berlin

Fakultät IV

Klausur MPGI 1 (A)

09.04.2009

Pepper, Kleeblatt, Frank, Beyer

Name: .....

Vorname: .....

Matr.-Nr.: .....

◯ Äquivalenz für Info A mit TechGI 2 aus dem Semester .....

◯ Äquivalenz für Info A mit TechGI 2 aus einem zukünftigen Semester

Bearbeitungszeit: 90 Minuten

- Benutzen Sie für die Lösung der Aufgaben nur das mit diesem Deckblatt ausgeteilte Papier. **Lösungen, die auf anderem Papier geschrieben werden, können nicht gewertet werden!**
- Schreiben Sie Ihre Lösungen auf das Aufgabenblatt der jeweiligen Aufgabe. Verwenden Sie auch die Rückseiten.
- Schreiben Sie deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen werden nicht gewertet! Streichen Sie gegebenenfalls eine Lösung durch!
- Schreiben Sie nur in **blau** oder **schwarz**. Benutzen Sie nur dokumentenechte Stifte. Lösungen, die mit Bleistift geschrieben sind, werden nicht gewertet!
- Erscheint Ihnen eine Aufgabe mehrdeutig, wenden Sie sich an die Betreuer.
- Sollten Sie eine Teilaufgabe nicht lösen können, so dürfen Sie die dort geforderte Funktion in anderen Teilaufgaben verwenden.
- Tragen Sie zu Beginn der Bearbeitungszeit auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein. **Blätter ohne Namen werden nicht gewertet.**
- Bachten Sie die Hinweise zu Opal-Programmen auf Seite 2.

Aufgabe	Punkte	erreicht
1	14	
2	12	
3	4	
4	7	
5	7	
6	6	
Summe	50	

Hinweise zur Bearbeitung der OPAL-Aufgaben

Bei der Bearbeitung der OPAL-Aufgaben kann auf `IMPORT`-, `SIGNATURE`- und `IMPLEMENTATION`-Deklarationen verzichtet werden, wenn es nicht explizit verlangt wird. Es können alle Funktionen aus der OPAL-Standardbibliothek verwendet werden. Eine **nicht vollständige** Liste der Strukturen und Funktionen aus der Bibliothek ist im Folgenden angegeben.

```
SIGNATURE Denotation
-- -----------------
FUN ++      : denotation ** denotation -> denotation
FUN < > =   : denotation ** denotation -> bool

SIGNATURE NatConv
-- --------------
FUN asReal : nat -> real

SIGNATURE Seq[data]
-- ----------------
SORT data
TYPE seq == <>
            ::(ft : data, rt : seq)
FUN # : seq -> nat
FUN = : (data ** data -> bool) -> seq ** seq -> bool

SIGNATURE SeqMap[from,to]
-- ----------------------
SORT from to
FUN map : (from -> to) -> seq[from] -> seq[to]

SIGNATURE SeqFilter[data]
-- ----------------------
SORT data
FUN filter : (data -> bool) -> seq[data] -> seq[data]

SIGNATURE SeqReduce[from,to]
-- -------------------------
SORT from to
FUN reduce : (from ** to -> to) ** to -> seq[from] -> to

SIGNATURE SeqZip[from1,from2,to]
-- -----------------------------
SORT from1 from2 to
FUN zip : (from1 ** from2 -> to) -> seq[from1] ** seq[from2] -> seq[to]

SIGNATURE Array[data]
-- ------------------
SORT data
SORT array
FUN ! : array ** nat -> data
FUN # : array -> nat

SIGNATURE ArrayReduce[from,to]
-- ---------------------------
SORT from to
FUN / : (from ** to -> to) ** to ** array[from] -> to

SIGNATURE ArrayMap[from,to]
-- ------------------------
SORT from to
FUN * : (from -> to) ** array[from] -> array[to]
```

1. Aufgabe (14 Punkte): Datenstrukturen

Gegeben sind untenstehende Datentypen zur Beschreibung eines einfachen Dateisystems. Eine Verzeichnisstruktur (`struktur`) ist entweder eine Datei (`datei`) oder ein Verzeichnis (`verz`). Jede Datei und jedes Verzeichnis besitzt zunächst einen eindeutigen Namen und Zugriffsrechte (lesend, schreibend, ausführbar). Dateien besitzen zusätzlich eine Dateigröße (`groesse`) und Verzeichnisse bestehen im Allgemeinen aus weiteren Unterverzeichnissen und Dateien.

```
DATA struktur == datei(name    : denotation,
                       rechte  : zugriffsrechte,
                       groesse : nat)

                 verz (name    : denotation,
                       rechte  : zugriffsrechte,
                       inhalt  : seq[struktur])


DATA zugriffsrechte == zugriffsrechte(lesend      : bool,
                                      schreibend  : bool,
                                      ausfuehrbar : bool)
```

1.1. Datentypen (2 Punkte) Um welche Art von Datentyp handelt es sich bei `struktur` und `zugriffsrechte` jeweils. Begründen Sie Ihre Antwort kurz.

1.2. Induzierte Signatur (3 Punkte) Geben Sie für den Datentyp `struktur` die *induzierte Signatur* an und *benennen Sie die einzelnen Teile.*

1.3. Umbenennen einer Datei bzw. eines Verzeichnisses (2 Punkte) Deklarieren und definieren Sie eine Funktion `umbenennen`, die eine `struktur` (Datei oder Verzeichnis) umbenennt, wenn das Zugriffsrecht (`schreibend`) es erlaubt. Andernfalls ist das Umbenennen zu unterlassen.

1.4. Datentypen definieren (2 Punkte) Definieren Sie einen Datentyp `medien`, der die Medientypen Text-, Audio-, Video- und Systemdatei repräsentiert und ergänzen Sie den nachfolgenden Quellcode der Verzeichnisstruktur so, dass jeder Datei ein Medientyp zugeordnet wird.

Um welchen Datentyp handelt es sich bei `medien`?

```
DATA struktur == datei(name    : denotation,
                       rechte  : zugriffsrechte,
                       groesse : nat


                      )

                 verz (name    : denotation,
                       rechte  : zugriffsrechte,
                       inhalt  : seq[struktur]


                      )
```

1.5. Existenz von Dateien (5 Punkte) Deklarieren und definieren Sie eine Funktion `exist?`, die rekursiv prüft, ob eine Datei bestimmten Namens in einer Verzeichnisstruktur enthalten ist.

Schreiben Sie **rekursive Funktionen**. Benutzen Sie **keine** Listenfunktionale.

2. Aufgabe (12 Punkte): Rekursion und Funktionale

Die Daten eines Containterterminals sollen verwaltet werden. Jeder Container wird verwaltet durch eine eindeutige Identifikationsnummer (`id`), seinen Versicherungswert (`wert`), sowie Reihenbezeichner (`reihe`) und Stellplatznummer in dieser Reihe (`platz`). Die Container eines gesamten Terminals werden als eine Sequenz von Containern (`seq[container]`) dargestellt.

```
DATA container == container(id   :nat,
                            wert :real,
                            reihe:denotation,
                            platz:nat)
```

2.1. Durchschnittlicher Versicherungswert (rekursiv) (3 Punkte) Deklarieren und definieren Sie eine rekursive Funktion `mittelwert`, die den durchschnittlichen Wert der Container eines Terminals (`seq[container]`) bestimmt. Schreiben Sie **rekursive Funktionen**. Benutzen Sie **keine** Listenfunktionale.
Standardlistenfunktionen (d. h. Funktionen, die keine Funktionale sind) sind erlaubt.

2.2. Container finden (1 Punkt) Definieren Sie eine Funktion `containerReihe`, die aus einer Sequenz von Containern alle Container zurückliefert, die in einer als Parameter zu übergebenden Reihe stehen. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

```
FUN containerReihe : seq[container] ** denotation -> seq[container]
```

2.3. Höchster Versicherungswert (2 Punkte) Deklarieren und definieren Sie ein Funktion `maxWert`, die aus einem Terminal (Sequenz von Containern) den höchsten Versicherungswert zurückliefert, den ein Container besitzt. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

2.4. Wertvollste Container (2 Punkte) Deklarieren und definieren Sie eine Funktion `wertvoll`, die aus einer Sequenz von Containern die Reihe und den Platz des Containers mit dem höchsten Versicherungswert liefert.

Es kann angenommen werden, dass es nur einen solchen Container gibt. Benutzen Sie dazu die Funktion `maxWert` aus der vorigen Unteraufgabe. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

2.5. Containerzahl (2 Punkte) Definieren Sie eine Funktion `anzahl`, die für ein Array von Terminals die Anzahl der insgesamt in allen Terminals lagernden Container liefert.

```
FUN anzahl : array[seq[container]] -> nat
```

2.6. Versicherungswerte von Terminals (2 Punkte) Deklarieren und definieren Sie eine Funktion `werte`, die für ein Array von Terminals und eine Liste von Terminalnummern (Indizes) eine Sequenz der Gesamtversicherungswerte der Terminals mit diesen Indizes liefert. Alle nicht in der Liste aufgeführten Terminals sollen also ignoriert werden.

3. Aufgabe (4 Punkte): Tailrekursion

Gegeben ist folgende Funktion `f`.

```
FUN f : seq[denotation] -> denotation

DEF f(<>)      == ""
DEF f(x :: xs) == f(xs) ++ x
```

3.1. Umformung (3 Punkte) Definieren sie eine Funktion `g` mit demselben Typ wie `f`. Diese Funktion `g` soll *nicht* rekursiv sein, sondern nur eine Hilfsfunktion `h` mit geeigneten Parametern aufrufen, so dass `g(L)` das gleiche Resultat liefert wie `f(L)` für beliebige Listen `L`.

Deklarieren und definieren sie die benötigte Hilfsfunktion `h`. Diese soll *nur Tailrekursion* (auch bekannt als Endrekursion oder repetitive Rekursion) verwenden.

Sie müssen keine Herleitung und keinen Rechenweg angeben.

3.2. Vorteile (1 Punkt) Nennen sie kurz die Vorteile der tailrekursiven (bzw. endrekursiven oder repetitiv rekursiven) Umformung gegenüber der linear rekursiven Funktion `f`.

4. Aufgabe (7 Punkte): Terminierung und Aufwand

4.1. Terminierungsbeweis (5 Punkte) Gegeben sind die nachfolgenden Funktionen `double` und `sub`. Untersuchen Sie, ob die gegebenen Funktionen terminieren. Falls die jeweilige Funktion terminiert, geben Sie eine Terminierungsfunktion an und benutzen Sie diese, um zu zeigen, dass die Funktion terminiert. Andernfalls zeigen Sie, dass die Funktion nicht terminiert.

```
FUN double: seq[int] -> seq[int]
DEF double(<>)   == <>
DEF double(h::t) == (2*h)::double(t)
```

```
FUN sub: int ** int -> int
DEF sub(a,b) == IF b=0 THEN a
                IF b<0 THEN sub(a+1, b+1)
                IF b>0 THEN sub(a-1, b-1)
                FI
```

4.2. Aufwandsklassen (2 Punkte) Gegeben sind untenstehende Funktionsdefinitionen. Geben Sie an, in welchen Aufwandsklassen die Laufzeiten der Funktionen jeweils liegen. Wählen Sie Aufwandsklassen, die das tatsächliche Laufzeitverhalten möglichst gut beschreiben. Sie müssen keinen Rechenweg angeben.

	Rekurrenzrelation		$A \in$
1	$A(n) = A(n-1) + bn^k$		$\mathcal{O}(n^{k+1})$
2	$A(n) = cA(n-1) + bn^k$	mit $c > 1$	$\mathcal{O}(c^n)$
3	$A(n) = cA(n/d) + bn^k$	mit $c > d^k$	$\mathcal{O}(n^{\log_d c})$
4	$A(n) = cA(n/d) + bn^k$	mit $c < d^k$	$\mathcal{O}(n^k)$
5	$A(n) = cA(n/d) + bn^k$	mit $c = d^k$	$\mathcal{O}(n^k \log_d n)$

```
FUN f : nat -> nat
DEF f(0) == 0
DEF f(n) == f(n-1) + f(n-1) + 2
```

```
FUN g : nat -> nat
DEF g(0) == 0
DEF g(n) == g(n/2) + n + 2
```

5. Aufgabe (7 Punkte): Suchbäume

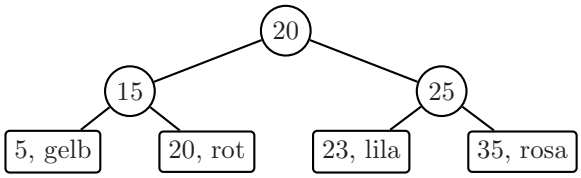
5.1. Varianten (3 Punkte) Geben Sie für folgende Varianten von Suchbäumen an, ob sie balanciert sind, und wenn ja, wie Balanciertheit in diesem Fall definiert ist.

a) einfache binäre Suchbäume

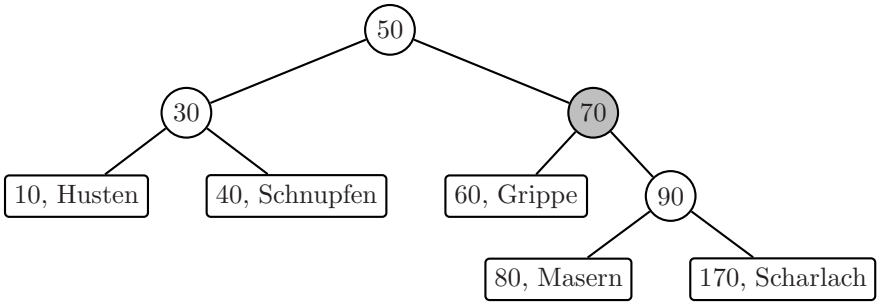
b) 2-3-Bäume

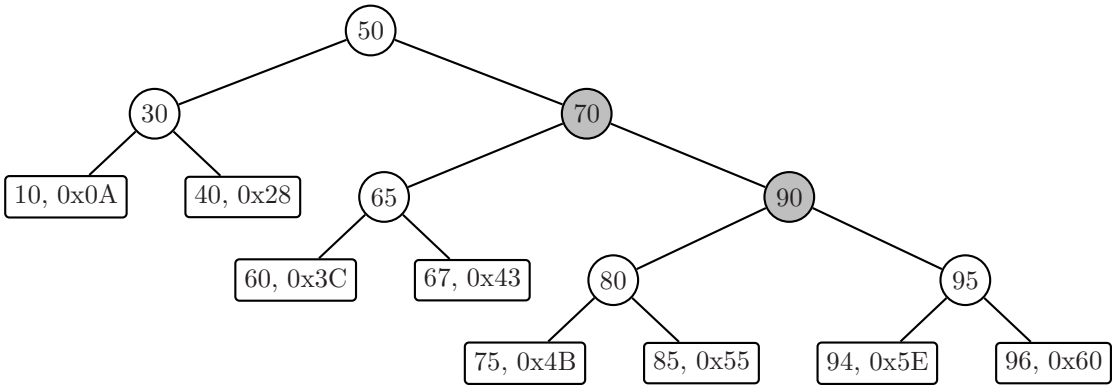
c) Rot-Schwarz-Bäume

5.2. Einfügen in binäre Suchbäume (2 Punkte) Geben Sie an, wie der Suchbaum aussieht, der entsteht, wenn in untenstehenden binären, blattorientierten Suchbaum die Schlüssel-Werte-Paare (10, grün) und (30, blau) eingefügt werden. Der resultierende Baum muss nicht balanciert sein.



5.3. Invarianten in Rot-Schwarz-Bäumen (2 Punkte) Geben Sie für folgende Bäume an, ob sie gültige Rot-Schwarz-Bäume sind, und falls nicht, welche Invariante verletzt wird. In den Abbildungen sind rote Knoten grau hinterlegt, schwarze Knoten sind normal dargestellt.





6. Aufgabe (6 Punkte): Parser

6.1. Beispiele (1 Punkt) Nennen sie zwei konkrete Beispiele für die Verwendung von Parsern.

6.2. Implementierung (5 Punkte) Es sei folgender Datentyp für blattorientierte Binärbäume gegeben.

```
DATA tree == leaf(n : nat)
             node(left: tree, right : tree)
```

Ein Parser soll derartige Bäume anhand folgender Grammatik erkennen.

$$\begin{aligned} \mathit{Tree} &\rightarrow \texttt{value} \\ &\mid \texttt{open}\ \mathit{Tree}\ \mathit{Tree}\ \texttt{close} \end{aligned}$$

Dabei wird folgender Typ für die Token verwendet.

```
DATA token == value(number : nat)
              open
              close
```

Beispiel: Die Sequenz „`open :: open :: value(1) :: value(2) :: close :: value(3) :: close :: <>`“ soll als Baum „`node(node(leaf(1), leaf(2)), leaf(3))`“ erkannt werden.

Definieren Sie die Funktion `parseTree`, die für die Erkennung von Binärbäumen notwendig ist. Dabei können Sie davon ausgehen, dass die Tokensequenzen korrekt sind und auf Fehlerbehandlungen verzichten.

```
FUN parseTree : seq[token] -> tree ** seq[token]
```