# Übungen zu Organisation und Architektur von Rechnern
# SS 2002
## Serie 2 — Musterlösung

**Aufgabe 4** (8 Punkte)

Der Scheme-Code für die Print-Routinen (`semcd-basic-print.scm`):

```
;; Prints a list of expressions found inside of an Ap{} or a Lam{} structure
(define (print-exprlist exprlist)
        (match exprlist [()               ""]
                        [(head tail ...)   (string-append " "
                                                          (print-expr head)
                                                          (print-exprlist tail))]
                        [_                 "exprlist:?"]))

;; Prints an expression
(define (print-expr expr)
        (match expr [($ Ap cnt exprs)       (string-append "Ap{"
                                                           (number->string cnt)
                                                           "}"
                                                           (print-exprlist exprs))]
                    [($ ApM cnt1 cnt2)      (string-append "Ap{"
                                                           (number->string cnt1)
                                                           ","
                                                           (number->string cnt2)
                                                           "}")]
                    [($ Lam cnt exprs)      (string-append "Lam{"
                                                           (number->string cnt)
                                                           "}"
                                                           (print-exprlist exprs))]
                    [($ Clos binds expr)    (string-append "["
                                                           (print-stack binds)     ;; This is an E-stack!
                                                           " | "
                                                           (print-expr expr)
                                                           "]")]
                    [($ Bind name expr)     (string-append "<"
                                                           name
                                                           " "
                                                           (print-expr expr)
                                                           ">")]
                    [($ Var name)           name]
                    [($ Num val)            (number->string val)]
                    [($ Bool val)           (if val "#t" "#f")]
                    ;;
                    ;; All patterns failed -> Error message
                    [_                      "expr:?"]))

;; Prints a S-, E-, M-, or C-stack
(define (print-stack stack)
        (match stack [()               "nil"]
                     [(head tail ...)   (string-append (print-expr head)
                                                       " : "
                                                       (print-stack tail))]
                     [_                "stack:?"]))

;; Prints a D-stack
(define (print-dump stack)
        (match stack [()        "nil"]
                     [(e c d)   (string-append "( "
                                               (print-stack e)
                                               " , "
                                               (print-stack c)
                                               " , "
                                               (print-dump d)
                                               " )")]
                     [_        "dump:?"]))

;; Prints a state of SEMCD machine.
```

```
;; Returns a single string (good for printing).
(define (print-semcd-s stacks)
        (match stacks [(s e m c d)   (string-append "(  "
                                                    (print-stack s)
                                                    "  ,  "
                                                    (print-stack e)
                                                    "  ,  "
                                                    (print-stack m)
                                                    "  ,  "
                                                    (print-stack c)
                                                    "  ,  "
                                                    (print-dump d)
                                                    "  )")]
                      [_             "semcd:?"]))

;; Prints a state of SEMCD machine.
;; Returns a list of strings (needed for GUI).
(define (print-semcd-l stacks)
        (match stacks [(s e m c d)   (list (print-stack s)
                                           (print-stack e)
                                           (print-stack m)
                                           (print-stack c)
                                           (print-dump d))]
                      [_             "semcd:?"]))
```

**Aufgabe 5+6** (10+14 Punkte)

Bei der Implementierung der SEMCD-Maschine sind folgende Punkte besonders zu beachten:

- Die Funktion `semcd` soll nur *einen* Transformationsschritt ausführen. Transformationssequenzen lassen sich mit Hilfe des Trace-Tools aus Aufgabe 9 erzeugen.
- Es ist darauf zu achten, daß bezüglich der Regeln (4, 6, 8b) die Nebenbedingung ($i > 0$) eingehalten wird. Auf eine explizite Abfrage dieser Nebenbedingung kann verzichtet werden, wenn die Regeln (9, 10) eine höhere Priorität als (4, 6, 8b) erhalten, also die entsprechenden Muster innerhalb des `match`-Konstruktes weiter vorne plaziert werden. Beispiele zum Testen:

  $$\overline{@}^{(2)}\ \overline{@}^{(2)}\ 1\ 2\ 3 \quad , \quad \overline{@}^{(2)}\ \overline{@}^{(2)}\ a\ b\ c \quad .$$

- Falls die Regel (2a) implementiert wird, muß sichergestellt werden, daß die neuen Bindungen in der richtigen Reihenfolge im Environment abgelegt werden. Die Auswertung des Beispiels

  $$\overline{@}^{(4)}\ 3\ 2\ 1\ \lambda^{(4)}\ x\ x\ x\ x$$

  sollte als Ergebnis 3 und nicht etwa 1 liefern.

Die Funktion `semcd` wurde in der Musterlösung so angelegt, daß sie genau dann einen (neuen) Zustand zurückliefert, wenn die Transformation erfolgreich durchgeführt werden konnte. Ist bereits ein Endzustand erreicht oder ein Fehler aufgetreten, ist der Rückgabewert eine Zeichenkette mit einer entsprechenden (Fehler-) Meldung.

Der Scheme-Code für die SEMCD-Maschine (`semcd-basic.scm`):

```
(require-library "match.ss")
(require-library "defstru.ss")

(define-structure (Ap   cnt exprlist))
(define-structure (ApM  cnt1 cnt2))
(define-structure (Lam  cnt exprlist))
(define-structure (Clos bindlist abs))
(define-structure (Bind name expr))
(define-structure (Var  name))
(define-structure (Num  val))
(define-structure (Bool val))

;; Takes the first 'cnt' elements of a list
```

```
(define (take lst cnt)
        (if (= cnt 0)
            ()
            (cons (car lst)
                  (take (cdr lst)
                        (- cnt 1)))))

;; Drops the first 'cnt' elements of a list
(define (drop lst cnt)
        (if (= cnt 0)
            lst
            (drop (cdr lst)
                  (- cnt 1))))

;; Appends a new environment entry in front of 'env-tail'
(define (cons-bind varname expr env-tail)
        (cons (make-Bind varname
                         expr)
              env-tail))

;; Appends new environment entries in front of 'env-tail'
(define (append-env vars exprs env-tail)
        (match (list vars exprs)
               [(()                      ())                 env-tail]
               [((($ Var name) vars ...) (expr exprs ...))   (append-env vars
                                                                         exprs
                                                                         (cons-bind name
                                                                                    expr
                                                                                    env-tail))]))

;; Looks up a binding in an environment
(define (lookup name binds)
        (match binds [()                               (make-Var name)]
                     [(($ Bind name2 expr) binds ...)   (if (string=? name name2)
                                                            expr
                                                            (lookup name binds))]))

;; Decrements the counter of the top-most ApM-constructor on the given M-stack
(define (dec-top m)
        (match m [()                          m]
                 [(($ ApM cnt1 cnt2) m ...)   (cons (make-ApM cnt1
                                                              (- cnt2 1))
                                                    m)]))

;; SEMCD machine:
;; Returns the new stack constellation iff a transformation can be performed,
;; returns a string iff an error occurs or the final state is reached already,
;; returns the unmodified argument otherwise.
(define (semcd stacks)
        (match stacks ;;----------------------------------------------------------------------------
                      ;; Rule 2a, 2b: Abstraction on S and application on M
                      [((($ Clos e2 ($ Lam cnt1 exprs)) s ...)
                        e
                        (($ ApM cnt2 0) m ...)
                        c d)                                (let ((cnt (- cnt1 1)))
                                                                 (if (= cnt1 cnt2)
                                                                     (list (drop s cnt)
                                                                           (append-env (take exprs cnt)
                                                                                       (take s cnt)
                                                                                       e2)
                                                                           m
                                                                           (list (list-ref exprs cnt))
                                                                           (list e c d))
                                                                     (string-append "semcd: "
                                                                                    "Illegal application found!"
                                                                                    " (expected #args: "
                                                                                    (number->string cnt)
                                                                                    ", found #args: "
                                                                                    (number->string (- cnt2 1))
                                                                                    ")")))]
                      ;;----------------------------------------------------------------------------
                      ;; Rule 9, 10: Application on M but no proper item on S
                      ;; This rule must have a higher priority than rules 3-6 and 8!!!
                      [(s e
```

```
  (($ ApM cnt1 0) m ...)
  c d)                                (list (cons (make-Ap cnt1
                                                           (take s cnt1))
                                                  (drop s cnt1))
                                            e
                                            (dec-top m)
                                            c d)]
;;----------------------------------------------------------------------------
;; Rule 3, 4: Variable on C
[(s e m
  (($ Var name) c ...)
  d)                                  (list (cons (lookup name e) s)
                                            e
                                            (dec-top m)
                                            c d)]
;;----------------------------------------------------------------------------
;; Rule 5, 6: Abstraction on C
[(s e m
  (($ Lam cnt exprs) c ...)
  d)                                  (list (cons (make-Clos e
                                                             (make-Lam cnt
                                                                       exprs))
                                                  s)
                                            e
                                            (dec-top m)
                                            c d)]
;;----------------------------------------------------------------------------
;; Rule 7: Application on C
[(s e m
  (($ Ap cnt exprs) c ...)
  d)                                  (list s e
                                            (cons (make-ApM cnt
                                                            cnt)
                                                  m)
                                            (append exprs c)
                                            d)]
;;----------------------------------------------------------------------------
;; Rule 8a, 8b: Atom on C
[(s e m
  (atom c ...)
  d)                                  (list (cons atom s)
                                            e
                                            (dec-top m)
                                            c d)]
;;----------------------------------------------------------------------------
;; Rule 0: Return from sub-evaluation
[(s e m ()
  (e2 c2 d2))                         (list s e2 m c2 d2)]
;;----------------------------------------------------------------------------
;; Final state reached
[(s e () () ())                       "semcd: Final state reached!"]
;;----------------------------------------------------------------------------
;; All patterns failed -> Do nothing
[_                                    stacks]))
```

Der Scheme-Code für die Testumgebung (`semcd-basic-exams.scm`):

```
(load "semcd-basic.scm")
(load "semcd-basic-print.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Tracing

;; Creates a list containing the numbers of the intervall ['from', 'to'] in ascending order.
(define (from-to from to) ...)

;; Applies 'semcd' until a final state is reached or all states corresponding with the numbers given
;; in the list 'cnt-list' have been generated.
;; Returns a list of pairs with each pair consisting of a state number and a state (which has been
;; applied to 'print-semcd').
(define (trace-semcd print-semcd cnt-list) (lambda (stacks) ...))

```

```
;; Applies 'trace-semcd' to an expression
(define (trace-semcd-expr print-semcd cnt-list) (lambda (expr)
        ((trace-semcd print-semcd cnt-list)
                      (list () () () (list expr) ()))))          ;; Build initial state

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Examples

;;-----------------------------------------------------------------------------
;; Exercise 6, 1st example
(define exer6-1  (make-Ap 2 (list (make-Num 15)
                                  (make-Lam 2 (list (make-Var "x")
                                                    (make-Var "x"))))))

;;-----------------------------------------------------------------------------
;; Exercise 6, 2nd example
(define exer6-2  (make-Ap 2 (list (make-Num 34) exer6-1)))

;;-----------------------------------------------------------------------------
;; Exercise 6, 3rd example
(define exer6-3  (make-Ap 3 (list (make-Var "b") (make-Var "a")
                                  (make-Lam 3 (list (make-Var "x") (make-Var "y")
                                                    (make-Var "x"))))))

;;-----------------------------------------------------------------------------
;; Exercise 7, this version does not work
(define exer7-1  (make-Ap 2 (list (make-Num 7)
                                  (make-Ap 2 (list (make-Num 11)
                                                   (make-Lam 3 (list (make-Var "x") (make-Var "y")
                                                                     (make-Var "y"))))))))

;;-----------------------------------------------------------------------------
;; Exercise 7, this version works
(define exer7-2  (make-Ap 2 (list (make-Num 7)
                                  (make-Ap 2 (list (make-Num 11)
                                                   (make-Lam 2 (list (make-Var "x")
                                                                     (make-Lam 2 (list (make-Var "y")
                                                                                       (make-Var "y"))))))))))

;;-----------------------------------------------------------------------------
;; Exercise 7, this version also works
(define exer7-3  (make-Ap 3 (list (make-Num 7) (make-Num 11)
                                  (make-Lam 3 (list (make-Var "x") (make-Var "y")
                                                    (make-Var "y"))))))

;;-----------------------------------------------------------------------------
;; S combinator (used in exercise 8)
(define s        (make-Lam 2 (list (make-Var "x")
                    (make-Lam 2 (list (make-Var "y")
                       (make-Lam 2 (list (make-Var "z")
                                         (make-Ap 2 (list (make-Ap 2 (list (make-Var "z")
                                                                           (make-Var "y")))
                                                          (make-Ap 2 (list (make-Var "z")
                                                                           (make-Var "x"))))))))))))

;;-----------------------------------------------------------------------------
;; K combinator (used in exercise 8)
(define k        (make-Lam 2 (list (make-Var "x")
                                   (make-Lam 2 (list (make-Var "y")
                                                     (make-Var "x"))))))

;;-----------------------------------------------------------------------------
;; Exercise 8: ((SK)K)
(define exer8-1  (make-Ap 2 (list k
                                  (make-Lam 2 (list (make-Var "k")
                                                    (make-Ap 2 (list (make-Var "k")
                                                                     (make-Ap 2 (list (make-Var "k")
                                                                                      s)))))))))
               ; (make-Ap 2 (list k (make-Ap 2 (list k s)))))

;;-----------------------------------------------------------------------------
;; Exercise 8: (((SK)K) x) evaluates to x  ->  ((SK)K) = lam{2} x x
(define exer8-2  (make-Ap 2 (list (make-Var "x") exer8-1)))
```

```

;;----------------------------------------------------------------------------
;; Tests whether environment is build in correct order -> Should evaluate to 3
(define test1    (make-Ap 4 (list (make-Num 3) (make-Num 2) (make-Num 1)
                                  (make-Lam 4 (list (make-Var "x") (make-Var "x") (make-Var "x")
                                                    (make-Var "x"))))))

;;----------------------------------------------------------------------------
;; Tests whether the ApM-Rules have the correct priority
(define test2-1  (make-Ap 2 (list (make-Ap 2 (list (make-Num 3)
                                                   (make-Num 2)))
                                  (make-Num 1))))
(define test2-2  (make-Ap 2 (list (make-Ap 2 (list (make-Var "a")
                                                   (make-Var "b")))
                                  (make-Var "c"))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Testing

(map (trace-semcd-expr print-semcd-s (from-to 0 5000))         ;; Show all states
     (list exer6-1 exer6-2 exer6-3 exer7-1 exer7-2 exer7-3
           test1 test2-1 test2-2))

(map (trace-semcd-expr print-semcd-s (list 5000))              ;; Show final state only
     (list exer8-1 exer8-2))

(stepper2 10 180                                               ;; Use teachpack 'semcd-gui.ss'
          ((trace-semcd-expr print-semcd-l (from-to 0 5000))
                             exer8-2))
```

**Aufgabe 7** (4 Punkte)

Die SEMCD-Maschine für Scheme-Ausdrücke kann die Applikation einer Abstraktion nur dann auswerten, wenn Applikation und Abstraktion die gleiche Stelligkeit besitzen.

In dem vorliegenden Beispiel erwartet die Abstraktion zwei Argumente ($x$, $y$). Die innerste Applikation führt jedoch nur ein Argument zu (11), das zweite Argument (7) wird mit Hilfe einer weiteren Applikation bereitgestellt. Die innerste Applikation ist also eine sogenannte Partielle Funktionsanwendung und in Scheme nicht erlaubt.

Es gibt zwei Möglichkeiten diesen Term so zu modifizieren, daß er vollständig ausgewertet werden kann, ohne die intendierte Funktionalität zu verändern. Entweder wird die Abstraktion so umgeschrieben, daß die Argumente schrittweise zugeführt werden können:

$$\overline{@}^{(2)}\ 7\ \overline{@}^{(2)}\ 11\ \lambda^{(2)}\ x\ \lambda^{(2)}\ y\ y \qquad ,$$

oder die beiden Applikationen werden zusammengefaßt:

$$\overline{@}^{(3)}\ 7\ 11\ \lambda^{(3)}\ x\ y\ y \qquad .$$

**Aufgabe 8** (4 Punkte)

a) Der innerste Lambda-Term

$$\lambda^{(2)}\ z\ \overline{@}^{(2)}\ \overline{@}^{(2)}\ z\ y\ \overline{@}^{(2)}\ z\ x$$

ist die einzige Abstraktion, die in keine zugehörige Applikation eingebettet ist. D. h. der gegebene Ausdruck wertet sich zu einer Funktion aus, die ein Argument ($z$) erwartet. Die Berechnung mit Hilfe der SEMCD-Maschine liefert daher eine Closure, die den obigen Lambda-Term und ein Environment mit den Werten für die relativ freien Variablen $x$ und $y$ enthält:

```
[ <y ...> : <x ...> : <k ...> : nil  Lam{2} z Ap{2} Ap{2} z y Ap{2} z x ]    .
```

b) Die SEMCD-Maschine wertet den Rumpf einer Abstraktion/Closure erst dann aus, wenn eine Applikation mit entsprechender Stelligkeit vorhanden ist. Um die Funktionalität des gegebenen Terms zu ermitteln, muß er also in eine geeignete Applikation eingebettet werden.

Da die SEMCD-Maschine mit freien Variablen umgehen kann (Scheme kann das nicht!), kann man den Term einfach auf eine solche (z. Bsp. $x$) anwenden

$$\overline{@}^{(2)}\ x\ \boxed{\overline{@}^{(2)}\ \lambda^{(2)}\ x\ \lambda^{(2)}\ y\ x\ \lambda^{(2)}\ k\ \overline{@}^{(2)}\ k\ \overline{@}^{(2)}\ k\ \lambda^{(2)}\ x\ \lambda^{(2)}\ y\ \lambda^{(2)}\ z\ \overline{@}^{(2)}\ \overline{@}^{(2)}\ z\ y\ \overline{@}^{(2)}\ z\ x}$$

und erhält als Resultat wieder die freie Variable selbst. Offenbar entspricht der gegebene Ausdruck also der Identitätsfunktion.

Daraus läßt sich ein allgemeines Prinzip zur Berechnung von Funktionen mit Hilfe der SEMCD-Maschine ableiten. Sei $T$ der zu berechnende (Funktions-) Term und $P$ ein initial leeres Prefix. Dann:

1) Berechne den Ausdruck $T$ mit Hilfe der SEMCD-Maschine zu $T'$.
2) Falls $T'$ eine $n$-stellige Closure ist, wähle $n$ Bezeichner $x_1, \ldots, x_n$, setze

$$P := \lambda^{(n)} x_1 \ldots x_n\ P \qquad , \qquad T := \overline{@}^{(n)} x_1 \ldots x_n\ T' \qquad ,$$

und gehe wieder zu 1). Andernfalls stellt $PT'$ das gesuchte Ergebnis dar.

Für das obige Beispiel ergibt sich nach diesem Schema der Term $\lambda^{(2)} x\ x$ — die Identitätsfunktion.