Check und Prepare

Lösungsvorschläge

TECHNISCHE UNIVERSITÄT DARMSTADT

Autoren: Nhan Huynh und Darya Nikitina
Fachbereich: Informatik
Übungsblatt: 06
Version: 20. November 2021
Semesterübergreifend

V1 Theoriefragen

★★★

Erklären Sie kurz in eigenen Worten die folgenden Konzepte:

1. Was sind Funktionen höherer Ordnung? Wo liegen ihre Vorteile?
2. Was ist ein Lambda-Ausdruck?
3. Wieso sollte man Abstraktion beim Programmieren verwenden?

Lösungsvorschlag:

1. Funktionen höherer Ordnung sind Funktionen, die als Parameter eine Funktion erhalten und oder eine Funktion als Rückgabewert haben. Ihre Vorteile liegen darin, dass sie ihre eigene Funktion anhand des Parameters anpassen (größere Abstraktion) oder Funktionen anhand bestimmter Parameter erstellen können. Durch die Abstraktion ergibt sich eine breitere Verwendungsmöglichkeit, d.h. eine Anpassbarkeit an eine Vielzahl gleichartiger Aufgaben.
2. Lambda-Ausdrücke sind Literale von Funktionstypen und haben daher, wie Literale anderer Typen erst einmal keinen Namen (anonyme Funktion), solange man sie nicht einer Variablen/Konstanten zuweist.

   In Bezug auf Java kann man sich einen Lambda-Ausdruck folgendermaßen vorstellen: Ein Lambda-Ausdruck ist eine anonyme Funktion, welche nur durch Referenzen und Verweise angesprochen werden kann, aber nicht über einen Bezeichner.
3. Durch das Unterteilen in Unterklassen und Methoden muss bei gleichartigen Aufgaben nicht bei jeder dieser Aufgabe alles von Grund auf neu programmiert werden, sondern es wird das Gemeinsame in eine einzige Entität (Klasse, Interface oder Methode in Java; Funktion in Racket) heraus faktorisiert. Es müssen also nur noch die spezifischen Aspekte bei jeder Aufgabe neu implementiert werden. Durch dieses Unterteilen können dann auch in Zukunft zusätzlich Aufgaben von einem ähnlichen Typ gelöst werden, die heute noch gar nicht bekannt sind. Der Hauptvorteil ist, dass man Wiederholungen vermeidet, was nicht nur aufwändig, sondern auch fehleranfällig ist. Ein weiterer Vorteil ist, dass man mit etwas Übung in abstraktem Denken einen solchen Code besser versteht und ihn daher auch besser weiter pflegen kann.

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V2 Ausdrücke mit Funktionen höherer Ordnung

In der Vorlesung haben Sie die Funktionen `my-map`, `my-filter` und `my-fold` kennengelernt und diese selbst implementiert. Alle drei Funktionen gibt es mit identischer Funktionalität bereits vordefiniert in DrRacket und haben die Namen `map`, `filter` und `foldr`.

Was liefern die folgenden Ausdrücke zurück? Arbeiten Sie hier ausschließlich mit Stift und Papier und verwenden Sie DrRacket erst hinterher, nur zur Überprüfung Ihrer Ergebnisse.

1. `(map + (list 1 2 3)(list 4 5 6))`
2. `(filter positive? (list 1 -2 3 4 -5))`
3. `(foldr + 0 (list 5 -9 3 2 5 6))`
4. `(filter string? (list 1 2 "3" "4")`
5. `(first (map list (list "x" "y" "z")))`
6. `(map list (list "a" "b" "c") (list 1 2 3)(list true false true))`
7. `(foldr cons (list -10 -1)(list 1 10 100 1000))`

Lösungsvorschlag:

1. `(list 5 7 9)`
2. `(list 1 3 4)`
3. `12`
4. `(list "3" "abc")`
5. `(list "x" ')`
6. `list (list "a" 1 true) "b" 2 false) "c" 3 true)`
7. `(list 1 10 100 1000 -10 -1)`

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V3 Funktionen höherer Ordnung verwenden ★★★

Definieren Sie die folgenden Funktionen. Außerhalb der Funktionen `map`, `filter` und `foldr` darf keine Rekursion verwendet werden.

- Eine Funktion `zip`, die aus zwei gleich langen Listen eine Liste von geordneten Paaren macht. Beispiel:
  `(zip (list "a" "b") (list 1 2))` → `(list (list) "a" 1) (list "b" 2))`
- Eine Funktion `vec-mult`, die zwei gleich lange Listen von Zahlen erhält und das Skalarprodukt, also die Summe der paarweisen Produkte berechnet. Beispiel:
  `(vec-mult (list 1 2 3)(list 4 5 6))` → `(+ (* 1 4)(* 2 5)(* 3 6))` → `32`

Lösungsvorschlag:

- `zip`

```
1  ;; Type: (list of ANY) (list of ANY) -> (list of (list of ANY))
2  ;; Returns: a list of ordered pairs from two lists of equal length
3  (define (zip list1 list2)
4    (map list list1 list2))
5
6  ;; Tests
7  (check-expect (zip (list "a" "b")(list 1 2)) (list (list "a" 1) (list "b" 2)))
8  (check-expect (zip (list 23 213 321 21 31221 05438)(list "a" "b" "c" "d" "e" "f"))
9                (list (list 23 "a") (list 213 "b") (list 321 "c")
10                     (list 21 "d") (list 31221 "e") (list 5438 "f")))
11 (check-expect (zip (list 4) (list 2)) (list (list 4 2)))
```

- `vec-mult`

```
1  ;; Type: (list of number) (list of number) -> number
2  ;; Returns: the dot product of two vectors as list
3  (define (vec-mult vec1 vec2)
4    (foldr + 0 (map * vec1 vec2)))
5
6  ;; Tests
7  (check-expect (vec-mult (list 1 2 3) (list 4 5 6)) 32)
8  (check-expect (vec-mult (list 1 2 3 4 5 6 7 8 9) (list 9 8 7 6 5 4 3 2 1)) 165)
9  (check-expect (vec-mult (list 213 422 42 312) (list 4324 2131 431 42141))
10               14986388)
```

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V4 Lambda-Ausdrücke

```
1 ;;
2 (define (z x)
3   ;;
4   (lambda (y) (* x y)))
```

Ergänzen Sie den Vertrag, sowohl für die Funktion z, als auch für den Lambda-Ausdruck. Was liefert `((z 3)4)` zurück

Lösungsvorschlag:

```
1 ;; Type: number -> (number -> number)
2 ;; Returns: a function that takes a number x as input and multiplies it by y
3 (define (z x)
4   ;; Type: number -> number
5   ;; Returns: Returns the product of parameter y with the stored
6   ;; value x of the outer procedure
7   (lambda (y) (* x y)))
```

Das Ergebnis von `((z 3)4)` ist 12.

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V5 Foo Reloaded I

Erinnern Sie sich noch an die Funktion `foo` aus Aufgabe V7 vom letzten Übungsblatt? Zur Erinnerung: Gegeben ist ein Struct-Typ `abc` mit zwei Feldern `a` und `b`. Die Funktion `foo` bekommt einen Parameter `p` und liefert falls `p` vom Typ `abc` und zudem der Wert im Feld `b` von `p` eine Liste ist eine Liste zurück, deren erstes Element der Wert von Feld `a` in `p` ist, und der Rest der zurückgelieferten Liste ist die Liste im Feld `b` von `p` (also eine Liste in der Liste). Andernfalls liefert `foo` einfach `false` zurück.

Definieren Sie nun eine Funktion `bar1`, die einen Parameter `lst` übergeben bekommt. Für jedes Element `x` in `lst`, das vom Typ `abc` ist, soll die Ergebnisliste von `bar1` das Ergebnis der Anwendung von `foo` auf `x` enthalten. Weitere Elemente darf die Ergebnisliste von `bar1` nicht enthalten.

> **Verbindliche Anforderung:** Sie dürfen in dieser Aufgabe noch keine Funktionen höherer Ordnung wie `map` oder `filter` verwenden. Diese Funktionalitäten müssen von Ihnen selbst implementiert werden.

Lösungsvorschlag:

```
1 ;; Type: (list of ANY) -> (list of ANY (list of ANY))
2 ;; Returns: a list of the results of using the function foo on abc-structs
3 (define (bar1 lst)
4   (cond
5     [(empty? lst) empty]
6     [(abc? (first lst)) (cons (foo (first lst)) (bar1 (rest lst)))]
7     [else (bar1 (rest lst))]))
```

V6 Foo Reloaded II

Definieren Sie nun eine Funktion `bar2`. Diese besitzt die gleiche Funktionalität wie `bar1` aus Aufgabe V5. In dieser Aufgabe wird die Funktionalität allerdings nicht mehr selbstgeschrieben, sondern an die vordefinierten Funktionen `map` und `filter` delegiert. Nutzen Sie Lambda-Ausdrücke, welche Sie innerhalb der Aufrufe von `map` und `filter` definieren.

Lösungsvorschlag:

```
1 ;; Type: (list of ANY) -> (list of ANY (list of ANY))
2 ;; Returns: a list of the results of using the function foo on abc-structs
3 (define (bar2 lst)
4   (map
5    (lambda (p)
6      (if (and (abc? p) (list? (abc-b p)))(list (abc-a p) (abc-b p)) false))
7    (filter abc? lst)))
```

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V7 Kartesisches Produkt

Definieren Sie eine Funktion `cartesian-prod`, die zwei Zahlenlisten erhält und das kartesische Produkt der beiden bildet. Beispiel:
`(cartesian-prod (list 1 2)(list 3 4))` → `(list (list 1 3)(list 1 4)(list 2 3)(list 2 4))`

Lösungsvorschlag:

```
;; Type: (list of number) (list of number) -> (list of (list of number))
;; Returns; the cartesian product of the two lists
(define (cartesian-product lst1 lst2)
  (foldr
   ; merges the list element x with the intermediate
   ; list "done"
   (lambda (x done)
     (append
      (map
       ; makes a new list element by combining the current
       ; element x from the first list and the
       ; element s from the second list
       (lambda (s) (list x s)) lst2) done))
   empty  lst1))
```

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V8 Bibliothek Leihgebühren ★★★

Eine Bibliothek verwaltet ihr Leihsystem nun in Racket. Dazu wird ein neuer Struct-Typ `br` definiert.

```
1 (define-struct br (id pop type))
```

Das Feld `id` ist dabei ein String und stellt die ID-Nummer des ausgeliehenen Buches dar. Das Feld `pop` ist eine Zahl zwischen 1 bis 6 und gibt die Beliebtheit des Buches an (je größer die Zahl, desto beliebter das Buch). Das letzte Feld `type` ist wieder ein String, der entweder `"Single"` oder `"Subscription"` sein kann, je nachdem, ob es sich um eine einmalige Ausleihe oder einen Abonnenten handelt.

Folgende Regeln gelten in der Bibliothek: Abonnenten zahlen für jedes ausgeliehene Buch pauschal 1,50€. Bei normalen Kunden berechnet sich der Preis über die Beliebtheit des Buches. Pro Beliebtheitsstufe kostet das Buch 1,75€. Somit kostet ein Buch mit Beliebtheitsstufe 3 beispielsweise 5,25€.

Ihre Aufgabe ist es nun eine Funktion `fee-total` zu definieren. Diese enthält eine Liste von `br`-Structs (die Ausleihliste) und gibt die Gesamteinnahmen aus eben dieser Ausleihliste zurück.

Lösungsvorschlag:

```
1  ;; Type: (list of br) -> real
2  ;; Returns: the total fee of borrowed books
3  (define (fee-total books)
4    (foldl + 0
5           ;; Type: br -> real
6           ;; Returns: the fee of a book
7           (map (lambda (b)
8                  (if (string=? (br-type b) "Single")
9                      (* (br-pop b) 1.75)
10                     1.5))
11               books)))
```

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V9 Wer bekommt die Zulassung? ★★★

Schreiben Sie eine Prozedur zur Prüfung der Zuteilung einer Studienleistung im Modul X. Dort sind 50 Hausaufgaben-, 35 Zwischenklausur- und 50 Projektpunkte sowie insgesamt mindestens 180 Punkte aus den drei Bereichen zusammen erforderlich. Definieren Sie dazu eine Funktion `passed` mit folgender Signatur

(`list` of `number`) (`list` of (`list` of `number number number`)) → (`list` of `number`)

Diese Funktion erhält aus Datenschutzgründen separat die Liste der Matrikelnummern sowie eine Liste von Listen mit Punkten für Hausaufgaben, Zwischenklausur und Projekt (in dieser Reihenfolge). Die Precondition ist dabei, dass die Listen gleich lang sind und dass die Matrikelnummer an Position `i` der ersten Liste zu den Punkten an Position `i` der zweiten Liste gehört (vergessen Sie die Precondition nicht im Vertrag der Funktion). Die Ergebnisliste enthält die Matrikelnummern aller Studierenden, die die Bedingungen für die Studienleistung erfüllt haben. Die Reihenfolge der Studierenden soll dabei erhalten bleiben.

Lösungsvorschlag:

```
1  ;; Type: (list of number) (list of (list of number number number))
2  ;; -> (list of number)
3  ;; Returns: a list of matriculation numbers of the people, who received admission
4  (define (passed ids points)
5    (map first ; use only student ID
6         (filter
7          ;; Type: (list of number number number) -> boolean
8          ;; Returns: true, if all requirements for an admission are met
9          (lambda (entry)
10           (and (>= (second entry) 50)
11                (>= (third entry) 35)
12                (>= (fourth entry) 50)
13                (>= (+ (second entry) (third entry) (fourth entry)) 180)))
14         (map cons ids points))))
```

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

V10 Bildverarbeitung in Racket ★★★

Um diese Aufgaben in DrRacket ausführen zu können, setzen Sie bitte `(require 2htdp/image)` in die oberste Zeile Ihrer Datei ein.

Bilder bestehen aus vielen aufeinanderfolgenden Pixeln. Jedes Pixel nimmt dabei genau die Farbe an, die durch sein sogenanntes RGB-Tripel beschrieben werden. Dies ist durch die Darstellung im sogenannten RGB-Farbraum, ein sogenannter technischer Farbraum, der die Farbwahrnehmung durch das additive Mischen der drei Grundfarben nachbildet, begründet. Jede Farbe lässt sich dabei durch ein Tripel *(R, G, B)* darstellen, wobei die drei Zahlen jeweils den Anteil der jeweiligen Grundfarbe angeben. So ist das klassische rot durch $(255, 0, 0)$, gelb als Mischung zweier Grundfarben durch $(255, 255, 0)$ und braun als Mischung aller Grundfarben als $(153, 102, 51)$ dargestellt.

Der Einfachheit wegen benutzen wir nur Bilder im PNG-Format (d.h. Dateiendung `.png`), die keine transparenten Farben enthalten, also keinen Alphakanal besitzen. Ein Bild ist in Racket immer ein Struct vom Typ `image`. Jedes Bild besteht aus seinen aufeinanderfolgenden Pixeln. In Racket ist ein Pixel als `color`-Struct definiert

```
1 (define-struct color (red green blue alpha))
```

Die ersten drei Felder sind das Tripel des RGB-Farbraums und liegen zwischen 0 und 255. Den Alpha-Wert(Siehe Transparenz, Kapitel 02 der Vorlesung) ignorieren wir in dieser Übung, er soll immer auf 255 gesetzt werden. Folgende Funktionen gibt es bereits für die Bildverarbeitung in Racket:

- Um aus einem `image` die entsprechenden `color`-Structs zu bekommen, gibt es die Funktion `(image->color-list img)`.
  Diese gibt eine Liste von `color`-Structs für das übergebene Bild zurück.
- Um aus einer Liste von `color`-Structs ein Bild zu generieren gibt es die Funktion `(color-list->bitmap clr-lst width height)`. Diese benötigt neben der Liste von `color`-Structs auch die Breite und Höhe des zu generierenden Bildes (über die Funktionen `(image-width img)` und `(image-height img)` abrufbar).
- Mit `(bitmap/file "image.png")` laden Sie das Bild im PNG-Format namens "image", welches im gleichen Verzeichnis wie die `.rkt` Datei liegt. Mittels `(save-image img " out.png")` speichern Sie ein `image`-Struct unter dem Namen `"out"` dort.

Nutzen Sie für die folgenden beiden Aufgaben Funktionen höherer Ordnung

1. Definieren Sie eine Funktion `(count-black-white img)`. Diese bekommt ein Bild übergeben, welches nur aus schwarzen $(0, 0, 0)$ und weißen Pixeln $(255, 255, 255)$ besteht. Zurückgegeben werden soll eine zweielementige Liste, welche an erster Position die Anzahl an schwarzen und an zweiter Position die Anzahl an weißen Pixeln enthält.
2. Definieren Sie eine Funktion `(negative-transformation img)`. Diese bekommt ein Bild als `image`-Struct übergeben und gibt die Negativtransformation dieses Bildes zurück. Dazu berechnen Sie die RGB-Werte für jeden Pixel neu über den folgenden Zusammenhang: $(R_{\text{neg}}, G_{\text{neg}}, B_{\text{neg}}) = (255 - R, 255 - G, 255 - B)$

Übungsblatt 06: Lösungsvorschläge von Nhan Huynh und Darya Nikitina

Lösungsvorschlag:

```
;; Type: image -> (list of number)
;; Returns: a list with two elements containing the total number of black and white
;; pixel of the given image
(define (count-black-white img)
  (list
   ;; Type: color -> boolean
   ;; Returns: true if the red component of a color is zero
   (length (filter (lambda (x) (= 0 (color-red x))) (image->color-list img)))
   ;; Type: color -> boolean
   ;; Returns: true if the red component of a color is 255
   (length (filter (lambda (x) (= 255 (color-red x))) (image->color-list img)))))

;; Type: image -> image
;; Returns: the negative transformation of the given image
(define (negative-transformation img)
  (color-list->bitmap
   (map
    ;; Type: color -> color
    ;; Returns: the negative color of the given color
    (lambda (x)
      (make-color
       (- 255 (color-red x))
       (- 255 (color-green x))
       (- 255 (color-blue x)) 255))
    (image->color-list img))
   (image-width img) (image-height img)))
```