Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Zusatzübungsblatt 5

(Kapitel 6)

Hinweis: Dieses Zusatzübungsblatt wird von den Tutoren der Vorlesung "Programmierung 1" (WS 2012/13) erstellt. Es enthält unter anderem Aufgaben von den Übungsblättern der gleichen Vorlesung aus dem Wintersemester 2011/12 gehalten von Prof. Hermanns, sowie des Nachklausurtutoriums aus dem gleichen Jahr. Außerdem enthält es einige neue, von den Tutoren erstellte Aufgaben.

Die hier gestellten Aufgaben und die damit abgedeckten Themenbereiche sind weder für die Klausur relevant, noch irrelevant. Sie dienen vor allem dazu Ihnen andere Blickrichtungen auf den Stoff zu präsentieren.

Aufgabe 5.1 (*Zusätzliche Aufgabe aus dem Wintersemester 11/12*)
Vervollständigen Sie folgenden Ansatz so, dass eine Person mehrere Kleidungsstücke "tragen" kann. Der Datentyp kleidung soll dabei mindestens vier Konstruktortypen haben.

```
datatype kleidung = ...
datatype person   = ...
```

Aufgabe 5.2 (*Zusätzliche Aufgabe aus dem Wintersemester 11/12*)
Legen Sie einen Datentyp tier (Tiere sind zum Beispiel Hund, Katze, Maus und Wellensittich) an, der mindestens vier Konstruktortypen hat. Stellen Sie außerdem einen Typ haustier auf. Ein Haustier ist dabei ein Tier mit Namen. Welche abstrakte Gleichheit kann man auf Haustieren definieren? Erklären Sie und geben Sie ein Beispiel an!

Aufgabe 5.3 (*Offizielle Übungsaufgabe aus dem Wintersemester 11/12*)
Schreiben sie eine Prozedur

$last : \alpha\ list \to \alpha\ option$,

die das letzte Element einer Liste liefert.

Aufgabe 5.4 (*Offizielle Übungsaufgabe aus dem Wintersemester 11/12*)
Wir versuchen nun, einen Datentyp zu konstruieren, mit dem wir verschiedene Arten von Computern darstellen können. Ein Computer kann entweder ein Netbook, ein Notebook oder ein Tower-PC sein. Alle Computer bestehen bei uns aus Speicher und Prozessor, wobei sowohl Speicher als auch Prozessor eine Ganzzahl speichern, die ihre Performance angibt.

(a) Implementieren Sie geeignete Datentypen, um Computer in SML gemäßdieser Angaben darstellen zu können.

(b) Schreiben Sie eine Prozedur

$compareC : computer * computer \rightarrow order$

die zwei Computer vergleicht, wobei bei einem Netbook die Performance des Speichers zu zwei Drittel und die des Prozessors zu einem Drittel in die Bewertung eingehen, bei einem Notebook jeweils zur Hälfte und bei einem Tower der Prozessor zu zwei Dritteln und der Speicher zu einem Drittel.

(c) Machen Sie sich klar, dass Sie nun in der Lage sind, mithilfe polymorpher Sortier-Prozeduren eine Liste von Computern anhand ihrer Performance zu sortieren.

Aufgabe 5.5 (*Offizielle Übungsaufgabe aus dem Wintersemester 11/12*)
VIVA ist noch nicht ganz zufrieden mit Ihrer Arbeit. Sie sollen den *Love-Generator* so erweitern, dass für einen Harem nicht mehr nur ein einzelner Wert, sondern eine nach Kompatibilität geordnete Liste von Paarungen zurückgegeben wird. Schreiben Sie also eine Prozedur

$loveGenSort : string\ list \rightarrow (string * string)\ list$

die für eine Liste von Namen zunächst alle möglichen Paarungen berechnet und diese anschließ gend anhand ihres LoveGen-Kompatibilität sortiert.

Aufgabe 5.6

(a) Schreiben Sie eine Prozedur

$$partition\colon\ (\alpha * \alpha \rightarrow order) \rightarrow \alpha\ list \rightarrow \alpha\ list * \alpha\ list * \alpha\ list \tag{1}$$

die eine Liste xs gemä einer Prozedur f in drei Listen us, vs, ws zerlegt, sodass f für die Elemente von us *LESS*, für die Elemente von vs *EQUAL* und für die Elemente von ws *GREATER* liefert.

(b) Verwenden Sie diese Prozedur, um Quicksort *(Blatt 5, Aufgabe 16)* polymorph zu implementieren.

(c) Zeichnen Sie einen Rekursionsbaum für den folgenden Aufruf ihrer in *b)* geschriebenen Prozedur.

```
quicksort Int.compare [5, 5, 7, 6, 3, 5, 8, 7]
```

Aufgabe 5.7 (*Aufgabe aus dem Nachklausurtutorium 11/12*)
Achtung! *Die folgende Aufgabe ist wirklich knifflig. Lassen Sie sich nicht entmutigen, wenn Sie nicht alle Aufgabenteile lösen können! Sie sollten allerdings trotzdem alle Aufgabenteile bearbeiten. Wenn Sie keine vollständige Lösung für einen Teil angeben können, versuchen Sie eine möglichst vollständige Lösung auszuarbeiten. Sie dürfen die deklarierten Prozeduren auch dann in den folgenden Aufgabenteilen benutzen, wenn sie diesen Aufgabenteil nicht gelöst haben.*

Da Sie Dieters Hausaufgaben erledigt haben, hatte Dieter Zeit, einige Sudokus zu lösen - zumindest hat er das versucht. Da Dieter aber kein besonders guter Sudoku-Spieler ist, braucht er wieder Ihre Hilfe!

Dieter hat bereits eine Datenstruktur *Feld* entwickelt, die angibt, dass ein Feld entweder eine Lücke *L* ist oder mit einer Zahl *Z of int* belegt ist:

```
datatype feld = L | Z of int
```

Jetzt ist er aber mit seinem Latein am Ende. Gut, dass er auf Sie zählen kann! Sie machen sich natürlich sofort an die Arbeit:

(a) Sie haben sich entschieden, Sudokus mit Listen darzustellen:

```
type zeile = feld list
type sudoku = zeile list
```

Beispiel:

$$\begin{matrix}1 & 2\\ 3 & 4\end{matrix} \rightarrow [[Z\ 1, Z\ 2], [Z\ 3, Z\ 4]]$$

Schreiben Sie eine Prozedur $getSize : sudoku \rightarrow int$, die die Gröe (Anzahl der Zeilen und Spalten) ermittelt. Wenn das Sudoku ungültig ist (mindestens eine der Zeilen hat nicht genau so viele Felder wie das Sudoku Spalten), dann soll die Ausnahme *falschessudoku* geworfen werden.

(b) Natürlich müssen Sie feststellen können, ob ein Sudoku gelöst ist. Schreiben Sie eine Prozedur $isSolved : sudoku \rightarrow bool$, die dies erledigt. Benutzen Sie *foldl* und keine andere Rekursion.

Tipp von Dieter: Ein Sudoku ist gelöst, wenn es keine Lücken mehr hat.

(c) Bei Sudokus hilft es oft, die Perspektive zu wechseln. Schreiben Sie eine Prozedur $mirror : sudoku \rightarrow sudoku$, die die Zeilen und Spalten eines Sudokus vertauscht.

Beispiel: $\begin{matrix}1 & 2\\ 3 & 4\end{matrix} \rightarrow \begin{matrix}1 & 3\\ 2 & 4\end{matrix}$

(d) Natürlich muss man beim Sudoku einige Regeln beachten - selbst bei der vereinfachten Variante, die Dieter spielt: Es dürfen nur Zahlen zwischen 1 und der Gröe des Sudokus verwendet werden. Auerdem dürfen in jeder Zeile und Spalte alle Zahlen nur einmal benutzt werden.

Schreiben Sie eine Prozedur $isValid : sudoku \rightarrow bool$, die prüft, ob alle Regeln eingehalten wurden.

Achtung: Auch ein Sudoku mit Lücken kann den Regeln entsprechen!

(e) Ausgetrickst! Nachdem Sie Dieter ein kniffliges Sudoku geschickt hatten, hatte er Ihnen eine valide Lösung geschickt. Erst viel später ist Ihnen aufgefallen, dass Dieter nicht nur Lücken ausgefüllt hat, sondern auch schon gegebene Zahlen geändert hat. Das wollen Sie sich nicht gefallen lassen!

Schreiben Sie eine Prozedur $antiDieter : sudoku \rightarrow sudoku \rightarrow bool$, die eine Aufgabe (ein Sudoku mit Lücken) und eine Lösung (von Dieter) nimmt und prüft, ob alle Zahlen, die in der Aufgabe angegeben sind, auch in Dieters Lösung unverändert an der gleichen Stelle stehen.

(f) *Achtung! Jetzt wird es richtig scharf!*

Jetzt haben Sie Dieter endgültig ausgetrickst: Er kann nicht mehr schummeln. Frustriert behauptet er: "*Du* kannst ja selbst gar keine Sudokus lösen!"

Das können Sie natürlich nicht auf sich sitzen lassen. Schreiben Sie eine Prozedur *solve* : *sudoku* → *sudoku*, die (einfache) Sudokus lösen kann. Gehen Sie dazu so vor:

(i) Schreiben Sie eine Prozedur *solveStep* : *sudoku* → *sudoku*, die alle Zeilen und Spalten des Sudokus *einmal* überprüft, ob sie nur noch eine Lücke haben. Falls ja, soll an dieser Stelle der einzige noch mögliche Wert eingesetzt werden, ansonsten bleibt die Zeile unverändert.

(ii) Schreiben Sie nun die Prozedur *solve*. Sie soll so lange *solveStep* anwenden, bis

- *entweder* das entstandene Sudoku nicht mehr valide ist (weil die Aufgabe unlösbar war)
- *oder* das Sudoku gelöst ist
- *oder* das weitere Anwenden von *solveStep* keine weiteren Lücken mehr ausfüllt