Grundlagen der Informatik 1

Sommersemester 2011

Dr. Guido Rößling
https://moodle.informatik.tu-darmstadt.de/

Übung 4 Version: 1.1 09. 05. 2011

---

1 Mini Quiz

Kreuzen Sie die wahren Aussagen an.

1. ☐ Der allgemeine Vertrag contract: (listof X)−> **boolean** ist mit diesem Vertrag vereinbar: f: (listof **number**)−> **number**.
2. ☐ Der allgemeine Vertrag contract: (listof X)−> (X −> **boolean**) ist mit diesem Vertrag vereinbar: f: (listof **number**)−> (**number** −> **boolean**).
3. ☐ f: ((listof X)−> X)−> **boolean** ist ein sinnvoller Vertrag.
4. ☐ **lambda** ist ein Spezialfall von **local** zur Definition anonymer Funktionen.
5. ☐ Die Funktion (**lambda** (x y)(+ x y)) erfüllt den Vertrag **number number** −> **number**.

2 Fragen

1. Was sind Prozeduren höherer Ordnung? Was sind ihre Vorteile?
2. Warum sollte man Abstraktion beim Programmieren verwenden?

Hinweis: Verwenden Sie für diese Übung bitte die Sprachstufe „Intermediate Student with lambda“ bzw. „Zwischenstufe mit lambda“.

3 Local und Lambda (K)

Schreiben Sie die folgende Prozedur um, indem Sie einen äquivalenten **local**-Ausdruck anstelle von **lambda** verwenden.

```
1 ;; multiply : number −> (number −> number)
2 ;; returns a function that takes a number x as input
3 ;; and returns the number multiplied by x
4 ;; example: ((multiply 3) 4) is 12
5 (define (multiply x)
6     ;; number −> number
7     ;; multiplies the parameter y with the stored value x of the outer procedure
8   (lambda (y) (* x y)))
```

4 Prozeduren höherer Ordnung (K)

Sie kennen bereits Prozeduren wie +, die mehrere Argumente konsumieren. In dieser Aufgabe wollen wir nun zeigen, wie man solche Funktionen erzeugen kann aus Funktionen, die nur ein Argument konsumieren.

1. Wie lautet der Vertrag einer Funktion *f*, die zwei Zahlen konsumiert und eine Zahl liefert?
2. Wie lautet der Vertrag einer Funktion *g*, die eine Zahl konsumiert und eine Funktion liefert, die wiederum eine Zahl konsumiert und eine Zahl liefert?
3. Definieren Sie die Funktion add2, die 2 zu einer übergebenen Zahl hinzuaddiert.
4. Definieren Sie die Funktionen add3 und add4, die 3 bzw. 4 zu einer übergebenen Zahl hinzuaddieren.
5. Definieren Sie die Funktion **make**−adder: **number** −> (**number** −> **number**). Sie soll eine Zahl x konsumieren und eine Funktion erzeugen. Die erzeugte Funktion soll eine Zahl y konsumieren und als Ergebnis x + y zurückliefern. So soll (**make**−adder 3) äquivalent zu add3 sein. Verwenden Sie **lambda**, um die zurückgelieferte Funktion zu definieren. Definieren Sie add2, add3 und add4 mit Hilfe von **make**−adder.

   Hinweis: Eine Beispielnutzung von make-adder sieht wie folgt aus: ((**make**−adder 3) 4) mit Ergebnis 7.
6. Definieren Sie eine Funktion add, die unter Verwendung von **make**−adder zwei Argumente addiert.
7. Definieren Sie die Funktion uncurry, die eine Funktion f mit einem Argument konsumiert und eine Funktion g mit zwei Argumenten zurückliefert. Dabei soll die erste Funktion f eine Funktion höherer Ordnung sein, die Funktionen erzeugt, die auf dem zweiten Argument von g operieren. Der Vertrag von uncurry ist also: uncurry: (X −> (Y −> Z))−> (X Y −> Z).
8. Definieren Sie nun add unter Verwendung von **make**−adder und uncurry.
9. Wozu könnte die Beschränkung auf Funktionen von nur einem Argument sinnvoll sein? Warum erlaubt man in der Praxis dennoch Funktionen auf mehreren Variablen?

5 Fold, Map und Co (K)

1. Wie lautet der Vertrag von **map**?
2. Definieren Sie die Funktion my−**map**, die für eine Funktion und eine Liste als Parameter das Selbe tut wie **map**, aber ohne **map** zu verwenden.
3. Zu was werden (**foldl cons empty** '(1 2 3 4)) und (**foldr cons empty** '(1 2 3 4)) ausgewertet?
4. Zu was wird (**map** + '(1 2 3)'(4 5 6)) ausgewertet?
5. Definieren Sie die Prozedur zip, die aus zwei gleich langen Listen eine Liste von geordneten Paaren macht. Beispiel: (zip '(a b c)'(1 2 3)) ergibt (**list** '(a 1)'(b 2)'(c 3)).
6. Definieren Sie eine Funktion vec−mult, die zwei gleich lange Vektoren (Listen von Zahlen) erhält und das Skalarprodukt—also die Summe der paarweisen Produkte, $\sum_i a_i * b_i$—berechnet. Verwenden Sie **map** und/oder **foldl**.
7. Definieren Sie eine Funktion cartesian−product, die zwei Listen $l_1 : (l_{11}...l_{1n})$ und $l_2 : (l_{21}...l_{2n})$ erhält und das kartesische Produkt $((l_{11}, l_{21}), ...(l_{11}, l_{2n}), ...(l_{1n}, l_{21}), ...(l_{1n}, l_{2n}))$ dieser beiden Listen zurückgibt. Verwenden Sie **map** und **foldr**.

Hausübung

Die Vorlagen für die Bearbeitung werden im Lernportal Informatik bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im Lernportal Informatik abgegeben werden.
Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe Ihrer Hausübung bestätigen Sie, dass Sie bzw. Ihre Gruppe alleiniger Autor des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitieren.
Falls Sie die Hausübung in einer Lerngruppe bearbeitet haben, geben Sie dies bitte deutlich bei der Abgabe an. Alle anderen Mitglieder der Lerngruppe müssen als Abgabe einen Verweis auf die gemeinsame Bearbeitung einreichen, damit die Abgabe im Lernportal Informatik auch für sie bewertet werden kann. Beachten Sie dazu die Hinweise bei der Aufgabenabgabe im Lernportal Informatik!

Abgabedatum: Freitag, 20. 05. 2011, 16:00 Uhr

Denken Sie bitte daran, Ihren Code hinreichend gemäß den Vorgaben zu kommentieren (Racket: Vertrag, Beschreibung und Beispiel sowie zwei Testfälle pro Funktion; Vertrag, Beschreibung und Beispiel für jede **local** definierte Funktion; Vertrag (ohne Namen) und kurze Beschreibung für jeden **lambda**-Ausdruck; Java: JavaDoc). Zerlegen Sie Ihren Code sinnvoll und versuchen Sie, wo es möglich ist, bestehende Funktionen wiederzuverwenden. Wählen Sie sinnvolle Namen für Hilfsfunktionen und Parameter.

Hinweis:

Im Portal stehen neben dem eigentlichen Kurs noch die folgenden Hilfsmittel als separate Kurse für Sie bereit:

- Kurs „Hilfen zur Nutzung des Portals“: hier finden Sie Information zum Portal und wie man es möglichst effizient nutzen kann. Das Portal kann sehr viel, und nicht alles davon sieht man auf den ersten Blick. Daher empfehlen wir Ihnen sehr, sich auch in diesen Kurs einzutragen und die dortigen Materialien durchzuarbeiten!

- Kurs „Tipps zum effektiven Studieren“: hier erhalten Sie Tipps rund um das Studium, etwa zum Bilden von Lerngruppen oder der Vorbereitung auf Klausuren. Eine Einschreibung in diesen Kurs ist daher ebenfalls ratsam. Zusätzlich arbeiten wir an Tipps zur Programmierung, etwa Hinweisen zum korrekten Verfassen von Verträgen in Scheme.

Nutzen Sie diese Hilfsangebote! Kommentare und Anregungen zu weiteren Themen sind willkommen.

6 Bildkompression (10 Punkte)

Bilder werden im Computer in verschiedenen Formaten codiert. Das einfachste Format (auch „Bitmap“ genannt) verwendet für jeden einzelnen Bildpunkt—oder „pixel“ für **pi***cture* **el***ement*—die Angabe der Rot-, Grün- und Blauanteile: die sogenannten RGB-Daten. Racket modelliert Bildpunkte mit einer color-Struktur ähnlich der aus Übung 3.3 mit den Feldnamen red, green und blue.
Bereits ein einfaches Bild wie die deutsche Nationalfahne führt bei dieser Darstellung zu einer sehr großen Anzahl an Daten: bei 150 Pixeln Breite und 90 Pixeln Höhe sind 13,500 color-Strukturen mit je 3 Zahlenwerten zu speichern!
In dieser Hausaufgabe befassen wir uns mit einer vergleichsweise einfachen Möglichkeit, wie die Datenmenge bei vielen—aber nicht allen—Bildern teilweise drastisch reduziert werden kann. So werden aus der Deutschlandfahne aus den erwähnten 40,500 Zahlen lediglich 12 (!). Es handelt sich dabei um die sogenannte *Lauflängenkodierung*, auf Englisch *Run Length Encoding*, kurz *RLE*.
Hinweis: Im Template werden mehrere Pixel vordefiniert—für die Farben weiß, grün, rot, blau und schwarz—, die Sie natürlich für Ihre Beispiele und Tests nutzen können. Bei der gesamten Hausaufgabe müssen Sie *keine eigenen Tests* schreiben; wir empfehlen aber, dies dennoch zu tun, um sicherzugehen, dass Ihr Code sauber funktioniert!

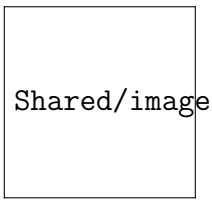

Abbildung 1: Die Deutsche Nationalfahne, © Wikimedia Commons

6.1 Erklärung des RLE-Verfahrens (0 Punkte)

Die Lauflängenkodierung nutzt eine an sich ganz einfache Idee, die man wie folgt beschreiben kann: „Wenn im Bild n Mal die gleiche Farbe direkt nacheinander auftritt, merke Dir nur die Farbe und wie oft sie auftrat“.
In der Deutschlandfahne aus der Abbildung treten insgesamt nur drei Farben auf: schwarz, ein Rot- und ein Goldfarbton. Da alle Farbstreifen die gleiche Höhe haben, kommen bei einer Höhe von 90 und einer Breite von 150 Pixeln zunächst $150 \times 30 = 4{,}500$ schwarze Pixel, dann 4,500 rote und am Ende 4,500 goldfarbene Pixel. Das Ergebnis der Kompression ist also sinngemäß „schwarz: 4,500 Mal; rot: 4,500 Mal; gold: 4,500 Mal“.
Im Template wird die folgende Datenstruktur für die RLE-Daten vorgegeben:

```
1 ;; rle: structure that represents run−length encoded colors
2 ;; color−value: color − the actual color entry
3 ;; count: number − the count of the "run" for the color; at least 1
4 (define−struct rle (color−value count))
```

Das Ergebnis der Lauflängencodierung der Deutschlandfahne als (listof rle) ist also wie folgt[1]. Die Felder in **make**−color stehen jeweils für den Rot-, Grün- und Blauanteil der Bildpunkte:

```
1 (list
2  (make−rle (make−color 0 0 0) 4500)
3  (make−rle (make−color 221 0 0) 4500)
4  (make−rle (make−color 255 206 0) 4500))
```

Zu beachten: Intern wird auch mit einer Variante der color-Struktur mit *vier* Werten gearbeitet. Der letzte Wert gibt dabei die Deckungskraft bzw. Durchsichtigkeit (Transparenz) der Farbe an und ist hier immer 255. Da gilt (equal? (**make**−color x y z)(**make**−color x y z 255)), können Sie den vierten Wert einfach ignorieren.

6.2 Vorbereitung (0 Punkte)

Laden Sie zunächst die Vorlage für die Hausaufgabe sowie die als germany.png bereitgestellte deutsche Nationalfahne aus dem Lernportal Informatik. Neben den Definitionen der verschiedenen Datentypen finden Sie in der Vorlage in Zeile 12 folgende Definition:

```
12 (define german−flag ...) ;; put actual image here!
```

Entfernen Sie die drei Pünktchen und klicken Sie an dieser Stelle im Menü auf „Einfügen → Bild einfügen. . . “ und wählen Sie das Bild germany.png aus. Dieses sollte nun hier erscheinen. Stellen Sie durch Klick auf „Sprache“ sicher, dass das Teachpack image.ss hinzugefügt ist[2]. Falls das nicht der Fall ist, fügen Sie es über „Sprache → Teachpack hinzufügen. . . “ und Auswahl von `image.ss` hinzu.

[1] Für das Rot wird nur ein Rotanteil von 221, für Gold hingegen Rot 255 und Grün 206 verwendet
[2] Wenn das Teachpack hinzugefügt ist, wird eine Option „Teachpack image.ss entfernen“ angezeigt.

Als Sprachstufe für diese Hausaufgabe **muss** *Zwischenstufe* bzw. *Intermediate* verwendet werden, **nicht** *Zwischenstufe mit lambda*. Aus unklaren Gründen muss sonst einerseits eine Prozedur umbenannt werden[3], und—viel wichtiger—die Tests auf Gleichheit schlagen teilweise fehl.
Hinweis: Eine Übersicht über die Funktionalität des `image.ss` Teachpacks finden Sie im Hilfezentrum bzw. unter `http://docs.racket-lang.org/teachpack/image.html`. Für uns wichtig sind die folgenden Funktionen:

- image−>color−**list**: image −> (listof color) erstellt aus einem Bild eine Liste von color-Strukturen. Eine color-Struktur hat die Felder red, green, blue, deren Werte jeweils **ganze Zahlen** zwischen 0 (Farbe nicht enthalten) und 255 (Farbe in voller Intensität vorhanden) sein **müssen**.
- image−width: image −> **number** und image−height: image −> **number** bestimmen die Breite bzw. Höhe eines Bildes.
- color−**list**−>bitmap: (listof color)**number number** −> image erstellt ein neues Bild aus einer Liste von color−Strukturen. Der zweite und dritte Zahl geben sind die Breite und Höhe des Zielbildes an.

Zu beachten: Entfernen Sie bitte **unbedingt alle Bilder aus Ihrem Quelltext** *bevor* Sie ihn abgeben!

6.3 Implementierung der Kompression (4 Punkte)

Implementieren Sie eine Prozedur rle−compress: (listof color)−> (listof rle). Diese erhält die (flache) Liste von Farbwerten für jeden einzelnen Bildpunkt—für die deutsche Nationalfahne also eine Liste mit 13,500 (!) color-Strukturen mit insgesamt 40,500 Zahlen. Die Prozedur liefert eine Liste der RLE-codierten Daten. Für die deutsche Nationalfahne ist das Ergebnis also identisch—oder gemäß dem Hinweis am Ende von Teil 6.1 bezüglich der Prozedur equal? äquivalent—zu der Liste aus Aufgabe 6.1 und hat die Länge 3.
Um diese Aufgabe sinnvoll bearbeiten zu können, sollten Sie vorab einige Überlegungen und Vorbereitungen treffen:

1. Überlegen Sie sich, wie sie einen „run“, also ein zusammenhängendes Auftreten von Pixeln der gleichen Farbe, erkennen können. Überlegen Sie sich auf dieser Basis, wie Sie in diesem Fall vorgehen wollen.
2. Überlegen Sie sich, was zu tun ist, wenn Sie an das Ende eines „runs“ kommen.
3. Überlegen Sie sich, was beim Erreichen des Endes der gesamten color-Liste zu tun ist.
4. Implementieren Sie Code für die drei vorherigen Fälle—als unabhängige Prozeduren oder als Teil einer größeren Prozedur—und testen Sie den Code schrittweise.
5. Konvertieren Sie das Bild der Nationalfahne in eine Farbliste und prüfen Sie, ob Ihr Ergebnis korrekt ist: es sollte bei Nutzung von (equal? ...) verglichen mit dem Ergebnis aus Abschnitt 6.1 **true** liefern.

Hinweis: Statt mit einem komplexen Bild wie der bereitgestellten Deutschlandfahne zu arbeiten, können Sie rle−compress auch mit einer selbsterstellten Liste, etwa (**list** red red green green), aufrufen (diese Namen werden im Template als Instanzen von color definiert). Dann „sehen“ Sie schneller, ob das Verhalten korrekt sein kann oder nicht. Wenn Sie mit echten Bildern arbeiten wollen, nehmen Sie nur *kleine* Bilder und nutzen Sie die in Abschnitt 6.2 genannten Prozeduren.
Sie sollten im eigenen Interesse bei dieser Aufgabe mit lokal definierten Prozeduren und vor allem gebundenen Namen arbeiten. Die Konvertierung eines Bilds in eine (listof color) dauert

[3]Wer es genau wissen will: color−**list**−>bitmap ist dann durch color−**list**−>image zu ersetzen

einen Moment; binden Sie daher das Ergebnis an einen Namen, um es nicht mehrfach berechnen zu müssen! Beachten Sie aber, dass lokale definierte Prozeduren *nicht getestet werden können*: schreiben Sie diese daher **erst global, testen Sie sie, und integrieren Sie dann als lokale Prozeduren**!

6.4 Dekompression von RLE-Daten (4 Punkte)

Implementieren Sie nun eine analoge Prozedur rle−decompress: (listof rle)−> (listof color). Diese konsumiert eine Liste von rle-Strukturen und wandelt sie in eine Liste von color-Strukturen.
Überlegen Sie sich auch hierzu der Reihe nach, ...

1. wie Sie über alle rle-Strukturen iterieren können;
2. wie Sie *eine* rle-Struktur in eine Liste von „entsprechend“ vielen color-Strukturen verwandeln können;
3. wie das Ergebnis aus dem vorherigen Schritt mit dem nächsten Ergebnis kombiniert werden muss.

Hinweis: Auch hier sind lokale Funktionen und Namensbindungen—nach der Faustformel „global schreiben, testen, dann lokal einbetten“!—sinnvoll. Sie sollten auch eigene (einfache) Tests mit von Hand erstellten (kurzen) Parametern nutzen.

6.5 Ausgabe des Ergebnisses (2 Punkte)

Zum Abschluss der Aufgabe wollen wir noch sehen, wie gut die RLE-Kompression arbeitet. Implementieren Sie dazu eine Methode rle−information: image −> **String**. Diese konsumiert ein *Bild*—keine Liste von color-Strukturen!—und gibt einen String im folgenden Format zurück:

```
1 Original␣size:␣13500,␣RLE␣size:␣3,␣Factor:␣1/4500␣−␣OK
```

Beachten Sie dazu die folgenden Hinweise:

- Auf Folie T2.17 finden Sie die Prozeduren **string**−**append**: **string**+ −> **string**, die beliebig viele Strings zu einem neuen String zusammenfügt, sowie **number**−>**string**: **number** −> **string**.
- Auch hier *sollten* Sie in jedem Fall mit gebundenen lokalen Namen arbeiten, um (viel) Rechenzeit einzusparen! Eine ineffiziente Mehrfachberechnung der gleichen Werte führt hier zu **0,5 Punkten Abzug**.
- Der String muss exakt den gewünschten Aufbau—einschließlich der oben extra markierten Leerzeichen—haben. Die Zahl 1/4500 entsteht durch die Division der Anzahl rle-Strukturen (3) durch die Anzahl color-Strukturen (13,500). Dass dies als Bruch angezeigt wird, ist die Entscheidung von DrRacket.
- Die letzte Stelle ist entweder der Text OK oder der Text—**nicht** ein über (**error** ...) ausgelöster Fehler!—compression/decompression **error**. OK darf nur verwendet werden, wenn die innerhalb der Methode *komprimierte* Darstellung nach Dekompression dem Ursprungsbild (hinsichtlich equal?) entspricht.