Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 06

Prof. Karsten Weihe

Übungsblattbetreuer:	Nhan Huynh
Wintersemester 23/24	v1.1
Themen:	Racket, Methoden und Rekursion
Relevante Foliensätze:	03c + 04a + Folien 1-50 von 04b
Abgabe der Hausübung:	08.12.2023 bis 23:50 Uhr

Hausübung 06 — Gesamt: 32 Punkte

Maze Runner

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h06` und ggf. `src/test/java/h06`.

Einleitung

In diesem Übungsblatt beschäftigen wir uns mit dem Thema Racket und Rekursion. Sie werden Racket-Code in Java übersetzen und die Funktionalitäten einmal rekursiv und einmal iterativ lösen.

Unser Fokus liegt auf der Implementierung des MazeSolvers, der dazu dient, einen MazeRunner durch ein Labyrinth zu führen. Das Labyrinth wird als Raster mit verschiedenen Zellen repräsentiert, von denen einige durch Mauern blockiert sind. Der MazeSolver beginnt an einer bestimmten Startzelle und versucht, den Weg zur Zielzelle zu finden, indem der MazeRunner sich von Zelle zu Zelle bewegt und dabei die Mauern umgeht.

Im ersten und zweiten Abschnitt werden Sie das Grundgerüst für das Lösen des Labyrinths definieren. Im dritten und vierten Abschnitt werden Sie eine rekursive bzw. iterative Lösung des MazeSolvers implementieren.

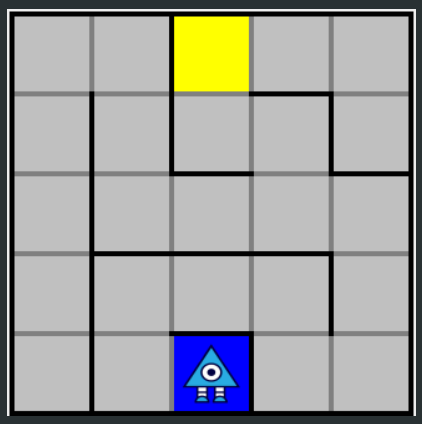

Abbildung 1: Beispiel - Labyrinth

Hinweis:

Für die Hausübung ist die Klasse `java.awt.Point` aus der Java Standardbibliothek relevant. Diese Klasse `Point` repräsentiert einen Punkt in einem zweidimensionalen Koordinatensystem mit einer x- und y-Koordinate. Mittels der Attribute x und y von `Point` können Sie auf die Koordinaten zugreifen.

Schauen Sie sich ebenfalls die Klassen in der Vorlage an, um sich mit den Klassen vertraut zu machen.

H1: Richtungsvektor

Um die Bewegungsrichtung in einem Labyrinth anzugeben, verwenden wir hier Richtungsvektoren[1]. Wir beschränken uns hier auf vier Richtungsvektoren:

- Vektor nach oben: $\vec{u} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
- Vektor nach rechts: $\vec{r} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
- Vektor nach unten: $\vec{d} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$
- Vektor nach links: $\vec{l} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$

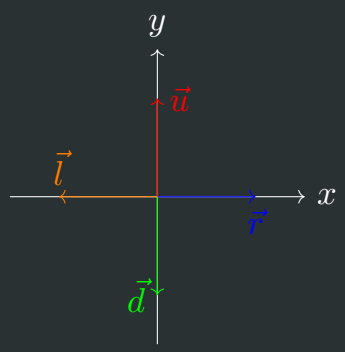

Abbildung 2: Richtungsvektoren

In Racket stellen wir die Richtungsvektoren folgendermaßen dar:

DirectionVector

```
4  (define-struct direction-vector (x y))
5  (define UP (make-direction-vector 0 1))
6  (define RIGHT (make-direction-vector 1 0))
7  (define DOWN (make-direction-vector 0 -1))
8  (define LEFT (make-direction-vector -1 0))
```

Im Package `h06.world` finden Sie eine Implementierung der Klasse `DirectionVector` als Enumeration, die den obigen Racket-Code äquivalent repräsentieren soll.

Wir werden in dieser Aufgabe zwei Operationen in `h06.world.DirectionVector` implementieren, die uns erlauben, einen Richtungsvektor zu rotieren.

[1] `https://de.wikipedia.org/wiki/Vektor#Orts-_und_Richtungsvektoren`

H1.1: Vektor um 270 Grad drehen — 1 Punkt

Implementieren Sie die Methode `rotate270`, die einen Richtungsvektor um $270°$ dreht. Die Implementierung soll äquivalent zur Implementierung in Racket sein.

rotate270()

```
4  ;; Type: direction-vector ->direction-vector
5  (define (rotate-270 d)
6    (if (equal? d UP)
7        LEFT
8        (if (equal? d LEFT)
9            DOWN
10           (if (equal? d DOWN)
11               RIGHT
12               UP
13               )
14           )
15       )
16   )
```

Die einzige Ausnahme hier ist, dass die Methode `rotate270()` kein Argument besitzt. Das Argument `d` ist hier implizit und wird durch das aktuelle Objekt ersetzt.

Verbindliche Anforderung:

Verwenden Sie in dieser Aufgabe nur den Bedingungsoperator.

H1.2: Vektor um 90 Grad drehen — 1 Punkt

Analog zu H1.1 implementieren Sie die Methode `rotate90`, die einen Richtungsvektor um $90°$ dreht. Die Implementierung soll ebenfalls äquivalent zur Implementierung in Racket sein.

rotate90()

```
4  ;; Type: direction-vector ->direction-vector
5  (define (rotate-90 d)
6    (cond
7      [(equal? d UP) RIGHT]
8      [(equal? d RIGHT) DOWN]
9      [(equal? d DOWN) LEFT]
10     [else UP]
11     )
12   )
```

Die einzige Ausnahme hier ist, dass die Methode `rotate90()` kein Argument besitzt. Das Argument `d` ist hier implizit und wird durch das aktuelle Objekt ersetzt.

Verbindliche Anforderung:

Verwenden Sie in dieser Aufgabe nur die `if-else`-Anweisung.

H2: Ist das Feld frei? 4 Punkte

In einem Labyrinth möchten wir überprüfen, ob ein Feld frei ist. Hierfür existiert eine Methode `isBlocked(Point, DirectionVector)` in der Klasse `h06.world.World`, die wir implementieren werden. Diese Methode erhält einen Punkt und einen Richtungsvektor als Parameter und gibt `true` zurück, wenn das Feld in der angegebenen Richtung blockiert ist.

Ein Feld gilt als blockiert, wenn

1. das Feld außerhalb des Labyrinths liegt oder
2. sich eine Wand auf dem Weg zum Feld befindet.

Zur Überprüfung, ob sich eine Wand auf einem Feld befindet, kann die Methode `isBlocked(int, int, boolean)`, welche sich in der selbigen Klasse befindet, verwendet werden. Diese Methode erhält eine Koordinate und eine Wandorientierung als Parameter und gibt `true` zurück, falls sich auf dem Feld eine Wand befindet.

Beachten Sie, dass eine vertikale Wand immer rechts von einem Feld liegt und eine horizontale Wand immer oberhalb eines Feldes platziert wird.

In Abbildung 3 sehen Sie die Orientierung der Wände. Die horizontale Wand befindet sich an der Position $(1, 1)$ und die vertikale Wand befindet sich an der Position $(2, 2)$.

Beispiele:

(a) Wir befinden uns auf dem Feld $(1, 1)$ und möchten in die Richtung `UP` gehen. Das Feld ist blockiert, da sich eine Wand auf dem Weg zum Feld befindet.

(b) Wir befinden uns auf dem Feld $(1, 0)$ und möchten in die Richtung `UP` gehen. Das Feld ist nicht blockiert, da sich keine Wand auf dem Weg zum Feld befindet.

(c) Wir befinden uns auf dem Feld $(2, 2)$ und möchten in die Richtung `RIGHT` gehen. Das Feld ist blockiert, da sich eine Wand auf dem Weg zum Feld befindet.

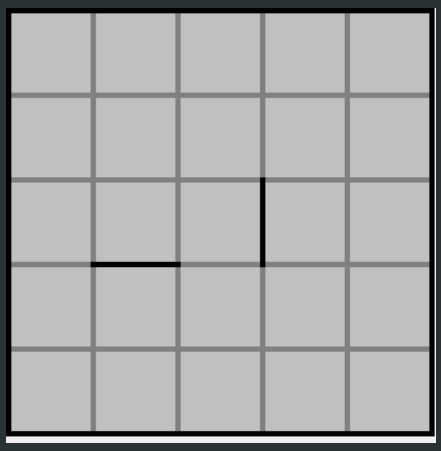

Abbildung 3: Orientierung der Wände

Hinweis:

(1) An den Grenzen der Welt befinden sich keine Wand und dienen lediglich als Visualisierung. Sie müssen explizit Punkt 1. überprüfen müssen, ob sich das Feld außerhalb des Labyrinths befindet.

(2) Die Methode `isBlocked(int, int, boolean)` erhält als Eingabe eine Koordinate und eine Wandorientierung. Die Wandorientierung gibt an, ob die Wand horizontal oder vertikal ist. Die Wandorientierung ist `true`, falls die Wand horizontal ist und `false`, falls die Wand vertikal ist.

H3: Labyrinth - Rekursiv

Nun können wir uns an die Implementierung des Maze Solvers machen. Hierfür implementieren wir die Klasse `MazeSolverRecursive` im Package `h06.problems`. Wie der Name schon sagt, soll die Implementierung **rekursiv** erfolgen.

> **Verbindliche Anforderung** (Für H3)**:**
>
> Alle zu implementierenden Methoden und Hilfsmethoden sind rein rekursiv, das heißt, Schleifen sind nicht erlaubt.

H3.1: nextStep — 4 Punkte

Um das Labyrinth zu lösen, müssen wir nach jedem Schritt überprüfen, wo wir danach hingehen werden. Dazu implementieren Sie die Methode `nextStep`, die den nächsten Schritt berechnet.

nextStep(World, Point, DirectionVector)

```
4  ;; Type: world point direction-vector ->direction-vector
5  (define (next-step world p d)
6    (cond
7      [(not ((world-is-blocked world) p (rotate-270 d))) (rotate-270 d)]
8      [(not ((world-is-blocked world) p d)) d]
9      [(not ((world-is-blocked world) p (rotate-90 d))) (rotate-90 d)]
10     [else (rotate-90 (rotate-90 d))]
11     )
12   )
```

Die Funktion erhält als Parameter die Welt, in der wir uns befinden, sowie die aktuelle Position und Richtung von uns. Anhand dieser Informationen berechnen wir den nächsten Schritt. Wir implementieren hier einen einfachen Algorithmus, auch bekannt als *rechte-Hand-Regel* in abgewandelter Form.

> **Erinnerung:**
>
> Die Funktion `world-is-blocked` haben wir bereits in H2 und `rotate-270` bzw. `rotate-90` in H1 implementiert.

> **Verbindliche Anforderungen:**
>
> (i) Verwenden Sie in dieser Aufgabe genau einen Bedingungsoperator. Verschachtelte Bedingungsoperatoren sind nicht erlaubt.
>
> (ii) Es sind keine Hilfsmethoden erlaubt.

Unbewertete Verständnisfragen:

(1) Was bedeutet das Doppel-Semikolon in der ersten Zeile des Racket-Codes? Was ist das Gegenstück in Java hierzu?

(2) Welche der Klammerpaare und Whitespaces im Racket-Code oben müssen vorhanden sein? Welche hätte man weglassen können und welche könnte man noch hinzufügen?

(3) Warum sind die Identifier im Racket-Code oben korrekt gemäß den Regeln von Racket? Wären sie korrekt gemäß den Regeln von Java?

(4) Wir gehen davon aus, dass mindestens eine Richtung nicht blockiert ist, d.h. im Idealfall rotieren wir den Richtungsvektor um $90°$ bis eine der Richtungen nicht blockiert ist.

Was würde passieren, wenn keine Richtung frei ist und wie geht Ihre Implementierung damit um?

H3.2: numberOfSteps — 5 Punkte

Listen sind in Racket dynamisch, das heißt, sie können beliebig viele Elemente enthalten. In Java verwenden wir Arrays, die eine feste Größe haben. Bevor wir die Hauptmethode `solve` implementieren können, müssen wir wissen, wie viele Schritte wir benötigen werden.

Dazu implementieren Sie die Methode `numberOfSteps`, die die Anzahl der Schritte berechnet, die wir benötigen werden, um das Labyrinth zu lösen.

Folgender Racket-Code wird zum Lösen des Labyrinths verwendet:

solve(World, Point, Point, DirectionVector)

```
;; Type: world point point direction-vector ->(list point)
(define (solve world s e d)
  (if (equal? s e)
      (list s)
      (cons s
            (solve
               world
               (get-movement (next-step world s d) s)
               e
               (next-step world s d))
            )
      )
  )
```

Statt Elemente in einer Liste hinzuzufügen, berechnen wir die Anzahl der Schritte, die wir benötigen werden, um das Labyrinth zu lösen.

Die Funktion `solve` erhält vier Argumente:

1. `world`: Die Welt, in der wir uns befinden.
2. `s`: Der Startpunkt, an dem wir uns befinden.
3. `e`: Der Endpunkt, an dem wir uns befinden wollen.
4. `d`: Die Richtung, in die wir schauen.

> **Erinnerung:**
>
> Die Funktion `get-movement` finden Sie in der Klasse `h06.world.DirectionVector`.

H3.3: solve 5 Punkte

Nun haben wir alles gegeben und können mit der Hauptmethode `solve` beginnen, die das Labyrinth löst.

Übersetzen Sie folgenden Racket-Code in Java:

solve(World, Point, Point, DirectionVector)

```
4  ;; Type: world point point direction-vector ->(list point)
5  (define (solve world s e d)
6    (if (equal? s e)
7        (list s)
8        (cons s
9              (solve
10                world
11                (get-movement (next-step world s d) s)
12                e
13                (next-step world s d))
14             )
15       )
16   )
```

Zuerst müssen Sie die Größe des Arrays berechnen, das Sie für die Lösung des Labyrinths benötigen. Anschließend initialisieren Sie das Array und können zum Schluss den Pfad zum Lösen des Labyrinths in das Array schreiben.

> **Verbindliche Anforderungen:**
>
> (i) Delegieren Sie das Befüllen des Arrays an die Methode `solveHelper`.
>
> (ii) Es sind keine weiteren Hilfsmethoden erlaubt.

> **Unbewertete Verständnisfragen:**
>
> Warum wird die eigentliche Rekursion jeweils in eine Hilfsmethode ausgelagert? Kann man die eigentliche Rekursion auch ohne Hilfsmethode lösen?

H4: Labyrinth - Iterativ

Analog zu H3 implementieren Sie die Klasse `MazeSolverIterative` im Package `h06.problems`. Im Gegegensatz zu H3 soll die Implementierung hier **iterativ** erfolgen.

> **Verbindliche Anforderung** (Für H4)**:**
>
> Alle zu implementierenden Methoden und Hilfsmethoden sind rein iterativ, das heißt, Rekursion ist nicht erlaubt.

H4.1: nextStep 4 Punkte

Um das Labyrinth zu lösen, müssen wir nach jedem Schritt überprüfen, wo wir danach hingehen werden. Dazu implementieren Sie die Methode `nextStep`, die den nächsten Schritt berechnet.

nextStep(World, Point, DirectionVector)

```
4  ;; Type: world point direction-vector ->direction-vector
5  (define (next-step world p d)
6    (cond
7      [(not ((world-is-blocked world) p (rotate-270 d))) (rotate-270 d)]
8      [(not ((world-is-blocked world) p d)) d]
9      [(not ((world-is-blocked world) p (rotate-90 d))) (rotate-90 d)]
10     [else (rotate-90 (rotate-90 d))]
11     )
12   )
```

Die Funktion erhält als Parameter die Welt, in der wir uns befinden, sowie die aktuelle Position und Richtung von uns. Anhand dieser Informationen berechnen wir den nächsten Schritt. Wir implementieren hier einen einfachen Algorithmus, auch bekannt als *rechte-Hand-Regel* in abgewandelter Form.

> **Erinnerung:**
>
> Die Funktion `world-is-blocked` haben wir bereits in H2 und `rotate-270` bzw. `rotate-90` in H1 implementiert.

H4.2: numberOfSteps 4 Punkte

Listen sind in Racket dynamisch, das heißt, sie können beliebig viele Elemente enthalten. In Java verwenden wir Arrays, die eine feste Größe haben. Bevor wir die Hauptmethode `solve` implementieren können, müssen wir wissen, wie viele Schritte wir benötigen werden.

Dazu implementieren Sie die Methode `numberOfSteps`, die die Anzahl der Schritte berechnet, die wir benötigen werden, um das Labyrinth zu lösen.

Folgender Racket-Code wird zum Lösen des Labyrinths verwendet:

solve(World, Point, Point, DirectionVector)

```
4  ;; Type: world point point direction-vector ->(list point)
5  (define (solve world s e d)
6    (if (equal? s e)
7        (list s)
8        (cons s
9              (solve
10                world
11                (get-movement (next-step world s d) s)
12                e
13                (next-step world s d))
14             )
15       )
16   )
```

Statt Elemente in einer Liste hinzuzufügen, berechnen wir die Anzahl der Schritte, die wir benötigen werden, um das Labyrinth zu lösen.

Die Funktion `solve` erhält drei Argumente:

1. `world`: Die Welt, in der wir uns befinden.
2. `s`: Der Startpunkt, an dem wir uns befinden.
3. `e`: Der Endpunkt, an dem wir uns befinden wollen.
4. `d`: Die Richtung, in die wir schauen.

> **Erinnerung:**
> Die Funktion `get-movement` finden Sie in der Klasse `h06.world.DirectionVector`.

H4.3: solve — 4 Punkte

Nun haben wir alles gegeben und können mit der Hauptmethode `solve` beginnen, die das Labyrinth löst.

Übersetzen Sie folgenden Racket-Code in Java:

solve(World, Point, Point, DirectionVector)

```
4  ;; Type: world point point direction-vector ->(list point)
5  (define (solve world s e d)
6    (if (equal? s e)
7        (list s)
8        (cons s
9              (solve
10                world
11                (get-movement (next-step world s d) s)
12                e
13                (next-step world s d))
14             )
15       )
16   )
```

Zuerst müssen Sie die Größe des Arrays berechnen, das Sie für die Lösung des Labyrinths benötigen. Anschließend initialisieren Sie das Array und können zum Schluss den Pfad zum Lösen des Labyrinths in das Array schreiben.

H5: Testen

In dieser Aufgabe beschäftigen wir uns damit, wie wir nun unseren Maze Solver testen können. Hierfür steht Ihnen ein `h06.ui.MazeVisualizer` zur Verfügung, um das Labyrinth zu visualisieren und die einzelnen Schritte zu verfolgen.

Wir wollen nun die Abbildung 4 visualisieren. Gehen Sie hierfür wie folgt vor:

(i) Gehen Sie zu der Methode `main`, die sich in der Klasse `h06.Main` befindet.

(ii) Erstellen Sie eine Instanz von der Klasse `h06.world.World` in der Methode.

(iii) Mittels der Methode `placeWall(int, int, boolean)` von `World` können Sie Wände in der Welt platzieren. Die Methode kommt eine Koordinate gegeben und die Orientierung der Wand (`true` = horizontale Wand) Platzieren Sie die Wände entsprechend der Abbildung 4.

(iv) Erstellen Sie nun eine Instanz von der Klasse `h06.ui.MazeVisualizer`.

(v) Initialisieren Sie den Visualizer mittels der Methode `init(World)`.

(vi) Visualizeren Sie nun die Visualizer mittels der Methode `show()`.

(vii) Zuletzt können Sie nun einen Problem mittels der Methode

```
run(ProblemSolver, Point, Point, Direction)
```

vom Visualizer ausführen. Die Methode erhält einen Algorithmus, einen Startpunkt, Endpunkt und eine Anfangsrichtung.

Führen Sie nun Ihr Programm aus und überprüfen Sie, ob die Schritte korrekt sind. Als `ProblemSolver` können Sie Ihre rekursive und iterative Implementierung einsetzen und vergleichen, ob beide Implementierung genau das Gleiche machen.

Falls Ihnen die Ausführunge der Schritte zu schnell ist, können Sie die Methode `setDelay(int)` verwenden, um die Verzögerung zwischen den Schritten zu erhöhen.

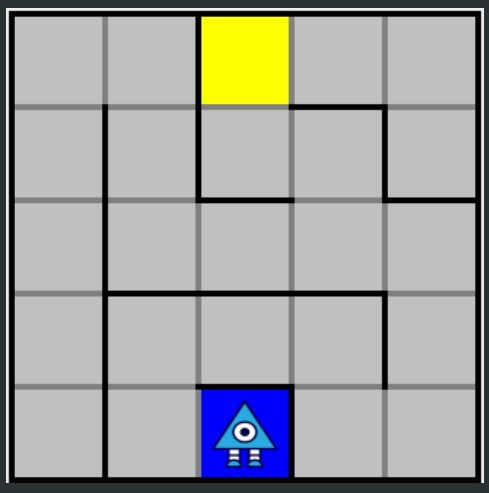

Abbildung 4: Beispiel - Labyrinth