Übungen zu Softwareentwicklung III, Funktionale Programmierung

Blatt 10, Woche 11

Funktionen höherer Ordnung und kombinatorische Probleme

Leonie Dreschler-Fischer

WS 2015/2016

Ausgabe: Freitag, 1.1.2016,

Abgabe der Lösungen: bis Montag, 15.1.2016, 12:00 Uhr per email bei den Übungsgruppenleitern.

Ziel: Die Aufgaben auf diesem Zettel dienen dazu, sich mit dem Entwurf von Funktionen höherer Ordnung und Rekursion zur Lösung kombinatorischer Probleme vertraut zu machen.

Bearbeitungsdauer: Die Bearbeitung sollte insgesamt nicht länger als 6 Stunden dauern.

Homepage:
http://kogs-www.informatik.uni-hamburg.de/~dreschle/teaching/Uebungen_Se_III/Uebungen_Se_III.html

Bitte denken Sie daran, auf den von Ihnen eingereichten Lösungsvorschlägen *Ihren Namen und die Matrikelnummer, den Namen der Übungsgruppenleiterin / des Übungsgruppenleiters* und *Wochentag und Uhrzeit der Übungsgruppe* anzugeben, damit wir ihre Ausarbeitungen eindeutig zuordnen können.

1 Sudoku

(Bearbeitungszeit 4 Std.)

Sudoku ist eine Gattung von Logikrätseln, in denen es darum geht, eine bestimmte vorgegebene Menge von Zahlen durch Anwendung von logischen Regeln zu vervollständigen. Im klassischen Falle eine 9x9 großen Sudoku-Feldes ergeben sich folgende drei logische Regeln, nach denen ausgefüllt werden darf. Jede Zahl (von 1 bis 9) darf

1. in jeder Zeile nur einmal vorkommen,
2. in jeder Spalte nur einmal vorkommen und
3. in jedem Quadranten nur einmal vorkommen.

Bei gut gestellten Rätseln gibt es stets eine eindeutige Lösung, die zudem lediglich durch Anwendung der oben genannten Regeln zu bestimmen ist. Ein Backtracking ist im Allgemeinen nicht notwendig.

Ein Beispiel für ein solches Rätsel und dessen Lösung zeigt die untere Abbildung. In dieser sind die einzelnen Quadranten jeweils durch einen dickeren Rahmen hervorgehoben.

Initial:

					9		7	
				8	2		5	
3	2	7					4	
	1	6		4				
	5					3		
				9		7		
			6					5
8		2						
		4	2					8

Lösung:

6	8	5	4	3	9	2	7	1
4	9	1	7	8	2	6	5	3
3	2	7	5	6	1	8	4	9
9	1	6	3	4	7	5	8	2
7	5	8	1	2	6	3	9	4
2	4	3	8	9	5	7	1	6
1	3	9	6	7	8	4	2	5
8	6	2	9	5	4	1	3	7
5	7	4	2	1	3	9	6	8

In diesem Aufgabenzettel bauen Sie auf einer Repräsentation eines solchen Rätsels auf und werden ein Werkzeug zur Hilfe sowie zur Lösung eines Sudoku-Rätsels programmieren. Um sich die Arbeit zu vereinfachen, verwenden Sie bitte Funktionen höherer Ordnung wann immer dies möglich ist.

1.1 Konsistenz eines Spielzustands

Eva Lu Ator hat sich schon ein wenig mit dem Rätsel beschäftigt, und folgende Repräsentation für den (initialen) Spielzustand entworfen. Hierbei setzt sich auf den Datentyp Vektor, zu erkennen an der Raute vor den Zahlen. Nullen repräsentieren noch nicht ausgefüllte Felder:

```
(define spiel  #(0 0 0 0 0 9 0 7 0
                 0 0 0 0 8 2 0 5 0
                 3 2 7 0 0 0 0 4 0
                 0 1 6 0 4 0 0 0 0
                 0 5 0 0 0 0 3 0 0
                 0 0 0 0 9 0 7 0 0
                 0 0 0 6 0 0 0 0 5
                 8 0 2 0 0 0 0 0 0
                 0 0 4 2 0 0 0 0 8))
```

Helfen Sie Eva nun dabei, festzustellen, ob ein (teilweise) gelöstes Rätsel einen konsistenten Zustand aufweist:

1. Definieren Sie eine Hilfsfunktion `(xy->index x y)` mit der Sie zwischen der Darstellung `x`= Spalte, `y`=Zeile und der Darstellung als Index in der Spielzustandsrepräsentation wechseln können: — 2 Pnkt.

   ```
   (xy->index 0 0) ⟶ 0
   (xy->index 3 1) ⟶ 12
   (xy->index 8 8) ⟶ 80
   ```

2. Definieren Sie drei Funktionen, die Ihnen jeweils Zugriff auf die Indizes der Zeilen, Spalten und Quadranten des Zustandsvektors geben: — 6 Pnkt.

   ```
   (zeile->indizes 0) ⟶ '(0 1 2 3 4 5 6 7 8)
   (spalte->indizes 5) ⟶ '(5 14 23 32 41 50 59 68 77)
   (quadrant->indizes 8) ⟶ '(60 61 62 69 70 71 78 79 80)
   ```

3. Definieren Sie eine Funktion, die ausgehend von einem Spielzustand und einer Indexmenge die Einträge des Spielzustands ermittelt: — 1 Pnkt.

   ```
   (spiel->eintraege spiel (quadrant->idx 8))
           ⟶ '(0 0 5 0 0 0 0 0 8)
   ```

4. Definieren Sie ausgehend von einem Spielzustand Funktionen, die unter Anwendung der logischen Regeln prüfen, ob ein Spielzustand insgesamt konsistent oder gelöst ist. Beachten Sie, dass Nullen keinen Einfluss auf die Konsistenz haben! — 6 Pnkt.

   ```
   (spiel-konsistent? spiel) ⟶ #t
   (spiel-geloest? spiel) ⟶ #f
   ```

1.2 Sudoku lösen (ohne Backtracking)

Um Sudoku-Spieler zu unterstützen, hatte Eva sich überlegt eine Hilfestellung anzubieten, die anzeigt wo eine neue Zahl eindeutig gesetzt werden kann. Hierbei können Sie zweischrittig vorgehen, wie dies in (1.2.1) und (1.2.2) beschrieben ist. Anschließend sollten Sie in der Lage sein, einen Sudoku-Löser ohne Backtracking zu implementieren.

1. Definieren Sie eine Funktion, die ein Spielfeld anhand einer Zahl annotiert. In diesem sind alle diejenigen Nullen durch das Symbol `'X` ersetzt, die gemäß den Regeln nicht mehr für die entsprechende Zahl infrage kommen. Dies ist dann der Fall, wenn die Zeile, Spalte oder der Quadrant die Zahl bereits enthält. 5 Pnkt.

   ```
   (markiere-ausschluss spiel 5) ⟶ '#(0 X 0 0 0 9 X 7 X
                                      X X X X 8 2 X 5 X
                                      3 2 7 0 0 0 X 4 X
                                      X 1 6 0 4 0 0 X X
                                      X 5 X X X X 3 X X
                                      X X X 0 9 0 7 X X
                                      X X X 6 X X X X 5
                                      8 X 2 0 0 0 X X X
                                      0 X 4 2 0 0 X X 8)
   ```

   Hinweis: Verwenden Sie initial `vector-copy` um den Spielzustand zu kopieren und verändern Sie ihn dann gezielt mithilfe von `vector-set!`.

2. Ausgehend von dem annotierten Spielfeld können Sie nun recht einfach bestimmen, wann eine Zahl eindeutig auf eine Position gesetzt werden kann. Dies ist immer dann der Fall, wenn es pro Zeile, Spalte oder Quadrant nur noch eine Null gibt. Schreiben Sie eine Funktion, die eine Liste dieser Positionen für eine Zahl zurück liefert. 5 Pnkt.

   ```
   (eindeutige-positionen spiel 5) ⟶ '(2 33 72)
   ```

3. Schreiben Sie eine rekursive Funktion `(loese-spiel spiel)`, die mittels der bisher definierten Funktionen ein Sudoku ohne Backtracking löst. Gehen Sie dazu in jedem Rekursionsschritt wie folgt vor: 10 Pnkt.

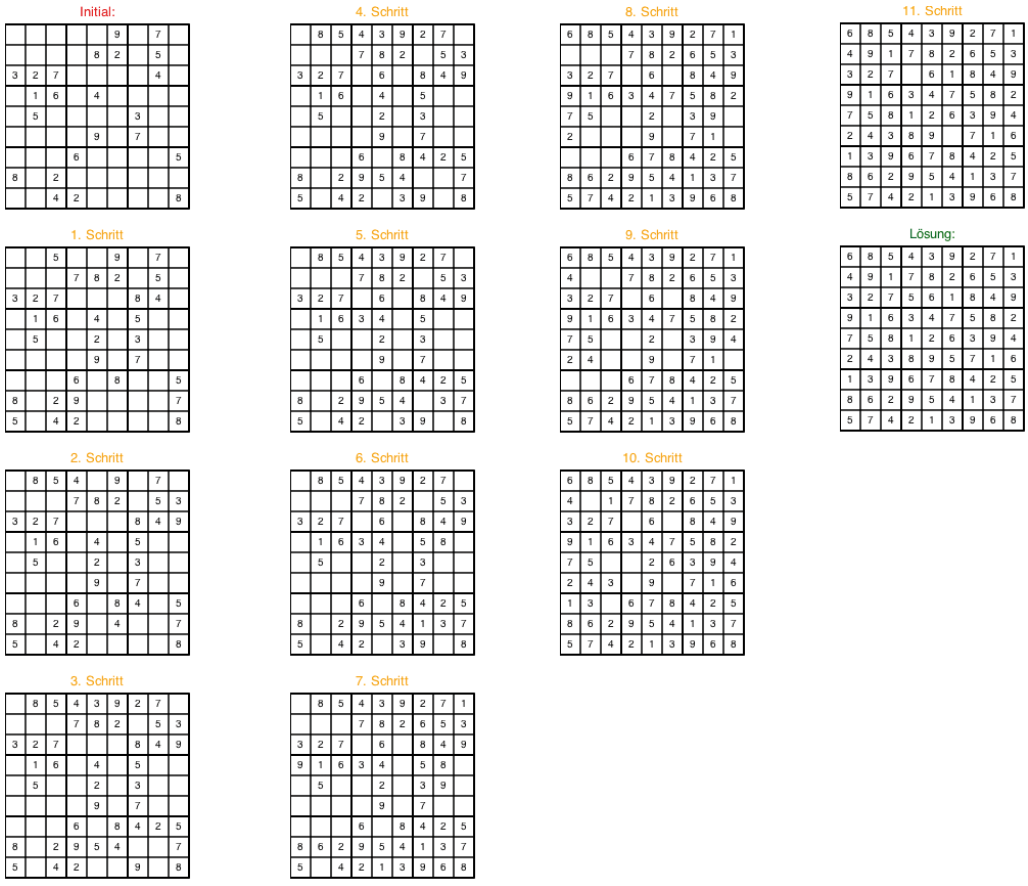
   - Ermittle die eindeutigen Positionen für alle Zahlen `(range 1 10)`.
   - Setze die Zahlen auf die ermittelten Positionen.
   - Falls keine Positionen ermittelt werden konnten, brich ab - das Rätsel ist ohne Backtracking nicht lösbar.

1.3 Grafische Ausgabe

Um dem Benutzer die Sicht auf die Spielzustände zu erleichtern, bietet es sich an, eine grafische Darstellung mittels des Pakets (require 2htdp/image) zu implementieren.

1. Schreiben Sie eine Funktion `zeichne-spiel`, die einen Spielzustand grafisch anzeigt. Annotierte Spielfelder (siehe 1.2.2) sollten als leere Felder mit rotem Hintergrund dargestellt werden, nicht ausgefüllte Felder als leere Felder mit weißem Hintergrund dargestellt werden. Alle anderen Spielfelder zeigen die eingetragene Zahl vor weißem Hintergrund. — 5 Pnkt.

2. Zusatzaufgabe: Schreiben Sie eine Funktion `loese-spiel-grafisch`, die für jeden Rekursionsschritt das erzielte Zwischenergebnis anzeigt. Wenn Sie mögen. können Sie auch einen weiteren Parameter integrieren, der die Zwischenergebnisse (ähnlich zur Darstellung auf Seite 2) ausblendet. Zum Testen Ihrer Lösung finden Sie eine beispielhafte Abbildung direkt unter dieser Aufgabenstellung. — 5 Zusatzpnkt.



2 Wiederholung, Klausurvorbereitung

(Bearbeitungszeit 2 Std.)

1. Zu welchen Werten evaluieren die folgenden Racket-Ausdrücke? 6 Zusatzpnkt.

   (a) (**max** (**min** 2 (− 2 3)))

   (b) ‘(+ ,(− 2 4) 2)

   (c) (car '(Alle meine Entchen))

   (d) (cdr '(schwimmen auf (dem See)))

   (e) (**cons** 'Listen '(sind einfach))

   (f) (**cons** 'Paare 'auch)

   (g)
   ```
   (equal?
        (list 'Racket 'Prolog 'Java)
        '(Racket Prolog Java))
   ```

   (h)
   ```
   (eq?
        (list 'Racket 'Prolog 'Java)
        (cons 'Racket '(Prolog Java)))
   ```

   (i)
   ```
   (map
        (lambda (x) (* x x x))
        '(1 2 3))
   ```

   (j) (filter odd? '(1 2 3 4 5))

   (k) ((curry **min** 6) 2)

   (l) ((curry = 2) 2)

2. Zur Syntax von Racket: Gegeben seien die folgenden Definitionen: 5 Zusatzpnkt.

   ```
   (define *a* 10)
   (define *b* '*a*)
   (define (merke x) (lambda () x))
   (define (test x)
     (let ((x (+ x *a*)))
       (+ x 2)))
   ```

   Haben die folgenden Ausdrücke einen wohldefinierten Wert und zu welchem Wert evaluieren Sie?

(a) ∗a∗

(b) (+ ∗a∗ ∗b∗)

(c) (+ (**eval** ∗a∗) (**eval** ∗b∗))

(d) (**and** (> ∗a∗ 10) (> ∗b∗ 3))

(e) (**or** (> ∗a∗ 10) (/ ∗a∗ 0))

(f) (+ 2 (merke 3))

(g) (+ 2 ((merke 3)))

(h) (test 4)

3. Geben Sie geeignete Racket-Ausdrücke an, um die folgenden Werte zu berechnen: 2 Zusatzpnkt.

   (a) $3 \times 4 + 5 \times 6$

   (b) $\sqrt{1 - \sin^2(x)}$

4. Definieren Sie die folgenden beiden Funktionen c und mytan als Racket-Funktionen: 2 Zusatzpnkt.

$$\begin{aligned} \mathrm{c}(a, b) &= \sqrt{a^2 + b^2}, \quad a, b \in R \\ \mathrm{mytan}(\alpha) &= \frac{\sin \alpha}{\sqrt{1 - \sin^2 \alpha}}, \quad -\frac{\pi}{2} < \alpha < \frac{\pi}{2} \end{aligned}$$

5. Wandeln Sie die folgenden Ausdrücke in die Präfixnotation von Racket um: 2 Zusatzpnkt.

   (a) $1 + 4/2 - 1$

   (b) $\frac{2 - \frac{1+3}{3+2*3}}{\sqrt{3}}$

6. Wandeln Sie den folgenden Ausdruck in Infixnotation um: 1 Zusatzpnkt.

   (∗ (+ 1 2 3) (− 2 3 (− 2 1)))

7. Reduktionsmodelle: 3 Zusatzpnkt.

   (a) Was ist der Unterschied zwischen innerer und äußerer Reduktion?

   (b) Unter welcher Voraussetzung führen die innere und die äußere Reduktion garantiert zum gleichen Ergebnis? Welche Sprachelemente können diese Voraussetzung verletzen?

8. Definieren Sie eine *rekursive* Funktion (laengen xss) in Racket, die folgendes leistet: Gegeben sei eine Liste von Listen. Bilden Sie diese ab auf die Liste der Längen der Unterlisten. 4 Zusatzpnkt.

   `(laengen (1 3 4) (Auto Bus) ()) -> (3 2 0)`

   Geben Sie die Funktion in zwei Varianten an: einmal linear rekursiv ohne Akkumulator und einmal endrekursiv. Woran erkennen Sie eine endrekursive Funktion?

9. Ein abstrakter Datentyp für Längenangaben: 10 Zusatzpnkt.

   Für viele Anwendungen ist es notwendig, nicht nur den numerischen Wert einer physikalischen Größe zu kennen sondern auch die Einheit, in der dieser Wert angegeben ist.

   Implementieren Sie die Zugriffsfunktionen für einen Datentyp „laenge“:

   Eine Längenangabe sei repräsentiert als Liste mit zwei Elementen, nämlich dem numerischen Wert und der Einheit, beispielsweise:
   `(0.5 m), (3 cm), (6.3 km), (6 inch)`

   (a) Definieren Sie eine Konstruktorfunktion `(make-length value unit)`, die aus einem Wert und einer Längeneinheit ein Element vom Typ Längenangabe erzeugt:
   `(make-length 3 'cm)` $\longrightarrow$ `(3 cm)`

   (b) Definieren Sie Selektorfunktionen für den Zugriff auf den Wert einer Längenangabe `(value-of-length len)`
   und die Längeneinheit `(unit-of-length len)`:
   `(value-of-length (make-length 3 'cm))` $\longrightarrow$ `3`
   `(unit-of-length (make-length 3 'cm))` $\longrightarrow$ `cm`

   (c) Definieren Sie eine Skalierungsfunktion `(scale-length len fac)`, mit der Längenangaben um einen bestimmtem Faktor vergrößert (verkleinert) werden können:
   `(value-of-length (scale-length (make-length 3 'cm) 2)`
   $\longrightarrow$ `6`

   (d) Gegeben sei eine Tabelle `*conversiontable*` mit Umrechnungsfaktoren für Längenangaben. Diese Tabelle sei als Assoziationsliste organisiert, die zu jeder Längeneinheit den Umrechnungsfaktor gegenüber der Angabe in Metern enthält:

```
(define *conversiontable* ;
   '( ;  (unit . factor)
```

```
(m . 1)
(cm . 0.01)
(mm . 0.001)
(km . 1000)
(inch . 0.0254)
(feet . 0.3048)
(yard . 0.9144)))
```

Definieren Sie die folgenden Operationen auf Längenangaben unter Verwendung der Funktionen Konstruktoren und Selektoren `make-length, unit-of-length, value-of-length, scale-length`:

- Auslesen des Umrechnungsfaktors aus der Tabelle:
  `(factor 'mm)` $\longrightarrow$ `0.001`

- Normalisierung (Wandlung in Meterangaben):
  `(length->meter '(3 cm))` $\longrightarrow$ `(0.03 m)`

- Vergleich zweier Längenangaben :
  `(length< '(3 cm) '(6 km))` $\longrightarrow$ `#t`
  `(length= '(300 cm) '(3 m))` $\longrightarrow$ `#t`

- Addition und Subtraktion zweier Längenangaben `(length+ len1 len2)`:
  `(length+ '(3 cm) '(1 km))` $\longrightarrow$ `(1000.03 m)`
  `(length- '(1.001 km) '(1 m))` $\longrightarrow$ `(1000.0 m)`

(e) Gegeben sei eine Liste von Längenangaben:
`'((6 km) (2 feet) (1 cm) (3 inch))`.
Verwenden Sie Funktionen höherer Ordnung (`reduce, map, filter, iterate, curry`...), um funktionale Ausdrücke für folgende Werte zu schreiben:

- Die Abbildung der Liste auf eine Liste, die die Längenangaben in Metern enthaelt,
- eine Liste aller Längen, die kürzer als 1m sind,
- die Summe der Längenangaben in Metern,
- die Längenangabe mit der kürzesten Länge in der Liste.

Erreichbare Punkte: 40

Erreichbare Zusatzunkte: 40