Lösung Aufgabe A


```
(define bsp-matrix (list '(1 2 3) '(4 5 6) '(7 8 9)))

;; eigene Version von list-ref, um die Indizierung bei 1 beginnen zu lassen
(define (list-ref i liste)
  (cond ((or (< i 1) (null? liste)) #f)
        ((= i 1) (car liste))
        (else (list-ref (- i 1) (cdr liste)))))

; Beispiele
(list-ref 0 '(1 2 3))
(list-ref 2 '(1 2 3))
(list-ref 4 '(1 2 3))

; matrix-col ist eine Anwendung von list-ref auf jede Zeile
(define (matrix-col j matrix)
  (map (lambda (x) (list-ref j x)) matrix))

(matrix-col 3 bsp-matrix)

;; scalar-prod berechnet das Skalarprodukt zweier Vektoren
(define (scalar-prod row col)
  (cond
    ((and (null? row) (null? col)) 0)
    ((or (null? row) (null? col)) #f)
    (else (+ (* (car row) (car col)) (scalar-prod (cdr row) (cdr col))))))

;; kuerzer waere:
;(define (skalar-prod v1 v2)
;  (apply + (map * v1 v2)))

(scalar-prod '(4 5 6) '(3 6 9))

; Multiplikation eines Vektors mit einer Matrix
(define (line-mul line matrix)
  (define (iter erg-liste j)
    (if (= 0 j) erg-liste
        (iter (cons (scalar-prod line (matrix-col j matrix)) erg-liste) (- j 1))))
  (iter '() (length matrix)))


(line-mul '(4 5 6) bsp-matrix)

; nun die eigentliche Matrix-Multiplikation
(define (matrix-mul mat-a mat-b)
  (if (null? mat-a) '()
      (cons (line-mul (car mat-a) mat-b) (matrix-mul (cdr mat-a) mat-b))))

(matrix-mul bsp-matrix bsp-matrix)
```

Lösung Aufgabe B


```
(define bsp-matrix (list '(1 2 3) '(4 5 6) '(7 8 9)))

; matrix-ref extrahiert das j-te Element der i-ten  Spalte
; der Matrix  matrix.
; Verwendet wird eine eigene Version von list-ref
(define (matrix-ref i j matrix)
  (define (list-ref i liste)
    (if (= i 1) (car liste)
        (list-ref (- i 1) (cdr liste))))
  (list-ref j (list-ref i matrix)))


(matrix-ref 2 3 bsp-matrix)

;; Fingeruebungen zur transpose-Proezedur
(map car bsp-matrix)
(map cdr bsp-matrix)
(map cdr (cdr bsp-matrix))
(cdr (car bsp-matrix))

; hier die transpose-Prozedur
(define (transpose matrix)
  (if (null? matrix) '()
      (cons (map car matrix)
            (map cons (cdr (car matrix)) (transpose (map cdr (cdr matrix)))))))

(transpose bsp-matrix)

; Die Gleichheit von Strukturen prueft man am einfachsten mit equal?
(define (symmetric? matrix)
  (equal? matrix (transpose matrix)))

(symmetric? bsp-matrix)

(symmetric? (list '(1 2 3) '(2 4 5) '(3 5 6)))
```

Lösung Aufgabe C


```
(define bsp-matrix (list '(1 2 3) '(4 5 6) '(7 8 9)))

;;;
;;; matrix-ref ist im Teachpack
;;;
(define (matrix-ref i j matrix)
  (define (list-ref i liste)
    (if (= i 1) (car liste)
        (list-ref (- i 1) (cdr liste))))
  (list-ref j (list-ref i matrix)))
;;;
;;;  Ende des Teachpacks
;;;

; Erzeugen einer Kopie einer Liste ohne das j-te Element
(define (delete i liste)
  (cond ((or (null? liste) (< i 1)) #f)
        ((= i 1) (cdr liste))
        (else (cons (car liste) (delete (- i 1)(cdr liste))))))

(delete 2 '(1 2 3 4))

; Entfernen der i-ten Zeile und j-ten Spalte aus einer Matrix
(define (delete-ij i j matrix)
  (delete i
         (map (lambda (x) (delete j x)) matrix)))

(delete-ij 2 3 bsp-matrix)

; Berechnung der Determinante
(define (det matrix)
  (define (iter j n f sum)
    (cond ((= n 1) (caar matrix))
          ((> j n) sum)
          (else (iter (+ j 1) n (- f)
                      (+ sum (* (matrix-ref 1 j matrix)
                                (det (delete-ij 1 j matrix))
                                f))))))
  (iter 1 (length matrix) 1 0))

;Beispiele
(det bsp-matrix)
(det (list '(1 0 0 0) '(0 1 0 0) '(0 0  1 0) '(0 0 0  1)))
(det (list  '(0 1 0 0) '(1 0 0 0) '(0 0  1 0) '(0 0 0  1)))
(det (list '(1 2 3) '(2 2 3) '(4 1 4)))
```

Lösung Aufgabe D


```
;; det ist im Teach-Pack
;;
(define (matrix-ref i j matrix)
  (define (list-ref i liste)
    (if (= i 1) (car liste)
        (list-ref (- i 1) (cdr liste))))
  (list-ref j (list-ref i matrix)))

(define (delete i liste)
  (cond ((or (null? liste) (< i 1)) #f)
        ((= i 1) (cdr liste))
        (else (cons (car liste) (delete (- i 1)(cdr liste))))))

(define (delete-ij i j matrix)
  (delete i
          (map (lambda (x) (delete j x)) matrix)))

(define (det matrix)
  (define (iter j n f sum)
    (cond ((= n 1) (caar matrix))
          ((> j n) sum)
          (else (iter (+ j 1) n (- f)
                      (+ sum (* (matrix-ref 1 j matrix)
                                (det (delete-ij 1 j matrix))
                                f))))))
  (iter 1 (length matrix) 1 0))
;;;
;;; Ende des Teachpacks
```

;;ab hier beginnt Aufgabe D

```
;;
(define bsp-matrix (list '(1 2 3) '(4 5 6) '(7 8 9)))
(define bsp-matrix-cra (list '(1 0 2) '(-3 4 6 ) '(-1 -2 3)))

; list-replace ersetzt das j-te Element einer Liste durch elem
(define (list-replace liste elem j)
  (cond ((or (null? liste)(< j 1)) #f)
        ((= j 1) (cons elem (cdr liste)))
        (else (cons (car liste) (list-replace (cdr liste) elem (- j 1))))))

(list-replace '(1 2 3 4 5) 10 2)

; matrix-replace-col ersetzt die j-te Spalte der Matrix durch col
(define (matrix-replace-col matrix col j)
  (map (lambda (x e)
         (list-replace x e j)) matrix col))

(matrix-replace-col bsp-matrix '(1 1 1) 3)

; hier die eigentliche Prozedur cramer
(define (cramer A b)
  (let ((det-A (det A)))
    (define (iter erg i)
      (if (= i 0) erg
          (iter (cons (/ (det (matrix-replace-col A b i)) det-A) erg) (- i 1))))
    (iter '() (length A))))

(cramer bsp-matrix-cra '(6 30 8))
```

Lösung Aufgabe E


```
(define (chebyshev n)
  (cond ((= n 0) 1)
        ((= n 1) 'x)
        (else (list '+ (list '* 2 'x (chebyshev (- n 1))) (chebyshev (- n 2))))))

(chebyshev 0)
(chebyshev 1)
(chebyshev 2)
(chebyshev 3)

(define (legendre n)
  (cond ((= n 0) 1)
        ((= n 1) 'x)
        (else (list '/
                    (list '-
                          (list '*
                                (- (* 2 n) 1)
                                'x
                                (legendre (- n 1)))
                          (list '*
                                (- n 1)
                                (legendre (- n 2))))
                    n))))

(legendre 0)
(legendre 1)
(legendre 2)
(legendre 3)

;; Der zweite Teil der Aufgabenstellung zeigt die Moeglichkeiten
;; von Scheme -  Auswertung der so erzeugten Ausdruecke an einer
;; Stelle a am Beispiel der Legendre-Polynome

; Hier die kompakte Version
; (define (legendre-wert n a)
;   (eval (list (list 'lambda '(x) (legendre n)) a)))
;
;(legendre-wert 0 3)
;(legendre-wert 1 3)
;(legendre-wert 2 3)
;(legendre-wert 3 3)

; Hier die etwas einfachere Version:

(define (leg-w-help n)
 (eval (list 'lambda '(x) (legendre n))))

(leg-w-help 2)
((leg-w-help 2) 3)

(define (legendre-wert n a)
  ((leg-w-help n) a))

(legendre-wert 0 3)
(legendre-wert 1 3)
(legendre-wert 2 3)
(legendre-wert 3 3)
```

Lösung Aufgabe F


```
;; zunaechst eine Version der ggT-Funktion, die einen rekursiven Prozess erzeugt.
;; Als arithmetische Operationen sind nur die folgenden drei erlaubt:
;; links-shift verdoppelt das Argument, rechts-shift halbiert und rundet das Argument
;;  minus subtrahiert das zweite Argument vom ersten. Eine mögliche Implementation der Funktionen wäre

(define (links-shift x) (* x 2))
(define (rechts-shift x) (quotient x 2))
(define (minus x y) (- x y))

;; Die Definition der ggT-Prozedur folgt direkt der Definition

(define (ggT m n)
  (cond ((= m n) m)
        ((and (even? m) (even? n)) (links-shift (ggT (rechts-shift m)  (rechts-shift n))))
        ((even? m) (ggT (rechts-shift m) n))
        ((even? n) (ggT m (rechts-shift n)))
        ((> n m) (ggT m (rechts-shift (minus n m))))
        (else (ggT (rechts-shift (minus m n)) n))))

(ggT 64 1)
(ggT 1 64)
(ggT 64 32)
(ggT 123456 789)

;; nun die zweite Version der ggT-Prozedur, die einen iterativen Prozess erzeugt.
;; Hier sind zwei weitere Funktionen erlaubt:
;;  erhoehe erhoeht das Argument um 1
;;   vermindere vermindert das Argument um 1
;; Eine mögliche Implementation der Funktionen wäre

(define (erhoehe n) (+ n 1))
(define (vermindere n) (- n 1))

;;  Die lokale Prozedur ggT-iter hat einen weiteren Parameter r, mit dem gezaehlt wird, wie haeufig das Ergebnis
;; mit 2 multipliziert werden muss. Dies realisiert eine weitere Hilfsfunktion links-shift-n, die den ersten
;;  Parameter so oft nach links schiebt, wie der zweite Parameter angibt.

(define (ggT m n)
  (define (links-shift-n x r)
    (if (= 0 r) x
        (links-shift-n (links-shift x) (vermindere r))))

  (define (ggT-iter m n r)
    (cond ((= n m) (links-shift-n m r))
          ((and (even? m) (even? n)) (ggT-iter (rechts-shift m) (rechts-shift n) (erhoehe r)))
          ((even? m) (ggT-iter (rechts-shift m) n r))
          ((even? n) (ggT-iter m (rechts-shift n) r))
          ((> n m) (ggT-iter m   (rechts-shift (minus n m))  r))
          (else (ggT-iter (rechts-shift (minus m n))
                          m
                          r))))
  (ggT-iter m n 0))

(ggT 64 1)
(ggT 1 64)
(ggT 64 32)
(ggT 123456 789)
```

Lösung Aufgabe G


```
;; Zunächst ist die Abfage auf die leere Menge in dieser Darstellung
;; natürlich gleich der Abfrage auf die leere Liste

(define empty? null?)

;; Die Implementation der Prozedur potenzmenge folgt genau der rekursiven
;  Definition der Potenzmenge in der Aufgabenstellung. Man entfernt das
;; erste Element der Liste und bildet rekursiv die Potenzmenge der Restliste.
;; Diese verbindet man über append mit der Liste, die man erhält, indem
;; das erste Element der Ausgangsliste in jede Liste (mit cons) der
;; Potenzmenge der Restliste einfügt.

(define (potenzmenge m)
 (if (empty? m) '(())
    (let ((pm1 (potenzmenge (cdr m))))
      (append pm1
            (map (lambda (x)
                  (cons (car m) x)) pm1)))))

(potenzmenge '())
(potenzmenge '(1))
(potenzmenge '(2 3))
(potenzmenge '(4 5 6))

;; Die Prozedur potenzmnge-n arbeitet im Prinzip wie die Prozedur potenzmenge.
;; Es gibt aber ein weiteres Abbruchkriterium. Man entfernt das
;; erste Element der Liste und bildet rekursiv die n-elementigen Teilmengen der
;; Restliste. Diese verbindet man über append mit der Liste, die man erhält, indem
;; das erste Element der Ausgangsliste in jede Liste (mit cons) der (n-1)-elementigen
;; Teilmengene der Restliste einfügt.

;; leere

(define (potenzmenge-n m n)
  (cond ((= n 0)'(()) )
       ((empty? m) '())
       (else
        (append (potenzmenge-n (cdr m) n)
              (map (lambda (x) (cons (car m) x))
                  (potenzmenge-n (cdr m) (- n 1)))))))

(potenzmenge-n '(2 3 5) 0)
(potenzmenge-n '(2 3 5) 1)
(potenzmenge-n '(2 3 5) 2)
(potenzmenge-n '(2 3 5) 3)
(potenzmenge-n '(2 3 5) 4)
```