Karlsruher Institut für Technologie

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

Prof. Dr. Ralf Reussner
reussner@kit.edu

2 215000 006350

Klausur Programmierparadigmen (Wiederholer)

SS2012, 21. September 2012, 8:00 - 10:00 Uhr

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

Bearbeitungszeit: 120 min

Aufgabe	max. Punkte	err. Punkte
1	25	
2	25	
3	15	
4	5	
5	12	
6	18	
7	20	
Σ	**120**	

Jeder Punkt entspricht ca. 1 min Bearbeitungszeit. Es ist garantiert, dass die Klausur mit 60 Punkten bestanden ist, und dass 110 Punkte für die Note „sehr gut“ ausreichen.

Name: _____

Matrikelnummer: _____

Studiengang: _____

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

Die **Klausureinsicht** findet am 23.10.2012 um 12:30h im Raum 010, Informatik-Gebäude statt.

Aufgabe 1 (Haskell) [25 Punkte]

Teilaufgaben sind unabhängig voneinander lösbar.

(a) Definieren Sie eine Funktion [7 Punkte]

```
hIndex :: [Int] -> Int
```

die, gegeben eine Liste von nicht-negativen Zahlen, die größte Zahl n berechnet, so dass n der Listenelemente größer oder gleich n sind.

Hinweis: Sortieren Sie dazu die Liste absteigend und zählen Sie von vorne durch, bis der Eintrag Nr. i kleiner i ist. Sie dürfen die Hilfsfunktionen **sort** und **reverse** verwenden.

Hintergrundinformation: Der sogenannte *H-Index* oder *Hirsch-Index* ist ein umstrittenes, aber einflussreiches Maß für die Sichtbarkeit eines Wissenschaftlers. So besagt zum Beispiel ein H-Index von mindestens 5, dass der Forscher mindestens fünf Veröffentlichungen hat, die jeweils mindestens fünf mal zitiert wurden. Obige Funktion berechnet den H-Index, wenn jeder Eintrag in der Eingabeliste für die Zahl der Zitationen einer Publikation des Wissenschaftlers steht.

Beispiellösung:

```
helper [] acc = acc
helper (z:ls) acc = if z > acc
                    then helper ls (acc + 1)
                    else acc

hindex l = helper (reverse (sort l)) 0
```

(b) Wir verwenden binäre Bäume nach der Definition aus der Vorlesung: [15 Punkte]

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

Definieren Sie eine Funktion

```
allTrees :: Int -> t -> [Tree t]
```

die eine Liste aller möglichen Bäume von genau der durch den ersten Parameter gegebenen Größe berechnet. Sämtliche Knoten sollen den als zweiten Parameter übergebenen Wert speichern. Die Größe eines Baumes ist die Anzahl seiner Knoten.

Hinweis: Sie können List Comprehensions verwenden.

Beispiellösung:

```
allTrees n x = if n == 0
               then [ Leaf ]
               else [ Node t1 x t2 |
                        i <- [0..n-1],
                        t1 <- allTrees i x,
                        t2 <- allTrees (n-i-1) x
                    ]
```

(c) Definieren Sie die Liste [3 Punkte]

```
trees_4_13 :: [Tree Int]
```

die alle Bäume mit 13 Knoten und Höhe 4 enthält. Sie dürfen `allTrees`, die Standard-Funktionen für Listen und die Hilfsfunktion `height :: Tree t -> Int` verwenden. In den Blättern soll stets der Wert `0` gespeichert sein.

Beispiellösung:

```
trees_4_13 = filter (\t -> height t == 4) (allTrees 13 0)
```

Aufgabe 2 (Prolog) [25 Punkte]

Ein Mann mit einem Wolf, einer Ziege und einem Kohlkopf will mit seinem Boot einen Fluss überqueren. Neben dem Mann hat maximal eines der drei Dinge im Boot Platz. Weiterhin gilt:

- Befinden sich Wolf und Ziege unbeaufsichtigt am gleichen Ufer, so frisst der Wolf die Ziege.
- Befinden sich Ziege und Kohl unbeaufsichtigt am gleichen Ufer, so frisst die Ziege den Kohl.

(Wie) kann der Mann alle drei Dinge heil vom Ufer `links` an das Ufer `rechts` transportieren?

In Prolog lässt sich die Situation vor und nach jeder Flussüberfahrt als Tupel `(Mann,Ziege,Wolf,Kohl)` von Ufern darstellen. Die unerwünschte Situation, dass sich der Mann und der Kohl am rechten, die Ziege und der Wolf jedoch am linken Ufer befinden, z.B. als: `(rechts,links,links,rechts).`

(a) Definieren Sie einen einstelligen *Tester* [7 Punkte]

`erlaubt(Situation)`

welcher für genau die Situationen erfüllt ist, in denen keines der Dinge gefressen wird.

(b) Wir benötigen einen dreistelligen *Generator* `fahrt(S1,F,S2)` [8 Punkte]
der zu Situation `S1` bei Reerfüllung alle Situationen `S2` generiert, die durch einfache Flussüberfahrt entstehen können. Die Zeichenkette `F` beschreibt die Fahrt, besteht also aus dem mitgenommen Ding (oder „leer“, falls der Mann nichts mit nimmt).

Definieren Sie

`fahrt(S1,F,S2)`

Geben Sie dabei lediglich die Regeln für Überfahrten mit Ziege, sowie die für leere Fahrten an.

Beispiel:

```
? fahrt((links,links,links,links),F,S2).
F = 'Ziege',
S2 = (rechts, rechts, links, links) ;

F = 'leer',
S2 = (rechts, links, links, links);

...
```

(c) Das Prädikat `lösung(Fahrten)` generiert alle Lösungen des Problems: [10 Punkte]

```
lösung(Fahrten) :- start(S), ziel(Z), erreichbar(S,[],Fahrten,Z).
start((links,links,links,links)).
ziel((rechts,rechts,rechts,rechts)).
```

Definieren Sie das hierzu benötigte vierstellige Prädikat

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

welches für Start-Situation `S` und Ziel-Situation `Z` erfüllt ist, falls `Z` von `S` durch eine Folge von Flussüber**`fahrt`**en erreichbar ist. Dabei dürfen nur **`erlaubt`**e Zwischensituationen entstehen. Um Endlosschleifen zu vermeiden darf dabei weiterhin keine der in der Liste `Besucht` enhaltenen Situationen nochmals vorkommen. Die Liste `Fahrten` soll bei Erfüllung die Beschreibungen der einzelnen **`fahrt`**en in richtiger Reihenfolge enthalten.

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

Beispiellösung:

```
gegenueber(links,rechts).
gegenueber(rechts,links).

harmlos(_,X,Y) :- gegenueber(X,Y).
harmlos(Ufer,Ufer,Ufer).

erlaubt((Mann,Ziege,Wolf,Kohl)) :- harmlos(Mann,Ziege,Wolf), harmlos(Mann,Ziege,Kohl).

fahrt((U,U,Wolf,Kohl),  "␣Ziege␣", (UNeu,UNeu,Wolf,Kohl)) :- gegenueber(U,UNeu).
fahrt((U, Ziege,Wolf,Kohl), "␣leer␣",  (UNeu,Ziege,Wolf,Kohl)) :- gegenueber(U,UNeu).

erreichbar(S,_,[],S).
erreichbar(S,Besucht,[Fahrt|Fahrten],Z) :-
        fahrt(S,Fahrt,ZwischenS), erlaubt(ZwischenS), not(member(ZwischenS,Besucht)),
        erreichbar(ZwischenS,[ZwischenS|Besucht],Fahrten,Z).
```

Aufgabe 3 (λ-Kalkül: Kombinatoren, β-Reduktion) [15 Punkte]

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

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

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

Weiterhin werden Kombinatoren Φ, B definiert als:

$\Phi = \lambda\mathtt{o}.\ \lambda\mathtt{f}.\ \lambda\mathtt{g}.\ \lambda\mathtt{x}.\ \mathtt{o\ (f\ x)\ (g\ x)} \qquad B = \lambda\mathtt{f}.\ \lambda\mathtt{g}.\ \lambda\mathtt{x}.\ \mathtt{f\ (g\ x)}$

B dient also der Funktionskomposition, Φ der punktweisen Verknüpfung von Funktionen $\mathtt{f}, \mathtt{g}$ durch Operator $\mathtt{o}$. Punktweise Addition zweier Funktionen $\mathtt{f}, \mathtt{g}$ z.B. wird geschrieben: $(\Phi\ (+)\ \mathtt{f\ g})$, denn damit gilt dann: $\Phi\ \mathtt{(+)\ f\ g} \Rightarrow^3 \lambda\mathtt{x}.\ \mathtt{(+)\ (f\ x)\ (g\ x)}$

(a) Geben Sie einen allgemeinsten Typ von Φ an! [7 Punkte]

Hinweis: $\mathtt{o}$ muss nicht unbedingt ein arithmetischer Operator sein

Beispiellösung:

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

(b) Es gilt (bis auf β, η-Konversion): $B = \Phi\ \ I\ \ (K\ (\Phi\ I))\ \ K$ [8 Punkte]
Beweisen Sie dies, indem Sie zeigen, dass für λ-Terme f, g, x gilt:

$$\Phi\ I\ (K\ (\Phi\ I))\ K\ f\ g\ x \Rightarrow^* f\ (g\ x)$$

Beispiellösung:

$$\begin{array}{rr}
\Phi\ I\ (K\ (\Phi\ I))\ K\ f\ g\ x & \Rightarrow^4 \quad I\ ((K\ (\Phi\ I))\ f)\ (K\ f)\ g\ x \\
& \Rightarrow \quad\ \ (K\ (\Phi\ I)\ f)\ (K\ f)\ g\ x \\
& \Rightarrow^2 \quad\ \ \Phi\ I\ (K\ f)\ g\ x \\
& \Rightarrow^4 \quad\ \ I\ (K\ f\ x)\ (g\ x) \\
& \Rightarrow^2 \quad\ \ I\ f\ (g\ x) \\
& \Rightarrow \quad\ \ f\ (g\ x)
\end{array}$$

Aufgabe 4 (Unifikation) [5 Punkte]

Bestimmen Sie einen allgemeinsten Unifikator für die Gleichung:

$$\texttt{a}([1,2,3],[3,4],\texttt{L}) = \texttt{a}([\texttt{X}|\texttt{Xs}],[\texttt{Y}|\texttt{Ys}],\texttt{L2})$$

Verwenden Sie Prolog-Notation: $\texttt{X}, \texttt{Xs}, \texttt{Y}, \texttt{Ys}, \texttt{L}, \texttt{L2}$ sind Variablen, `[_|_]`, `[_,_,_]` etc. Listen.

Beispiellösung: $\sigma = [\texttt{L} \mapsto \texttt{L2}, \texttt{X} \mapsto 1, \texttt{Xs} \mapsto [2,3], \texttt{Y} \mapsto 3, \texttt{Ys} \mapsto [4]]$

Aufgabe 5 (C: Zeiger, Arrays) [12 Punkte]

Welche Ausgabe erzeugt das folgende C-Programm? Machen Sie den Lösungsweg deutlich. Sie können dazu unter die Programmzeilen jeweils Ihre Auswertungen (Variablenbelegungen, Zwischenausdrücke) schreiben.

Hinweis: In `printf("...%s...",x)` ist `%s` Platzhalter für die Zeichenkette `x`

```
1  #include <stdio.h>
2
3  int global[] = {10, 20, 30, 60, 70, 80};
4  char mitteilung[] = "ist nicht sortiert";
5
6  int checkOrder(int local[], int y) {
7
8      global[2] = *(local+2) + y;
9
10     for(int i = 0; i < y; i++)
11        printf ( "%i\n", local[i] );
12
13     for (int i = 0; i < y-1; i++) {
14
15        if (local[i] > local[i+1]) {
16
17           return 4;
18        }
19     }
20
21     return 10;
22 }
23
24 int main() {
25     int compareLength = 4;
26     printf("Die Zahlenfolge ist %s\n",
27            &mitteilung[checkOrder(&global[2], compareLength)]);
28     return 0;
29 }
```

74

74,60,70,80 für i=0,1,2,3

74 60 für i=0

4

4

Ausgabe:

```
74
60
70
80
Die Zahlenfolge ist nicht sortiert
```

Aufgabe 5 (MPI: Matrixmultiplikation) [18 Punkte]

Gegeben sei die 3x3-Matrix A (als **int**-Array `A[3][3]`) und der 3-Vektor v (als **int**-Array `v[3]`).

Die **Matrix-Vektor-Multiplikation** ist wie folgt definiert:

$$e = A * v = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} a_{11}v_1 + a_{12}v_2 + a_{13}v_3 \\ a_{21}v_1 + a_{22}v_2 + a_{23}v_3 \\ a_{31}v_1 + a_{32}v_2 + a_{33}v_3 \end{pmatrix}$$

$$\text{oder auch } e_i = \sum_{j=1}^{3} A_{ij} * v_j \text{ mit } i = 1 \ldots 3$$

Eine sequenzielle Berechnung sieht wie folgt aus:

```
for(int i = 0; i < 3; i++)
   for(int j = 0; j < 3; j++)
      e[i] += A[i][j] * v[j];
```

Die **Matrix-Vektor-Multiplikation** soll mit MPI auf drei Knoten der Communication-Group `comm` (Rank 0, 1, 2) verteilt werden. Dabei berechnet jeder Knoten einen Eintrag des Ergebnisvektors e.

Am Anfang liegen auf dem Masterknoten 0 in `A` die Matrix und in `v` der Vektor vor. Die Berechnung soll folgendermaßen ablaufen:

- Jeder Knoten bekommt von Knoten 0 eine Zeile der Matrix A mit einer kollektiven Operation **(KO1)** übermittelt. Die Knoten speichern die empfangene Zeile jeweils im **int**-Array `a[]`.

  Hinweis: Bei mehrdimensionalen Arrays liegen die Zeilen hintereinander im Speicher.

- Jeder Knoten bekommt von Knoten 0 den Vektor v mit einer kollektiven Operation **(KO2)** übermittelt.

- Jeder Knoten i berechnet in der **int**-Variable `ei` den Eintrag e_i des Ergebnisvektors.
  Die Einträge werden schließlich mit **einer einzigen** kollektiven Operation **(KO3)** auf Knoten 0 im Ergebnisvektor e (**int**-Array `e[]`) zusammengeführt.

Aufgaben:

Schlagen Sie für das beschriebene Verfahren geeignete kollektive Operationen **(KO1, KO2, KO3)** vor. (Jeweils genau eine kollektive Operation!) Geben Sie jeweils den **Namen** und die **komplette Parameterbelegung** an. Die verwendbaren Variablen finden Sie im obigen Aufgabentext (z. B. `A`, `a`, `v`, `e`, `ei`, `comm`).

Beispiellösung:

KO1:	`MPI_Scatter(A, 3, MPI_INT, a, 3, MPI_INT, 0, comm);`	6 Punkte
KO2:	`MPI_Bcast(v, 3, MPI_INT, 0, comm);`	6 Punkte
KO3:	`MPI_Gather(&ei, 1, MPI_INT, e, 1, MPI_INT, 0, comm);`	6 Punkte

Aufgabe 6 (Syntaktische Analyse) [20 Punkte]

Eine Grammatik für einfache reguläre Ausdrücke ist (Terminale sind unterstrichen):

$$R \;\rightarrow\; \underline{\textbf{char}} \mid \underline{\varepsilon} \mid \underline{(}\; R \;\underline{\cup}\; R \;\underline{)} \mid \underline{(}\; R \;\underline{\cdot}\; R \;\underline{)} \mid \underline{(}\; R \;\underline{)}\; \underline{*}$$

Zusammengesetzte Ausdrücke müssen also geklammert werden. Nach Linksfaktorisierung erhält man:

$$\begin{aligned} R &\rightarrow \underline{\textbf{char}} \mid \underline{\varepsilon} \mid \underline{(}\; R\, R' \\ R' &\rightarrow \underline{\cup}\; R \;\underline{)} \mid \underline{\cdot}\; R \;\underline{)} \mid \underline{)}\; \underline{*} \end{aligned}$$

Die *abstrakte* Syntax für reguläre Ausdrücke sei vorgegeben als:

$$\begin{aligned} \mathit{RegExp} &= \mathit{Char} \mid \mathit{Epsilon} \mid \mathit{Union} \mid \mathit{Concatenation} \mid \mathit{KleeneClosure} \\ \\ \mathit{Union} &:: \mathit{RegExp}\ \mathit{RegExp} \\ \mathit{Concatenation} &:: \mathit{RegExp}\ \mathit{RegExp} \\ \mathit{KleeneClosure} &:: \mathit{RegExp} \\ \mathit{Char} &= \textbf{char} \\ \mathit{Epsilon} &= \varepsilon \end{aligned}$$

Implementieren Sie einen Parser für reguläre Ausdrücke. Geben Sie dazu Parser-Prozeduren in Pseudo-Code an.

Der Parser soll als Ergebnis einen abstrakten Syntaxbaum zurückliefern. Verwenden Sie Klassen und Konstruktoren, die der abstrakten Syntax entsprechen. Die globale Variable `token` enthält immer das aktuelle Token. Dieses besitzt die Methode `getType()`, die einen der folgenden Tokentypen zurückgibt:

`CHAR`	für Zeichen ohne besondere Bedeutung, z.B. `a`, `b`, ..., `A`, ...
`EPSILON`	für das Zeichen $\underline{\varepsilon}$
`LP`, `RP`, `UNION`, `DOT` und `STAR`	für die Zeichen $\underline{(}$, $\underline{)}$, $\underline{\cup}$, $\underline{\cdot}$ und $\underline{*}$

Für `CHAR`-Token liefert **`getChar()`** das entsprechende Zeichen. Die globale Prozedur `nextToken()` fordert das nächste Token an. Brechen Sie bei Parsefehlern durch Aufruf der globalen `error()`-Prozedur ohne Fehlermeldung ab.

Beispiellösung:

```
RegExp parseRegExp() {
        switch (token.getType()) {
                case EPSILON:
                        nextToken();
                        return new Epsilon();
                case LP:
                        nextToken();
                        return parseRegExp_(parseRegExp());
                case CHAR:
                        Char c = new Char(token.getChar());
                        nextToken();
                        return c;
                default:
                        error();
        }
}

RegExp parseRegExp_(RegExp left) {
        RegExp rx;
        switch (token.getType()) {
                case UNION:
                        nextToken();
                        rx = new Union(left, parseRegExp());
                        if (token.getType() != RP)
                                error();
                        nextToken();
                        return rx;
                case DOT:
                        nextToken();
                        rx = new Concatenation(left, parseRegExp());
                        if (token.getType() != RP)
                                error();
                        nextToken();
                        return rx;
                case RP:
                        nextToken();
                        rx = new KleeneClosure(left);
                        if (token.getType() != STAR)
                                error();
                        nextToken();
                        return rx;
                default:
                        error();
        }
}
```