Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Programming Concepts and Paradigms (PCP)

Programmierübung zu Scheme 5+6

Hauptthemen:	Funktionale Programmierung: Funktionen höherer Ordnung, Funktionen mit Gedächtnis
Zeitfenster:	ca. 4-8 Lektionen

R. Diehl, HS2018

Diese Übung repetiert und vertieft den in Scheme 5 und 6 vermittelten Stoff. Gehen Sie zur Vorbereitung nochmals durch die Unterrichtsfolien durch und schauen Sie insbesondere, dass Sie alle auf den Folien angegebenen Code-Beispiele verstanden haben.

Die mit * gekennzeichneten Aufgaben müssen dem Dozenten oder Assistenten als Teil des Testats gezeigt werden.

Lokale Definitionen und Lexikalisches Scoping

(Einstellung in DrRacket: "Intermediate Student")

1. Aufgabe

Sie haben das Sortierens durch Einfügen aus „Programmierübung zu Scheme 3+4“ mit einem zusätzlichen Parameter implementiert:

```
; Sortieren durch Einfügen
(define (sort-a-list op a-list)
  (cond
    ((empty? a-list) empty)
    (else (insert op (first a-list)
                  (sort-a-list op (rest a-list))))
    ))

; Einfügen in sortierter Liste
(define (insert op item a-list)
  (cond
    ((empty? a-list) (list item))
    ((op item (first a-list)) (cons item a-list))
    (else (cons (first a-list) (insert op item (rest a-list))))
    ))
```

Die Hilfsfunktion `insert` wird eigentlich nur innerhalb der Funktion `sort-by-insert` benötigt. Integrieren Sie deshalb die Hilfsfunktion `insert` als lokale Funktion in die Funktion `sort-by-insert`.

2. Aufgabe *

Sie kennen die Fibonacci-Folge:

$$\text{fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & n \geq 2 \end{cases}$$

Die Definition einer strukturellen Rekursion könnte so aussehen

```
(define (fib n)
  (cond
    ((or (= n 0) (= n 1)) n)
    (else (+ (fib (- n 1))
             (fib (- n 2))))))
)
```

a) Implementieren Sie eine Rekursion mit Akkumulator. Nutzen Sie dazu die `local` Definition.
b) Führen Sie Vergleich Tests mit und ohne Akkumulator durch. Welche Unterschiede sehen Sie?

3. Aufgabe *

a) Was ist die Ausgabe des folgenden Scheme-Programms?

```
(define a 42)

(let ((a 1)
      (b (+ a 1)))
  b)

(let* ((a 1)
       (b (+ a 1)))
  b)
```

b) Erklären Sie, warum sich die beiden Ausdrücke unterscheiden.

Anonyme Funktionen

(Einstellung in DrRacket: "Advanced Student")

4. Aufgabe *

a) Was ist die Ausgabe des folgenden Scheme-Programms?

```
(define x 1)
(define y 5)

((lambda (x y)
   (+ (* 2 x) y))
 y x)

((lambda (a b)
   (+ (* 2 x) y))
 y x)
```

b) Erklären Sie, warum sich die beiden Ausdrücke unterscheiden.

5. Aufgabe *

Angenommen, man hat die Liste

```
(define a-list (list (list 1 2 3) (list 1 2) (list 1 2 3 4)))
```

und möchte jede Liste mit 0 beginnen lassen. Wie kann man dies erreichen, ohne, dass extra eine Funktion (mit Namen) geschrieben werden muss?

6. Aufgabe *

Angenommen, man hat eine Liste mit Funktionen zur Berechnung von Eigenschaften eines Rechteckes, hier Fläche und Umfang:

```
(define rect-calc-list
  (list (lambda (a b) (* a b)) (lambda (a b) (* 2 (+ a b)))))
```

Implementieren Sie eine Funktion, der man die Liste mit Funktionen und die Seiten eines Rechteckes übergeben kann, dann die Eigenschaften berechnet und ausgibt.

Eine mögliche Interaktion könnte so aussehen:

```
> (calc-a-list rect-calc-list 2 3)
6
10
finished
> (calc-a-list rect-calc-list 5 5)
25
20
finished
>
```

Funktionen mit Gedächtnis

(Einstellung in DrRacket: "Advanced Student")

7. Aufgabe

Wir haben ein globales Verzeichnis mit Vornamen und Telefonnummer von Personen:

```
(define my-phone-dir (list (list 'Adam 4711) (list 'Eva 4712)))
```

Ein Programm soll zwei Möglichkeiten bieten:

1. Suchen der Nummer anhand des Vornamens, etwa durch

```
; nachschauen: liste symbol --> zahl oder false
(define (look-at phone-dir name) (...))
```

2. Hinzufügen einer neuen Person mittels Vornamen und Nummer im globalen Verzeichnis

```
; hinzufuegen: symbol zahl --> void
(define (add-entry name number) (...))
```

Eine mögliche Interaktion könnte dann folgendermaen verlaufen:

```
> (look-at my-phone-dir 'Adam)
4711
> (look-at my-phone-dir 'Erna)
#false
> (add-entry 'Erna 4715)
(void)
; dies bedeutet, dass die Funktion add-entry nichts zurück gibt
> (look-at my-phone-dir 'Erna)
4715
```

a) Warum widerspricht diese Interaktion fundamental unserem bisherigen funktionalen Programmiergrundsatz?
b) Schreiben Sie die Funktion `look-at` für das Suchen der Nummer anhand des Vornamens
c) Schreiben Sie die Funktion `add-entry` für das Hinzufügen einer neuen Person mit Vornamen und Nummer.

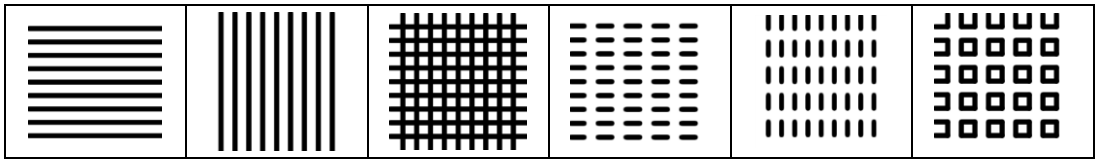
GIMP Skript *

Zu dieser Aufgabe erhalten Sie die Vorlage `gridlines.scm`. Zum besseren Verständnis der Vorlage empfiehlt es sich das Script-Fu Tutorial der GIMP Dokumentation durchzulesen:

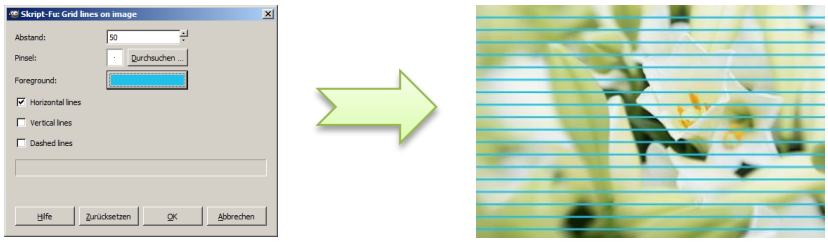
https://docs.gimp.org/de/gimp-using-script-fu-tutorial-first-script.html

Als Editor wird DrRacket (Spracheinstellung: "R5RS") empfohlen. Sie können damit den Tiny Scheme Code und die Verwendung der Namen prüfen (Check Syntax), aber nicht ausführen.

a) Schreiben Sie ein GIMP Skript, das ein konfigurierbares Gitternetz im GIMP zeichnet. Die Vorlage enthält, dass ein neues Bild erstellt wird, Hintergrund- und Strichfarbe konfigurierbar sind, der Hintergrund transparent eingestellt werden kann. Sie müssen das Zeichnen der folgenden Gitternetz-Varianten umsetzen:



b) Erstellen Sie ein neues GIMP Skript, basierend auf dem Skript aus a), mit dem man auf ein bestehendes Bild ein konfigurierbares Gitternetz im GIMP zeichnen kann. Der Konfigurationsdialog soll nur noch die Parameter enthalten, die tatsächlich notwendig sind.



Hinweise

- Tiny Scheme kennt keine `local` Definition für Funktionen, nur das `let`/`let*` für Variablen.
- Eine Linie zeichnen in Script-Fu:


```
; Anfangs- und Endpunkt der Linie definieren x1, y1, x2, y2
(point (cons-array 4 'double))
; Variablen x1, y1, x2, y2 den Anfangs- und Endpunkten der Linie zuordnen
(aset point 0 x1)
(aset point 1 y1)
(aset point 2 x2)
(aset point 3 y2)
; Linie zeichnen
(gimp-paintbrush-default layer 4 point)
```

- Damit Ihr Skript auf einem geöffneten Bild operiert, muss der Parameter SF-IMAGE der ERSTE nach den erforderlichen Parametern sein. GIMP übergibt in diesem Parameter eine Referenz auf das Bild.

```
SF-IMAGE "Image" 0 ; current image
```