Fakultät IV

Probeklausur MPGI 1
06.02.2013

Prof. Dr. Glesner
Göthel, Hostettler, Tetzlaff

Name: .....

Vorname: .....

Matr.-Nr.: .....

Bearbeitungszeit: 75 Minuten

- Als Hilfsmittel sind zwei doppelseitig handschriftlich beschriebene DIN-A4 Seiten erlaubt. Elektronische Geräte wie Laptops, Mobilfunktelefone jeglicher Art oder Netbooks 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	8	
3	Listenfunktionale	5	
4	Eingabeverarbeitung	5	
5	Ein- und Ausgabe	5	
6	Aufwandsanalyse	4	
Summe		30	

1. Aufgabe (3 Punkte): Datenstrukturen

Zur Verwaltung von Politiker-Gehältern sind Ihnen die beiden Datentypen `politician` (Politiker) und `salary` (Gehalt) gegeben. Diese Datentypen finden noch in späteren Aufgaben Verwendung und sind auch auf dem Referenzblatt/S. 11 zu finden.

```
TYPE salary == salary(euro  : nat,  -- Euro
                      cent  : nat,  -- Cent
                      bonus : nat   -- Bonus in Euro
                     )
```

```
TYPE politician == pol(first : denotation,  -- Vorname
                       last  : denotation,  -- Nachname
                       wage  : salary       -- Gehalt
                      )
```

1. Zu welcher Art von Datentyp gehört `salary`? Produkttyp
2. Geben Sie die induzierte Signatur für den Datentyp `politician` an und benennen Sie die Komponenten.

```
--Sorte
SORT politician

-- Konstruktor
FUN pol : denotation ** denotation ** salary -> politician

-- Diskriminator
FUN pol? : politician -> bool

-- Selektoren
FUN first last : politician -> denotation
FUN wage       : politician -> salary
```

2. Aufgabe (8 Punkte): Binäre Suchbäume

In dieser Aufgabe programmieren Sie einen generischen binären Suchbaum. **Alle** Knoten enthalten **sowohl Daten als auch** die **Schlüssel**, mit denen die Daten sortiert sind. Dabei sind die Schlüssel im linken Unterbaum kleiner als der Schlüssel des aktuellen Knotens, im rechten Unterbaum größer.

Die Abbildung 1 zeigt einen Beispielbaum, der als Daten Politiker (`politician`) mit einer natürlichen Zahl als Schlüssel speichert.

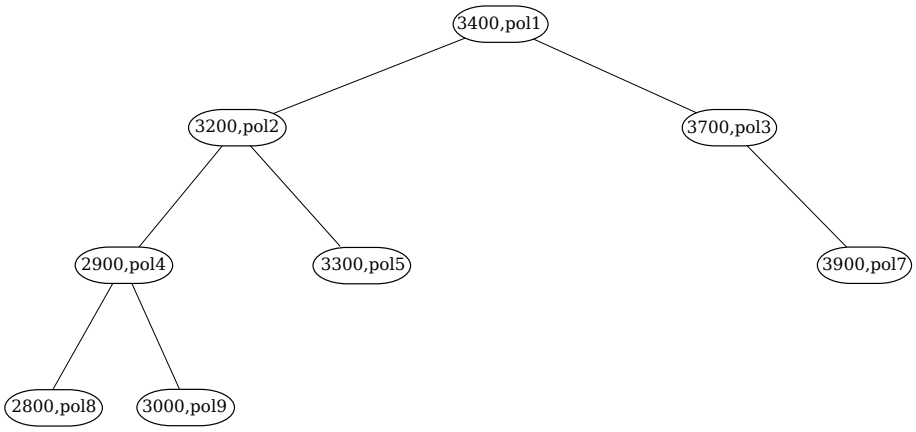

Abbildung 1: Beispiel für einen Suchbaum zur Speicherung von Politikern.

2.1. Baumdatenstruktur (4 Punkte) Definieren Sie eine Struktur `BST` mit einem **generischen** Datentypen `bst` für die oben beschriebene Art von Suchbäumen. Der Baum soll sowohl **Schlüssel** als auch **Daten beliebigen Typs** enthalten können. Nach außen hin soll zunächst **ausschließlich** eine Funktion `empty` zum Erzeugen eines leeren Baumes zur Verfügung gestellt werden. Geben Sie sowohl die **Signatur** als auch die **Implementierung der Struktur** vollständig an.

```
SIGNATURE BST[key, val, <]

SORT key val
SORT bst

FUN < : key ** key -> bool

FUN empty: bst

---------------------------------------------------

IMPLEMENTATION BST

DATA bst == nil
            node(k:key, v:val, l:bst, r:bst)

DEF empty == nil
```

2.2. Einfügen (4 Punkte) Deklarieren und definieren Sie eine rekursive Funktion `insert`, die als Argumente einen Schlüssel, ein Datum und einen Suchbaum vom Typ `bst` bekommt.

Die Funktion `insert` fügt das Datum mit dem Schlüssel in den Suchbaum ein und gibt den resultierenden Baum zurück. Ist der **Schlüssel** bereits im Eingabebaum **vorhanden**, soll das entsprechende **Datum ersetzt** werden.

```
FUN insert : key ** val ** bst -> bst

DEF insert(newK, newV, node(k,v,l,r)) ==
        IF newK<k THEN node(k,v, insert(newK,newV,l), r)
        IF k<newK THEN node(k,v,l, insert(newK,newV,r))
        ELSE  node(k,newV,l,r)
        FI

DEF insert(newK, newV, nil) ==  node(newK,newV,nil,nil)
```

3. Aufgabe (5 Punkte): Listenfunktionale

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

3.1. Gehälter von Politikern (3 Punkte) Deklarieren und definieren Sie eine Funktion `getSal`, die einen Vornamen und Nachnamen jeweils als `denotation` sowie eine Sequenz von Politikern übergeben bekommt und von **allen** Politikern mit diesem Namen die **Gehälter** (`salary`) zurück gibt. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

```
FUN getSal: denotation ** denotation -> seq[politician] -> seq[salary]
DEF getSal(f,l) == map(wage) o filter(\\p.first(p)=f and last(p)=l)
```

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

```
FUN lastNames : seq[politician] -> denotation
```

`lastNames` bekommt eine Sequenz von Politikern übergeben und soll die **Nachnamen** aller Politiker als `denotation` durch **Kommata getrennt** zurückgeben. Benutzen Sie **Listenfunktionale**. Schreiben Sie **keine** rekursiven Funktionen.

```
DEF lastNames == reduce(\\p,d. last(p) ++ ", " ++ d,"")
```

4. Aufgabe (5 Punkte): Eingabeverarbeitung

Zur Erstellung einer Parlamentarierdatenbank wird eine Funktion benötigt, die eine Liste von Politikern einliest. Diese Liste liegt als Textdokument vor, das nach den Regeln folgender Grammatik aufgebaut ist:

$$\textit{List} \rightarrow \textit{Politician}^{\star}$$
$$\textit{Politician} \rightarrow \textit{Name Name Num } \underline{\texttt{,}} \textit{ Num } \underline{\texttt{;}}$$

Dabei bezeichnet das linke *Name* den Vornamen, das rechte *Name* den Nachnamen und die beiden *Num* den Euro- bzw. Centbetrag des Grundgehalts eines Politikers. Eine Liste aus drei Politikern könnte bspw. so aussehen:

```
Uwe Schünemann 6000,00 ; Dieter Wiefelspütz 7668,00 ; Ansgar Heveling 7668,00 ;
```

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                    -- ,
              semicolon                -- ;
```

Zusätzlich zum Scanner sind bereits entsprechende **Funktionen** zum Parsen der Token `name`, `num`, `comma` und `semicolon` **gegeben**, die Sie verwenden können:

```
FUN parseName : seq[token] -> denotation ** seq[token]
FUN parseNum  : seq[token] -> nat ** seq[token]
FUN parseComma parseSemicolon : seq[token] -> seq[token]
```

Definieren Sie nun auf Basis der gegebenen Funktionalität die Funktion `parseList`, die eine Sequenz von Tokens nach den Regeln der obigen Grammatik in eine Sequenz von Politikern umwandelt, wobei der `bonus` des Gehalts (Typ `salary`) auf `0` zu setzen ist. Gehen Sie davon aus, dass in der Eingabesequenz keine Syntaxfehler enthalten sind. Das heißt, Sie müssen in Ihrem Parser **keine Fehlerbehandlung** vorsehen.

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

DEF parseList(input)  ==
    LET (first, r1)  == parseName(input)
        (last, r2) == parseName(r1)
       (euro, r3) == parseNum(r2)
              r4  == parseComma(r3)
      (cent, r5)  == parseNum(r4)
              r6 == parseSemicolon(r5)
    IN pol(first, last, salary(euro,cent,0)) :: parseList(r6)

DEF parseList(<>) == <>
```

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

Gegeben sind die beiden Datentypen `salary` und `politician` (siehe Referenzblatt/S. 11). Darüber hinaus ist eine Suchfunktion `filterByBonus` gegeben, die aus einer Sequenz von Politikern diese heraussucht, deren Bonuszahlungen eine gegebene Grenze überschreiten.

```
FUN filterByBonus : nat ** seq[politician] -> seq[politician]
```

Um dem interessierten Bürger die Möglichkeit zu geben, die Bonuszahlungen aller in einer Parlamentarierdatenbank geführten Politiker zu überprüfen, soll ein entsprechendes Kommando implementiert werden.

```
FUN listByBonus : seq[politician] -> com[void]
```

Definieren Sie das Kommando `listByBonus`, das nach einem Euro-Betrag fragt und die Liste aller Politiker, deren Bonuszahlungen den gegebenen Betrag überschreiten, zeilenweise in eine Datei `bonus.txt` schreibt. Sie können für diese Aufgabe eine Backtick-Funktion für den Datentyp `politician` als gegeben annehmen.

```
FUN ‘ : politician -> denotation
```

Verwenden Sie zur Implementierung der Funktion `listByBonus` entsprechende Kommandos aus der Struktur `File` (Siehe Referenzblatt/S. 11).

```
DEF listByBonus(l) == ask("Bonusgrenze: ") & (\\ b .
                writeToFile(filterByBonus(b,l)) )


FUN writeToFile : seq[politician] -> com[void]
DEF writeToFile(ps) == open("bonus.txt", "w") & (\\ file .
                                write(file,lines(ps))  &
                                close(file) )


FUN lines : seq[politician] -> denotation
DEF lines(<>)    == ""
DEF lines(p::ps) == `(p) ++ "\n" ++ lines(ps)
```

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
```

A_f(n) = A_g(n) + A_f(n - 1) + const1

A_g(n) = A_f(n-1) + const2

A_f(n) = A_f(n-1) + A_f(n - 1) + const3
= 2A_f(n - 1) + const3*n^0

-> 2. Zeile, c=2, b=const3, k=0 => f in O(2^n)

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

baumartig rekursiv

Wichtige Strukturen

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

```
SIGNATURE Politician

TYPE politician == pol(first : denotation,  -- Vorname
                       last  : denotation,  -- Nachname
                       wage  : salary       -- Gehalt
                      )
```

```
SIGNATURE Salary

TYPE salary == salary(euro  : nat,  -- Euro
                      cent  : nat,  -- Cent
                      bonus : nat   -- Bonus in Euro
                     )
```

Auszug aus der Bibliotheca Opalica

```
SIGNATURE Pair [data1,data2]

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

```
SIGNATURE Denotation

FUN ++ : denotation ** denotation -> denotation
FUN +/+ : denotation -> denotation ** denotation -> denotation -- put first denotation inbetween the tupel
```

```
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 BasicIO

FUN ask : denotation -> com[nat]
```

```
SIGNATURE ComCompose[first,second]

SORT first second
FUN & : com[first] ** (first -> com[second]) -> com[second]
FUN & : com[first] ** com[second] -> com[second]
```

```
SIGNATURE File

-- open(filename, mode) opens a file in "r" read, "w" write or "a" append mode
FUN open      : denotation ** denotation -> com[file]
FUN write     : file ** denotation -> com[void]
FUN writeLine : file ** denotation -> com[void]
FUN close     : file -> com[void]
```