Funktionale und objektorientierte Programmierkonzepte

Florian Kadner und Lukas Röhrig
(Gesamtleitung: Prof. Karsten Weihe)

Wintersemester 18/19
v1.0

Übungsblatt 10

Themen:	Wrap-Up I
Relevante Folien:	alle bis dahin
Abgabe der Hausübung:	25.01.2019 bis 23:55 Uhr

Dies ist eines der drei Wrap-Up Blätter. Auf diesen Blättern begegnen Ihnen keine völlig neuen Konzepte mehr, sondern sie stellen vielmehr einen Rundumschlag über alle vergangenen Themen und spezielle Vertiefungen dar.

Die vorbereitenden Übungen enthalten auf diesen Blättern keine Sternkennzeichnung mehr.

Außerdem sind die Hausübungen etwas kniffliger als die bisherigen. Auf diesem fortgeschrittenen Level müssen Sie sich davon verabschieden, dass wir Ihnen genau mittels Teilaufgaben strukturieren, wie Sie ein Problem zu lösen haben. Überlegen Sie sich selbst eine sinnvolle Strukturierung und lösen Sie die Aufgaben genau so, wie Sie sich das vorstellen.

V Vorbereitende Übungen

V1 Fibonacci: rekursiv vs. iterativ

Die Fibonacci Funktion ist eines der klassischen Beispiele für Rekursion. Sie ist wie folgt definiert:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{falls } n \geq 2 \end{cases}$$

1. Implementieren Sie eine Methode in Java, welche die Fibonacci Funktion nach obiger Definition rekursiv berechnet.
2. Implementieren Sie nun erneut eine Methode in Java, welche die Fibonacci Funktion berechnet. Diesmal muss Ihre Lösung allerdings iterativ (also über Schleifen) gelöst werden!

V2 Ackermann Funktion

Die Ackermann-Funktion ist eine 1926 von Wilhelm Ackermann gefundene mathematische Funktion, die in der theoretischen Informatik eine wichtige Rolle bezüglich Komplexitätsgrenzen von Algorithmen spielt. Sie wird als Benchmark zur Überprüfung rekursiver Prozeduraufrufe verwendet, wenn man die Leistungsfähigkeit von Programmiersprachen oder Compilern überprüfen will.

Gegeben sei folgende vereinfachte Version der Ackermann Funktion:

$$\begin{aligned} \mathrm{Ack}(0,m) &= m+1 \\ \mathrm{Ack}(n,0) &= \mathrm{Ack}(n-1,1) \\ \mathrm{Ack}(n,m) &= \mathrm{Ack}(n-1,\mathrm{Ack}(n,m-1)) \end{aligned}$$

Implementieren Sie die Funktion in Racket.

1. Berechnen Sie den Aufruf `Ack(2,2)`.

2. Berechnen Sie die ersten Schritte des Aufrufs `Ack(4,2)`. Was fällt Ihnen hierbei sehr schnell auf?

V3 Pascal'sches Dreieck

Definieren Sie in Racket eine **rekursive** Funktion `(pascal-line n)`, die die n-te Zeile des Pascal'schen Dreiecks berechnet. Jede Zeile n im Dreieck, beginnend mit Zeile 0, hat $n+1$ Elemente.

Für die n-te Zeile und die $n+1$ Elemente und die Positionen i pro Zeile gilt dann:

$$p_i(n) = p_{i-1}(n-1) + p_i(n-1) \quad \text{mit} \quad i = 1 \ldots n+1 \quad \text{und} \quad p_0(n-1) = 0$$

Nachfolgend ist das Pascal'sche Dreieck mit den ersten 7 Zeilen angegeben. Jeder Wert in einer gegebenen Spalte entsteht durch die Summe des Wertes direkt darüber und links schräg darüber. Falls der Wert links darüber nicht existiert, ist der Wert 0 dafür anzunehmen.

$n=0$						1							
$n=1$					1		1						
$n=2$				1		2		1					
$n=3$			1		3		3		1				
$n=4$		1		4		6		4		1			
$n=5$	1		5		10		10		5		1		
$n=6$	1		6		15		20		15		6		1

Ihre Funktion soll die n-te Zeile jeweils als Liste von Zahlen zurückliefern.

V4 Selection Sort

Bei dem Sortierverfahren *Selection Sort* wird beim Durchlaufen des Arrays der Größe n im Durchlauf $i = 0, \ldots, n-1$ das kleinste Element ab Position $i+1$ bestimmt und mit dem

Element an Position i vertauscht. Daher verringert sich die Länge der zu sortierenden Arrayteilfolge mit jeder Iteration um eins, womit sichergestellt ist, dass der Algorithmus terminiert.

Implementieren Sie nun Selection Sort in Java als `void selectionSort(int[] array)`.

Beispiel für das Array `[4,2,1,6,3,5]`:

i = 0	`[4,2,1,6,3,5]`	$4 \leftrightarrow 1$
i = 1	`[1,2,4,6,3,5]`	–
i = 2	`[1,2,4,6,3,5]`	$4 \leftrightarrow 3$
i = 3	`[1,2,3,6,4,5]`	$6 \leftrightarrow 4$
i = 4	`[1,2,3,4,6,5]`	$6 \leftrightarrow 5$
i = 5	`[1,2,3,4,5,6]`	

Für weitere Informationen zum Algorithmus finden Sie beispielsweise in der deutschsprachigen Wikipedia unter:

`https://de.wikipedia.org/wiki/Selectionsort`

V5 String Matching

In dieser Aufgabe bekommen Sie in Racket zwei Strings und sollen prüfen, ob der eine String ein Teilstring des anderen ist, also ob der eine String im anderen enthalten ist. So ist beispielsweise `"kuchen"` Substring des Strings `"Apfelkuchen"`.

Definieren Sie also eine Funktion `(is-substring? s sub)`. Diese liefert `true` genau dann zurück, wenn der String `sub` ein Teilstring des Strings `s` ist.

Hinweis:

Sie können die eingebaute Funktion `explode` verwenden. Diese bekommt einen String übergeben und überführt diesen in eine Liste von einelementigen Strings.

Beispielsweise liefert `(explode "abc")` die Liste `(list "a" "b" "c")` zurück.

V6 Einführung Backtracking - Das n-Damen Problem

In der Hausübung des ersten Wrap-Up Blatts wollen wir uns mit sogenannten Backtracking-Algorithmen beschäftigen. Diese Aufgabe dient als erste Einführung, um das Prinzip dieser Algorithmen zu verinnerlichen.

Backtracking-Algorithmen

Es gibt Probleme, die wir mit der kennengelernten Rekursion aus der Vorlesung nicht oder zumindest nicht optimal gelöst bekommen. Dafür gibt es Backtracking-Algorithmen, die auf folgender Vorgehensweise basieren:

1. Verfolge einen möglichen Lösungsweg, bis ...
   (a) ... die Lösung gefunden wurde: **Erfolg**!
   (b) ... oder der Weg nicht fortgesetzt werden kann.
2. Wenn der Weg nicht fortgesetzt werden kann: Gehe den Weg zurück, bis zur letzten Verzweigungsmöglichkeit, wo es noch nicht gewählte Alternativen gibt und wähle dort noch eine nicht gewählte Alternative und mache weiter bei Schritt 1.
3. Wenn der Ausgangspunkt erreicht ist und es keine Alternativen mehr gibt: **Misserfolg**!

V6.1 Das n-Damen Problem

Bei dem n-Damen Problem handelt es sich um ein mathematisches Problem aus der Welt des Schachs.

Problemstellung: Wir wollen n Damen auf einem $n \times n$-Schachbrett so positionieren, dass keine Dame eine andere gemäß der Schachregeln schlagen kann. Anders ausgedrückt: Es dürfen keine zwei oder mehr Damen auf derselben Reihe, Linie oder Diagonale stehen.

Für weitere Details bietet sich der deutschsprachige Wikipedia Artikel `https://de.wikipedia.org/wiki/Damenproblem` an. Hier findet sich bereits die Formulierung eines Algorithmus, welcher die Lösung mittels rekursivem Backtracking ermittelt. Versuchen Sie es aber zunächst unbedingt alleine!

Aufgabe: Implementieren Sie in Java die Methode `int[][] solveNQueens(int n)`. Die Funktion soll für ein gegebenes `n` eine mögliche Lösung des Problems mittels rekursivem Backtracking bestimmen. Zurückgegeben wird ein `n` × `n` Array vom Typ `int`, wobei eine 0 bedeutet keine Dame und eine 1 bedeutet hier wurde eine Dame platziert.

Der Aufruf `solveQueens(4)` kann also beispielsweise folgendes Array zurückliefern:

```
0  0  1  0
1  0  0  0
0  0  0  1
0  1  0  0
```

H Zehnte Hausübung Gesamt 11 Punkte
Racket Reloaded: Backtracking

In dieser zehnten Hausübung beschäftigen wir uns mit Backtracking (vergleiche V6) in Racket.

H1 Graphen traversieren 4 Punkte

Als Graph bezeichnen wir eine Sammlung von Knoten und Kanten. In einem gerichteten Graphen repräsentieren die Kanten gerichtete Verbindungen zwischen den Knoten. Betrachten Sie das nachfolgende Beispiel eines solchen Graphens:

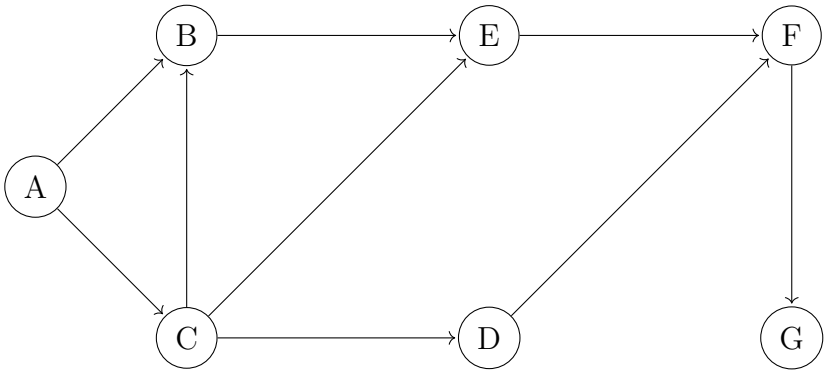

Abbildung 1: Beispielgraph

Sie dürfen im nachfolgenden davon ausgehen, dass wir nur gerichtete Graphen betrachten, welche keine Zyklen enthalten.

Wir wollen den eben gezeigten Graphen nun in Racket modellieren und verwenden dazu Strukturen und Listen. Die Knoten modellieren wir mit einem Namen (als String) und einer Liste mit den Namen seiner Nachbarn (Liste von Strings). Für unseren Beispielgraphen sieht dies wie folgt aus:

```
(define-struct node (name neighbors))
(define A (make-node "A" (list "B" "C")) )
(define B (make-node "B" (list "E")) )
(define C (make-node "C" (list "B" "D" "E")) )
(define D (make-node "D" (list "F")) )
(define E (make-node "E" (list "F")) )
(define F (make-node "F" (list "G")) )
(define G (make-node "G" empty) )
```

Der Graph selbst besitzt ebenfalls einen Namen und eine Liste all seiner Knoten (als Liste von node-Structs). Für unseren Beispielgraphen sieht dies wie folgt aus:

```
(define-struct graph (name nodes))
(define G1 (make-graph "G1" (list A B C D E F G)))
```

Ihre Aufgabe:

Ergänzen Sie die folgende Funktion aus der Vorlage:

```
;; Type: node node graph -> (list of String)
;; Returns: a path from origin to dest in the given graph
;; Precondition: cycle-free graph
(define (find-route origin dest graph)
        ...
)
```

Beispielsweise liefert der Aufruf `(find-route B F G1)` die Liste `(list "B" "E" "F")` zurück. Die Namen des Start- bzw. Zielknotens soll also auch nochmal in der Liste vorhanden sein.

Wird kein Pfad gefunden, so soll `false` zurückgegeben werden.

Wichtige Hinweise:

- In der Vorlage ist die Definition des Beispielgraphens gegeben, nutzen Sie diese um Ihre Implementierung zu testen.
- Nutzen Sie das vorgestellte Prinzip des Backtrackings!
- Sollte es mehrere Pfade geben, so ist es egal, welchen Pfad Sie zurückgeben. Die Funktion soll lediglich einen gültigen Pfad zurückliefern.

Beispiel für Backtracking:

Suchen wir beispielsweise den Pfad von A nach D, so sehen wir schnell, dass lediglich die Route `(list "A" "C" "D")` in Frage kommt.

Nehmen wir an unser Algorithmus entscheidet sich nach C zunächst für den Knoten E. Er würde weiter nach F laufen und dann nach G und hätte keine Optionen mehr. Da wir nicht am Ziel sind gehen wir einen Knoten zurück und sind wieder bei F und schauen nach Alternativrouten, da es keine gibt gehen wir zurück nach E. Auch hier haben wir keine anderen Möglichkeiten, wir gehen also zurück nach C. Bei C haben wir zwei andere Alternativen und zwar B oder D. Entscheiden wir uns zunächst für B passiert das gleiche wie zuvor wir laufen zu G und dann wieder zurück zu C. Wählen wir jetzt D sind wir am Ziel und der Algorithmus terminiert mit einer korrekten Lösung.

H2 Einstellung neuer FOP-Tutoren 4 Punkte

Wir stehen mal wieder vor einem Problem. Wir wollen neue FOP-Tutoren einstellen und haben lediglich ein begrenztes Budget. Unsere Entscheidung welche Tutoren wir nun einstellen, wollen wir dabei möglichst optimiert treffen. Tutoren besitzen zwei essentielle Eigenschaften: Schnelligkeit und Korrektheit. Beide Eigenschaften können wir auf einer Skala von 0-100 angeben, wobei höhere Werte besser sind. Jeder Tutor verlangt außerdem ein festes Gehalt für seine gesamte Arbeitszeit. Zur Modellierung verwenden wir folgende Struktur:

```
(define-struct tutor (name salary speed correctness))
```

Daraus ergibt sich für uns das Problem, dass wir nicht alle Tutoren anstellen können, wir müssen also eine Auswahl treffen!

Ihre Aufgabe:

Ergänzen Sie die folgende Funktion aus der Vorlage:

```
;; Type: (list of tutor) number (tutor -> number) -> (list
   of String)
;; Returns: the best combination of tutors that have a total
   price less than the number passed in, regarding the given
   criterion
(define (choose-tutors possible-tutors budget criterion)
    ... )
```

Die Funktion bekommt der Reihe nach die Liste der potenziellen Tutoren, das maximale zur Verfügung stehende Budget, sowie eine Auswertungsfunktion, welche angibt welches Kriterium wir maximieren möchten (hier also beispielsweise `tutor-speed` bzw. `tutor-correctness`) übergeben.

Geliefert werden soll eine Liste mit den Namen der Tutoren, für die die ausgewählte Auswertungsfunktion das Maximum, also die für unser Kriterium beste Zusammenstellung, liefert. Beachten Sie dabei unbedingt, dass das maximal verfügbare Budget nicht überschritten werden darf! Ihre Aufgabe ist es nun, entsprechend der konkret zu nutzenden Auswertungsfunktion die bestmögliche Auswahl an Tutoren zu treffen. Dabei können Sie also eine Tutorenliste mit entweder der maximalen Schnelligkeit, oder aber der maximalen Korrektheit erstellen.

Verbindliche Anforderung: Die Liste darf zu keinem Zeitpunkt sortiert werden. Ebenso darf nicht explizit die Potenzmenge berechnet und daraus dann die beste Lösung bestimmt werden. Beides ist für die Lösung auch nicht nötig. Sie dürfen also nicht einfach alle Lösungen bestimmen und dann die beste heraussuchen, dies wäre höchst ineffizient, verwenden Sie Backtracking!

Für diese Aufgabe sind bereits Beispieldaten und Tests in der Vorlage gegeben, Sie können auf die Erstellung von eigenen Tests bei dieser Aufgabe verzichten.

H3 Das n-Damen Problem - Reloaded 3 Punkte

In allen Backtracking-Problemen bisher ging es darum, dass Sie **eine** mögliche Lösung finden. In dieser Aufgabe interessieren wir uns hingegen für die mögliche Anzahl **aller** Lösungen. Dazu betrachten wir erneut das n-Damen Problem aus Aufgabe V6.

Definieren Sie eine Funktion `(n-queens n)`, welche wieder n Damen auf einem $n \times n$ großen Schachbrett verteilen soll. Die Funktion liefert die gesamte Anzahl an möglichen Lösungen zurück, nicht die Lösungen selbst! Lösungen die gespiegelte oder rotierte Varianten von anderen Lösungen sind zählen auch!

Beispiel: Für ein 8×8 Schachbrett gibt es nur 12 vollständig unterschiedliche Lösungen, allerdings insgesamt 92.

Unverbindliche Hilfestellung:

Sie können zur Lösung ein Struct `queen` verwenden:

```
(define-struct queen (x y))
```

Außerdem die folgende Funktion, welche testet ob eine Dame platziert werden kann:

```
;; Type: queen (list of queen) -> boolean
;; Returns: true if a queen can be attacked by a list of
   other queens
(define (under-attack new-queen queens)
 (cond
  [(empty? queens) false]
  [else
    (or
    ;; queens on same row
    (= (queen-x new-queen) (queen-x (first queens)))
    ;; queens on same col
    (= (queen-y new-queen) (queen-y (first queens)))
    ;; queens on same diagonal
    (= (abs (- (queen-x new-queen) (queen-x (first queens))))
    (abs (- (queen-y new-queen) (queen-y (first queens)))))
    ;; next queen
    (under-attack new-queen (rest queens)))]))
```

So ergibt `(under-attack (make-queen 1 1)(list (make-queen 1 2)))` beispielsweise `true`.