# Sprachspezifikation der „Matrix And Vector Language“

FG Eingebettete Systeme und ihre Anwendungen

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Version 2023.1

# Inhaltsverzeichnis

# 1 Einführung

Die „Matrix and Vector Language“, kurz auch MAVL, ist eine Programmiersprache. Ihr Zweck liegt hauptsächlich darin, Konzepte zu vermitteln, die für den Compilerbau wichtig sind. Die MAVL-Syntax orientiert sich stark an gängigen Sprachen wie JAVA, C/C++ und Scala. Im Gegensatz zu diesen Sprachen bietet MAVL eine native Unterstützung von Matrix- und Vektoroperationen. MAVL ist eine statisch typisierte, imperative Programmiersprache mit strengen Typregeln, die alle Operationen betreffen.

Die folgenden Abschnitte spezifizieren verschiedene Konzepte der Sprache. In jedem Abschnitt wird dabei auf die Syntax und die Semantik eingegangen. Abschließend finden Sie Beispiele, die die Verwendung der Sprachkonzepte verdeutlichen.

# 2 Lexikalische Regeln

Lexikalische Regeln beschreiben, wie der Programmtext in eine Reihe von atomaren grammatikalschen Bausteinen, sogenannten *Tokens*, zerlegt wird. Diese Spezifikation schreibt nicht vor, wie die lexikalische Analyse einer MAVL-Implementierung intern organisiert sein muss, jedoch müssen gewisse Regeln eingehalten werden, damit jede Implementierung MAVL-Programme gleich interpretiert.

Allgemein besteht ein MAVL-Programm aus ASCII-kodiertem Text.

## 2.1 Whitespace

MAVL ist nicht Whitespace-sensitiv: Leerzeichen, Tabs und Zeilenumbrüche werden in MAVL größtenteils ignoriert und können vom Programmierer fast beliebig verwendet werden. Jedoch werden die meisten Token durch Whitespace beendet, somit kann an vielen Stellen das Vorhandensein von Whitespace (nicht jedoch die Art und Anzahl) beeinflussen, wie der Programmtext in Tokens zerlegt wird. Das folgende Beispiel demonstriert das für Bezeichner und Schlüsselwörter:

Listing 1: Whitespace trennt Tokens

```
val int x = 1; // Neue Wertdefinition mit Namen x
valintx = 1; // Zuweisung der existierenden Variable valintx
```

## 2.2 Bezeichner

Bezeichner müssen aus mindestens einem Zeichen bestehen und können beliebig lang sein. Sie beginnen mit einem Groß- oder Kleinbuchstaben, die folgenden Zeichen können Buchstaben, Zahlen und Unterstriche (`_`) sein. Groß- und Kleinschreibung ist dabei relevant: `ab`, `Ab`, `aB` und `AB` sind allesamt verschiedene Bezeichner.

Die folgenden Zeichenketten sind reservierte Schlüsselwörter und können nicht als Bezeichner verwendet werden:

```
if else for foreach switch case default return function record val var true false
                    int float bool matrix vector string void
```

## 2.3 Atomare Literale

Integer-Literale bestehen aus beliebig vielen dezimalen Ziffern, führende Nullen sind jedoch nicht erlaubt. Ein eventuelles Vorzeichen ist nicht Teil des Literals sondern ein separater Operator.

Float-Literale bestehen aus beliebig vielen dezimalen Ziffern, gefolgt von einem Punkt, gefolgt von weiteren (wieder beliebig vielen) dezimalen Ziffern. Die erste dieser beiden Zifferfolgen darf keine führenden Nullen haben, außer die Ziffer null steht alleine, z.B. `0.123` oder ähnliches. Beide Zifferfolgen müssen nicht-leer sein, es muss also mindestens eine Ziffer vor und eine Ziffer nach dem Punkt vorhanden sein.

Boolesche Literale sind die Schlüsselwörter `true` und `false`.

String-Literale beginnen und enden mit Anführungsstrichen und können dazwischen beliebige ASCII-Symbole enthalten, mit Ausnahme von Anführungsstrichen und Zeilenumbrüchen. Die folgenden fünf Escape-Sequenzen werden unterstützt: (`\"` → "), (`\\` → \), (`\n` → ASCII LF 0x0A), (`\r` → ASCII CR 0x0D), (`\t` → ASCII TAB 0x09)

## 2.4 Kommentare

Kommentare in MAVL gibt es in zwei Varianten. Die erste Variation sind Zeilenkommentare, welche mittels // begonnen werden und dann bis zum Ende der Zeile reichen. Darüber hinaus gibt es mehrzeilige Kommentare, welche mit einem /* begonnen werden und mit einem */ enden. Alles, was sich zwischen diesen Symbolkombinationen befindet, wird als Kommentar behandelt.

# 3 Module

MAVL-Programme bestehen aus genau einem Modul, welches in einer einzelnen Datei enthalten ist. Dieses Modul kann beliebig viele Funktionen und Record-Typen enthalten, wobei eine dieser Funktionen die **main**-Funktion sein muss. Ihre Signatur sieht immer folgendermaßen aus:

Listing 2: Signatur der main-Funktion

```
function void main() {...}
```

Darüber hinaus spielt die Reihenfolge der Funktionsdefinitionen und Record-Typ-Deklarationen keine Rolle. Die Datei enthält ausschließlich die Funktionsdefinitionen und Record-Typ-Deklarationen. Das Modul muss nicht extra gekennzeichnet werden. MAVL unterstützt keine globalen Variablen. Variablen und Wertbezeichner sind nur innerhalb der Funktion gültig, in welcher sie definiert wurden, nicht jedoch über Funktionsgrenzen hinweg.

# 4 Typsystem

MAVL unterstützt verschiedene primitive, komplexe und benutzerdefinierte Typen. Im Folgenden werden die Typen und ihre Eigenschaften vorgestellt.

## 4.1 Typklassifizierung

Alle Typen in MaVL lassen sich folgendermaßen hierarchisch klassifizieren:

- **MaVL Typen** Die allgemeinste Klasse von Typen. Umfasst alle Typen, die MaVL zu bieten hat.
  - `void` Nur als Rückgabetyp von Funktionen gültig, gibt dort an, dass kein Wert zurückgegeben wird.
  - **Wert-Typen** Umfasst alle Typen, die einem Wert oder einer Variable zugewiesen werden können.
    - **Nutzerdefinierte Typen** Alle als `record` deklarierten Typen.
    - **Member-Typen** Umfasst alle Typen, die für Element-Deklarationen in einem Record gültig sind.
      - `string` Eine Zeichenkette, nützlich für Ausgabe von Text.
      - **Primitive Typen** Atomare, vordefinierte Typen, die genau ein Datenwort belegen.
        - `bool` Wahrheitswert (`true` oder `false`).
        - **Numerische Typen** Alle Typen, die Zahlenwerte speichern.
          - `int` Vorzeichenbehafteter 32 Bit Ganzzahlwert.
          - `float` 32 Bit (single precision) Fließkommazahl nach dem IEEE-754-Standard.
      - **Strukturtypen** Gruppiering mehrerer Zahlenwerte.
        - `vector` Eindimensionale Liste fester Länge von Zahlenwerten.

- `matrix` Zweidimensionales Feld fester Größe von Zahlenwerten.

## 4.2 Typregeln und Typprüfung

MAVL ist statisch typisiert: Jedem Bezeichner, jedem Ausdruck, jeder Funktion wird zur Compilezeit ein Datentyp zugeordnet. Diese Datentypen sind verpflichtend und bindend: ein Programm, das gegen die Typregeln verstößt, muss vom Compiler als fehlerhaft zurückgewiesen werden.

Die Typen von Variablen und Wertdefinitionen sowie die Signaturen von Funktionen werden im Programmtext explizit annotiert und lassen sich danach nicht mehr ändern. Die Datentypen von Literalen ergeben sich direkt aus der Art des Literals, während komplexen Ausdrücke rekursiv anhand der Typisierungsregeln der verwendeten Operatoren und den Typen der Operanden analysiert werden.

So liefert etwa die Addition von zwei Fließkommazahlen wiederum eine Fließkommazahl, Elementselektion auf einem Integer-Vektor einen Integer, und so weiter. Die Typen von Operanden und Ergebnis können aber auch verschieden sein. Das Ergebnis einer Matrixmultiplikation ist zwar wieder eine Matrix, diese hat im allgemeinen jedoch andere Dimensionen als die Operanden. Ein weiteres Beispiel ist das Skalarprodukt, welches aus zwei Vektorwerten einen skalaren Wert berechnet.

MAVL enthält keine automatischen oder impliziten Typkonvertierungen, auch gibt es kein *subtyping*. Folglich ist in Kontexten wie Zuweisungen, Operanden von Operatoren, Funktionsargumenten, etc. stets **Typgleichheit** erfordert. So kann zum Beispiel keine Integer-Zahl einer Float-Variable zugewiesen werden oder eine Matrix mit Dimension $n \times 1$ oder $1 \times n$ als Vektor behandelt werden. Besonders zu beachten ist, dass die Dimensionen von Vektoren und Matrizen Teil des statischen Datentyps sind, so kann also z.B. keine Variable existieren, die manchmal einen Vektor mit 2 und manchmal einen Vektor mit 3 Elementen speichert.

Die genauen Typisierungsregeln der Sprachkonstrukte werden in späteren Abschnitten beschrieben, die sich mit diesen Sprachkonstrukten befassen. Der Rest dieses Abschnittes behandelt die verschiedenen Datentypen, die in MAVL existieren.

## 4.3 Primitive Datentypen

Die primitiven Datentypen sind von MAVL atomare, vordefinierte Datentypen.

- Das Schlüsselwort `int` bezeichnet den Integer-Typ, also mathematisch ganze Zahlen, allerdings auf 32 Bit im Zweier-Komplement beschränkt.
- Das Schlüsselwort `float` bezeichnet 32 Bit große Fließkommazahlen zur Basis 2 nach dem IEEE-754-Standard (*single precision*).
- Das Schlüsselwort `bool` bezeichnet Wahrheitswerte bzw. boolesche Werte, von denen es zwei gibt, bezeichnet `true` und `false`.

Das folgende Programmfragment zeigt Beispiele für die Verwendung dieser primitiven Datentypen:

Listing 3: Primitive Datentypen und Prozedur in MAVL

```
//Beispiele fuer Integer-Zahlen
val int integerValue = 42;

//Beispiele fuer Float-Zahlen
val float floatValue = 3.1514;

//Beispiele fuer Bools
val bool boolValue1 = true;
val bool boolValue2 = false;
```

Die Schlüsselwörter **var** und **val** deklarieren Variablen und Wertdefinitionen. Mehr zu diesen Schlüsselwörtern finden Sie im Kapitel Befehle unter den Abschnitten Wertdefinition und Variablendeklaration.

## 4.4 Void

Der leere Datentyp `void` wird ausschließlich als Rückgabetyp von Funktionen verwendet. Er gibt dabei an, dass die entsprechende Funktion keinen Rückgabewert liefert. Er ist in diesem Sinne also kein wirklicher Datentyp, sondern dient nur der Kennzeichnung rückgabeloser Funktionen. Er kann nicht als Typ für Variablen oder Wertefinitionen verwendet werden. Funktionen, welche durch `void` gekennzeichnet sind, sind folglich Prozeduren.

Listing 4: Prozedur in MAVL

```
function void someProcedure(){
    // ...
}
```

## 4.5 String

Der String-Typ ist der erste von drei vordefinierten, komplexen Typen in MAVL. Das Schlüsselwort, welches Strings kennzeichnet lautet `string`. String-Literale werden durch Anführungsstriche eingeleitet und auch beendet. Für genauere Hinweise zu den erlaubten Zeichen und verfügbaren Escape-Sequenzen, siehe Abschnitt 2.3. String-Operationen (wie etwa Konkatenation) werden in MAVL nicht unterstützt.

## 4.6 Vektoren

Vektoren sind eine Art von komplexen Datentypen in MAVL, die mehrere Zahlen beinhalten. Im Gegensatz zu den obig besprochenen Datentypen, welche durch ein einzelnes Schlüsselwort angegeben werden, ist der Vektortyp parametrisiert durch einen Elementtyp und die Größe des Vektors. Es gibt also viele verschiedene Vektortypen.

Der Elementtyp bezeichnet den Typ eines einzelnen Elements im Vektor und kann nur einer der numerischen Typen `int` oder `float` sein. Darüber hinaus ist die Größe des Vektors, also die Anzahl der Elemente, Teil des Vektortyps. Die Größe muss eine positive ganze Zahl sein und als konstanter Ausdruck (siehe Abschnitt 7.3) angegeben werden. Im folgenden Beispiel etwa ist ein Vektor mit Elementtyp Integer und Größe vier zu sehen:

Listing 5: Wertdefinition eines vierdimensionalen int-Vektors

```
val vector<int>[4] a = [1,2,3,4];
```

## 4.7 Matrizen

Matrizen sind das zweidimensionale Äquivalent zu den eindimensionalen Vektoren. Dementsprechend sind Matrixtypen durch Elementtyp und zwei Größen - die Anzahl der Zeilen und Spalten - parametrisiert. Wie bei Vektoren können beide Größen nur positive ganze Zahlen sein und müssen als konstante Ausdrücke (siehe Abschnitt 7.3) angegeben sein. Ebenso können Matrizen nur Integer- und Float-Elemente enthalten.

Gemäß mathematischer Konventionen wird zuerst die Anzahl der Zeilen, dann die Anzahl der Spalten angegeben. Das folgende Beispiel zeigt also die Definition einer Matrix mit Elementtyp Float, drei Zeilen, und zwei Spalten.

Listing 6: Definition einer 3x2-Matrix aus float-Werten

```
val matrix<float>[3][2] A = [[1.1,2.2],
                             [4.4,5.5],
                             [7.7,8.8]];
```

## 4.8 Records

Bei Records handelt es sich um vom Programmierer neu definierte Typen, die aus mindestens einem Element bestehen. Für die Elemente sind alle Member-Typen zulässig, also alle Typen außer `void` und weiteren `record`s. Eine Verschachtelung von Records ist demnach nicht möglich. Records ermöglichen es, zusammengehörige Werte über eine gemeinsame Variable oder Wertdefinition zu referenzieren.

### 4.8.1 Deklaration von Record-Typen

Um einen Record-Typ verwenden zu können, muss dieser deklariert werden. Die Deklarationen müssen sich innerhalb eines Moduls und außerhalb von Funktionsdefinitionen befinden. Die Reihenfolge von Verwendung und Deklaration ist dabei beliebig, die Deklaration kann auch nach der ersten Verwendung liegen.

Listing 7: Deklaration eines Record-Typs

```
record myRecord {
  var float r;
  var matrix<int>[2][2] M;
  val int i;
}
```

### 4.8.2 Record-Instanziierung

Um eine Instanz eines Record-Typs zu erzeugen, wird eine ähnliche Syntax wie bei Vektoren verwendet, allerdings muss der Name des Typs vorangestellt werden, damit klar ersichtlich ist, welcher Typ erzeugt wird.

Listing 8: Definition eines Wertes des Record Typs aus Listing 7

```
val myRecord x = @myRecord[1.0, [[1, 2], [3, 4]], 10];
```

# 5 Funktionen

Funktionen sind integraler Bestandteil von MAVL-Modulen. Sie können, wie schon erwähnt, beliebig viele Funktionen in beliebiger Reihenfolge definieren, solange Sie eine **main**-Funktion definieren. Prinzipiell sieht die Definition einer Funktion folgendermaßen aus:

Listing 9: Definition einer einfachen MAVL-Funktion

```
//Definition einer Inkrementierungsfunktion
function int increment(int i){
    i = i + 1;
    return i;
}
```

Die Definition wird durch das `function`-Schlüsselwort eingeleitet. Anschließend folgt der Typ des Rückgabewerts und der Name der Funktion. Die Argumentliste beendet den Funktionskopf und es folgt in geschweiften Klammern der Funktionskörper. Der Funktionskörper enthält 0 oder mehr Befehle und ist, ähnlich wie der Befehlsblock (siehe 6.5), ein Geltungsbereich für Bezeichner. Ein Funktionskörper kann nur Befehle enthalten, keine verschachtelten Funktionen oder Record-Typ-Deklarationen.

## 5.1 Typregeln und return-Befehle

Prinzipiell kann der Typ einer Funktion in MAVL beliebig gewählt werden. Dies gilt insbesondere für den `void`-Typ. Jedoch unterliegen die Funktionstypen, wie auch die Typen von Variablen und Wertdefinitionen, strengen Regeln:

- Der definierte Typ der Rückgabe und der tatsächliche Typ des Rückgabewerts müssen typäquivalent sein. Das bedeutet, dass eine `int`-Funktion nur einen Wert des Typs `int` zurückgeben darf.
- Jede Funktion muss genau einen `return`-Befehl enthalten, außer sie ist vom Typ `void`. Darüber hinaus muss der `return`-Befehl, sofern die Funktionen einen enthält, immer der letzte Befehl einer Funktion sein. Dies bedeutet insbesondere, dass der **return**-Befehl nicht in Verzweigungen oder verschachtelten Blöcken verwendet werden darf.
- Funktionen des Typs `void` haben keinen Rückgabewert und dementsprechend keinen `return`-Befehl. Diese Funktionen werden als Prozeduren bezeichnet.

## 5.2 Argumente und Argumentlisten

Aufgrund der Tatsache, dass MAVL keine globalen Variablen unterstützt, sind Argumente und Argumentlisten von großer Bedeutung. Sämtliche Variablen und Wertdefinitionen, die innerhalb einer Funktion verwendet werden, müssen explizit übergeben werden. In der Funktionsdeklaration wird durch Argumente bzw. Argumentlisten spezifiziert, welche Parameter später übergeben werden müssen. Außerdem wird zu jedem Argument der Typ spezifiziert. Bei einem einzelnen Argument muss nur der Typ und ein Bezeichner angegeben werden. Wenn mehrere Argumente übergeben werden sollen, so werden diese als Komma-separierte Liste spezifiziert. Formale Parameter sind variabel, können also innerhalb der Funktion neu zugewiesen werden.

Listing 10: Argumente und Argumentlisten

```
//Funktionsdefinition mit einem Argument
function int oneArgument(int i) {
    ...
}

//Funktionsdefinition mit mehreren Argument
function int multiArgument(int i, float j, bool k, int l) {
    ...
}
```

# 6 Befehle

Es gibt in MAVL, wie auch in anderen Programmiersprachen, verschiedene Formen von Befehlen, welche meist durch ein Semikolon beendet werden. Diese werden wir nun der Reihe nach vorstellen. Für jeden vorgestellten Befehl werden wir ein einfaches Syntax-Beispiel als MAVL-Code betrachten. Darüber hinaus wird die Semantik dahinter erläutert. Anschließend findet sich eine Sammlung von verschiedenen Verwendungsmöglichkeiten.

## 6.1 Wertdefinition

Wertdefinitionen berechnen das Ergebnis eines Ausdrucks einmal und speichern es unter einem vom Programmierer gewählten Namen, dem *Wertbezeichner*, damit andere Ausdrücke später diesen Wert weiterverwenden können, ohne den Ausdruck erneut auszuwerten.

Listing 11: Einfache Wertdefinitionen

```
val int integerValue = 42;
```

Der Befehl wird eingeleitet durch das Schlüsselwort `val`. Anschließend folgt der Typ, welcher in diesem Beispiel `int` ist. Erlaubt sind hierbei alle Wert-Typen, also alle Typen außer `void`. Außerdem muss ein Bezeichner vergeben werden, welcher in diesem Beispiel „integerValue“ lautet. Anschließend folgt ein Gleichheitszeichen, hinter dem ein Ausdruck folgt. Die Wertdefinition wird durch ein Semikolon beendet.

Wird ein solcher Befehl ausgeführt, so wird der Ausdruck auf der rechten Seite evaluiert und anschließend dem Namen `integerValue` zugewiesen. Nach dieser initialen Zuweisung kann der Wert nicht mehr verändert werden. Wertdefinitionen sind also unveränderlich, ihr Wert muss jedoch nicht eine zur Compilezeit bekannte Konstante sein, sondern kann etwa vom aktuellen Wert einer Variable abhängen.

Der Typ des Ausdrucks muss mit dem im `val`-Befehl deklarierten Typ übereinstimmen. Beispielsweise kann ein `float`-Ausdruck nur auf eine `float`-Definition zugewiesen werden.

Wie Sie in den folgenden Beispielen sehen, ist es möglich, dass die Ausdrücke nach dem Gleichheitszeichen beliebig komplex werden.

Listing 12: Verschiedene Beispiele für Wertdefinitionen

```
//Wertdefinitionen mit einfachen Ausdruecken
val float floatValue = 3.1514;
val bool boolValue = true;

//Wertdefinitionen mit komplexeren Ausdruecken
val float floatValue2 = 3.1 + floatValue;
val bool boolValue2 = true & false | false;
```

## 6.2 Variablendeklaration

Bevor eine Variable verwendet werden kann, muss diese deklariert werden. Eine solche Deklaration sieht folgendermaßen aus:

Listing 13: Variablen Deklaration

```
var int integerVariable;
```

Der Befehl wird eingeleitet durch das Schlüsselwort `var`, gefolgt durch einen Typ, welcher in diesem Fall `int` ist. Auch hier sind wieder alle Typen außer `void` erlaubt. Anschließend folgt der Variablenname in Form eines Bezeichners. Ein Semikolon beendet den Befehl.

Der Speicherbereich, der einer Variablen zugeordnet ist, wird mit Nullen initialisiert.

## 6.3 Variablenzuweisung

Der Wert einer Variablen kann durch eine Variablenzuweisung verändert werden. Eine solche Zuweisung sieht wie folgt aus:

Listing 14: Einfache Variablenzuweisung

```
integerVar = 43 + 2;
```

Die Variablenzuweisung besteht dabei aus einer linken und rechten Seite, welche durch ein Gleichheitszeichen getrennt werden. Bei einer Zuweisung wird der Ausdruck auf der rechten Seite evaluiert und anschließend der Variable zugewiesen. Ein Semikolon beendet den Befehl. Mehr zu Ausdrücken finden Sie im Abschnitt Ausdrücke.

Der Bezeichner auf der linken Seite des Gleichzeichen muss eine zuvor mit `var` deklarierte Variable oder ein formaler Parameter sein. Es ist nicht möglich, auf einen nicht deklarierten Bezeichner oder einen Wertbezeichner (mit `val` deklariert) zuzuweisen. Weiterhin ist die Einhaltung der Typregeln wichtig: der Typ des Ausdrucks muss der gleiche sein wie der, mit dem die Variable deklariert wurde.

Neben einfachen Zuweisungen zu Bezeichnern können auch analog Vektoren- und Matrizen-Elementen zugewiesen werden. Das entsprechende Element wird durch einen oder zwei Indizes spezifiziert, welche beliebige Ausdrücke vom Typ Integer sein können. Liegt einer der zur Laufzeit berechneten Indizes außerhalb des gültigen Bereiches, terminiert das Programm mit einem Fehler. Das folgende Beispiel demonstriert die Elementzuweisung:

Listing 15: Vektor- und Matrix- Elementzuweisungen in MAVL

```
var vector<int> [3] vec1;
vec1 = [1,2,3];     //Vektorzuweisung
vec1[0] = 7;        //Vektorelementzuweisung
//vec1 sieht nun so aus: [7,2,3]

var matrix<int> [3][3] mat1;
mat1 = [[1,2,3],
        [2,3,4],
        [3,4,5]];
mat1[1][2] = 0;
//mat1 sieht nun so aus: [[1,2,3],
//                        [2,3,0],
//                        [3,4,5]]
```

Bei einer Zuweisung auf eine Matrix kann nur ein einzelnes Element zugewiesen werden. Es ist nicht möglich, eine ganze Zeile der Matrix zu überschreiben.

Um den Wert eines Record-Elements neu zuzuweisen, wird dieses mit **`@<elementname>`** referenziert. Handelt es sich bei dem Element z.B. um einen Vektor, so kann nur der gesamte Vektor neu zugewiesen werden. Eine Kombination mit den Zuweisungen aus Listing 15 ist nicht möglich. Nur mit `var` deklarierte Elemente dürfen neu zugewiesen werden.

Listing 16: Beispiel einer Zuweisung an ein Recordelement am Typ aus Listing 7

```
var myRecord x;
x = @myRecord[1.0, [[1, 2], [3, 4]], 10];
x@r = 3.141;
// x sieht nun so aus: [3.141,
//                      [[1, 2], [3, 4]],
//                      10]
```

## 6.4 Aufrufe

MAVL unterstützt sowohl vordefinierte als auch selbst definierte Funktionen. Aufrufe erlauben die Verwendung dieser Funktionen. Aufrufe können allein stehen oder zu Ausdrücken verschachtelt werden (siehe dazu Kapitel 7). Wird ein Funktionsaufruf als Befehl verwendet, wird dieser durch ein Semikolon beendet.

Listing 17: Funktionsaufrufe in MAVL

```
//Funktionsaufruf mit/ohne Wertrueckgabe
i = increment(i);
doSomething(i, j, k);
```

Wichtig ist bei Aufrufen vor allem die Übergabe der Parameter. MAVL verwendet bei der Parameterübergabe das Konzept Call-by-Value. Dementsprechend werden alle Parameter, welche innerhalb der Argumentliste einer Funktion stehen, als Kopie an die Funktion übergeben. Folglich sind die übergeben Parameter innerhalb der Funktion selbständig und unabhängig von ihren Werten außerhalb der Funktion. Darüber hinaus wird zur Compilezeit eine Typprüfung vorgenommen, sodass für entsprechende Argumente auch nur Werte des gleichen Typs übergeben werden können.

Im Kontext von Befehlen wird ein eventueller Rückgabewert der Funktion verworfen, es ist also möglich, Funktionen mit anderen Rückgabetypen als **void** als Befehl aufzurufen.

## 6.5 Blöcke

In MAVL können Befehle nicht nur einzeln auftreten, sondern auch als Befehlsblöcke (compound statements). Blöcke sind Sequenzen von Befehlen, die von geschweiften Klammern umschlossen werden, und können überall dort verwendet werden, wo ein beliebiger Befehl erwartet wird. Ein Beispiel hierfür sehen Sie in Listing 18. Wird der Anfang des Blocks erreicht, so werden nacheinander alle Befehle im Block ausgeführt, bis das Ende des Blocks erreicht wird. Ist ein Block leer, so wird er einfach übersprungen. Blöcke definieren neue Geltungsbereiche (engl. “scopes”), sodass innerhalb eines Blocks Variablennamen neu belegt werden können, ohne, dass Variablen oder Wertbezeichner außerhalb des Blocks davon betroffen sind. In MAVL können mehrere Geltungsbereiche ineinander verschachtelt sein. Folgendes Beispiel soll diesen Sachverhalt verdeutlichen:

Listing 18: Verwendung von Blöcken in MAVL

```
function void main() {
    var int i;
    i = 1;
    //hier beginnt ein neuer Block
    {
        val int k = i * 2;

        var int i;
        i = 5;

        val int j = i * 2;
        //Werte der var/vals zu diesem
        //Zeitpunkt: i = 5, j = 10, k = 2
    }
    //hier endet der Block
    ...
}
```

Wird ein Bezeichner verwendet, so geschieht dies immer innerhalb eines Blocks. Sofern der Bezeichner in diesem Block deklariert wurde, so wird diese Deklaration verwendet. Ist dies nicht der Fall, so wird im jeweils umschließenden Block nach einer Deklaration gesucht.

Im Beispiel sieht man dies an dem definierten Block. Durch den Block wird prinzipiell ein neuer Scope geöffnet, aber die Variable i ist zum Zeitpunkt des ersten Befehls nicht deklariert, also wird der umschließende Scope der **main**-Funktion durchsucht. Da dieser Scope eine passende Variable i enthält, wird diese verwendet. Der nächste Befehl innerhalb des Blocks definiert eine neue Variable i, welche nun innerhalb des Scopes liegt. Alle folgenden Befehle, die den Bezeichner i verwenden werden nun auf dieses i zugreifen, bis der Block endet. Pro Scope darf jeder Bezeichner nur maximal einmal deklariert werden.

## 6.6 Schleifen

MAVL unterstützt `for`- und `foreach`-Schleifen. Schleifen bestehen aus einem Schleifenkopf und einem Schleifenkörper. Der Schleifenkörper ist ein einzelner Befehl, bei dem es sich aber auch um einen Block handeln kann. Der Schleifenkopf bestimmt, ob und wie häufig der Körper ausgeführt wird.

### 6.6.1 For-Schleife

In MAVL leitet das Schlüsselwort `for` den Kopf einer gleichnamigen Schleife ein. Anschließend folgen, in einem Paar runder Klammern, die drei Teile des Schleifenkopfs:

- Eine initiale Zuweisung, welche einer bestehenden Variable den Wert eines Ausdrucks zuweist. Diese Zuweisung wird einmalig ausgeführt, wenn der Schleifenkopf im Programmcode erreicht wird. Die angegebene Zuweisung muss den Typregeln entsprechen.
- Eine Bedingung, welche vor jedem Schleifendurchlauf geprüft wird. Der Typ des Ausdrucks muss dabei `bool` sein und eine Iteration wird dann ausgeführt, wenn die Evaluierung des Ausdrucks `true` zurückliefert. Ansonsten wird die Schleife beendet und die Ausführung beim ersten Befehl nach der Schleife fortgesetzt.
- Eine Zuweisung, welche einen Ausdruck auf eine Variable zuweist. Typischerweise wird hier eine Zählvariable inkrementiert. Wichtig ist dabei nur, dass auf der linken Seite des Gleichheitszeichens ein Variablenbezeichner steht und rechts des Gleichheitszeichens ein Ausdruck. Darüber hinaus müssen die Typregeln eingehalten werden.

Nach der schließenden Kammer des Schleifenkopfs folgt ein Befehl, der in jeder Iteration ausgeführt wird. Jeder in Befehle spezifizierte Befehl kann als Schleifenkörper verwendet werden. Der Befehl steht in einem neuen Scope, auch wenn es sich nicht um einen Befehlsblock handelt.

Im folgenden Beispiel sehen Sie eine Schleife, welche eine Variable hoch zählt:

Listing 19: Einfache for-Schleife in MAVL

```
var int i;
var int counter;
i = 0;
for (counter = 0; counter < 10; counter = counter + 1) {
    i = i + 1;
}
```

### 6.6.2 Foreach-Schleife

Die foreach-Schleife dient dazu, über die Elemente von Vektoren und Matrizen zu iterieren, ohne explizit eine Zählvariable verwenden zu müssen. Der Schleifenkopf einer foreach-Schleife beginnt mit dem Schlüsselwort `foreach` und verfügt über zwei Teile, die durch einen Doppelpunkt voneinander getrennt sind.

- Eine Iterator-Deklaration, welche in einem eigenen Scope steht. Diese besteht aus dem Schlüsselwort `var` oder `val`, dem Typ des Iterators und dessen Namen. Ein mit `val` deklarierter Iterator erlaubt nur lesenden Zugriff auf die Vektor-/Matrixelemente. Im Gegensatz dazu kann über einen `var`-Iterator das jeweils aktuelle Element verändert werden.

- Ein Ausdruck der zu einem Vektor oder einer Matrix evaluiert und über dessen Elemente iteriert werden soll.

Dabei darf der Iterator nur dann als `var` deklariert werden, wenn es sich bei dem Ausdruck um eine `var`-Variable handelt. Zudem muss der Typ des Iterators mit dem Elementtyp des zu iterierenden Ausdrucks übereinstimmen.

Ist der Iterator als `var` deklariert, so bedeutet eine Zuweisungen an den Iterator, dass derselbe Wert auch im entsprechenden Element des Vektors oder der Matrix wiederzufinden ist, sobald die jeweilige Iteration abgeschlossen ist.

Die Iterationsreihenfolge entspricht der Reihenfolge, in der die Elemente auch bei einer Vektor- oder Matrix-Definition aufgeführt werden.

Wie bei der `for`-Schleife kann jeder Befehl als Schleifenkörper verwendet werden. Dieser steht in einem neuen Scope, womit die `foreach`-Schleife also zwei Scopes öffnet:

1. Ein Scope, in dem die Deklaration des Iterators steht.
2. Ein untergeordneter Scope, in dem der Schleifenkörper steht.

Im Folgenden sollen 2 Beispiele den Unterschied zwischen einem `val` und `var` Iterator verdeutlichen:

Listing 20: Beispiel einer foreach-Schleife in MAVL mit **val** Iterator

```
var vector<int>[3] v;
v = [1, 2, 3];
foreach(val int i : v * 2) { // 'var' nicht erlaubt
  // keine Zuweisungen an i erlaubt
    ...
}
```

Listing 21: Beispiel einer foreach-Schleife in MAVL mit **var** Iterator

```
var vector<int>[3] v;
v = [1, 2, 3];
foreach (var int i : v) {
    i = 3;
}
// v = [3, 3, 3]
```

## 6.7 Verzweigungen

Verzweigungen ermöglichen die Ausführung von Befehlen unter einer Bedingung. Hierfür verwendet MAVL das **If-Else**-Konstrukt. Der Befehl wird durch das `if`-Schlüsselwort eingeleitet und es folgt ein Ausdruck des Typs `bool`. Nach dem geklammerten Ausdruck folgt ein einzelner Befehl. Analog zum Schleifenkonstrukt kann es sich auch hier wieder um einen Befehlsblock handeln. Darüber hinaus kann ein weiterer Befehl mittels des Schlüsselworts `else` angehängt werden.

Listing 22: Verzweigung in MAVL

```
//If-Standalone Variante ohne else
val int i = 1;
val int j = 2;

if (i < j) {
    ...
}

//If-Else Variante
if (i > j) {
    ...
} else {
    ...
}

//Verschachtelt
if (i > j)                  //If1
    if (j < i - 1) ...      //If2
    else ...                //Else2
```

Der Befehl funktioniert folgendermaßen: Zuerst wird der Ausdruck evaluiert. Ergibt sich `true` als Wert des Ausdrucks, so wird der erste Befehl ausgeführt. Ergibt sich stattdessen `false`, so wird der erste Befehl nicht ausgeführt. Ist ein **else**-Schlüsselwort vorhanden, so wird der zugehörige Befehl oder Block ausgeführt. Ansonsten wird die Ausführung nach der Verzweigung fortgesetzt. Die bedingten Befehle werden in einem neuen Scope ausgeführt, auch wenn sie keine Befehlsblöcke sind. Es ist also nicht möglich, eine Variable unter einem `if`-Befehl zu deklarieren und außerhalb davon zu benutzen.

Darüber hinaus muss beachtet werden, dass bei mehreren verschachtelten Verzweigungen ein **else** immer dem direkt vorhergegangenen `if` zugeordnet wird, am verschachtelten Beispiel in Listing 22 wird die Zuordnung demonstriert. Das mit “Else2” gekennzeichnete `else` wird dem direkt vohergegangenen `if` (“If2”) zugeordnet. Dieses Problem (“dangling else” in der Literatur) kann durch die Verwendung von Blöcken vermieden werden.

Möchte man eine stark verschachtelte if-else-Konstruktion vermeiden, bietet es sich an, ein Switch-Case-Konstrukt zu verwenden:

Ähnlich wie beim if wird auch hier zuerst die Bedingung berechnet (diese steht in den Klammern nach dem `switch`). Die Bedingung darf an dieser Stelle aber nur den Datentyp `int` haben und wird anschließend mit den konstanten Ausdrücken (siehe Abschnitt 7.3) verglichen, welche jeweils nach dem Schlüsselwort `case` stehen. Der Case (bzw. der zugeordnete Befehl) bei dem Bedingung und Konstante übereinstimmen, wird ausgeführt. Darüber hinaus gibt es die Möglichkeit, maximal einen `default`-Case zu spezifizieren. Greift keiner der angegebenen Cases, wird das Statement, das mit dem Default-Case assoziiert ist, ausgeführt. Ist kein Default-Case vorhanden, wird kein Befehl ausgeführt und das Programm wird nach dem Switch-Case-Statement fortgeführt. Die Cases (bzw. ihre Konstanten) müssen eindeutig sein. Die Reihenfolge der Cases ist beliebig. Anders als in anderen Programmiersprachen gibt es kein *fall-through* von einem Case zum nächsten und auch keinen **break**-Befehl um *fall-though* zu verhindern, es wird also immer maximal ein Case ausgeführt. Die Befehle, die den Cases zugeordnet sind, werden in einem neuen Scope ausgeführt, auch wenn sie keine Befehlsblöcke sind. Es ist also nicht möglich, eine Variable in einem Case zu deklarieren und sie dann in einem anderen Case oder außerhalb des `switch`-Befehls zu verwenden.

Listing 23: Switch-Case in MAVL

```
var int a ;
a = 10;

// Einfaches Switch-Case-Konstrukt:
switch(a){
  case 10:
    a = 27;
  case 1:
    a = 7;
  default:
    a = 10;
  case -1:
    a = -42;
}

// Will man mehrere Statements in einem Case verwenden,
// muss man dies ueber ein Compound-Statement tun:
var int a ;
a = 10;

switch(a){
  case 1:
      {
        a = 7;
        a = a + 7;
      }
  case 10:
      {
        a = 27;
        a = a + 7;
      }
  default:
      a = 10;
}
```

## 6.8 Rückgabe

Rückgabe-Befehle gibt es nur in Funktionen, deren Rückgabetyp nicht `void` ist. Rückgabe-Befehle liefern einen Wert zurück an die aufrufende Funktion und beenden die aktuelle Funktion. Rückgaben sind gekennzeichnet durch das Schlüsselwort `return`. Ein Beispiel hierfür finden Sie in Listing 24.

Listing 24: return-Befehle in MAVL

```
function int function1() {
    val int i = 42;
    ...

    return i;
}
```

Wichtig ist auch bei der Verwendung von `return`-Befehlen, dass nur Werte des deklarierten Rückgabetyps der Funktion zurückgegeben werden dürfen. Darüber hinaus können beliebige Ausdrücke auf das Schlüsselwort folgen, welche dann evaluiert und zurückgegeben werden, sofern der Typ korrekt ist. Der Rückgabebefehl wird durch ein Semikolon abgeschlossen.

# 7 Ausdrücke

Berechnungen jeder Art werden in MAVL durch Ausdrücke dargestellt. Ausdrücke sind rekursiv definiert und können beliebig komplex sein:

- Atomare Ausdrücke sind der Basisfall.
- Struktur-Literale kombinieren einen oder mehrere Ausdrücke in einen Ausdruck, der einen zusammengesetzten Vektor, Matrix, oder Record produziert.
- Operatoren werden auf einen oder mehrere kleinere Ausdrücke angewandt, um ein Ergebnis zu produzieren.
- Funktionsaufrufe können nicht nur Befehle sondern auch Ausdrücke sein, sofern der Rückgabetyp nicht `void` ist.
- Klammern um einen Ausdruck können als unärer Operator betrachtet werden, sind jedoch besonders hervorzuheben, da sie keine Berechnung ausführen, sondern dazu dienen, die Präzedenz in verschachtelten Ausdrücken festzulegen und Mehrdeutigkeiten aufzulösen.

In den folgenden Abschnitten werden wir die verschiedenen Arten von Ausdrücken in MAVL, sowie die zugehörigen Typisierungsregeln, behandeln.

## 7.1 Atomare Ausdrücke

Die atomaren Ausdrücke in MAVL sind Integer-, Float-, Bool- und String-Literale (siehe Abschnitt 2.3) sowie Bezeichner, die sich auf zuvor deklarierte Funktionsparameter, Variablen, oder Wertdefinitionen beziehen.

## 7.2 Struktur-Literale

Literale von komplexen Strukturen wie Vektoren, Matrizen und Records kombinieren kleinere Ausdrücke, die die Elemente der Struktur vorgeben.

Ein Vektor-Literal enthält dabei Ausdrücke, die zu den Elementen des Vektors ausgewertet werden, während ein Matrix-Literal eine Reihe von Vektor-Ausdrücken enthält, deren Ergebnisse den Zeilen der Matrix entsprechen. Beispiele finden sich in Listing 15.

Record-Initialisierungen, d.h. Literale von Record-Typen enthalten einen Ausdruck für jedes Feld des Records, wie in Listing 8 zu sehen ist.

## 7.3 Konstante Ausdrücke

Konstante Ausdrücke sind eine Teilklasse von Ausdrücken, welche jede MAVL-Implementierung zur Compilezeit auswerten können muss. Sie werden an einigen Stellen in der Sprache verwendet, an denen der Wert eines Ausdrucks für die kontextuelle Analyse wichtig ist, wie etwa die Cases von switch-case-Befehlen oder die Dimensionen von (Sub-)Vektoren und (Sub-)Matrizen.

Konstante Ausdrücke sind in MAVL immer vom Typ `int` und setzen sich aus Integer-Literalen, Klammern und den primitiven arithmetischen Operatoren (im Sinne von Abschnitt 7.4) zusammen. Jeder Ausdruck, der diesen Anforderungen genügt, ist ein konstanter Ausdruck, und keine anderen Ausdrücke sind konstante Ausdrücke.

Es ist besonders zu beachten, dass kategorisch keine Bezeichner in konstanten Ausdrücken vorkommen können, weder Namen von Variablen noch Wertbezeichner. Im folgenden Beispiel sind die Vektordimensionen sowie die obere und untere Schranke des Subvektor-Operators konstante Ausdrücke, `i*(3+1)` jedoch nicht:

Listing 25: Beispiele für konstante Ausdrücke

```
var vector<float>[(3+1)*10] v1;
val vector<float>[(3+1)*5] v2 = v1{-2 * (3+1) : i*(3+1) : 2 * (3+1)};
```

## 7.4 Operatoren

Mit Hilfe von Operatoren können in MAVL komplexe Ausdrücke gebildet werden. MAVL definiert Operatoren für die folgenden Operationen:

- Arithmetische Operationen: Addition, Subtraktion, Multiplikation, Division, Potenz sowie Negation (unäres Minus)
- Vergleichsoperationen: Größer, Kleiner, Größer-Gleich, Kleiner-Gleich, Gleich und Ungleich
- Boolesche Operationen: Nicht, Und und Oder
- Spezielle Vektor- und Matrixoperationen: Submatrix, Subvektor, Matrixmultiplikation, Skalarprodukt, Abfrage der Dimension, Elementselektion sowie Matrix-Transponierung
- Selektion eines Recordelements

Beispiele für die Verwendung finden Sie im Abschnitt 7.11.

## 7.5 Ternärer Operator (bedingte Auswertung)

Desweitern gibt es den ternären *select*-Operator '?', welcher, analog zu Sprachen wie C und Java, eine bedingte Auswertung durchführt. Eine Anwendung des Operators beinhaltet drei Ausdrücke:

- Ein boolescher Ausdruck, welcher die Bedingung angibt.
- Wird die Bedingung zu `true` ausgewertet, wird der zweite Ausdruck ausgewertet.
- Sonst wird der dritte Ausdruck ausgewertet.

Hierbei gilt zu beachten das mehrere ?-Ausdrücke nicht ohne entsprechende Klammerung geschachtelt werden dürfen. Weiterhin müssen der zweite und dritte Ausdruck typgleich sein. Beispiele für die Verwendung finden Sie im Abschnitt 7.11.

## 7.6 Operationen auf primitiven Datentypen

Tabelle 1 beschreibt die Semantik der für primitive Typen definierten Operatoren.

| Linker Operand | Op | Rechter Operand | | Ergebnis | Assoz. | Beschreibung |
|---|---|---|---|---|---|---|
| `int` | `+` | `int` | → | `int` | L | Addition |
| `float` | `+` | `float` | → | `float` | L | |
| | `-` | `int` | → | `int` | - | Negation |
| | `-` | `float` | → | `float` | - | |
| `int` | `-` | `int` | → | `int` | L | Subtraktion |
| `float` | `-` | `float` | → | `float` | L | |
| `int` | `*` | `int` | → | `int` | L | Multiplikation |
| `float` | `*` | `float` | → | `float` | L | |
| `int` | `/` | `int` | → | `int` | L | Division |
| `float` | `/` | `float` | → | `float` | L | |
| `int` | `^` | `int` | → | `int` | R | Exponentiation |
| `float` | `^` | `float` | → | `float` | R | |
| `bool` | `&` | `bool` | → | `bool` | L | Logisches Und |
| `bool` | `\|` | `bool` | → | `bool` | L | Logisches Oder |
| | `!` | `bool` | → | `bool` | - | Logisches Nicht |
| `int` | `==` | `int` | → | `bool` | - | Gleich |
| `float` | `==` | `float` | → | `bool` | - | |
| `int` | `!=` | `int` | → | `bool` | - | Ungleich |
| `float` | `!=` | `float` | → | `bool` | - | |
| `int` | `<` | `int` | → | `bool` | - | Kleiner |
| `float` | `<` | `float` | → | `bool` | - | |
| `int` | `<=` | `int` | → | `bool` | - | Kleiner-gleich |
| `float` | `<=` | `float` | → | `bool` | - | |
| `int` | `>` | `int` | → | `bool` | - | Größer |
| `float` | `>` | `float` | → | `bool` | - | |
| `int` | `>=` | `int` | → | `bool` | - | Größer-gleich |
| `float` | `>=` | `float` | → | `bool` | - | |

Tabelle 1: Operationen auf primitiven Datentypen

Darüber hinaus unterstützt MAVL verschiedene Operationen auf Matrizen und Vektoren. Diese werden in den folgenden Abschnitten beschrieben.

## 7.7 Matrix- und Vektoroperationen

Auch für Matrizen und Vektoren sind die grundlegenden arithmetischen Operationen definiert:

- Addition (`+`): Addiert zwei Matrizen oder Vektoren elementweise.
- Subtraktion (`-`): Subtrahiert zwei Matrizen oder Vektoren elementweise.
- Multiplikation (`*`): Multipliziert zwei Matrizen oder Vektoren elementweise.

Die beiden Operanden müssen dabei typgleich sein, also sowohl den gleichen Elementtyp, als auch die gleichen Dimensionen haben.

Zusätzlich zu den genannten Struktur-Struktur Operationen gibt es die Möglichkeit, einen Vektor oder eine Matrix mit einem Skalar zu multiplizieren. Der Typ des skalaren Wertes muss hierbei mit dem Elementtyp der Struktur übereinstimmen.

Tabelle 2 zeigt die gültigen Typkombinationen für alle Matrix und Vektoroperationen. Dabei steht `T` stellvertretend für `int` oder `float`, sowie `d`, `r` und `c` für beliebige Dimensionen.

| Linker Operand | Op | Rechter Operand | | Ergebnis | Assoz. | Beschreibung |
|---|---|---|---|---|---|---|
| `vector<T>[d]` | `+` | `vector<T>[d]` | → | `vector<T>[d]` | L | Elementweise Addition |
| `matrix<T>[r][c]` | `+` | `matrix<T>[r][c]` | → | `matrix<T>[r][c]` | L | |
| `vector<T>[d]` | `-` | `vector<T>[d]` | → | `vector<T>[d]` | L | Elementweise Subtraktion |
| `matrix<T>[r][c]` | `-` | `matrix<T>[r][c]` | → | `matrix<T>[r][c]` | L | |
| `vector<T>[d]` | `*` | `vector<T>[d]` | → | `vector<T>[d]` | L | Elementweise Multiplikation |
| `matrix<T>[r][c]` | `*` | `matrix<T>[r][c]` | → | `matrix<T>[r][c]` | L | |
| `T` | `*` | `vector<T>[d]` | → | `vector<T>[d]` | L | Skalar-Multiplikation |
| `vector<T>[d]` | `*` | `T` | → | `vector<T>[d]` | L | |
| `T` | `*` | `matrix<T>[r][c]` | → | `matrix<T>[r][c]` | L | |
| `matrix<T>[r][c]` | `*` | `T` | → | `matrix<T>[r][c]` | L | |
| `vector<T>[d]` | `.dimension` | | → | `int` | - | Dimensionsoperatoren |
| `matrix<T>[r][c]` | `.rows` | | → | `int` | - | |
| `matrix<T>[r][c]` | `.cols` | | → | `int` | - | |
| `matrix<T>[r][d]` | `#` | `matrix<T>[d][c]` | → | `matrix<T>[r][c]` | L | Matrixmultiplikation |
| `vector<T>[d]` | `.*` | `vector<T>[d]` | → | `T` | - | Skalarprodukt |
| | `~` | `matrix<T>[r][c]` | → | `matrix<T>[c][r]` | - | Matrix-Transponierung |

Tabelle 2: Operationen auf Strukturtypen

### 7.7.1 Dimensionsoperatoren

Die folgenden Operatoren geben jeweils einen Integer-Wert zurück:

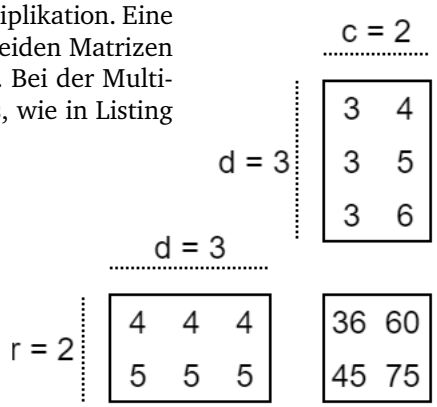
- `.dimension` kann nur auf Vektoren angewandt werden und liefert die Anzahl der Elemente des Vektors zurück.
- `.rows` kann nur auf Matrizen angewandt werden und liefert die Anzahl der Zeilen der Matrix zurück.
- `.cols` kann ebenfalls nur auf Matrizen angewandt werden und liefert die Anzahl der Spalten der Matrix zurück.

### 7.7.2 Skalarprodukt

Auf Vektoren kann das Skalarprodukt berechnet werden. MAVL unterstützt diese Operation mit dem `.*`-Operator auf zwei Vektoren, deren Elementtyp und Größe übereinstimmen. Der Typ des Ergebnisses ist identisch mit dem Elementtyp der Ausgangsvektoren.

### 7.7.3 Matrixmultiplikation

MAVL unterstützt mittels des binären # Operators auch eine echte Matrixmultiplikation. Eine Anwendung des Operators ist nur dann möglich, wenn der Elementtyp der beiden Matrizen identisch ist und die linke Matrix so viele Spalten hat, wie die rechte Zeilen. Bei der Multiplikation einer $r \times d$ und einer $d \times c$ Matrix kommt eine $r \times c$ Matrix heraus, wie in Listing 26 verdeutlicht wird.



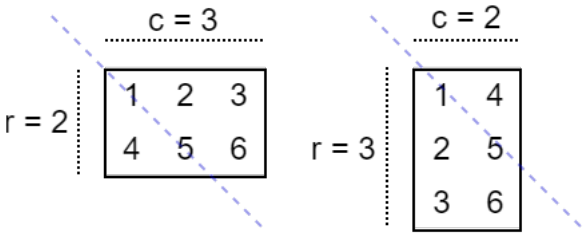

Listing 26: Beispiel einer Matrixmultiplikation

```
val matrix<int>[2][3] m1 = [[4, 4, 4], [5, 5, 5]];
val matrix<int>[3][2] m2 = [[3, 4], [3, 5], [3, 6]];

val matrix<int>[2][2] result = m1 # m2; //[[36, 60],
                                       // [45, 75]]
```

### 7.7.4 Matrix-Transponierung

Mit Hilfe des ~ Operators lässt sich eine Matrix transponieren, also an ihrer Diagonalen spiegeln. Die Anzahl von Zeilen und Spalten wird dabei vertauscht.




Listing 27: Beispiel für Matrix-Transponierung

```
val matrix<int>[2][3] m23 = [[1,2,3],[4,5,6]];
val matrix<int>[3][2] m32 = ~m23;
```

### 7.7.5 Submatrix und Subvektor

Aus einer Matrix oder einem Vektor kann nicht nur ein einzelnes Element selektiert werden, sondern auch eine Submatrix bzw. ein Subvektor ausgewählt werden. Beide verwenden eine Schreibweise mit geschweiften Klammern, um den ausgewählten Bereich zu beschreiben. Da Matrizen ein zweidimensionales Konstrukt sind, besteht der Submatrix-Operator aus zwei Bereichsspezifikationen, der Subvektor-Operator hingegen besteht nur aus einer Bereichsspezifikation. Die Beispiele in Listing 28 verdeutlichen die Syntax und Funktion der Operation.

Listing 28: Beispiele für Submatrix- und Subvektor-Selektionsoperationen

```
var matrix<int>[3][3] mat;
mat = [[1,2,3],
       [4,5,6],
       [7,8,9]];

var matrix<int>[2][2] mat2;
mat2 = mat{0:0:1}{0:0:1};
// mat2 sieht nun so aus: [[1,2],
//                         [4,5]]
val int i = 3;
var vector<float>[7] vec;
vec = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0];
// Selektiert vom i-1ten bis zum i+1ten Element
var vector<float>[3] vec2;
vec2 = vec{-1:i:1};
```

Innerhalb der Bereichsspezifikation werden jeweils 3 Werte angegeben. Wird beispielsweise $\{-1 : i : 1\}$ angegeben, so wird das Intervall $[i-1, i+1]$ selektiert. Das bedeutet, dass sowohl das $i-1$te Element, als auch das $i+1$te Element mit ausgewählt werden. Dementsprechend würde “vec2” aus Listing 28 folgende Werte enthalten: $[3, 4, 5]$. Verallgemeinert bedeutet dies, dass $\{l : e : u\}$ alles im Bereich von $e + l$ bis $e + u$ selektiert. Es gelten folgende Regeln:

- $l$ und $u$ müssen konstante Ausdrücke vom Typ `int` sein.
- $e$ ist ein beliebiger Integer-Ausdruck.
- $l$ ist kleiner oder gleich $u$.
- Das Ergebnis eines Submatrix-Ausdrucks ist immer eine Matrix und eines Subvektor-Ausdrucks immer ein Vektor.
- Liegt der ausgewählte Bereich nicht vollständig innerhalb der Struktur, terminiert das Programm mit einem Fehler.

Beide Operatoren können nur maximal einmal auf einen Ausdruck angewendet werden. Eine mehrfache Anwendung kann durch explizite Klammerung erreicht werden, wie Listing 29 zeigt.

Listing 29: Mehrfache Anwendung des Submatrix-Operators

```
var matrix<int>[5][5] mat3;
mat3 = getMatrix();

val matrix<int>[1][1] a = (mat3{-1:2:1}{-1:2:1}){0:1:0}{0:1:0};
//            nicht legal: mat3{-1:2:1}{-1:2:1} {0:1:0}{0:1:0}
```

### 7.7.6 Selektion von Strukturelementen

Mit Hilfe des Selektionsoperators kann auf ein Element eines Matrix- oder Vektortyps zugegriffen werden.

Selektionen können auf alle Werte dieser Typen angewandt werden. Die Anwendung des Selektionsoperators auf einem Vektor liefert das entsprechende Element zurück, während die Selektion auf einer Matrix den entsprechenden Zeilenvektor zurückgibt. Um auf ein Matrixelement zuzugreifen, muss der Selektionsoperator zweimal angewendet werden. Liegt der zur Laufzeit berechnete Index außerhalb des gültigen Bereiches, terminiert das Programm mit einem Fehler. Im folgenden Listing 30 sehen Sie Beispiele für die verschiedenen Selektionen.

Listing 30: Beispiele für Selektionen

```
var matrix<int>[3][3] mat;
mat = [[1,2,3],
       [4,5,6],
       [7,8,9]];

var vector<int>[3] vec;
// Die erste Zeile der Matrix wird selektiert.
vec = mat[0];

var int i;
// Der erste Eintrag des Vektors wird selektiert.
i = vec[0];
// Der erste Eintrag des dritten Vektors wird selektiert.
i = mat[2][0];
```

## 7.8 Selektion von Record-Elementen

Ein Element eines Records wird mittels **`@<elementname>`** referenziert. Der Operator kann nur auf Ausdrücke angewendet werden, deren Typ ein Record-Typ ist, welcher ein entsprechend benanntes Element enthält.

Listing 31: Beispiel einer Recordelementselektion am Typ aus Listing 7

```
val myRecord x = @myRecord[1.0, [[1, 2], [3, 4]], 10];
val int y = 2 * x@i;
// y hat den Wert 20
```

## 7.9 Sonderfall: Unäre Operatoren

Die unären Operatoren (‘-’, ‘~’, ‘!’, sowie die Dimensionsoperatoren) können nicht mehrfach hintereinander auftreten. Ein Ausdruck wie z.B. `!!!(a < b)` ist nicht legal in MAVL.

## 7.10 Operatorpräzedenzen

MAVL verwendet Operatorpräzedenzen, um eine eindeutige Auswertungsreihenfolge für zusammengesetzte Ausdrücke zu definieren. Ein bekanntes Beispiel für eine solche Präzedenz ist die Punkt-vor-Strich-Regel.

Tabelle 3 teilt die Operatoren in Präzedenzebenen ein. Innerhalb einer Ebene werden die Operatoren gemäß ihrer Assoziativität entweder links-lastig (z.B. a + b - c = (a + b) - c) oder rechts-lastig (z.B. a ^ b ^ c = a ^ (b ^ c)) gruppiert. Geklammerte Ausdrücke binden stärker als alle Operatoren, übertrumpfen also die Operatorpräzedenzen.

| Level | Operatoren | Beschreibung |
|---|---|---|
| 1 | `@` | Selektion von Record-Elementen |
| 2 | `[ ]` | Elementselektion |
| 3 | `{}{}, {}` | Submatrix, Subvektor |
| 4 | `.rows, .cols, .dimension` | Dimensionsoperatoren |
| 5 | ~ | Matrix-Transponierung |
| 6 | # | Matrixmultiplikation |
| 7 | `.*` | Skalarprodukt |
| 8 | ^ | Potenz |
| 9 | - | Unäres Minus |
| 10 | *, / | Multiplikation und Division |
| 11 | +, - | Addition und Subtraktion |
| 12 | <, >, <=, >=, ==, != | Vergleichsoperatoren |
| 13 | ! | Logische Negation |
| 14 | & | Logisches Und |
| 15 | \| | Logisches Oder |
| 16 | ? : | Ternärer Operator |

Tabelle 3: Übersicht aller Operatoren geordnet von höchster Präzedenz (Level 1) zu niedrigster Präzedenz (Level 16)

## 7.11 Beispiele

Im Folgenden Listing 32 finden Sie Beispiele unterschiedlich komplexer Ausdrücke:

Listing 32: Verschiedene Ausdrücke in MAVL

```
// Trivialer Ausdruck in Form eines einzelnen int-Literals
val int a = 1;

// Verschachtelter Ausdruck mit Funktionsaufruf
val int i = 1 + (2 * 3) + increment(3);

// Komplexer Ausdruck mit verschiedenen Typen
val bool b = 3 > 2 + 2 & 2 * increment(a) > 1;
// Schritt 1: increment(a) liefert int
// Schritt 2: 2 * increment a liefert int
// Schritt 3: 2 + 2 liefert int
// Schritt 4: 3 > 4 liefert bool
// Schritt 5: 2 * increment(a) > 1 liefert bool
// Ergebnis: Konjunktion zweier booleschen Werte liefert bool

// Einfache Verwendung des ternaeren Operators ?:
val int e = b ? 10 : 20 + 37;

// Geschachtelte Verwendung des ternaeren Operators ?:
val int d = false ? 10 : (a>e ? a : e);
// Schritt 1: false liefert bool
// true -> Schritt 2: Wert des ? Operators ist 10
// false -> Schritt 2: a>e liefert bool
//   true -> Schritt 3: a liefert int
//   false -> Schritt 3: e liefert int
// Ergebnis: Aeusserer ?-Operator liefert einen Integer
```