Musterlösung XIV. Übungsblatt “Informatik I” WS 95/96

Jürgen Zimmer, Christian Schulte

Aufgabe XIV.1 (Erzwingen von Auswertung) Um die Auswertung aller Knoten, die von `x` erreichbar sind, zu erzwingen, muß man eine gierige Prozedur auf diese applizieren. Das Prädikat `pair?` ist eine solche (gierige) Prozedur.

```
(define erzwinge-tief
  (lambda (x)
    (if (pair? x) (seq (erzwinge-tief (car x))
                       (erzwinge-tief (cdr x)))
        () )))
```

Aufgabe XIV.2 (Delay und force) Man erweitert den Scheme2+3 Interpretierer um die beiden reservierten Symbole `delay` und `force` und führt diese in der globalen Umgebung als Namen von primitiven Prozeduren ein, die faul (delay) bzw. gierig (force) sind. Die entsprechenden Implementierungen nutzen die bereits vorhandenen Konzepte `versprechen` und `erzwinge`.
Zusätzlich braucht man noch die beiden Prädikate `delay?` und `force?`, die das bekannte Prädikat `sonderform?` zur Hilfe nehmen.
Um mit dem Aufbau des Scheme2+3 Interpretierers konsistent zu bleiben, definiert man auch die Selektoren `delay-ausdruck` und `force-ausdruck`. Um die Ausdrücke auswerten zu können muß man natürlich noch die Prozedur `werte-aus` entsprechend anpassen indem man `werte-aus-delay` und `werte-aus-force` appliziert.
Die Änderungen als Fragment :

```
(define reservierte-symbole
  '(quote if then else end lambda := seq new in define delay force))
   .
   .
(define (delay? a)
  (sonderform? a 'delay 1))

(define (force? a)
  (sonderform? a 'force 1))

(define delay-ausdruck second)
(define force-ausdruck second)
```

```
in werte-aus :
                  .
        ((delay? ausdruck)
         (werte-aus-delay ausdruck umgebung)
        ((force? ausdruck)
         (werte-aus-force ausdruck umgebung))
        (else (fehler 'intern "werte-aus"))))

   (define (werte-aus-delay a u)
      (verzoegere (delay-ausdruck a) u))
```

```
(define (werte-aus-force a u)
   (erzwinge (werte-aus (force-ausdruck a) u)))
```

In Scheme 3 kann man die beiden Prozeduren sehr einfach definieren. `Delay` entspricht der Identität und `force` wendet eine gierige primitve Prozedur (z.B. `eq?`) auf das Argument an, so daß dieses ausgewertet wird.

```
(define delay (lambda (x) x))

(define force (lambda (x) (if (eq? x x) x () )))
```

In Scheme 2 existiert der Mechanismus der faulen Auswertung nicht. Die Argumente einer Applikation werden immer ausgewertet. Eine Definition von delay ist deshalb nicht möglich.

Aufgabe XIV.3 (Verzögerte Auswertung und Zuweisung) Ich erkläre die Vorgehensweise an der ersten Teilaufgabe.
`(+ (car ys) (car (cdr ys)))`
In Scheme3 werden die Argumente - falls überhaupt - von vorne nach hinten ausgewertet. Da hier die primitive Prozedur + appliziert wird, die den mode (g g) besitzt, also gierig ist, werden beide Argumente ausgewertet. Die Evaluierung von (car ys) setzt x auf $2 * x = 4$ [(mul 2)] und liefert auch 4. Die Auswertung von `(car (cdr ys))` setzt x auf $2 * x = 2 * 4 = 8$ [(mul 3)] und liefert 8. Die Summe evaluiert also zu $4 + 8 = 12$.
Die anderen Ausdrücke untersucht man mit der gleichen Methode. man muß jedoch darauf achten, daß die Argumente bzw. die Teilausdrücke einer Sequentialisierung nur ausgewertet werden, wenn eine gierige Prozedur appliziert wird. Dies ist z.B. beim ersten Ausdruck der Sequentialisierung in 4. nicht der Fall. Dort wird das Versprechen (mul 4) nicht ausgewertet! Die Ergebnisse auf eine Blick:

1. 16 (x=12)
2. 18 (x=12)
3. 72 (x=48)
4. 18 (x=12)
5. 18 (x=12)
6. 18 (x=12)

Aufgabe XIV.4 (Ströme in Scheme 3) Mittels des Diagonalisierungs-Verfahrens durchläuft man mit i und j den ersten Quadranten (von $N \times N$) entlang den Diagonalen und erzeugt für jedes Paar (i, j) alle Tripel (i, j, k) für die gilt $k < i + j$.

```
(define gen-tripel-strom
  (lambda (i j k)
    (if (< k (+ i j))
        (cons (cons i (cons j k))
              (gen-tripel-strom i j (+ k 1)))
        (if (> i j)
            (gen-tripel-strom i (+ j 1) 1)
```

```
            (gen-tripel-strom (+ i 1) j 1)))))

(define tripel-strom (gen-tripel-strom 1 1 1))
```

Aufgabe XIV.5 (Sequentialisierung) Da Scheme2 gierig ist, werden bei der Applikation von `(begin x y)` automatisch beide Parameter ausgewertet. Es reicht, `begin` als Projektionsprozedur zu definieren:

```
(define begin (lambda (x y) y))
```

In Scheme, wie es im Buch beschrieben ist, wird keine Reihenfolge bei der Auswertung der Parameter festgelegt. In Scheme 2 werden diese der Reihe nach ausgewertet.

Aufgabe XIV.6 (Sequentialisierung) Die Lösung lautet `(define x (cons 5 (cons 2 ())))`, denn dann wird `(cdr x)` zu dem angegebenen Wert evaluiert und `(begin (cdr x) x)` zu `(5 2)`, da hier die Applikation der gierigen Prozedur `eq?` dazu führt, daß alle Versprechen ausgewertet werden.
Um den Unterschied zu beseitigen, muß man im Scheme2+3 Interpretierer folgende Änderung vornehmen:

```
(define (werte-aus-sequentialisierung a u)
  (erzwinge (werte-aus (sequentialisierung-links a) u))
  (erzwinge (werte-aus (sequentialisierung-rechts a) u)))
```