Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 06

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Karsten Weihe

Wintersemester 22/23	v2.0
Themen:	Racket, Methoden und Rekursion
Relevante Foliensätze:	03c + 04a + Folien 1-41 von 04b
Abgabe der Hausübung:	09.12.2022 bis 23:50 Uhr

Hausübung 06 — Gesamt: 24 Punkte

Strange Functions

> **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 den letzten beiden Hausübungen ging es primär um Klassen und Interfaces – die Methodenimplementationen in den Klassen waren meist eher einfach gehalten. In dieser Hausübung sind nun wieder Methodenimplementationen im Fokus. Neben verschiedenen „kleineren“ Aspekten von Methoden geht es um eine sehr wichtige, aber nicht unbedingt leicht zu verstehende Art von Methoden: ***rekursive*** Methoden, das heißt, Methoden, die sich selber aufrufen.

In Kapiteln 04a, 04b, 04c und 04d finden Sie neben Java noch eine weitere Programmiersprache namens Racket: Racket ist eine *funktionale Programmiersprache* – das heißt: funktionale Sprachkonstrukte prägen die Sprache, während Java durch objektorientierte Sprachkonstrukte geprägt wird. Die Unterschiede werden kurz am Beginn von Kapitel 04a der FOP und dann noch einmal ausführlich in Kapitel 04d der FOP betrachtet.

Einen Teil der von Ihnen zu schreibenden Methoden geben wir im Folgenden bereits als Quelltext vor – allerdings in der Programmiersprache Racket. Testen Sie Ihre Methoden, indem Sie dieselben Methoden in Racket aufrufen und das Verhalten vergleichen.

> Wenn die Rede davon ist, dass eine Racket-Funktion 1:1 in Java umgesetzt werden soll, können Sie sich bei der Umsetzung an den Beispielen aus den oben genannten Kapiteln orientieren.

> Auf diesem Übungsblatt werden Aufzählungen überschneidender Namen verkürzt, indem nur disjunkte Teile von Namen innerhalb geschweifter Klammern aufgezählt werden und bei vorangegangener Aufzählungen einsetzbare Elemente durch `*` ersetzt werden. Beispiel: `set{X,Y}For{A,B}` ist die Abkürzung für `setXForA`, `setXForB`, `setYForA` und `setYForB`. `set*For*` ist die Abkürzung für `set{X,Y}For{A,B}`.

H1: Rekursion auf Zahlen ?? Punkte

H1.1: Erste Rekursion auf Zahlen ?? Punkte

Implementieren Sie in Klasse `StrangeFunctions` die Klassenmethoden `strangeFunction{1,2}` mit jeweils zwei Parametern vom formalen Typ `int` und Rückgabetyp `int`. Beide Objektmethoden realisieren 1:1 die folgende Racket-Funktion `strange-function` in Java unter Einhaltung der unten genannten verbindlichen Anforderungen.

```
;; Type: natural natural -> real
( define ( strange-function m n )
    ( if ( < m n )
        m
        ( if ( > 0 ( modulo m n ) )
            ( strange-function ( - m ( modulo m n ) ) n )
            ( / m n ) ) ) )
```

Ihre Implementationen dürfen (genauso wie die gegebene Implementation in Racket) ohne Überprüfung davon ausgehen, dass beide aktualen Parameterwerte nicht-negativ (also gleich 0 oder positiv) sind.

Verbindliche Anforderungen:

1. `strangeFunction1` darf *nur* mit `if`-Anweisungen realisiert werden – also ohne Bedingungsoperator und ohne `switch`-Anweisungen.
2. `strangeFunction2` darf *nur* mit Bedingungsoperator (siehe Kapitel 01b, ab Folie 231 der FOP) realisiert werden – also ohne `if`- oder `switch`-Anweisungen.

H1.2: Zweite Rekursion auf Zahlen ?? Punkte

Implementieren Sie in Klasse `StrangeFunctions` nun die Klassenmethoden `understandable{1,2}`, die jeweils 1:1 die folgende Racket-Funktion `understandable?` in Java realisieren. Beide Parameter sind vom Typ `double`, die Rückgabe ebenfalls.

```
;; Type: real real -> boolean
( define ( understandable? m n )
    ( cond
        [  ( and ( <= m 0 ) ( <= n 0 ) )  #t  ]
        [  ( or ( <= m 0 ) ( <= n 0 ) )  #f  ]
        [  ( < m n )  ( understandable? ( - m 1 ) ( + m ( * 2 n ) ) ) ]
        [  else  ( understandable? m ( - m n ) ) ] ) )
```

Verbindliche Anforderungen:

- `understandable1` darf wie in H1.1 nur mit `if`-Anweisungen realisiert werden.
- `understandable2` darf wie in H1.1 nur mit Bedingungsoperator realisiert werden.

Hinweis:

Es gibt Kombinationen von aktualen Parameterwerten für `m` und `n`, bei denen die Rekursion nicht terminiert.

Da der Call-Stack (siehe Kapitel 04a, ab Folie 143 der FOP) endliche Größe hat, führt eine nicht terminierende Rekursion normalerweise dazu, dass das Laufzeitsystem das Programm mit einer entsprechenden Fehlermeldung abbricht, sobald der Call-Stack voll ist. Da es sich hier um eine *Endrekursion* handelt, kann es aber auch sein, dass die Rekursion von DrRacket bzw. vom Java-Compiler intern in eine Schleife umgewandelt wird, so dass sich keine Frames auf dem Call-Stack ansammeln.[a] Machen Sie sich klar, dass sich alle Kombinationen von aktualen Parameterwerten von `m` und `n`, bei denen die Rekursion nicht terminiert, durch ein kurz und bündig formulierbares Prädikat auf `m` und `n` beschreiben lässt. Identifizieren Sie vorab dieses Prädikat, um beim Testen keine Überraschungen zu erleben.

[a] Siehe de.wikipedia.org/wiki/Endrekursion

Unbewertete Verständnisfragen:

1. Das `if` in Racket ist nicht so recht vergleichbar mit der `if`-Verzweigung in Java, dafür 1:1 vergleichbar mit dem Bedingungsoperator in Java. Warum ist das so? Welcher Bestandteil eines Ausdrucks mit Bedingungsoperator in Java entspricht welchem Bestandteil im obigen Racket-Code? Die Antwort auf die letzte Frage sollten Sie sich durch Vergleich Ihrer beiden Implementationen geben.
2. Was bedeutet das Doppel-Semikolon in der ersten Zeile des Racket-Codes? Was ist das Gegenstück in Java hierzu?
3. Welche der Klammerpaare und Whitespaces im Racket-Code oben müssen sein – welche hätte man auch weglassen können, welche könnte man noch hinzufügen?
4. Warum sind die Identifier im Racket-Code oben korrekt gebildet gemäß den Regeln von Racket? Wären sie korrekt gebildet nach den Regeln von Java?
5. Auf natürlichen Zahlen `m` und `n` berechnet die Funktion `strange-function` einen recht einfachen mathematischen Ausdruck in `m` und `n`. Wie lautet der? Vielleicht helfen Ihnen die Ausgaben von `strange-function`? Wie würden Sie diesen mathematischen Ausdruck umformulieren, um auch negative Zahlen `n` zu erfassen? Rufen Sie `strange-function` zur Beantwortung der letzten Frage auch mit negativen aktualen Parameterwerten auf.

H2: Iterativ/Rekursiv auf Arrays ?? Punkte

Implementieren Sie in Klasse `StrangeFunctions` die Methoden `transformArray{1,2}`, die jeweils einen Parameter vom formalen Typ „Array von `double`“ haben.

Die einzelnen Elemente des zurückgelieferten Arrays sollen nach derselben Logik wie in der folgenden Racket-Funktion berechnet werden.

```
;; Type: ( list of real ) -> ( list of real )
( define ( strange-transformation lst )
( cond
    [ ( empty? lst )  ( list 2 )  ]
    [ ( empty? ( rest lst ) )  ( cons ( + 1 ( first lst ) ) empty )  ]
    [ else  ( cons ( + ( first lst ) ( first ( rest lst ) ) )
        ( strange-transformation ( rest lst ) ) )  ] ) )
```

In dieser Aufgabe müssen wir von der Anforderung, dass die gegebene Racket-Funktion 1:1 in Java umgesetzt werden soll, abweichen: Einerseits ist Ihnen vielleicht schon aufgefallen, dass in funktionalen Programmiersprachen wie Racket

Rekursion die primäre Möglichkeit für wiederholte Durchläufe ist und Schleifen eigentlich nicht vorgesehen sind[1] – in der Methode `transformArray1` müssen Sie aber Schleifen verwenden. Andererseits ersetzen wir Racket-Listen durch Java-Arrays – einfach, weil Java-Listen in der Vorlesung noch nicht behandelt wurden.

Um `transformArray2` realisieren zu können, implementieren Sie in Klasse `StrangeFunctions` weiter die Methode `doTheRecursion`, in der die Rekursion erst stattfindet. `transformArray2` selber ist frei von Rekursion und dient nur dazu `doTheRecursion` mit dem eigenen aktualen Array-Parameter und einem `int`-Wert aufzurufen. Der aktuale `int`-Wert gibt den in der Methode zu behandelnden Index an.

Bei der Implementation von `transformArray{1,2}` ist Ihnen überlassen, in welcher Reihenfolge die Elemente durchlaufen werden.

Verbindliche Anforderungen:

1. Die Methode `transformArray1` ist rein iterativ.
2. Die Methode `transformArray2` ruft nur die die Methode `doTheRecursion` auf
3. Die Methode `doTheRecursion` ist rein rekursiv ist.
4. Keine der drei Methode erstellt dabei neuen Arrays und befüllt diese mit `System.arraycopy()`, o.ä., sondern arbeitet ausschließlich mit dem übergebenen Array.

Unbewertete Verständnisfragen:

1. Warum wird die eigentliche Rekursion jeweils in eine Hilfsmethode `doTheRecursion` ausgelagert?
2. Wie sieht es aus, wenn Sie in Racket das `i`-te Element einer Liste `lst` setzen möchten? Schreiben Sie dazu in Racket eine Methode `set-value-at` mit Parametern `lst`, `i` und `elem`, die eine Liste liefert, die identisch zur Liste `lst` ist, nur dass das Element von `lst` an Position `i` durch `elem` ersetzt ist (erstes Element: wie üblich Position 0). Falls die in `i` angegebene Position nicht zur Liste gehört, soll die Ergebnisliste gleich der Eingabeliste sein.

H3: Korrektheit von Klammerausdrücken ?? Punkte

In dieser Aufgabe beschäftigen Sie sich mit der Korrektheit von *Klammerausdrücken*. Wir bezeichnen einen String als *korrekten Klammerausdruck*, wenn dieser die folgenden Eigenschaften erfüllt:

1. Der String besteht ausschließlich aus den Zeichen `'('`, `')'`, `'['`, `']'`, `'{'` und `'}'`. Wir sagen, dass `')'` die zu `'('`, `']'` die zu `'['` und `'}'` die zu `'{'` passende schließende Klammer ist.

2. Sei $2k$ mit $k \in \mathbb{N}$ die Länge des Strings, das heißt, der String hat die Positionen $0, \ldots, 2k-1$. Dann existieren die Paare $(\ell_0, r_0), \ldots, (\ell_{k-1}, r_{k-1})$ aus $\{0, \ldots, 2k-1\} \times \{0, \ldots, 2k-1\}$, so dass

   a) jeder Wert $0, \ldots, 2k-1$ genau einmal unter den Werten $\ell_0, \ldots, \ell_{k-1}, r_0, \ldots, r_{k-1}$ vorkommt;

   b) für $i \in \{0, \ldots, k-1\}$ ist an Position ℓ_i eine öffnende Klammer und an der zugehörigen Position r_i eine dazu passende schließende Klammer im String ist;

   c) für $i, j \in \{0, \ldots, k-1\}$ mit $i \neq j$ und $\ell_i < \ell_j$ gilt entweder $r_i < \ell_j$ oder $r_j < r_i$ (das heißt, zwei eingeklammerte Bereiche folgen entweder disjunkt nacheinander, oder einer ist im anderen ganz enthalten).

[1] In gewissem Rahmen sind Schleifen auch in Racket eingebaut, aber das ist nicht Thema der FOP.

Beispiele:

`"([{}])"` und `"([{}{}{}]){[()()()]}"` sind korrekte Klammerausdrücke. Der zweite Klammerausdruck besteht aus disjunkten Klammerausdrücken.

Für diese Aufgabe werden Klammerausdrücke mittels Objekten der Klasse `BracketExpression` dargestellt: `BracketExpression` hat einen Konstruktor mit einem Parameter `expression` vom Typ `String`, der `expression` dargestellt mittels „Array von `char`“ im gleichnamigen Objektattribut speichert.

In Aufgabe H3.1 implementieren Sie zunächst das Verhalten für non-disjunkte Klammerausdrücke, in Aufgabe H3.2 implementieren Sie dann das Verhalten für disjunkte Klammerausdrücke.

H3.1: Nicht-Disjunkte Klammerausdrücke ?? Punkte

Implementieren Sie in Klasse `BracketExpression` die Objektmethode `evaluate`, die einen Parameter `i` vom Typ `int` hat und ein Objekt vom formalen Typ `EvaluationResult` liefert. `i` hat die gleiche Bedeutung wie i in der Einleitung von H3. `evaluate` darf ohne Überprüfung davon ausgehen, dass `index` ein Index im String ist, also nicht-negativ und kleiner als die Länge des Strings ist.

Verbindliche Anforderung:

Die Methode `evaluate` *muss* nach folgendem Schema implementiert werden. Sie dürfen aber mehrere Schritte zusammenfassen, sofern das Resultat das gleiche ist.

Hinweis:

Bei der Implementierung von `evaluate` empfiehlt sich die Auslagerung von Abfragen in Hilfsmethoden, um Redundanz zu reduzieren.

Wir betrachten einen beliebigen rekursiven Aufruf von `evaluate`.

Als erstes betrachten wir folgende Fälle, bei denen die Rekursion abgebrochen wird, weil die Rückgabe auch ohne rekursiven Aufruf berechnet werden kann:

1. Wenn an Index `i` keine Klammer steht, soll direkt `EvaluationResult` mit `type` = `INVALID_CHARACTER` und `nextIndex` = `i` geliefert werden.

2. Wenn an Index `i` eine schließende Klammer steht, soll direkt `EvaluationResult` mit `type` = `NO_OPENING_BRACKET` und `nextIndex` = `i` geliefert werden.

3. Wenn Index `i` der letzte Index des Klammerausdrucks ist, soll direkt `EvaluationResult` mit `type` = `NO_CLOSING_BRACKET` und `nextIndex` = `i` geliefert werden.

Wenn Schritte 1 bis 3 ausgeführt wurden, ohne, dass ein `EvaluationResult` geliefert wurde, wissen wir, dass an Index `i` eine öffnende Klammer steht und, dass `i` nicht der letzte Index ist.

4. Wenn an Index `i + 1` eine schließende Klammer steht, die das Gegenstück zur öffnenden Klammer an Index `i` ist, soll direkt `EvaluationResult` mit `type` = `CORRECT` und `nextIndex=i + 2` geliefert werden.

5. Wenn an Index `i+1` eine schließende Klammer steht, die nicht das Gegenstück zur öffnenden Klammer an Index `i` ist, soll direkt `EvaluationResult` mit `type` = `INVALID_CLOSING_BRACKET` und `nextIndex` = `i+1` geliefert werden.

Wenn weiter auch Schritte 4 bis 6 ausgeführt wurden, ohne dass ein `EvaluationResult` geliefert wurde, wissen wir, dass an Index `i+1` keine schließende Klammer steht. In diesem Fall ruft sich `evaluate` selber auf, wobei für `index` nun `i+1` verwendet wird. Wir bezeichnen die Rückgabe dieses Aufrufs als `r`.

6. Wenn `r.type() != CORRECT`, soll direkt `r` geliefert werden.

7. Wenn `r.nextIndex()` kein Index des Strings ist, soll direkt ein `EvaluationResult` mit `type = NO_CLOSING_BRACKET` und `r.nextIndex` geliefert werden.

Wenn weiter auch Schritte 6 bis 7 ausgeführt wurden, ohne dass ein `EvaluationResult` geliefert wurde, wissen wir weiter, dass sich Index `r.nextIndex` innerhalb des Strings befindet.

8. Wenn an Index `r.nextIndex()` eine schließende Klammer steht, die das Gegenstück zur öffnenden Klammer an Index `i` ist, soll direkt `EvaluationResult` mit `type = CORRECT` und `nextIndex = r.nextIndex()+1` geliefert werden.

9. Wenn an Index `r.nextIndex()` eine schließende Klammer steht, die nicht das Gegenstück zur öffnenden Klammer an Index `i` ist, soll direkt `EvaluationResult` mit `type = INVALID_CLOSING_BRACKET` und `nextIndex = r.nextIndex()` geliefert werden.

10. Wenn an Index `nextIndex` eine öffnende Klammer steht, soll direkt `EvaluationResult` mit `type = NO_CLOSING_BRACKET` und `nextIndex = r.nextIndex()` geliefert werden.

Wenn weiter auch Schritte 8 bis 10 ausgeführt wurden, ohne dass ein `EvaluationResult` geliefert wurde, wissen wir weiter, dass sich an Index `r.nextIndex()` keine Klammer befindet. In diesem Fall wird `EvaluationResult` mit `type = INVALID_CHARACTER` und `nextIndex = r.nextIndex()` geliefert.

H3.2: Disjunkte Klammerausdrücke ?? Punkte

Nach korrekter Bearbeitung von H3.1 kann Methode `evaluate` Klammerausdrücke nur dann korrekt auswerten, wenn diese *nicht* aus disjunkten Klammerausdrücken bestehen.

Modifizieren Sie Ihre Implementation aus H3.1 nun so, dass diese einen *kompletten* Klammerausdruck auswerten kann, wenn dieser aus disjunkten Teil-Klammerausdrücken besteht.

Teil-Klammerausdrücke sollen in der Reihenfolge abgearbeitet werden, in der diese im Klammerausdruck vorkommen. Wenn innerhalb der Auswertung eines Klammerausdrucks ein `EvaluationResult r` mit `r.type() != CORRECT` erzeugt wird, soll *dieses* Objekt bis hin zum ersten Aufruf von `evaluate` weitergereicht werden, ohne dass weitere Teile des Klammerausdrucks ausgewertet werden. ihre Implementation soll ebenfalls erkennen, wenn auf einen korrekten Klammerausdruck ein ungültiges Zeichen folgt und entsprechenden ein `EvaluationResult` mit `type = INVALID_CHARACTER` und `nextIndex` gleich der Position des ungültigen Zeichen liefern.

Verbindliche Anforderung:

Das Schema aus H3.1 darf für diese Aufgabe natürlich so verändert werden, dass die Anforderungen dieser Aufgabe erfüllt werden.

Die Methode `evaluate` muss weiter rekursiv arbeiten.