Grundlagen der Informatik 1
WS 2008/09

Prof. Mühlhäuser, Dr. Rößling, Melanie Hartmann, Daniel Schreiber
http://proffs.tk.informatik.tu-darmstadt.de/gdi1

Übung 4 Lösungsvorschlag v1.0 10.11.2008

---

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?

Lösungsvorschlag:

1. *Prozeduren höherer Ordnung sind Prozeduren, die andere Prozeduren als Parameter konsumieren oder als Ergebnis zurückliefern. Sie können z.B. allgemeine, häufige Berechnungsmethoden von den beteiligten Funktionen abstrahieren.*

   So löst z.B. `map`, `foldr`, `filter`... die Implementierung von Prozeduren, die eine Liste transformieren sollen, von den spezifischen Details, wie die einzelnen Elemente der Liste extrahiert und wieder kombiniert werden. Dies kann den Code sowohl kürzer als auch verständlicher machen (z.B. T5.51 vs. T5.50)

2. *Abstraktion führt dazu, dass wir anders über ein Problem denken und allgemeine Lösungsansätze wiederverwenden können. Bei der Implementierung führt sinnvolle Abstraktion dazu, dass jede Programmfunktion nur an einer Stelle konzentriert ist. Sollte an dieser Funktion eine Änderung nötig werden, so sollte das keine Auswirkung auf andere Teile des Programms haben. Obwohl dies in der Praxis nicht immer erreicht werden kann, ist dies das angestrebte Ziel.*

3 Local und Lambda (K)

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

```
;; multiply : number -> (number -> number)
;; returns a function that takes a number as input and
;; and returns the number multiplied by x
(define (multiply x)
   (lambda (y) (* x y)))
```

Lösungsvorschlag:

```
;; multiply : number -> (number -> number)
;; returns a function that takes a number as input and
;; and returns the number multiplied by x
(define (multiply x)
  (local
    ((define (multiplier y) (* x y)))
    multiplier))
```

4 Prozeduren Höherer Ordnung (K)

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

- Wie lautet der Vertrag einer Funktion, die zwei Zahlen konsumiert und eine Zahl liefert?
- Wie lautet der Vertrag einer Funktion, die eine Zahl konsumiert und eine Funktion liefert, die wiederum eine Zahl konsumiert und eine Zahl liefert?
- Definieren Sie die Funktion `add2`, die 2 zu einer übergebenen Zahl hinzuaddiert.
- Definieren Sie die Funktionen `add3` und `add4`, die 3 bzw. 4 zu einer übergebenen Zahl hinzuaddieren.
- 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` zurück liefern. Zum Beispiel 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`.
- Definieren Sie die Funktion `add`, die zwei Argumente addiert, unter Verwendung von `make-adder`.
- 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)`.
- Definieren Sie nun `add` unter Verwendung von `make-adder` und `uncurry`.

- Wozu könnte die Beschränkung auf Funktionen von nur einem Argument sinnvoll sein? Warum erlaubt man in der Praxis dennoch Funktionen mehrer Variablen?

Lösungsvorschlag:

- `number number -> number`
- `number -> (number -> number)`

```
1  ;; add2 : number -> number
2  (define (add2 x) (+ x 2))
3  ;; add3 : number -> number
4  (define (add3 x) (+ x 3))
5  ;; add4 : number -> number
6  (define (add4 x) (+ x 4))
7
8  ;;make-adder: number -> (number -> number)
9  (define (make-adder x)
10   (lambda (y) (+ x y)))
11
12 ;;add2
13 (define add2-with-make-adder (make-adder 2))
14
15 ;;add3
16 (define add3-with-make-adder (make-adder 3))
17
18 ;;add4
19 (define add4-with-make-adder (make-adder 4))
20
21 ;;add: number number -> number
22 (define (add x y)
23   ((make-adder x) y))
24
25 ;;uncurry: (number -> (number -> number)) -> (number number -> number)
26 (define (uncurry f)
27   (lambda (x y) ((f x) y)))
28
29 ;;add
30 (define add-with-uncurry (uncurry make-adder))
31
32 (check-expect
33  (add-with-uncurry 3 4)
34  7)
```

- *Die Struktur der Sprache wird vereinfacht. Die Regeln des "ultimativen Lambdas" können noch weiter vereinfacht werden, es reichen Lambda-Ausdrücke einer Variablen aus. In der Praxis werden Programme, die nur aus einstelligen Lambda Ausdrücken aufgebaut sind, sehr lang. Daher ist es sinnvoll, Funktionen mehrerer Variablen als* Abstraktion *einzuführen.*

5 Fold, Map und Co (K)

- Wie ist der Vertrag von `map`?

- Definiern Sie die Funktion `mymap`, die für eine Funktion und eine Liste als Parameter das Selbe tut wie `map`, aber ohne `map` zu verwenden.
- Zu was wird `(foldl + 4 '(1 2 3))` ausgewertet?
- Zu was wird `(map + '(1 2 3) '(4 5 6))` ausgewertet?
- 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))`.
- Definieren Sie eine Funktion `vec-mult`, die zwei gleich lange Vektoren (Listen von Zahlen) erhält und das Skalarprodukt berechnet. Verwenden Sie `map` und `foldl`.
- * 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 zurück gibt. Verwenden Sie `map` und `foldr`.

Lösungsvorschlag:

```
1  ;;mymap: (X -> Y) listof-X -> listof-Y
2  (define (mymap f l)
3    (cond
4      [(empty? l) empty]
5      [else (cons (f (first l)) (mymap f (rest l)))]))
6
7  (check-expect
8   (mymap add1 '(1 2 3))
9   '(2 3 4))
10
11 (check-expect
12  (foldl + 4 '(1 2 3))
13  10)
14
15 (check-expect
16  (map + '(1 2 3) '(4 5 6))
17  '(5 7 9))
18
19 ;;zip: listof-X listof-Y -> listof-(X;Y)
20 (define (zip l1 l2)
21   (map list l1 l2))
22
23 (check-expect
24  (zip '(1 2 3) '(a b c))
25  (list '(1 a) '(2 b) '(3 c)))
26
27 ;;zip: lon lon -> n
28 (define (vec-mult v1 v2)
29   (foldl + 0 (map * v1 v2)))
30
31 (check-expect
32  (vec-mult '(1 2 3) '(1 0 2))
33  7)
34
35 ;;cartesian-product list-of-X list-of-Y -> list-of-(X;Y)
36 (define (cartesian-product l1 l2)
```

```
37   (foldr (lambda (f done) (append (map (lambda (s) (list f s)) l2) done))
        empty l1))
38
39 (check-expect
40  (cartesian-product '(a b c) '(1 2 3))
41  (list '(a 1) '(a 2) '(a 3) '(b 1) '(b 2) '(b 3) '(c 1) '(c 2) '(c 3)))
```

Hausübung

Die Vorlagen für die Bearbeitung werden im GdI1-Portal bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im GdI1-Portal 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.

Abgabedatum: Fr., 21.11.08, 16:00

6 Datenbanken (6P)

1. Schreiben Sie eine Funktion `make-where`, die eine Funktion `f: X -> boolean` konsumiert und einen Listen-Filter zurück gibt. Ein Listen-Filter soll auf eine `list-of-X` angewandt, diejenigen Elemente liefern, für die `f true` liefert. Ein Beispiel: `where-larger-4` soll ein Filter sein, der Zahlen aus einer Liste filter, die größer 4 sind. Ein Aufruf würde dann so aussehen:

   ```
   ;; Test
   (where-larger-4 '(1 2 3 4 5 6))
   ;; should be
   '(5 6)
   ```

2. Schreiben Sie eine Funktion `make-selector`, die eine Funktion `f: X -> list` konsumiert und einen Listen-Selektor zurück gibt. Ein Listen-Selektor soll auf eine `list-of-X` angewandt werden, wobei `X` eine Struktur ist. Zurückgeben soll er Listen bestimmter Elemente der Struktur. Ein Beispiel: `select-name` soll ein Selektor sein, der die Namen von Studenten selektiert:

   ```
   ;; Test
   (select-name
      (list (make-student 'daniel 1) (make-student 'melanie 5)))
   ;; should be
   (list '(daniel) '(melanie))
   ```

   Noch ein Beispiel: `select-name-matrikel` soll neben dem Namen auch die Matrikelnummer von Studenten selektieren:

   ```
   ;; Test
   (select-name-matrikel
      (list (make-student 'daniel 1) (make-student 'melanie 5)))
   ;; should be
   (list '(daniel 1) '(melanie 5))
   ```

   Ein drittes Beispiel: `select-matrikel-name` selektiert auch Namen und Matrikelnummer, aber in umgekehrter Reihenfolge:

```
;;Test
(select-matrikel-name
   (list (make-student 'daniel 1) (make-student 'melanie 5)))
;;should be
(list '(1 daniel) '(5 melanie))
```

3. Schreiben Sie eine Funktion `query`, die einen Selektor, einen Filter und eine Liste übergeben bekommt. Im Ergebnis soll der Selektor auf die Bestandteile der Liste angewandt werden, auf die der Filter zutrifft.

```
;;Test
(query select-name where-matrikel-larger-4
 (list (make-student 'daniel 1) (make-student 'melanie 5)))
;;should be
(list '(melanie))
```

7 Bild-Operationen (7P)

In dieser Übung wollen wir uns mit Bildverarbeitung beschäftigen. Um in Scheme Bilder verabeiten zu können, muss das Teachpack `image.ss` installiert werden. Hierzu rufen Sie in der Menüleiste die Option "Sprache → Teachpack hinzufügen..." auf. Der Pfad der Datei lautet PLT/collects/teachpack/image.ss.
Fügen Sie das Bild robocup.bmp das mit der Übung veröffentlicht wurde in die Definitionen ein. Gehen Sie dazu folgendermaßen vor: Laden Sie das Bild aus dem Portal herunter. Rufen Sie in der Menüleiste die Option "Spezial → Bild einfügen..." auf. Wählen Sie die Bilddatei aus. Das Bild sollte nach kurzer Zeit erscheinen. Geben Sie dem Bild mit `define` den Namen `robocup`. Sie können nun die Funktion `image->color-list` verwenden, um das Bild in eine List von Punkten zu transformieren. Probieren Sie die Funktion aus, indem Sie `(image->color-list robocup)` ausführen.

Betrachten Sie die Funktion copy, die ein Bild erhält, es in eine Punktliste verwandelt und aus dieser eine Kopie des ursprünglichen Bildes erzeugt:

```
(define (copy img)
   (color-list->image
    (image->color-list img)
    (image-width img)
    (image-height img)
```

```
6       0 0))
```

Sie verwendet die Funktion
`color-list->image: list-of-color number number number number -> image`. Die vier numerischen Parameter dieser Funktion sind die Breite und die Höhe des Bildes sowie ein Koordinatenpaar, das hier nicht weiter wichtig ist. Da die Breite und Höhe des Ergebnisbildes mit denen des Ausgangsbildes übereinstimmen, werden hier `(image-width)` und `(image-height)` verwendet, um die Breite und Höhe des Ausgangsbildes zu ermitteln. Für die Koordinaten kann `0;0` verwendet werden.

1. Schreiben Sie eine Funktion `filter-img`, die eine Funktion
   `f: make-color -> make-color` als Parameter erhält, diese auf jeden Punkt des Bildes anwendet und anschließend die Punkte wieder zu einem Bild zusammensetzt. Orientieren Sie sich an der Prozedur `copy`. Verändern Sie die Punktliste in der dritten Zeile der Prozedur `copy`. `(filter-img robocup (lambda (x) (make-color 255 0 0)))` ergibt ein völlig rotes Bild.

2. Schreiben Sie eine Prozedur `check-pixel`, die eine Funktion `f: bildpunkt -> boolean` erhält und eine Funktion von Bildpunkt zu Bildpunkt liefert. Dabei soll die Funktion einen schwarzen Bildpunkt (`(make-color 0 0 0)`) liefern, wenn `(f P)` `false` ist, sonst eine weißen (`(make-color 255 255 255)`). Wenn Sie diese Prozedur an `filter-img` übergeben können Sie das Bild schwarz und weiß einfärben. `(filter-img robocup (check-pixel (lambda (x) true)))` ergibt ein völlig weißes Bild.

3. Erinnern Sie sich an die Prozedur `hue` aus dem 3. Übungsblatt. Diese rechnet von RGB Werten wie sie in Punktlisten verwendet werden in Farbwerte um. Schreiben Sie eine Prozedur `compare-hue`, die zwei Zahlen: einen Farbwert und eine Toleranz übergeben bekommt und eine Funktion von Bildpunkten zu boolean zurückliefert. Die zurückgelieferte Funktion soll `true` ergeben, wenn die Farbe des Bildpunktes weniger als die Toleranz vom Farbwert abweicht.

4. Verwenden Sie "chaining" um mit `compare-hue` erzeugte Funktionen in `check-pixel` und diese wiederum in `filter-img` zu verwenden. Schaffen Sie es, nur den orangenen Ball weiß zu färben? Auf ähnliche Art und Weise erkennen die Roboter beim RoboCup den Spielball.