch ein Element umfasst, soll dieses zurückgegeben werden.

(a) Ergänzen Sie die `compute`-Methode der `Action` so, dass sie das oben spezifizierte Verhalten aufweist. Stellen Sie dabei sicher, dass bei einem Array der Länge n nicht mehr als n Threads gestartet werden. [8 Punkte]

Beispiellösung:

siehe Code
Es ist wichtig, dass von den beiden neu erzeugten Actions nur für eine ein neuer Thread gestartet wird, da sonst insgesamt $2n - 1$ Threads erzeugt werden anstatt n.

(b) Sie stellen fest, dass die Berechnung bei großen Datenmengen (bspw. bei 100.000 Array-Elementen) wesentlich länger dauert als bei sequentieller Ausführung der Operation. Wie erklären Sie dies? Erläutern Sie außerdem, wie Sie das vermeiden könnten. [4 Punkte]

Beispiellösung:

Es werden durch die rekursive Aufteilung bis zu einem Element insgesamt n Threads für ein Array der Länge n erzeugt. Der Overhead für die Verwaltung der Threads, insbesondere durch die Instanziierung der `Actions`, ist höher als die Zeit, die durch parallele Verarbeitung eingespart werden kann.

Um das Problem zu lösen, müsste ein anderes Abbruchkriterium definiert werden, als dass nur noch ein Element in der verarbeiteten Teilliste vorhanden ist. Beispielsweise könnte die Anzahl rekursiver Aufteilungen abhängig von der Anzahl zur Verfügung stehender Prozessoren definiert werden. Unter der Voraussetzung, dass die Ausführungsdauer der Kombinationsoperation nicht von den Daten abhängt und somit alle Ausführungen etwa gleich viel Zeit benötigen, brauchen die Threads für die Ausführung ungefähr gleich lang.

(c) Die Operation des `ValueCombinator` entspricht einer Reduktion, wofür auf Java Streams die Methode `Optional<T> reduce(BinaryOperator<T>)` angeboten wird. `BinaryOperator<T>` ist ein Functional Interface, welches die Methode `T apply(T, T)` bereitstellt. Ersetzen Sie den Inhalt der `execute(ValueCombinator, Integer[])`-Methode von `GenericOperator` unter Verwendung der `reduce`-Operation von Java Streams und des gegebenen `ValueCombinator`. Ergänzen Sie dazu die bereits vorgegebene Umwandlung des Arrays in eine Liste: [3 Punkte]

```
public static int execute(ValueCombinator combinator, Integer[] array) {
    return Arrays.asList(array)
        .stream().reduce((a, b) -> combinator.combine(a, b)).get();
}
```

Beispiellösung: *siehe Quellcode*

Statt des Lambda-Ausdrucks ist auch die Verwendung einer Methodenreferenz möglich:

```
return Arrays.asList(array).stream().reduce(combinator::combine).get();
```

Design by Contract (5 Punkte)

Betrachten Sie für die folgenden Teilaufgaben diese Beispielimplementierung von `ValueCombinator`:

```
1  public class MaxAbsCombinator implements ValueCombinator {
2      /*@
3        @ requires left != Integer.MIN_VALUE && right != Integer.MIN_VALUE;
4        @ ensures \result <= left || \result <= right;
5        @ ensures \result >= left && \result >= right;
6        */
7      @Override
8      public int combine(int left, int right) {
9          return Math.max(Math.abs(left), Math.abs(right));
10     }
11 }
```

(d) Der Vertrag der Methode `combine` wird *vom Aufgerufenen* verletzt. Begründen Sie dies und geben Sie an, wie die verletzte Nachbedingung geändert oder erweitert werden könnte, ohne sie vollständig zu entfernen. [3 Punkte]

Beispiellösung:

Die erste Nachbedingung wird nicht erfüllt, da bei einem negative Eingabewert, dessen Betrag größer ist als der andere Eingabewert, keine der mit *Oder* verknüpften Bedingungen erfüllt wird. Dies ließe sich korrigieren, indem in der Bedingung die Beträge der Eingabewerte verwendet werden oder die Bedingungen um die additiven Inversen der Eingabewerte ergänzt werden:

```
@ ensures \result <= left || \result <= right || \result <= -left
          || \result <= -right;
```

(e) Betrachten Sie den folgenden Aufruf der `combine`-Methode. Wird der Vertrag hier *vom Aufrufer* erfüllt? Begründen Sie Ihre Antwort kurz. [2 Punkte]
Hinweis: Es gilt `random.nextInt() ∈ [Integer.MIN_VALUE, Integer.MAX_VALUE]`.

```
java.util.Random random = new java.util.Random();
new MaxAbsCombinator().combine(random.nextInt(), random.nextInt());
```

Beispiellösung:

Nein, der Vertrag wird nicht erfüllt. Die Vorbedingung verlangt, dass keines der Argumente den Wert `Integer.MIN_VALUE` hat, dieser kann jedoch von `random.nextInt()` erzeugt werden, wodurch der Vertrag verletzt wird.

Aufgabe 7 (Kompilierung von `do until`) [23 Punkte]

In dieser Aufgabe sollen Sie Java um ein neues Schleifenkonstrukt erweitern, nämlich die do-until-Schleife. Diese führt ihren Rumpf mindestens einmal aus und läuft solange, bis die Bedingung erfüllt ist.

Zum Beispiel setzt **`do`** `i++;` **`until`** `(isPrime(i));` die Variable `i` auf die nächste Primzahl.

(a) Erweitern Sie den folgenden Ausschnitt der Java-Grammatik um einen Fall für do-until-Schleifen. [5 Punkte]
Geben Sie dann eine Linksfaktorisierung der Regeln von *Statement* an. Sie dürfen annehmen, dass keine der ausgelassenen Alternativen von *Statement* mit **`do`** beginnt.

$$
\begin{array}{rcl}
\textit{Expression} & \rightarrow & \ldots \mid \ldots \mid \ldots \\
\textit{Statement} & \rightarrow & \ldots \mid \ldots \mid \ldots \quad \text{(Keine dieser Alternativen beginnt mit } \texttt{do}\text{)} \\
& \mid & \texttt{do}\ \textit{Statement}\ \texttt{while (}\ \textit{Expression}\ \texttt{);}
\end{array}
$$

Beispiellösung:

$$
\begin{array}{rcl}
\textit{Expression} & \rightarrow & \ldots \mid \ldots \mid \ldots \\
\textit{Statement} & \rightarrow & \ldots \mid \ldots \mid \ldots \quad \text{(Keine dieser Alternativen beginnt mit } \texttt{do}\text{)} \\
& \mid & \texttt{do}\ \textit{Statement}\ \texttt{while (}\ \textit{Expression}\ \texttt{);} \\
& \mid & \texttt{do}\ \textit{Statement}\ \texttt{until (}\ \textit{Expression}\ \texttt{);}
\end{array}
$$

Linksfaktorisiert:

$$
\begin{array}{rcl}
\textit{Expression} & \rightarrow & \ldots \mid \ldots \mid \ldots \\
\textit{Statement} & \rightarrow & \ldots \mid \ldots \mid \ldots \quad \text{(Keine dieser Alternativen beginnt mit } \texttt{do}\text{)} \\
& \mid & \texttt{do}\ \textit{Statement}\ \textit{WhileOrUntil} \\
\textit{WhileOrUntil} & \rightarrow & \texttt{while (}\ \textit{Expression}\ \texttt{);} \\
& \mid & \texttt{until (}\ \textit{Expression}\ \texttt{);}
\end{array}
$$

(b) Vervollständigen Sie den `DO`-Fall des auf der nächsten Seite gegebenen rekursiven Abstiegsparsers, sodass er Ihre linksfaktorisierte Grammatik parsen kann. [11 Punkte]

Sie dürfen wiederholte Codesequenzen durch geeignete Markierungen abkürzen.

Wenn sie Teilaufgabe (a) nicht lösen konnten, dürfen Sie einen nichtdeterministischen Parser angeben. Markieren Sie in diesem Fall deutlich, wo Ihr Parser nichtdeterministisch ist.

Lexer-Schnittstelle:
Die globale Variable `token` enthält immer das aktuelle Token. Tokens besitzen die Methode `getType()`, die den Typ des Tokens zurückgibt.

Folgende Token-Typen sind definiert:

`DO`	für das Schlüsselwort **`do`**
`WHILE`	für das Schlüsselwort **`while`**
`UNTIL`	für das Schlüsselwort **`until`**
`LP` und `RP`	für die linke und rechte Klammer **`(`** und **`)`**
`SC`	für das Semikolon **`;`**

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

```
1  // Diese Funktion darf als gegeben angenommen werden
2  void parseExpression() { ... }
3
4  void parseStatement() {
5    switch (token.getType()) {
6      // cases für die anderen Arten von Statement ausgelassen
7      ...
8    case DO:
9      // Hier ergänzen
```

Beispiellösung:

```
1      nextToken();
2      parseStatement();
3      parseWhileOrUntil();
4      break;
5    }
6  }
7
8  void parseWhileOrUntil() {
9    switch (token.getType()) {
10   case WHILE:
11     nextToken();
12     expect(LP);
13     parseExpression();
14     expect(RP);
15     expect(SC);
16     break;
17   case UNTIL:
18     nextToken();
19     expect(LP);
20     parseExpression();
21     expect(RP);
22     expect(SC);
23     break;
24   }
25 }
26
27 void expect(TokenType tt) {
28   if (token.getType() == tt) {
29     nextToken();
30   } else {
31     error();
32   }
33 }
```

(c) Im AST werden do-until-Schleifen durch Objekte der folgenden Klasse repräsentiert: [7 Punkte]

```
1 public class DoUntilLoop extends Statement {
2   public BooleanExpression condition;
3   public Statement body;
4
5   @Override
6   public void accept(StatementVisitor v) { v.visit(this); }
7 }
```

Sie sollen nun einen Codegenerator schreiben, der do-until-Schleifen in Java-Bytecode übersetzt.

Der Codegenerator soll wie aus der Vorlesung bekannt als Visitor von Statements implementiert werden. Vervollständigen Sie dazu die gegebene `visit`-Methode für `DoUntilLoop`-Objekte.

Folgende Funktionen (siehe auch Folie 445f.) können nützlich sein:

- `String makeLabel()`
  Erzeugt einen einmaligen Namen für ein Label.
- **`void`** `evaluateBooleanExpression(BooleanExpression expr, String trueLabel, String falseLabel)`
  Erzeugt Code, der den booleschen Ausdruck `expr` auswertet. Ist der Ausdruck wahr, springt der erzeugte Code zu `trueLabel`, sonst zu `falseLabel`.
- **`void`** `BytecodeWriter.printf(String code, Object arg)`
  Gibt den Bytecode `code` aus. „`%s`“ wird durch `arg.toString()` ersetzt.

```
1 class CodeGenerationVisitor extends StatementVisitor {
2   BytecodeWriter writer;
3
4   // visit-Methoden für andere Arten von Statements ausgelassen
5
6   void visit (DoUntilLoop loop) {
7     // Hier ergänzen
```

Beispiellösung:

```
1     String loopEntry = makeLabel();
2     String loopExit  = makeLabel();
3
4     writer.printf("%s:\n", loopEntry);
5     loop.body.accept(this);
6     evaluateBooleanExpression(loop.condition, loopExit, loopEntry);
7     writer.printf("%s:\n", loopExit);
8   }
9 }
```