Fakultät IV

Probeklausur MPGI 1
12.02.2014

Prof. Dr. Pepper
Höger, Reicherdt, Zuber, Tutoren

Name: .....

Vorname: .....

Matr.-Nr.: .....

Bearbeitungszeit: 90 Minuten

- Hilfsmittel jeglicher Art sind **nicht erlaubt**.
- 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. Wenn Sie zusätzliche, von uns ausgegebene Blätter verwenden, geben Sie unbedingt an, zu welcher Aufgabe die Lösung gehört!
- Schreiben Sie deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen werden nicht gewertet! Streichen Sie gegebenenfalls eine Lösung durch!
- Schreiben Sie nur in **blau** oder **schwarz**. 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.
- Wenn Sie die Heftung der Klausur entfernen, tragen Sie auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein. **Lose Blätter ohne Namen werden nicht gewertet.**
- Bei der Bearbeitung der Opal-Aufgaben kann auf `IMPORT`-Deklarationen verzichtet werden, sofern es von der Aufgabe nicht anders gefordert ist.

  Es können alle Funktionen aus der Bibliotheca Opalica benutzt werden, soweit es nicht anders angegeben ist.
- Am Ende der Klausur befindet sich ein Referenzblatt (S. 11), das Opal-Strukturen aufführt, die in mehreren Aufgaben verwendet werden. Zum leichteren Nachschlagen können Sie dieses Blatt von den restlichen Blättern abtrennen.

  Weiterhin befindet sich auf dieser Seite ein kurzer Auszug aus der Bibliotheca Opalica.

Aufgabe	Thema	Punkte	erreicht
1	Datenstrukturen	3	
2	Binäre Suchbäume	12	
3	Listenfunktionale	6	
4	Eingabeverarbeitung	8	
5	Ein- und Ausgabe	5	
6	Aufwandsanalyse	4	

1. Aufgabe (3 Punkte): Datenstrukturen

Zur Verwaltung der Bücher einer Bibliothek sind Ihnen die beiden Datentypen `book` und `person` gegeben. Diese Datentypen finden noch in späteren Aufgaben Verwendung und sind auch auf dem Referenzblatt/S. 11 zu finden.

```
TYPE book == book(author : person,      -- Autor des Buches
                  title  : denotation,  -- Titel des Buches
                  year   : nat          -- Erscheinungsjahr
                 )

TYPE person == person(first : denotation,  -- Vorname
                      last  : denotation,  -- Nachname
                      birth : nat          -- Geburtsjahr
                     )
```

1. Zu welcher Art von Datentypen gehören `book` und `person`?
2. Geben Sie die induzierte Signatur für den Datentyp `book` an.

2. Aufgabe (12 Punkte): Binäre Suchbäume

In dieser Aufgabe programmieren Sie einen generischen binären Suchbaum, der Daten ausschliesslich in den Blättern speichert. Die Wurzel und die inneren Knoten enthalten die Schlüssel, unter denen die Daten zu finden sind. Dabei sind die Schlüssel im linken Unterbaum kleiner als der Schlüssel der Wurzel, im rechten Unterbaum größer oder gleich dem Schlüssel der Wurzel.

Die Abb. 1 zeigt einen Beispielbaum, der Bücher (`book`) mit natürlichen Zahlen als Schlüssel speichert.

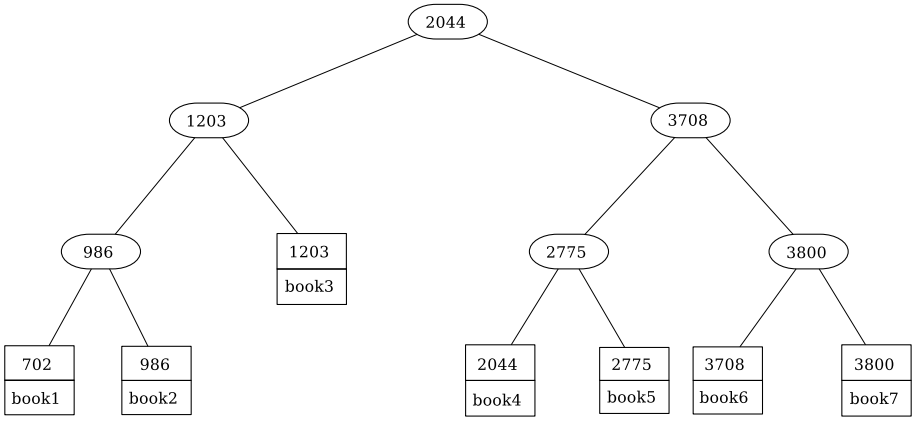

Abbildung 1: Beispiel für einen Binärbaum zur Speicherung von Büchern.

2.1. Baumdatenstruktur (4 Punkte) Definieren Sie eine Struktur `BinSearchTree` mit einem **abstrakten** generischen Datentypen `bst` *(binary search tree)* für die oben beschriebene Art von Suchbäumen. Der Baum soll sowohl Schlüssel als auch Daten beliebigen Typs enthalten können. Nach außen hin soll ausschließlich eine Funktion zum Erzeugen eines Baums aus einem Schlüssel und einem Datum zur Verfügung gestellt werden. Geben Sie sowohl die Signatur als auch die Implementierung vollständig an.

2.2. *Inorder*-Traversierung (1 Punkt) Geben Sie die Schlüssel aller besuchten Knoten in der Reihenfolge an, die einer *Inorder*-Traversierung des Baumes aus Abb. 1 entspricht.

2.3. Umwandlung in Listen (3 Punkte) Deklarieren und definieren Sie eine Funktion `asLists`, die einen Baum vom Typ `bst` in ein Tupel aus zwei Sequenzen umwandelt: Die erste Sequenz enthält alle Schlüssel der Blätter des Baumes in aufsteigender Reihenfolge, die zweite Sequenz enthält die zugehörigen Daten.

Für den Beispielbaum aus Abb. 1 liefert `asLists` die beiden Sequenzen

```
(<702,986,1203,2044,2775,3708,3800>, <book1,book2,book3,book4,book5,book6,book7>)
```

2.4. Einfügen (4 Punkte) Deklarieren und definieren Sie eine Funktion `update`, die als Argumente einen Baum vom Typ `bst` sowie einen einzufügenden Schlüssel und ein Datum bekommt. Die Funktion `update` fügt das Datum unter dem angegebenen Schlüssel in den Baum ein und gibt den resultierenden Baum zurück.

Ist der Schlüssel bereits im Eingabebaum vorhanden, soll das entsprechende Datum ersetzt werden.

3. Aufgabe (6 Punkte): Listenfunktionale

Gegeben sind Ihnen wieder die Datentypen `book` und `person` (siehe Referenzblatt/S. 11).

3.1. Suche nach Autor (2 Punkte) Deklarieren und definieren Sie eine Funktion `getBooks`, die einen Vornamen und Nachnamen als `denotation` sowie eine Sequenz von Büchern übergeben bekommt und nur die Bücher des entsprechenden Autors zurück gibt. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

3.2. Ausgabe aller Titel (2 Punkte) Definieren Sie eine Funktion `printTitles` mit folgender Funktionalität:

```
FUN printTitles : seq[book] -> denotation
```

`printTitles` bekommt eine Sequenz von Büchern übergeben und soll die Titel aller Bücher als `denotation` durch Kommata getrennt zurückgeben. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

3.3. Rekursion → Listenfunktionale (2 Punkte) Gegeben ist folgende OPAL-Funktion:

```
FUN authorsAge : seq[book] -> seq[pair[denotation,nat]]
DEF authorsAge(<>)    == <>
DEF authorsAge(x::xs) ==
    LET age == year(x) - birth(author(x))
    IN (title(x) & age) :: authorsAge(xs)
```

Wandeln Sie diese Funktion mit Hilfe von Listenfunktionalen in eine Funktion ohne Rekursion um.

4. Aufgabe (8 Punkte): Eingabeverarbeitung

Zur Erweiterung der Bibliotheksdatenbank wird eine Funktion benötigt, die eine Liste von Autoren einliest. Diese Liste liegt als Textdokument vor, das nach den Regeln folgender Grammatik aufgebaut ist:

$$\begin{aligned} \mathit{List} &\rightarrow \mathit{Person}^* \\ \mathit{Person} &\rightarrow \mathit{Name}\ \underline{,}\ \mathit{Name}\ \underline{(}\ \mathit{Num}\ \underline{)}\ \underline{;} \end{aligned}$$

Dabei bezeichnet das linke *Name* den Nachnamen, das rechte *Name* den Vornamen und *Num* das Geburtsjahr einer Person.

Eine Liste aus drei Autoren könnte bspw. so aussehen:

```
Brecht, Bertolt (1898); Shakespeare, William (1564); Dürrenmatt, Friedrich (1921);
```

Die Scanning-Funktion liefert Ihnen die Eingabeliste als eine Sequenz von Tokens des Datentyps `token`:

```
TYPE token == name(name : denotation)  -- Name
              num(val : nat)           -- Num
              comma                    -- ,
              open                     -- (
              close                    -- )
              semicolon                -- ;
```

Definieren Sie eine Funktion `parseList`, die eine Sequenz von Tokens in eine Sequenz von Personen umwandelt:

```
FUN parseList : seq[token] -> seq[person]
```

Gehen Sie davon aus, dass in der Eingabesequenz keine Syntaxfehler enthalten sind. Das heißt, Sie müssen in Ihrem Parser keine Fehlerbehandlung vorsehen.

Hinweis: Bedenken Sie, dass ein Parser in der Regel aus mehreren Funktionen besteht. Geben Sie für jede von Ihnen zusätzlich definierte Funktion den Typ an.

5. Aufgabe (5 Punkte): Ein- und Ausgabe

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.

6. Aufgabe (4 Punkte): Aufwandsanalyse

	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 $c = d^k$	$\mathcal{O}(n^k \log_d n)$

Abbildung 2: Rekurrenzrelationen zur Bestimmung von Aufwandsklassen.

Bestimmen Sie den Aufwandsterm der unten abgedruckten OPAL-Funktion `f` und leiten Sie daraus mit Hilfe der Rekurrenzrelationen aus Abb. 2 die Aufwandsklasse ab. Zur korrekten Lösung gehört auch die Angabe der verwendeten Zeile in der Tabelle, die Belegung der Variablen und die Angabe der daraus resultierenden Aufwandsklasse.

```
FUN f : nat -> nat
DEF f(0) == 1
DEF f(n) == g(n) + f(n-1) + 23

FUN g : nat -> nat
DEF g(n) == f(n-1) + 5
```

Geben Sie zusätzlich an, welche Rekursionsart die Funktion `f` verwendet.

Name: .....
Matr.-Nr. .....

Referenzblatt

Sie können dieses Blatt zum leichteren Nachschlagen von den restlichen Blättern der Klausur abtrennen.

Wichtige Strukturen

```
SIGNATURE Book

TYPE book == book(author : person, title : denotation, year : nat)
```

```
SIGNATURE Person

TYPE person == person(first : denotation, last : denotation, birth : nat)
```

Auszug aus der Bibliotheca Opalica

```
SIGNATURE Seq[data]

SORT data
TYPE seq == <>
            ::(ft : data, rt : seq)
FUN #  : seq -> nat
FUN ++ : seq ** seq -> 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 data com
FUN succeed : data -> com[data]
```

```
SIGNATURE BasicIO

FUN write     : denotation -> com[void]
FUN writeLine : denotation -> 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] ** com[second] -> com[second]
```

```
SIGNATURE Pair [data1,data2]

SORT data1 data2
TYPE pair == &(1st: data1, 2nd: data2 )
```