Funktionale Sprachen: LISP

(mehrdim) Arrays:

(make-array Dimensionsliste)

- erzeugt (mehrdim.) Matrix entsprechend der Dimensionsliste
- Anzahl der Einträge in Dim.Liste bestimmt Dimension der Matrix, Wert der Einträge bestimmt Größe je Dimension
- Beispiel:

  (make-array ´(3 4))

  erzeugt 2-dimensionale Matrix, Dimension 1: 3, Dimension 2: 4
- viele optionale Parameter

Funktionale Sprachen: LISP

optionale Parameter bei Arrays:

- :element-type Wert spezifiziert Typ jeden Elements, voreingestellt: t (bel. Typ)

```
(make-array 5 :element-type 'single-float)
```

- :initial-element Startwert, wird jedem Arrayelement zugewiesen
- :initial-contents pro Element ein Startwert

```
(make-array '(4  2  3) :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (3 1 2))
              ((g h i) (2 3 1))
              ((j k l) (0 0 0))))
```

- :adjustable Voreinstellung NIL, T: Arraygröße dynamisch änderbar

Funktionale Sprachen: LISP

optionale Parameter bei Arrays:

- :displaced-to M :displaced-index-offset wert

  Matrix ist wertbezogen Teilmenge der Matrix M, Start der Werte ab Index wert

```
(setq a (make-array '(4 3)))
(setq b (make-array 8 :displaced-to a :displaced-index-offset 2))
```

- Default-Wert für :displaced-to NIL
- :displaced-index-offset nur in Verbindung mit :displaced-to
- Größe von M (Multiplikation der Dimensionen) >= Größe der darauf bezogenen Matrix + offset

```
(aref b 0) == (aref a 0 2)
(aref b 1) == (aref a 1 0)
(aref b 2) == (aref a 1 1)
(aref b 3) == (aref a 1 2)
(aref b 4) == (aref a 2 0)
(aref b 5) == (aref a 2 1)
(aref b 6) == (aref a 2 2)
(aref b 7) == (aref a 3 0)
```

Funktionale Sprachen: LISP

Zugriff auf Matrizenelemente:

- (aref M index1 [index2 …])
- Indexnummerierung ab 0
- Beispiel:

```
(setf M (make-array ´(3 3)  :initial-element 1)
(setf (aref M 0 2) 3)
→ #2A((1 1 3) (1 1 1) (1 1 1))
```

$$\begin{pmatrix} 1 & 1 & 3 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

- erster Index: „Zeile“, zweiter Index: „Spalte“
- Eindimensionale Matrix:

```
(setf v (make-array 3))
(setf (aref v 0) 1)
(setf (aref v 1) 2)
(setf (aref v 2) 3)
→ #(1  2  3)
```

Funktionale Sprachen: LISP

Eindimensionale Matrix = Vektor:

- zusätzl. Option :fill-pointer (nicht zwingend)
- mögliche Werte:
  - (Voreinstellung) NIL: keine Füllstandsanzeige
  - t: es gibt Füllstandsanzeiger, Wert = Größe des Vektors
  - Integer-Wert:: zwischen 0 und Länge des Vektors
- Vektor mit fill-pointer: nach und nach füllbar, fill-pointer zeigt immer Position

des letzten „aktiven“ Elements im Vektor, alle Funktionen auf Vektoren arbeiten nur auf den aktiven Elementen, Ausnahme: aref

- (array-has-fill-pointer-p array) liefert t, wenn array fill-pointer hat
- (fill-pointer vector) liefert den fill-pointer, wenn vorhanden, oder error

Funktionale Sprachen: LISP

Eindimensionale Matrix = Vektor:

- (vector-push new-el vector)nur für Vektoren mit fill-pointer, neues Element wird hinzugefügt, fill-pointer erhöht, wird Anz. Elemente > Größe Vektor: NIL
- (vector-push-extend new-el vector) wie oben, bei adjustable Vektoren dyn. Vergrößerung, wenn max. Vektorgröße erreicht
- (vector-pop vector) für Vektoren mit fill-pointer, fill-pointer wird um 1 reduziert, altes letztes aktives Element wird zurückgeliefert (falls fill-pointer 0 war: error)

Funktionale Sprachen: LISP

Beispiel eindimensionale Matrix :

```
(setf vec1 (make-array 10 :initial-element 1 :fill-pointer 5))
 →  #(1 1 1 1 1)
(array-has-fill-pointer-p vec1)        →  T
(fill-pointer  vec1)                   →  5
(vector-push 4 vec1)                   →  5 (Index!)
(fill-pointer vec1)                    →  6
(print vec1)                           →  #(1 1 1 1 1 4)
(vector-pop  vec1)                     →  4
(print vec1)                           →  #(1 1 1 1 1)
```

Funktionale Sprachen: LISP

Arbeiten mit Matrizen :

- (array-element-type array) liefert Typ der Obekte, die in array gespeichert werden können
- (array-rank array) → Anzahl Dimensionen
- (array-dimension array nr) → Größe der nr.ten Dimension (Index ab 0)
- (array-dimensions array) → Dimensionen als Liste
- (array-total-size array) → Multiplikation der Dimensionen

- (array-in-bound array subscr) → subscr: Integerwerte, Anzahl = Dimensionen des arrays, T wenn subscr gültige Indexangaben
- (array-row-major-index array subscr) → liefert Index des Elements, welches durch subscr beschrieben, bei zeilen-weiser Ordnung
- (adjustable-array-p array) → T, wenn array adjustable, sonst NIL

Funktionale Sprachen: LISP

Beispiele :

```
(setf m (make-array '(3 4) :initial-element 1)
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

(array-rank m)	→	2
(array-dimension m 0)	→	3
(array-dimensions m)	→	(3 4)
(array-total-size array)	→	12
(array-in-bounds-p m 1 2)	→	T
(array-in-bounds-p m 3 2)	→	NIL
(array-row-major-index m 2 2)	→	10

Funktionale Sprachen: LISP

Multiplikation Matrix * Vektor :

```
(defun matmulvec (m v)
    (setf dims (array-dimensions m))
    (setf dimv (array-dimensions v))
    (setf ergm (make-array (first dims) :initial-element 0))
    (if (not (= (second dims) (first dimv)))
        (error "Dimensionen passen nicht!")
            (dotimes (i (first dims) ergm)
                (setf (aref ergm i)
                    (do(( j 0 (+ j 1))
                        ( sum 0) )
                        ( (= j (second dims)) sum)
                        (setf sum (+ sum (* (aref m i j) (aref v j))))
                    )
                )
            ); ende dotimes
        )
)
```

Funktionale Sprachen: LISP

Ändern der Dimension einer Matrix :

- (adjust-array m neue-Dimliste)

  ändert die Dimension von m entsprechend der neuen Dimensionsliste

  m muss mit Option :adjustable erzeugt sein

  Anzahl Dimensionen in alter u. neuer Dimliste muss übereinstimmen

- optionale Parameter:
  - :element-type muss der gleiche der Originalmatrix sein
  - :initial-element nur die Elemente außerhalb der Indexgrenzen der alten Matrix werden hiermit gesetzt
  - :initial-contents überschreibt alten Inhalt komplett
  - :displaced-to , :displaced-index-offset überschreibt ebenfalls alten Inhalt
  - :fill-pointer m musste schon fill-pointer haben, setzt ihn neu

Funktionale Sprachen: LISP

Ändern der Dimension einer Matrix :

- Beispiel:

```
m (4x4):
#2A( (a  b  c  d)
     (e  f  g  h)
     (i  j  k  l)
     (m n  o  p))

(adjust-array m ´(3 5) :initial-element ´z)  liefert

#2A( (a  b  c  d  z)
     (e  f  g  h  z)
     (i  j  k  l  z))
```

Funktionale Sprachen: LISP

Zusammenspiel adjust-array und displaced-to (1) :

Matrix M soll bzgl. Dimension geändert werden

1. M ist weder vor noch hinterher displaced-to: Dimensionen werden geändert, Inhalte entsprechend angepasst

2. M ist vorher nicht displaced-to, hinterher displaced-to C: vorherige Werte verschwinden, M enthält nun entspr. Index-Offset Werte aus C

3. M ist vorher displaced-to B, hinterher displaced-to C: vorherige Werte verschwinden, M enthält nun entspr. Index-Offset Werte aus C

4. M ist vorher displaced-to B, aber nicht displaced-to hinterher: M übernimmt Werte aus B, wo neue Dimension kleiner oder gleich ist, füllt mit NIL bzw. initial-element auf, wo Dimension größer wurde

Funktionale Sprachen: LISP

Zusammenspiel adjust-array und displaced-to (2):

Matrix B ist displaced-to Matrix A, A wird bzgl. Dimensionen geändert:
B ist weiterhin displaced-to geänderter Matrix A → evtl. Wertveränderungen in B, A muss aber weiterhin genügend Elemente (für B) haben
Beispiel:

```
(setf m1 (make-array ´(4 4) :adjustable T
    :initial-contents ´((1 2 3 4) (5 6 7 8) (9 10 11 12) (13 14 15 16))))
(setf m2 (make-array ´(2 3) :displaced-to m1  :displaced-index-offset 1))
(print m1)  →  #2A((1 2 3 4) (5 6 7 8) (9 10 11 12) (13 14 15 16))
(print m2)  →  #2A((2 3 4) (5 6 7))

(adjust-array m1 ´(3 5) :initial-element 0)
(print m1)  →  #2A((1 2 3 4  0) (5 6 7 8 0) (9 10 11 12 0))
(print m2)  →  #2A((2 3 4) (0 5 6))
```