berlin

# Fakultät IV

# Klausur MPGI 1 (A)

# 20.02.2009

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 | 11 | |
| 2 | 9 | |
| 3 | 7 | |
| 4 | 4 | |
| 5 | 5 | |
| 6 | 6 | |
| 7 | 8 | |
| 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 Seq[data]
-- ----------------
SORT data
TYPE seq == <>
            ::(ft : data, rt : seq)
FUN #       : seq -> nat
FUN ++      : seq ** seq -> seq
FUN =       : (data ** data -> bool) -> seq ** seq -> bool
FUN ft last : seq -> data
FUN split   : nat ** seq -> seq ** seq
FUN +%      : seq ** data -> seq

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 Com[data:SORT]
-- ----------------------
SORT data com
FUN succeed : data -> com

SIGNATURE BasicIO
-- --------------
FUN write: nat -> com[void]
FUN ask: denotation -> com[nat]
FUN ask: denotation -> com[denotation]

SIGNATURE ComCompose[first,second]
-- -------------------------------
SORT first second
FUN & : com[first] ** (first -> com[second]) -> com[second]
FUN ; : com[first} ** (ans[first] -> com[second]) -> com[second]
```

# 1. Aufgabe (11 Punkte): Datenstrukturen

Gegeben sind untenstehende Datentypen zur Verwaltung von Bankkonten. Ein Bankkonto ist zunächst durch seinen Typ (z.B. Tagesgeld, Sparbuch oder Girokonto) gekennzeichnet. Hinzu kommen Verwaltungsdaten, wie die eindeutige Identifikationsnummer (`kontoNr`) und eine Liste der bisherigen Transaktionen des Kontos.

```
DATA kontoart == tagegeld sparbuch giro

DATA konto == konto(typ:kontoart,
                    kontoNr:nat,
                    aktionen:seq[transaktion])

DATA transaktion == buchung(bId:nat,
                            zeit:time,
                            summe:real)
```

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

**1.2. Induzierte Signatur (2 Punkte)** Geben Sie für den Datentyp `konto` die *induzierte Signatur* an und *benennen Sie die einzelnen Teile.*

**1.3. Datentypen definieren (2 Punkte)** Definieren Sie einen Datentyp `kunde`, der einen Bankkunden repräsentiert. Ein solcher Kunde sei definiert durch eine eindeutige Kundennummer `kId` vom Typ `nat`, einen Namen des Typs `denotation`, und eine *beliebige Anzahl* von Konten (Typ `konto`). Der Typ `kunde` soll unterscheiden zwischen Privatkunden (`privatKunde`) und Firmenkunden (`firmenKunde`). Firmenkunden besitzen als zusätzliches Attribut eine Steuernummer (`steuerId`) des Typs `denotation`.

Um welchen Datentyp handelt es sich?

**1.4. Hinzufügen von Konten (5 Punkte)** Deklarieren und definieren Sie eine Funktion `addKonto`, die zu einem Bankkunden (`kunde`) ein Konto hinzufügt. Um doppelte Zuweisungen zu vermeiden, muss sichergestellt werden, dass ein Konto mit identischer `kontoNr` nicht schon dem Kunden zugewiesen wurde. In diesem Fall ist das Hinzufügen zu unterlassen.

Für die Überprüfung der Existenz einer Kontonummer in einer Kontensequenz, ist eine Hilfsfunktion `exist?` zu implementieren.

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

Betrachten Sie nur den Fall für **Privatkunden** (Typ-Variante `privatKunde`).

## 2. Aufgabe (9 Punkte): Listenfunktionale

Gegeben sind untenstehende Datentypen zur Verwaltung der Vorgänge im Meldeamt. Im Typ `anliegen` werden die Standardanliegen zusammengefasst. Die einzelnen Aufträge mit ihren Eigenschaften werden mit dem Datentyp `auftrag` definiert. Für jeden Auftrag werden die Art der Aktivität (`art`), der bearbeitende Schalter (`schalter`), die Wartezeit bis zum Bearbeitungsbeginn in Sekunden (`wZeit`) und die eigentliche Bearbeitungszeit in Sekunden (`bZeit`) vermerkt.

```
DATA anliegen == pass ausweis meldung bescheinigung

DATA auftrag == auftrag(art:anliegen,
                        schalter:nat,
                        wZeit:nat,
                        bZeit:nat)
```

**2.1. Zeiterfassung (2 Punkte)** Deklarieren und definieren Sie eine Funktion `time`, die aus einer Sequenz von Aufträgen, die Summe der Wartezeiten für alle Schalter liefert. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

**2.2. Schneller Service (2 Punkte)** Deklarieren und definieren Sie eine Funktion `fastService`, die aus einer Sequenz von Aufträgen, diejenigen Aufträge herausfiltert, deren Bearbeitungszeit unterhalb einer als Parameter zu übergebenen Zeit liegt. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

**2.3. Schnellster Schalter (5 Punkte)** Deklarieren und definieren Sie eine Funktion `fastest`, die aus einer Sequenz von Aufträgen und einem als Parameter zu übergebenen Diskriminator für Anliegen (`anliegen -> bool`), denjenigen Schalter ermittelt, der für dieses Anliegen die kleinste Bearbeitungszeit (`bZeit`) hat. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

# 3. Aufgabe (7 Punkte): Polymorphe Strukturen

**3.1. Duplikate entfernen (4 Punkte)** Implementieren Sie in einer polymorphen (dass heißt parametrisierten) Struktur `SeqDups` eine Funktion `filterdups`, die aus einer Sequenz sämtliche Duplikate entfernt; d.h., wenn ein Element mehrfach vorkommt, soll nur das erste erhalten bleiben. Die Implementierung soll mit verschiedenen Funktionen für den Gleichheitstest arbeiten können. Geben Sie die vollständige Implementierung (inklusive optionaler Parameter und aller Deklarationen) der Struktur an. Sie können allerdings auf die Deklaration benötigter Importe verzichten.

**3.2. Anwendung der polymorphen Struktur (3 Punkte)** Deklarieren und definieren Sie eine Funktion `natdups`, die aus einer Sequenz von *natürlichen Zahlen* sämtliche Duplikate eines mehrfach vorkommenden Elementes entfernt. Benutzen Sie hierfür die Struktur aus der vorigen Aufgabe und geben Sie *alle notwendigen Importe* an.

# 4. Aufgabe (4 Punkte): Typen

**4.1. Spaltensumme (1 Punkt)** Gegeben sei der folgende Funktion zum Berechnen der Spaltensumme einer Matrix, die als Sequenz von Sequenzen gegeben ist.

```
FUN spaltensumme : seq[seq[int]] -> seq[int]
DEF spaltensumme(M) == reduce(raetsel, ft(M))(rt(M))


DEF raetsel(a,b) == zip(+)(a,b)
```

Geben Sie die fehlende Deklaration der Funktion `raetsel` an.

**4.2. Quicksort (3 Punkte)** Gegeben sei der folgende OPAL-Code zum Sortieren einer Sequenz.

```
FUN sort : seq[nat] -> seq[nat]
DEF sort(<>) == (<>)
DEF sort(L) == combine(ft(L))(apply(sort)(partition(ft(L), rt(L))))


DEF partition(e, <>) == (<>, <>)
DEF partition(e, h::T) == LET (L, R) == partition(e, T) IN
  IF h<e THEN (h::L, R) ELSE (L, h::R) FI


DEF apply(f)(l,r) == (f(l), f(r))


DEF combine(M)(L,R) == L ++ (M :: R)
```

Geben Sie die fehlenden Deklarationen der Funktionen `partition`, `apply` und `combine` an.

# 5. Aufgabe (5 Punkte): Aufwand

**5.1. Aufwandsklassen (3 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) ^ 2 - f(n-1)
```

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

```
FUN h : seq[nat] -> nat
DEF h(<>) == 1
DEF h(a :: A) == a * h(A)
```

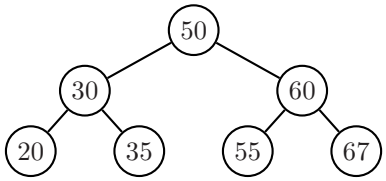
**5.2. Laufzeiten (2 Punkte)** Für die Verwaltung der Datenbank mit den Bestellungen eines Internet-Buchhändlers stehen zwei Implementierungen zur Verfügung. Hersteller A gibt die Aufwandsklasse seiner Implementierung mit $\mathcal{O}(n^2)$ an, Hersteller B mit $\mathcal{O}(n \log n)$. Dabei bezeichnet $n$ in beiden Fällen die Anzahl der Bestellungen in der Datenbank.

Wenn Sie nicht wissen, wie viele Bestellungen in Zukunft bearbeitet werden müssen, und auch sonst keine Details über die beiden Implementierungen bekannt sind: Welche Implementierung würden Sie kaufen? Begründen Sie Ihre Antwort.

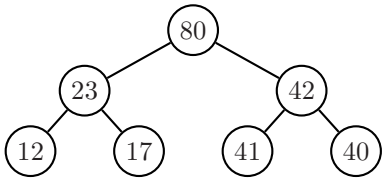
# 6. Aufgabe (6 Punkte): Heaps

**6.1. Invarianten und Darstellung (4 Punkte)** Geben Sie für folgende Bäume jeweils an, ob sie alle Heap-Invarianten erfüllen. Wenn nicht, geben Sie an, welche Invariante verletzt wird. Wenn alle Invarianten erfüllt werden, geben Sie den Heap in der Array-Darstellung an.

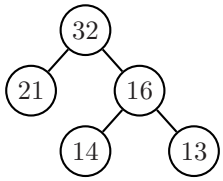
a)



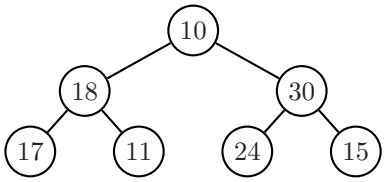

b)



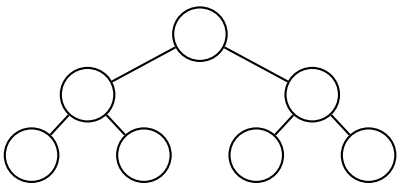

c)



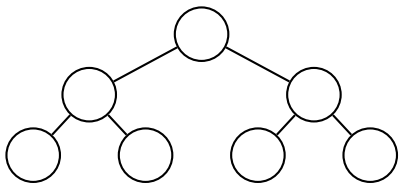

**6.2. Reparatur (2 Punkte)** Der folgende Baum erfüllt die Heap-Invarianten nicht. Mit der `sink`-Operation können diese wiederhergestellt werden. Simulieren Sie diese Reparatur. Zeigen Sie den resultierenden Baum nach jeder durchgeführten Vertauschung von zwei Elementen. Wenn Sie weniger als vier Schritte benötigen, lassen Sie die nicht benötigten Bäume einfach leer.



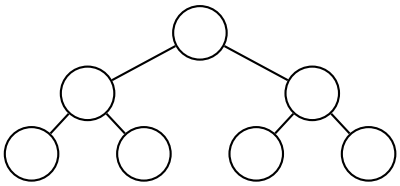

Schritt ___:



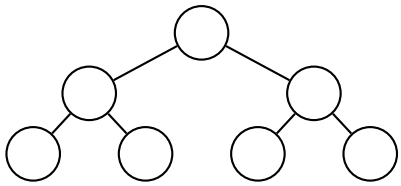
Schritt ___:



Schritt ___:



Schritt ___:



# 7. Aufgabe (8 Punkte): Ein- und Ausgabe

**7.1. Typ `com` (1 Punkt)** Erläutern Sie kurz die Rolle des polymorphen (dass heißt parametrisierten Typs `com[alpha]` und seines Typparameters `alpha`.

**7.2. Bind-Operator (2 Punkte)** Erläutern Sie kurz die Funktion des Operators `&` mit untenstehender Signatur in folgendem Kontext: `A & (\\ x . B(x))`.

```
FUN & : com[alpha] ** (alpha -> com[beta]) -> com[beta]
```

**7.3. Multiplikator (5 Punkte)** Deklarieren und definieren Sie ein Kommando `multiplikator` mit folgender Funktionsweise: Zunächst wird eine Zahl von der Tastatur eingelesen. Dann soll eine entsprechende Anzahl von Werten von der Tastatur eingelesen werden. Abschließend wird das Produkt dieser Werte ausgegeben.