Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation
Prof. Dr.-Ing. Gregor Snelting
gregor.snelting@kit.edu
Jun.-Prof. Dr. Oliver Hummel
hummel@kit.edu

2 215000 006695

Klausur Programmierparadigmen — Beispiellösung

WS 2012/13, 10. April 2013, 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, Lauflängenkodierung) [20 Punkte]

Die Lauflängenkodierung ist ein Verfahren zur Komprimierung von Daten. Dabei wird ausgenutzt, dass viele Datenströme lange Zeichenketten aufweisen, die nur aus einem einzigen sich immer wiederholenden Zeichen bestehen. Diese sich wiederholenden Zeichen werden zusammengefasst und nur noch die Anzahl der Wiederholungen sowie das Zeichen selbst gespeichert. In dieser Aufgabe werden Sie die Lauflängenkodierung in Haskell implementieren.

(a) Definieren Sie zunächst eine Funktion `splitWhen`, welche ein Prädikat `p :: (a -> Bool)` und eine Liste `xs :: [a]` nimmt und ein Tupel `(ys,zs) :: ([a],[a])` bestehend aus zwei Listen zurückgibt. Die Funktion `splitWhen` zerlegt die Liste `xs` an der Stelle, an der `p` das erste Mal gilt. [9 Punkte]

Beispiele:

```
splitWhen even [1,2,3] ⇒+ ([1],[2,3])
splitWhen (=='o') "Hello, World!" ⇒+ ("Hell","o, World!")
```

Beispiellösung:

```
splitWhen :: (a -> Bool) -> [a] -> ([a],[a])
splitWhen _ [] = ([],[])
splitWhen p (x:xs)
        | p x       = ([],x:xs)
        | otherwise = let (ys,zs) = splitWhen p xs in (x:ys,zs)
```

(b) Definieren Sie nun eine Funktion `group` und geben Sie ihren allgemeinsten Typ an. Die Funktion `group` nimmt eine Liste `xs` und gibt eine Liste von Listen zurück. Dabei soll jede Teilliste der Rückgabe immer genau alle aufeinanderfolgenden identischen Zeichen der Originalliste beinhalten. [7 Punkte]

Hinweis: Verwenden Sie die Funktion `splitWhen` aus Teilaufgabe (a).

Beispiele:

```
group [1,1,2,1,3,3,3] ⇒+ [[1,1],[2],[1],[3,3,3]]
group "aaabbcdddd" ⇒+ ["aaa","bb","c","dddd"]
```

Beispiellösung:

```
group :: Eq a => [a] -> [[a]]
group [] = []
group (x:xs) = let (group1, rest) = splitWhen (/=x) xs
        in (x:group1) : group rest
```

(c) Verwenden Sie `group`, um die Funktion `encode` zu definieren, die eine Liste `xs :: [a]` nimmt und die Lauflängenkodierung von `xs` als Liste von Tupeln `[(a,Int)]` zurückgibt. Jeder Eintrag der Ergebnisliste kodiert, wie oft das jeweilige Zeichen in `xs` hintereinander vorkommt. [4 Punkte]

Beispiele:

```
encode [1,1,2,1,3,3,3] ⇒+ [(1,2),(2,1),(1,1),(3,3)]
encode "aaabbcdddd" ⇒+ [('a',3),('b',2),('c',1),('d',4)]
```

Beispiellösung:

```
encode :: Eq a => [a] -> [(a,Int)]
encode xs = map (\x -> (head x, length x)) (group xs)
```

Aufgabe 2 (λ-Kalkül, Typinferenz, Church-Paare) [25 Punkte]

Aus der Vorlesung kennen Sie die Modellierung von Church-Zahlen und Church-Booleans. Auf ähnliche Weise kann man auch Paare von Elementen im λ-Kalkül darstellen. Der Paar-Konstruktor `pair` ist:

$$\texttt{pair} = \lambda\texttt{a}.\ \lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f a b}$$

Wichtige Funktionen auf Paaren sind die Destruktoren `fst` und `snd`, die jeweils das erste bzw. zweite Element aus dem Paar extrahieren. Die Definition von `fst` ist:

$$\texttt{fst} = \lambda\texttt{p}.\ \texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a})$$

(a) Definieren Sie `snd`. [1 Punkt]

Beispiellösung:

$$\texttt{snd} = \lambda\texttt{p}.\ \texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{b})$$

(b) Zeigen Sie mit Beta-Reduktion unter Verwendung der Auswertungsstrategie Call-by-name: [4 Punkte]

$$\texttt{fst (pair a b)} \Rightarrow^* \texttt{a}$$

Beispiellösung:

$$\begin{aligned}
\texttt{fst (pair a b)} &= (\lambda\texttt{p}.\ \texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a}))\ ((\lambda\texttt{a}.\ \lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f a b})\ \texttt{a b}) \\
&\Rightarrow (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f a b})\ \texttt{a b}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a}) \\
&\Rightarrow (\lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f a b})\ \texttt{b}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a}) \\
&\Rightarrow (\lambda\texttt{f}.\ \texttt{f a b})\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a}) \\
&\Rightarrow (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a})\ \texttt{a b} \\
&\Rightarrow (\lambda\texttt{b}.\ \texttt{a})\ \texttt{b} \\
&\Rightarrow \texttt{a}
\end{aligned}$$

(c) Eine weitere wichtige Operation auf Paaren ist `map`. [8 Punkte]
Definieren Sie eine Funktion `map g h p`, die gegebene Funktionen `g` und `h` auf die beiden Elemente eines Paares `p` anwendet. Geben Sie `map` in Normalform an.

Hinweis: In Haskell kann man `map` so definieren:

```
map g h p = (g (fst p),h (snd p))
```

Beispiellösung:

$$\begin{aligned}
\texttt{map} &= \lambda\texttt{g}.\ \lambda\texttt{h}.\ \lambda\texttt{p}.\ \texttt{pair}\ (\texttt{g}\ (\texttt{fst p}))\ (\texttt{h}\ (\texttt{snd p})) \\
&\Rightarrow^* \lambda\texttt{g}.\ \lambda\texttt{h}.\ \lambda\texttt{p}.\ \lambda\texttt{f}.\ \texttt{f}\ (\texttt{g}\ (\texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{a})))\ (\texttt{h}\ (\texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{b})))
\end{aligned}$$

Alternativen:

$\texttt{map} = \lambda\texttt{g}.\ \lambda\texttt{h}.\ \lambda\texttt{p}.\ \lambda\texttt{f}.\ \texttt{f}\ (\texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{g a}))\ (\texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \texttt{h b}))$

oder

$\texttt{map} = \lambda\texttt{g}.\ \lambda\texttt{h}.\ \lambda\texttt{p}.\ \texttt{p}\ (\lambda\texttt{a}.\ \lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f}\ (\texttt{g a})\ (\texttt{h b}))$

(d) Bestimmen Sie einen allgemeinsten Typ für `pair`. Gegeben ist der Ableitungsbaum. Geben Sie das zugehörige Constraintsystem, sowie den verwendeten Unifikator an. [12 Punkte]

Hinweis: Es genügt, wenn Sie den Unifikator direkt angeben. Sie müssen die einzelnen Berechnungsschritte des Unifikationsalgorithmus nicht explizit hinschreiben.

Es sei $\Gamma = \texttt{a} : \alpha_2, \texttt{b} : \alpha_4, \texttt{f} : \alpha_6$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\Gamma)\,(\texttt{f}) = \alpha_{10}}{\Gamma \vdash \texttt{f} : \alpha_{10}}\,\text{VAR} \quad \dfrac{(\Gamma)\,(\texttt{a}) = \alpha_{11}}{\Gamma \vdash \texttt{a} : \alpha_{11}}\,\text{VAR}}{\Gamma \vdash \texttt{f a} : \alpha_8}\,\text{APP} \quad \dfrac{(\Gamma)\,(\texttt{b}) = \alpha_9}{\Gamma \vdash \texttt{b} : \alpha_9}\,\text{VAR}}{\texttt{a} : \alpha_2, \texttt{b} : \alpha_4, \texttt{f} : \alpha_6 \vdash \texttt{f a b} : \alpha_7}\,\text{APP}}{\texttt{a} : \alpha_2, \texttt{b} : \alpha_4 \vdash \lambda\texttt{f}.\ \texttt{f a b} : \alpha_5}\,\text{ABS}}{\texttt{a} : \alpha_2 \vdash \lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f a b} : \alpha_3}\,\text{ABS}}{\vdash \lambda\texttt{a}.\ \lambda\texttt{b}.\ \lambda\texttt{f}.\ \texttt{f a b} : \alpha_1}\,\text{ABS}$$

Beispiellösung:

Constraintsystem:

$C = \{$

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

$\}$

Unifikator:

$\sigma = [\alpha_{11} \Rightarrow \alpha_2, \alpha_{10} \Rightarrow \alpha_2 \to \alpha_4 \to \alpha_7, \alpha_9 \Rightarrow \alpha_4, \alpha_8 \Rightarrow \alpha_4 \to \alpha_7, \alpha_6 \Rightarrow \alpha_2 \to \alpha_4 \to \alpha_7,$
$\alpha_5 \Rightarrow (\alpha_2 \to \alpha_4 \to \alpha_7) \to \alpha_7, \alpha_3 \Rightarrow \alpha_4 \to (\alpha_2 \to \alpha_4 \to \alpha_7) \to \alpha_7, \alpha_1 \Rightarrow \alpha_2 \to \alpha_4 \to (\alpha_2 \to \alpha_4 \to \alpha_7) \to \alpha_7]$

Allgemeinster Typ für `pair`*:*

$\alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma$

Aufgabe 3 (Prolog, Labyrinth) [20 Punkte]

Im Folgenden soll ein Prolog-Programm zum Finden aller möglichen Wege durch ein Labyrinth entwickelt werden. Das Labyrinth sei in Felder unterteilt, die entweder *frei* oder *blockiert* sind.

In Prolog modellieren wir das Labyrinth über das Prädikat `frei(Feld)`, das genau dann erfüllbar ist, wenn das entsprechende `Feld` *frei* ist. Ein `Feld` ist ein Tupel `(X, Y)` aus X- und Y-Koordinate.

(a) Definieren Sie ein zweistelliges Prädikat [3 Punkte]

```
lengthof(L, Laenge)
```

das die Länge der übergebenen Liste `L` berechnet.

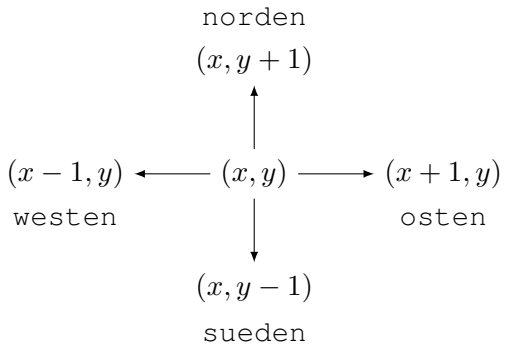
(b) Wir benötigen einen dreistelligen *Generator* [7 Punkte]

```
nachbar(Feld, Richtung, NachbarFeld)
```

der zu einem gegebenen `Feld` bei Reerfüllung alle *freien* Nachbarfelder `NachbarFeld` generiert. Schritte können nur in die Richtungen `norden`, `osten`, `sueden` und `westen` gemacht werden. Die Abbildung rechts gibt das verwendete Koordinatensystem an. Die Variable `Richtung` enthält eine Beschreibung, wie man vom Feld zum Nachbarfeld gelangt.

Beispiel:

```
? nachbar((1, 1), Richtung, NachbarFeld).
  Richtung = osten,
  NachbarFeld = (2, 1) ;
  Richtung = sueden,
  NachbarFeld = (1, 0) ;
  No
```



(c) Das Prädikat `wege(Schritte, MaxSchritte)` generiert alle möglichen Wege durch das Labyrinth, die höchstens `MaxSchritte` Schritte benötigen: [10 Punkte]

```
wege(Schritte, MaxSchritte) :-
        start(Start),
        ziel(Ziel),
        findeweg(Start, Ziel, MaxSchritte, [], Schritte).
```

Definieren Sie das hierzu benötigte fünfstellige Prädikat

```
findeweg(Start, Ziel, MaxSchritte, Besucht, Schritte)
```

welches für Startfeld `Start` und Zielfeld `Ziel` erfüllt ist, falls `Ziel` von `Start` aus durch eine Folge von höchstens `MaxSchritte` Schritten erreichbar ist. Weiterhin darf keines der in `Besucht` enthaltenen Felder nochmals betreten werden, um nicht im Kreis zu laufen. Die Liste `Schritte` sammelt die Richtungen der einzelnen Schritte.

Hinweis: Sie können die Prädikate `member` und `not` aus der Vorlesung, sowie `lengthof` aus Aufgabenteil (a) verwenden.

Beispiellösung:

```
% Aus Vorlesung:
member(X, [X|_]).
member(X, [_|R]) :- member(X, R).

% Aufgabenteil a)
lengthof([], 0).
lengthof([_|R], NewLength) :- lengthof(R, Length), NewLength is Length + 1.

% Aufgabenteil b)
nachbar((X, Y), osten,  (XNeu, Y)) :- XNeu is X + 1, frei((XNeu, Y)).
nachbar((X, Y), westen, (XNeu, Y)) :- XNeu is X - 1, frei((XNeu, Y)).
nachbar((X, Y), norden, (X, YNeu)) :- YNeu is Y + 1, frei((X, YNeu)).
nachbar((X, Y), sueden, (X, YNeu)) :- YNeu is Y - 1, frei((X, YNeu)).

% Vorgegeben:
wege(Schritte, MaxSchritte) :- start(Start), ziel(Ziel),
                               findeweg(Start, Ziel, MaxSchritte, [], Schritte).

% Aufgabenteil c)
findeweg(Ziel, Ziel, _, _, []).
findeweg(Start, Ziel, MaxSchritte, Besucht, [Schritt|Schritte]) :-
        lengthof(Besucht, NumBesucht),
        NumBesucht < MaxSchritte,
        nachbar(Start, Schritt, Nachbar),
        not(member(Nachbar, Besucht)),
        findeweg(Nachbar, Ziel, MaxSchritte, [Nachbar|Besucht], Schritte).

% Alternative (ohne lengthof):
findeweg2(Ziel, Ziel, _, _, []).
findeweg2(Start, Ziel, MaxSchritte, Besucht, [Schritt|Schritte]) :-
        MaxSchritte > 0,
        nachbar(Start, Schritt, Nachbar),
        not(member(Nachbar, Besucht)),
        NeuMaxSchritte is MaxSchritte - 1,
        findeweg2(Nachbar, Ziel, NeuMaxSchritte, [Nachbar|Besucht], Schritte).
```

Hinweis: Die Lösungen für `findeweg` verhalten sich unterschiedlich, wenn für `Besucht` nicht die leere Liste übergeben wird. Die Lösung, die `lengthof` verwendet, betrachtet die Felder in `Besucht` immer implizit als Teil des bereits gefundenen Wegstücks und bezieht sie in die Berechnung der gemachten Schrittzahl ein. Die andere Lösung betrachtet `Besucht` ausschließlich als Liste von "verbotenen" Feldern und berechnet die verbleibende Schrittzahl komplett unabhängig. Im Rahmen der Klausuraufgabe wurden beide Varianten akzeptiert, da innerhalb von `wege` immer nur die leere Liste für `Besucht` übergeben wird.

Aufgabe 4 (C, Deklarationen) [5 Punkte]

Benutzen Sie den in der Vorlesung (VL 4_4, Folie 28) vorgestellten C-Decoder-Ring zum „Entschlüsseln“ der folgenden C-Deklaration. Tragen Sie in unten stehender Tabelle jeweils ein, welches Token Sie mit Hilfe welches Decoder-Elements erkannt haben und wie die Deklaration zu lesen ist.

```
int * (* (*fp1) () ) [10];
```

Schrittnummer (aus Decoder)	**Bearbeitetes Token**	**Gelesener Text**
1	`fp1`	`fp1` is
5	`*`	pointer to
4	`(...)`	—
3	`()`	function returning
5	`*`	pointer to
4	`(...)`	—
2	`[10]`	array of
5	`*`	pointer to
6	`int`	`int`.

Aufgabe 5 (X10, Fibonacci) [5 Punkte]

Folgende X10-Funktion berechnet die n-te Fibonacci-Zahl.

```
1  static def fib(n: Int): Int {
2      if (n <= 1) return n; // fib(0) = 0, fib(1) = 1
3
4      var f1: Int;
5      var f2: Int;
6      ??????????? {
7          ??????????? f1 = fib(n - 1);
8          ??????????? f2 = fib(n - 2);
9      }
10
11     return f1 + f2;
12 }
```

Welche der unten aufgeführten Kombinationen von X10-Schlüsselwörtern können Sie in der angegebenen Reihenfolge an die grau markierten Stellen schreiben, um das Programm zu parallelisieren? Begründen Sie bei den nicht geeigneten Modifikationen kurz, warum diese nicht zur Parallelisierung geeignet sind.

(a) **finish**, **async**, **async** ☐ Geeignet ☐ Nicht geeignet
Geeignet.
Anmerkung: Das zweite **async** ist überflüssig, da die initiale Activity so selbst keine Arbeit verrichtet und nur wartet. Trotzdem werden beide `fib`-Aufrufe potentiell parallel ausgeführt und das korrekte Ergebnis berechnet.

(b) **finish**, **async**, *nichts* ☐ Geeignet ☐ Nicht geeignet
Geeignet.
Anmerkung: Die initiale Activity erstellt für den ersten `fib`-Aufruf eine neue Activity und führt dann, potentiell parallel dazu, den zweiten Aufruf selbst aus.

(c) **finish**, *nichts*, **async** ☐ Geeignet ☐ Nicht geeignet
Nicht geeignet.
Die initiale Activity führt den ersten `fib`-Aufruf selbst aus und erzeugt *anschließend* eine neue Activity für den zweiten Aufruf. Dadurch wird die Berechnung sequentialisiert. Das Programm läuft unter Umständen sogar langsamer als die sequentielle Variante, da zusätzlich noch der Aufwand für das Erstellen einer neuen Activity und das Warten auf deren Beendigung hinzukommt.

(d) *nichts*, **async**, **async** ☐ Geeignet ☐ Nicht geeignet
Nicht geeignet.
Durch den fehlenden **finish**-Block wird nicht auf die Beendigung der neu gestarteten Activities gewartet. Dadurch sind die Werte `f1` und `f2` beim Zugriff in Zeile 11 undefiniert.
Bemerkung: Der X10-Compiler lehnt dieses Programm sogar statisch ab und erzwingt den **finish**-Block.

Aufgabe 6 (MPI, Broadcast) [10 Punkte]

Implementieren Sie die kollektive Operation `MPI_Bcast` mit Hilfe der MPI-Funktionen `MPI_Send`, `MPI_Recv`, `MPI_Comm_size` und `MPI_Comm_rank`. Ergänzen Sie dazu den unten angegebenen Funktionsheader `my_bcast`, so dass ein Aufruf der Funktion die Daten in derselben Weise verteilt wie ein Aufruf von `MPI_Bcast`.

Hinweis: Sie dürfen davon ausgehen, dass `my_bcast` nur mit gültigen Argumenten aufgerufen wird. Sie brauchen sich also nicht um Fehlerbehandlung aufgrund falscher Argumente zu kümmern.

```
void my_bcast(void* data, int count, MPI_Datatype type,
              int root, MPI_Comm comm) {

    int my_rank;
    MPI_Comm_rank(comm, &my_rank);
    int comm_size;
    MPI_Comm_size(comm, &comm_size);

    if (my_rank == root) {
        // If we are the root process, send our data to everyone
        for (int i = 0; i < comm_size; i++) {
            if (i != my_rank) {
                MPI_Send(data, count, type, i, 0, comm);
            }
        }
    } else {
        // If we are a receiver process, receive the data from the root
        MPI_Recv(data, count, type, root, 0, comm, MPI_STATUS_IGNORE);
    }

}
```

Aufgabe 7 (Scala) [10 Punkte]

Das folgende Scala-Programm gibt eine Folge von Zahlen aus. Dazu werden mehrere Aktoren erstellt.

(a) Beschreiben Sie das Verhalten des Programms während der ersten vier Iterationen der **for**-Schleife aus Zeile 30 (`i` $\in \{2, 3, 4, 5\}$). [6 Punkte]
Tragen Sie dazu in die rechts stehende Tabelle für jede von einem Aktor empfangene Nachricht die jeweiligen Werte von `n` und `i` ein und beschreiben Sie kurz das Verhalten innerhalb der `act()`-Methode.

(b) Welche Eigenschaft haben alle in Zeile 11 ausgegebenen Zahlen gemeinsam? [2 Punkte]

(c) Welche Eigenschaft haben alle Zahlen `n`, die in den Aktoren gespeichert sind? [2 Punkte]
Wie sind jeweils welche Aktoren miteinander verknüpft?

```
1  import scala.actors.Actor
2
3  class SE(n: Int) extends Actor {
4      var successor: SE = null
5
6      def act() {
7          while (true) {
8              receive {
9              case i: Int =>
10                 if (i == n) {
11                     println(i)
12                 }
13                 if (i % n != 0) {
14                     if (successor == null) {
15                         successor = new SE(i)
16                         successor.start()
17                     }
18                     successor ! i
19                 }
20             }
21         }
22     }
23 }
24
25 object runIt {
26     def main(args: Array[String]) {
27         var se = new SE(2)
28         se.start()
29
30         for (i <- 2 to 1000)
31             se ! i
32     }
33 }
```

n	**i**	**Verhalten innerhalb von `act()`**
2	2	`i == n`, also wird 2 ausgegeben.
2	3	`i % n == 1`, also wird im Aktor a_2 die `successor`-Referenz mit dem neuen Aktor a_3 initialisiert. Anschließend schickt a_2 die Nachricht 3 weiter an a_3.
3	3	`i == n`, also wird 3 ausgegeben.
2	4	Weder `i == n`, noch `i % n != 0`, also wird die Nachricht 4 ignoriert.
2	5	`i % n != 0`, also schickt Aktor a_2 die Nachricht 5 weiter an seinen `successor`, den Aktor a_3.
3	5	`i % n != 0`, also wird im Aktor a_3 die `successor`-Referenz mit dem neuen Aktor a_5 initialisiert. Anschließend schickt a_3 die Nachricht weiter an seinen `successor`, den Aktor a_5.
5	5	`i == n`, also wird 5 ausgegeben.

Beispiellösung:

(a) Siehe Tabelle. Es sind natürlich auch andere Reihenfolgen korrekt, wenn sie einem gültigen Interleaving der Aktor-Aufrufe entsprechen (z.B. (2,4) vor (3,3) oder (2,4) und (2,5) vor (3,3)).

(b) Sie sind Primzahlen.

(c) Sie sind Primzahlen. Jeder Aktor a_{p_i} ist über die `successor`-Referenz mit dem Aktor $a_{p_{i+1}}$ für die nächstgrößere Primzahl verknüpft.

Aufgabe 8 (Compiler, Haskell, Erzeugung von Java-Bytecode) [16 Punkte]

Gegeben sei folgender Datentyp für ganzzahlige arithmetische Ausdrücke in Baumdarstellung:

```
data Exp = Const Int | Var Int | Neg Exp | Add Exp Exp
```

Ein Ausdruck ist also entweder eine Integer-Konstante, eine lokale Integer-Variable, die Negation eines Ausdrucks oder die Summe zweier Ausdrücke. Der **Int**-Wert bei einer Integer-Variablen `Var` gibt ihren Index im Activation Record der Methode an.

Um Java-Bytecode mit minimalem Stackverbrauch für einen solchen Ausdrucksbaum zu erzeugen, muss bei Knoten, die mehrere Unterbäume haben, immer zunächst Code für den höheren Unterbaum erzeugt werden.

(a) Erzeugen Sie Java-Bytecode mit minimalem Stackverbrauch für den Ausdruck [4 Punkte]

```
Neg (Add (Const 1) (Add (Const 2) (Var 3)))
```

Hinweis: Der Java-Bytecode `ineg` negiert das oberste Element des Operandenstacks.

Beispiellösung:

```
ldc 2
iload 3
iadd
ldc 1
iadd
ineg
```

Anmerkung: Lösungen mit verkürzten Opcodes (`bipush`, `iconst_X`, etc.) werden auch akzeptiert.

(b) Definieren Sie eine Haskell-Funktion [12 Punkte]

```
codegen :: Exp -> [String]
```

die Java-Bytecode mit minimalem Stackverbrauch für den übergebenen Ausdrucksbaum erzeugt. Verwenden Sie die vordefinierte Funktion `height :: Exp -> Int`, um die Höhe eines Ausdrucksbaums zu bestimmen. Sie müssen *keine* Definition von `height` angeben.
Die von `codegen` zurückgelieferte Liste soll die erzeugten Bytecode-Befehle in der richtigen Reihenfolge als Strings enthalten. Verwenden Sie Pattern Matching.

Hinweis: Verwenden Sie die Funktion `show` zur Umwandlung eines **Int** in einen **String**.

Beispiel: `codegen (Neg (Const 42))` $\Rightarrow^{+}$ `[ "ldc 42", "ineg" ]`

Beispiellösung:

```
codegen (Const value) = [ "ldc␣" ++ (show value) ]
codegen (Var index)   = [ "iload␣" ++ (show index) ]
codegen (Neg expr)    = codegen expr ++ [ "ineg" ]
codegen (Add e1 e2)
  | height e1 > height e2 = codegen e1 ++ codegen e2 ++ [ "iadd" ]
  | otherwise             = codegen e2 ++ codegen e1 ++ [ "iadd" ]
```

Aufgabe 9 (Compiler, Syntaktische Analyse) [9 Punkte]

Betrachten Sie die folgende Grammatik (Terminale sind unterstrichen):

$$\begin{array}{lcl} S & \rightarrow & L \ \underline{=} \ R \ \mid \ R \\ L & \rightarrow & \underline{*} \ R \ \mid \ \underline{\mathtt{id}} \\ R & \rightarrow & L \end{array}$$

Die Grammatik beschreibt Zuweisungen in der Programmiersprache C.

Bestimmen Sie die Mengen First_1 und Follow_1 für alle Nichtterminale. Geben Sie für jedes Terminal in den Follow_1-Mengen eine Folge von Ableitungsschritten an, die belegen, dass das Terminal Teil der Follow_1-Menge ist.

Beispiellösung:
First_1-Mengen:

$$\begin{array}{rcl} \mathsf{First}_1(L) & = & \{\underline{*}, \underline{\mathtt{id}}\} \\ \mathsf{First}_1(R) & = & \{\underline{*}, \underline{\mathtt{id}}\} \\ \mathsf{First}_1(S) & = & \mathsf{First}_1(L) \cup \mathsf{First}_1(R) = \{\underline{*}, \underline{\mathtt{id}}\} \end{array}$$

Follow_1-Mengen:

$$\begin{array}{rcl} \mathsf{Follow}_1(L) & = & \{\underline{\#}, \underline{=}\} \\ \mathsf{Follow}_1(R) & = & \{\underline{\#}, \underline{=}\} \\ \mathsf{Follow}_1(S) & = & \{\underline{\#}\} \end{array}$$

Es ist $\underline{\#} \in \mathsf{Follow}_1(L)$, da Ableitung $S \Rightarrow R \Rightarrow L$ möglich.
Es ist $\underline{=} \in \mathsf{Follow}_1(L)$, da Ableitung $S \Rightarrow L \ \underline{=} \ R$ möglich.
Es ist $\underline{\#} \in \mathsf{Follow}_1(R)$, da Ableitung $S \Rightarrow R$ möglich.
Es ist $\underline{=} \in \mathsf{Follow}_1(R)$, da Ableitung $S \Rightarrow L \ \underline{=} \ R \Rightarrow \underline{*}R \ \underline{=} \ R$ möglich.
Es ist $\underline{\#} \in \mathsf{Follow}_1(S)$, da Ableitung S möglich.