Module 8 Lijsten en verzamelingen

Onderwerp	Expressierijen, lijsten, verzamelingen en arrays.
Voorkennis	Module 1, 25, 26.
Expressies	`exprseq, seq, { }, [ ], nops, op, NULL, list, set, union, intersect, member, minus, map, zip, sort, select, remove, has, convert, Array, listlist`
Bibliotheken	`ListTools`
Zie ook	Module 9, 13, 14, 27, 29.

8.1 Expressierijen, lijsten en verzamelingen

Een collectie van objecten kunnen we in Maple bij elkaar groeperen door ze in een expressierij, lijst of verzameling onder te brengen.

Expressierijen. Een *expressierij* (Engels: *sequence*) wordt gevormd door een aantal expressies gescheiden door komma's, zoals

```
a, b, c
a, x+y, 4, x^3
a1, a2, a3, a4, a5
1, 4, 9, 16, 25
f, f, f, f
```

We kunnen zo'n rij ook toekennen aan een variabele:

```
A := 1,4,9,16,25;
```

`exprseq` De variabele `A` heeft dan het type `exprseq` (*expression sequence*)[16]. Wanneer er in het rijtje enige structuur zit, kunnen we het meestal `seq` gemakkelijk genereren met de procedure `seq`. Bijvoorbeeld de laatste drie rijtjes van hierboven hadden we kunnen maken met

```
seq(a||i, i=1..5);
seq(i^2, i=1..5);
seq(f, i=1..4);  of met  f$4;
```

`$` De constructie `f$4` is dus niets anders dan 'de sequence, bestaande

[16] In feite is `exprseq` het type dat teruggegeven wordt door `whattype`. Echter, `exprseq` is niet een type dat herkend wordt door de procedure `type`. Dat komt omdat het commando `type(A, exprseq)` door Maple wordt geïnterpreteerd als `type(1,4,9,16,25, exprseq)`, en dan klaagt het dat `type` met te veel parameters wordt aangeroepen.

uit 4 keer een `f`'.[17]
In de bovenstaande voorbeelden betekent `i=1..5` dat `i` achtereenvolgens $1, 2, 3, 4, 5$ wordt. Dit gaat dus precies zoals bij `add` (zie §6.8). Het enige verschil is dat `add` plustekens tussen de elementen zet, en `seq` komma's. Het is ook mogelijk `seq` een derde argument mee te geven dat de stapgrootte aangeeft.

```
seq( i^2, i=0.1..0.9, 0.2 );
```

resulteert in

$$0.01,\ 0.09,\ 0.25,\ 0.49,\ 0.81$$

Bij `add` hebben we die mogelijkheid niet.

Het j^{e} element uit zo'n rijtje met de naam `A` wordt aangegeven met `A[j]`. Het gebruik van deze notatie noemen we *indicering*; in Maple-uitvoer komt zo'n variabele te voorschijn als A_j (zie de voorbeeldsessie op blz. 393). Het getalletje `j` heet de *index* van het element `A[j]`. De variabele `A[j]` is van het (basis-)type `indexed`, net als `symbol` een subtype van het type `name`.
De argumenten van `seq` worden niet eerst volledig geëvalueerd (zie opgave 8.6).

Lijsten en verzamelingen. Er zijn in Maple niet zoveel bewerkingen mogelijk op expressierijen. Hiervoor moeten we er meer structuur in aanbrengen, bijvoorbeeld door er een *lijst* of *verzameling* van te maken. Dit gaat door er rechte haken (`[ ]`), respectievelijk accolades (`{ }`) omheen te zetten. Het verschil tussen lijsten en verzamelingen is dat in een *verzameling* elk element slechts eenmaal wordt opgenomen, en dat de ordening niet van tevoren vastligt. In een *lijst* daarentegen kan een element vaker dan één keer voorkomen en de volgorde van de elementen ligt vast.

`[ ]`
`{ }`

> **!** Het essentiële verschil tussen lijsten en verzamelingen enerzijds en expressierijen anderzijds is dat een lijst of verzameling door Maple als één object wordt beschouwd, terwijl van een expressierij elk element een apart object is.

Net als bij expressierijen worden de elementen van lijsten en verzamelingen via indicering aangegeven. Bij lijsten is dat uiteraard zinvol, bij verzamelingen haast nooit. Omdat de volgorde van de elementen in een verzameling niet vastligt, zal `A[2]` in feite betekenen: 'een willekeurig element van de verzameling `A`'.

[17] Het `$`-teken kan ook in andere gevallen gebruikt worden als alternatief voor de `seq`-procedure, maar dit vereist soms wat meer oplettendheid. Raadpleeg `?$` voor nadere informatie.

nops
op

Elementen van een expressierij, lijst of verzameling. Het *aantal* elementen in een lijst of verzameling vinden we door de procedure `nops`. (hoe zou dat in een expressierij moeten? zie opgave 8.2). De procedure `op` geeft deze elementen in de vorm van een expressierij. Het commando `op(A)` haalt dus als het ware de haken van de lijst of verzameling `A` weg.
Als `A` een expressierij, lijst of verzameling is dan is `A[j]` het j^{de} element van `A`. De index j mag ook negatief zijn, zo is `A[-1]` het laatste element, `A[-2]` het voorlaatste, enzovoort. Ook kunnen we bijvoorbeeld de eerste drie elementen van `A` krijgen met `A[1..3]`. *Alle* elementen van `A` verkrijgt men met `A[]`, dit heeft dus hetzelfde effect als `op(A)`

Voorbeeldsessie

Rij:

```
> A := seq(i^2, i=-5..5);
```

$$A := 25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25$$

Verzameling:

```
> B := {A};
```

$$B := \{0, 1, 4, 9, 16, 25\}$$

Lijst:

```
> C := [A];
```

$$C := [25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25]$$

```
> A[2];
```

$$16$$

```
> B[2];
```

$$1$$

```
> C[2..4];
```

$$[16, 9, 4]$$

```
> nops(B);
```

$$6$$

```
> op(B);
```

$$0, 1, 4, 9, 16, 25$$

```
> B[];
```

$$0, 1, 4, 9, 16, 25$$

```
> nops(C);
```

$$11$$

```
> op(C);
```

$$25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25$$

Aan een lijst (of verzameling) kunnen niet zonder meer elementen worden toegevoegd:

```
> C[12] := 100;
```

```
Error, out of bound assignment to a list
```

```
> C := [C[], 100]:  C[12];
```

$$100$$

```
> op(0,C);
```

$$list$$

```
> whattype(C);
```

$$list$$

```
> type( C, list(integer) );
```

$$true$$

```
> type( C, list({integer,constant}) );
```

$$true$$

Toelichting

Merk op dat bij het maken van de verzameling `B` automatisch de 'dubbele' elementen eruit gehaald zijn. In een verzameling ligt ook de volgorde van de elementen niet vast. Het is dan ook volstrekt niet denkbeeldig dat `B[2]` een volgende keer een andere waarde heeft dan de eerste keer.
Expressierijen zijn handig als we elementen (achteraan) willen *toevoegen*. In het voorbeeld hebben we aan lijst `c` het element 100 toegevoegd door er eerst met `C[]` een expressierij van te maken (`op(C)` had ook gekund). Deze kunnen we uitbreiden door het nieuwe element er met een komma achter te zetten. Ten slotte wordt van die aldus uitgebreide expressierij weer een lijst gemaakt door er vierkante haken omheen te zetten. *Tussen*voegen is iets lastiger maar gaat ook via expressierijen, zie opgave 8.3.

`set` `list` Verzamelingen zijn van het type `set`, en lijsten zijn van het type `list`. Dit zijn voorbeelden van zogenaamde *structured types*, dat wil zeggen dat we gemakkelijk kunnen nagaan of de elementen ervan van een bepaald type zijn. In het voorbeeld is getest of `C` een lijst is, waarvan alle elementen integers (dan wel constanten) zijn. ◇

`convert` Een verzameling `B` kunnen we omzetten naar een lijst met het commando `convert(B,list)` en omgekeerd de lijst `C` naar een verzameling met `convert(C,set)`. Van een *sequence* kunnen we een verzameling of lijst maken door er de juiste haken omheen te zetten; van een verzameling of lijst wordt een sequence gemaakt met `op`.

Overigens kan het best voorkomen dat de elementen in een lijst of verzameling van verschillend type zijn. Met name kan het gebeuren

dat een element van een lijst of verzameling zelf weer een lijst of verzameling is. Zie de volgende paragrafen voor voorbeelden.

{} De *lege verzameling* respectievelijk *lege lijst* geven we aan met {} respectievelijk met `[]`. De *lege sequence* kunnen we weergeven met `NULL`.

`NULL, []`

8.2 Bewerkingen op verzamelingen

We kunnen op *verzamelingen* de gebruikelijke operaties uitvoeren:

operatie	Maple-invoer
$A \cup B$	`A union B`
$A \cap B$	`A intersect B`
$A \setminus B$	`A minus B`
$x \in A$	`member(x,A)` of `x in A`

De Maple-woorden `union` voor de *vereniging*, `intersect` voor de *doorsnede*[18] en `minus` voor het *verschil* van verzamelingen zijn voorbeelden van *operatoren* (net als bijvoorbeeld `+` en `*`), dat wil zeggen dat ze *tussen* de operanden geplaatst worden. Men kan ze ook als 'gewone' Maple-commando's gebruiken en dat is vooral handig als bijvoorbeeld de vereniging van méér dat twee verzamelingen bepaald moet worden. Voor $A \cup B \cup C$ wordt dat:

```
`union`(A,B,C);
```

De *back-quotes* zijn nodig als een operator (zoals `union`) als procedurenaam wordt gebruikt. Zo kan men in plaats van `1+1` dus ook `` `+`(1,1) `` invoeren.
Het commando `member` of `x in A` werkt ook bij lijsten.

Voorbeeldsessie

```
> a := {1};
```

$$a := \{1\}$$

```
> b := {2};
```

$$b := \{2\}$$

```
> ab := a union b;
```

$$ab := \{1, 2\}$$

```
> power_set := {{}, a, b, ab};
```

$$power\_set := \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$$

[18] Denk eraan dat het hier om de doorsnede van *verzamelingen* gaat. Zie voor de doorsnede van *lineaire ruimten* Module 14.

```
> power_set[3];        # een verzameling
```

$$\{2\}$$

```
> {1} in power_set;
```

$$\{1\} \in \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$$

```
> is( {1} in power_set );
```

$$true$$

```
> is( 1 in power_set); # nee, 1 is geen element van power_set
```

$$false$$

```
> is(1 in power_set[2]);
```

$$true$$

Toelichting

We zien dat we de machtsverzameling van de verzameling $\{1, 2\}$ in Maple kunnen representeren. De elementen van `power_set` zijn zelf weer verzamelingen. Om na te gaan of $x \in A$ waar is of niet, kunnen
`is` we het commando `is` gebruiken. ◇

8.3 Bewerkingen op lijsten

Bewerken van alle elementen van één lijst. Het belangrijkste hulpmiddel voor bewerking van lijsten (en verzamelingen) is de
`map` procedure `map`. Het doel van `map` is het gemakkelijk te maken een procedure toe te passen op alle elementen van een lijst of verzameling. In het algemeen is `map` bedoeld om een functie toe te passen op alle operanden van een expressie en de operanden van een lijst of verzameling zijn juist de *elementen* ervan. Zie §26.6 voor meer voorbeelden. De algemene syntax van `map` is

```
map(f, A, eventueel een rij argumenten voor f)
```

Hierbij is `f` de procedure die moet worden toegepast op alle elementen in de lijst of verzameling `A`. Na de tweede parameter in `map` kunnen eventueel extra argumenten voor `f` worden meegegeven. Zie het volgende voorbeeld.

Voorbeeldsessie

```
> lijst := [seq(a||i, i=1..3)]:
> f := (x,n) -> x^2 + n:
> flijst := map(f, lijst, 5);
```

$$flijst := [a1^2 + 5,\ a2^2 + 5,\ a3^2 + 5]$$

Toelichting

Merk op dat nu `5` als derde parameter in `map(f,lijst,5)` is meegegeven. De betekenis van dit commando is dan:

> Bereken `f(lijst[i],5)` voor `i` = 1 t/m 3 (het aantal elementen in de lijst) en zet de resultaten weer in een lijst. ◇

Vele ingebouwde Maple-procedures werken automatisch alsof er een `map`-commando is gegeven. Zo zal `simplify(A);` ook werken op alle elementen van `A` als `A` een lijst is, terwijl strict genomen het commando `map(simplify,A);` gegeven zou moeten worden als men alle elementen van `A` vereenvoudigd wil hebben.

`zip`

Combineren van twee lijsten. Enigszins te vergelijken met `map` is de procedure `zip` waarmee bewerkingen op *twee* lijsten kunnen worden uitgevoerd. De syntax is:

```
zip(f, A, B);
```

Hierbij zijn `A` en `B` lijsten met evenveel elementen en `f` een procedure met twee formele parameters (functie van twee variabelen). Door het bovenstaande commando wordt een lijst gemaakt met `f(A[i],B[i])` als elementen.

Voorbeeldsessie

```
> A := [4,2,1,3]:    C := [500,400,200,300]:
```

Combineren van lijsten:

Voorbeeld (1) Elementen van A optellen bij die van C:

```
> f := (x,y) -> x+y:  zip(f, A, C);
```

[504, 402, 201, 303]

Dit is hetzelfde als:

```
> somlijst := zip( `+`, A, C );
```

somlijst := [504, 402, 201, 303]

Bij deze eenvoudige functie accepteert Maple zelfs:

```
> A + C;
```

[504, 402, 201, 303]

Voorbeeld (2) Combineren tot getallenparen:

```
> AC := [ seq( [A[i],C[i]], i=1..nops(A) ) ];
```

AC := [[4, 500], [2, 400], [1, 200], [3, 300]]

Hetzelfde met het zip-commando:

```
> zip( (a,c)->[a,c], A, C );
```

[[4, 500], [2, 400], [1, 200], [3, 300]]

Toelichting

In voorbeeld (1) is de `zip`-functie gebruikt om de elementen van de lijsten `A` en `C` bij elkaar op te tellen. Zie blz. 101 voor de betekenis van `‘+‘`.
Het praktische nut van voorbeeld (2) is het volgende. Als `A` een lijst van x-waarden is en `C` de lijst met *bijbehorende* y-waarden, dan kunnen we `A` en `C` combineren tot een lijst van coördinaatparen die we met `pointplot` kunnen tekenen (zie §9.3). De procedure `seq` is hier gebruikt om de lijsten `A` en `C` te combineren tot de lijst van lijsten `AC`. Eenvoudiger gaat dit met `zip`, waarbij de op `A` en `C` toe te passen procedure een element van de ene lijst samen met een element van de andere lijst tussen vierkante haakjes zet. ◇

`sort`

Sorteren. Voor het *sorteren* bestaat de procedure `sort`. Het commando `sort(A)` ordent de elementen van de lijst `A` in opklimmende grootte. Om volgens een ander criterium te sorteren kan `sort` ook met een tweede argument worden gebruikt: `sort(A,f)`. Hierbij is `f` een procedure met twee parameters die voor `f(x,y)` de waarde *true* afgeeft als een element met de waarde `x` vóór een element met de waarde `y` moet komen. Met andere woorden, als `sort` zónder tweede argument wordt aangeroepen, wordt gehandeld alsof `f := (x,y) -> evalb(x<y)`.[19]

Voorbeeldsessie

```
> A := [4,2,1,3]:    C := [500,400,200,300]:
  AC := zip( (a,c)->[a,c], A, C );
```

$$AC := [[4,500],[2,400],[1,200],[3,300]]$$

Sorteren:

```
> sort(C);
```

$$[200,300,400,500]$$

```
> sort( A, (i,j)->evalb(i>j) );
```

$$[4,3,2,1]$$

```
> sort(AC);
```

$$[[1,200],[2,400],[3,300],[4,500]]$$

Sorteert op het eerste element.

Als we op het *tweede* element willen sorteren:

```
> sort( AC, (x,y) -> evalb(x[2]<y[2]) );
```

$$[[1,200],[3,300],[2,400],[4,500]]$$

[19] Het commando `evalb` betekent: eval*uate as* b*oolean*, zie §27.2.

Toelichting

Als de elementen van de te sorteren lijst geen getallen[20] zijn, zoals bij de lijst `AC` het geval is waar de elementen *lijsten* zijn, werkt `sort` op een min of meer voor de hand liggende manier. In het voorbeeld wordt gesorteerd op het *eerste* element van `AC[i]`. Als we iets anders willen, dan moeten we het afwijkende 'sorteerprincipe' vermelden. ⋄

Deellijsten. Ten slotte zijn er nog twee procedures waarmee we uit een gegeven lijst (of verzameling) een *deel*lijst (of -verzameling) kunnen maken. De wijze waarop de procedure `select` wordt aangeroepen is precies als bij `map`. Met `select(f,A)` worden de elementen uit de lijst `A` gehaald waarvoor `f(A[i])` waar is (de waarde *true* oplevert). Vergelijkbaar daarmee is `remove`, met de voor de hand liggende werking.

`select`

`remove`

Voorbeeldsessie

```
> B := [4,2,1,a,3];
```

$$B := [4, 2, 1, a, 3]$$

```
> select( n -> n>2, B );
```

```
Error, selecting function must return true or false
```

```
> A := remove( type, B, symbol );
```

$$A := [4, 2, 1, 3]$$

Hetzelfde wordt bereikt met

```
> select( type, B, numeric );
```

$$[4, 2, 1, 3]$$

```
> select( n -> n>2, A );
```

$$[4, 3]$$

```
> select( type, A, even );
```

$$[4, 2]$$

Toelichting

Als we uit lijst `B` alleen de getallen > 2 willen hebben, dan lukt dat niet omdat een van de elementen geen waarde heeft. Dat verwijderen we eerst met het commando `remove`:

```
remove( type, B, symbol )
```

[20] Overigens werkt `sort` ook wel als de lijst bijvoorbeeld uit namen van variabelen (dus zonder waarde) bestaat; de elementen worden dan alfabetisch geordend. Zie `?sort` voor details.

heeft het effect: verwijder uit de lijst `B` alle elementen waarvoor `type(B[i],symbol)` de waarde *true* oplevert. Nu lukt het wel. ◇

Het commando `remove` kan ook worden gebruikt om niet-reële oplossingen 'automatisch' uit een verzameling oplossingen van een stelsel vergelijkingen te verwijderen (zoals aangekondigd in §5.5). We nemen het stelsel vergelijkingen uit de voorbeeldsessie op blz. 64.

Voorbeeldsessie

```
> vgln := {x^2-y=0, y^2-x=0}:
> s1 := solve( vgln, {x,y} );
```

$$s1 := \{x = 0,\ y = 0\},\ \{x = 1,\ y = 1\},$$
$$\{x = \mathrm{RootOf}(\_Z^2 + \_Z + 1),\ y = -1 - \mathrm{RootOf}(\_Z^2 + \_Z + 1)\}$$

```
> s2 := map(allvalues,[s1]);
```

$$s2 := [\{x = 0,\ y = 0\},\ \{x = 1,\ y = 1\},$$
$$\{x = -\frac{1}{2} + \frac{1}{2}\,I\,\sqrt{3},\ y = -\frac{1}{2} - \frac{1}{2}\,I\,\sqrt{3}\},$$
$$\{x = -\frac{1}{2} - \frac{1}{2}\,I\,\sqrt{3},\ y = -\frac{1}{2} + \frac{1}{2}\,I\,\sqrt{3}\}]$$

```
> s := remove( has, s2, I );
```

$$s := [\{x = 0,\ y = 0\},\ \{x = 1,\ y = 1\}]$$

Toelichting

De oplossingen worden gegeven in de vorm van de expressierij `s1`. Willen we op alle oplossingsparen `allvalues` toepassen, dan moeten we ervoor zorgen dat ze operanden van één object worden. Dat doen we door er vierkante haken om te zetten, waardoor het elementen van een lijst worden. Nu wordt `s2` een lijst met 4 elementen, waarvan er twee niet-reëel zijn. Deze worden gekarakteriseerd doordat er een `I` in staat. Of een bepaalde subexpressie in een expressie voorkomt, kunnen we onderzoeken met de functie `has`[21]. Dat betekent dat `has(s2[1],I)` *false* oplevert en `has(s2[4],I)` *true*. ◇

`has`

`ListTools` Tenslotte kunt u nog gebruik maken van de bibliotheek `ListTools` dat een aantal bruikbare procedures voor manipulatie met lijsten bevat. Raadpleeg `?ListTools` voor meer informatie.

[21] Meer daarover aan het eind van Module 26.

8.4 Arrays

In sommige gevallen kan het onhandig, dat wil zeggen: een mogelijke bron van vergissingen, zijn dat de nummering van de elementen van een lijst bij 1 begint. Men zou ze bijvoorbeeld als $a_0, a_1, a_2, \ldots$ willen aanduiden, inplaats van als $a_1, a_2, a_3, \ldots$. In dat geval kan een `Array` worden gebruikt inplaats van een lijst.

`Array`

Voorbeeldsessie

```
> A := Array(0..3, [p,q,r,s]):
> A[0], A[2];
```

$$p,\ r$$

```
> B := Array(-4..4):
> convert(B,list);
```

$$[0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0]$$

```
> B[-2]:=5:  B[0]:=c:  B[4]:=-9:
> convert(B,list);
```

$$[0,\ 0,\ 5,\ 0,\ c,\ 0,\ 0,\ 0,\ -9]$$

```
> B;
```

$$\mathrm{Array}(-4..4,\ \{(-2) = 5,\ (0) = c,\ (4) = -9\},\ datatype = anything,\ storage = rectangular,\ order = Fortran\_order)$$

Toelichting

In grote trekken kunnen de bedoelde `Array`'s op twee manieren worden gemaakt:

(1) Door aan een (eventueel al bestaande) lijst het laagste en hoogste 'rangnummer' toe te voegen, zoals bij Array `A` in het voorbeeld;
(2) Door eerst een 'lege' Array te maken, dat wil zeggen: een Array met allemaal nullen, en de gewenste waarden later in te vullen. Dit is bij Array `B` gebeurd.[22]

Van een `Array` kunnen we met een `convert`-commando gemakkelijk een lijst maken.
Aan de uitvoer bij het laatste statement kunnen we opmaken dat in de variabele `B` naast de relevante waarden, nog een aantal andere zaken wordt opgeslagen, de zogenaamde *attributen*. Deze zijn vooral van belang bij zeer grote, eventueel meerdimensionale, Arrays. In

[22] Meestal zal men daarvoor een `for`-lus gebruiken, zie Module 27.

feite zijn vectoren en matrices, die in Module 13 aan de orde komen, bijzondere gevallen van Arrays. ◇

8.5 Meerdimensionale arrays

Het type `Array` kunnen we ook gebruiken om een soort lijsten te maken waarvoor meer dan één index moet worden gebruikt om de elementen aan te geven. Het aantal indices dat we nodig hebben om de elementen aan te geven noemen we de *dimensie* van het array.

Voorbeeldsessie

```
> N := [[a,b],[c,d],[e,f]];
```

$$N := [[a, b], [c, d], [e, f]]$$

```
> M := Array(N);
```

$$M := \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

```
> op(M);
```

$$1..3, 1..2, \{(1, 1) = a, (1, 2) = b, (2, 1) = c, (2, 2) = d, (3, 1) = e, (3, 2) = f\}, datatype = anything, storage = rectangular, order = Fortran\_order$$

```
> P := [[[a,b],[c,d]],[[e,f],[g,h]],[[i,j],[k,l]]];
```

$$P := [[[a, b], [c, d]], [[e, f], [g, h]], [[i, j], [k, l]]]$$

```
> Q := Array(P);
```

$$Q := \begin{bmatrix} \text{1..3 x 1..2 x 1..2 Array} \\ \text{Data Type : anything} \\ \text{Storage : rectangular} \\ \text{Order : Fortran\_order} \end{bmatrix}$$

```
> P[2];
```

$$[[e, f], [g, h]]$$

```
> Q[2];
```

$$\begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

```
> Q[2,2,1];
```

$$g$$

```
> S := convert(Q,list);
```

$$S := [a, e, i, c, g, k, b, f, j, d, h, l]$$

```
> R := convert(Q,listlist);
```

$$R := [[[a, b], [c, d]], [[e, f], [g, h]], [[i, j], [k, l]]]$$

```
> evalb(P=R);
```

$$true$$

Toelichting

`N` is een lijst met drie elementen; elk element van `N` is zelf weer een lijst met twee elementen. Het array dat we van `N` maken is tweedimensionaal (wat ook te zien is aan de manier waarop Maple hem weergeeft); er zijn *twee* ranges voor de indices. In dit geval zouden we misschien eerder een gewone matrix hebben gemaakt, zie Module 13. Op een vergelijkbare manier kunnen we ook de driedimensionale array `Q` maken. Maple doet hier geen moeite meer om hem netjes weer te geven. Het volstaat met te melden wat voor soort ding `Q` is.

`listlist`

Om van een meerdimensionale array weer een lijst te maken, moeten we converteren naar `listlist`, een lijst van lijsten. Door het laatste statement wordt nagegeaan of `P=R` waar is of niet, met andere woorden: of de nieuwe lijst `R` gelijk is aan de oude lijst `P`. (Zie §27.2 voor nadere uitleg over het commando `evalb`.) ◇

Eén- en tweedimensionale Arrays dienen als bouwsteen voor de objecten Vector en Matrix, die in Module 13 ter sprake komen.
We kunnen een array ook maken door aan te geven tussen welke grenzen de indices mogen liggen. Dat is nodig als de ranges voor de indices niet bij 1 beginnen. Bijvoorbeeld

```
A := Array(2..3, -1..1);
```

De variabele `A` is nu een tweedimensionaal Array, waarbij we de beschikking hebben over de elementen

```
A[2,-1], A[2,0], A[2,1], A[3,-1], A[3,0], A[3,1]
```

Wanneer we nu aan al deze elementen een waarde willen toekennen, kunnen we dat doen door een reeks statements uit te voeren als `A[2,-1] := ...` enzovoort. Soms is het echter handiger dit direct te doen in de definitie van `A`. We demonstreren dit aan de hand van het volgende voorbeeld.

Voorbeeldsessie

```
> A := Array(2..3, -1..1):
> A[2,-1] := 1; A[2,0] := 6; A[2,1] := 7;
  A[3,-1] := 4; A[3,0] := 6; A[3,1] := 9;
```

$$A_{2,-1} := 1$$

$$A_{2,0} := 6$$

$$A_{2,1} := 7$$

$$A_{3,-1} := 4$$

$$A_{3,0} := 6$$

$$A_{3,\,1} := 9$$

Nu korter:

```
> A := Array(2..3, -1..1, [[1,6,7],[4,6,9]]);
```

$$A := \text{Array}(2..3,\ -1..1,\ \{(2,\ -1) = 1,\ (2,\ 0) = 6,\ (2,\ 1) = 7,\ (3,\ -1) = 4,\ (3,\ 0) = 6,\ (3,\ 1) = 9\},\ datatype = anything,\ storage = rectangular,\ order = Fortran\_order)$$

```
> f := x -> x^2;
```

$$f := x \rightarrow x^2$$

```
> B := map(f, A):
> A[3,0], B[3,0];
```

$$6,\ 36$$

```
> map(x->sqrt(x), A):
```

Toelichting

Bij de tweede (korte) definitie van `A` wordt als derde argument een lijst opgegeven, namelijk de lijst `[[1,6,7],[4,6,9]]`. De elementen van deze lijst zijn zelf weer lijsten. De eerste lijst, `[1,6,7]`, bevat de elementen `A[2,-1]`, `A[2,0]`, `A[2,1]`, waarbij dus de eerste index constant wordt gehouden (namelijk 2), en de tweede loopt van de minimaal toegestane tot de maximaal toegestane waarde (van -1 naar 1). Analoog bevat de tweede lijst de elementen `A[3,-1]`, `A[3,0]`, `A[3,1]`, waarbij dus weer de eerste index constant wordt gehouden (nu op 3) en de tweede varieert.
Ten slotte zien we dat de procedure `map` ook op Arrays kan worden toegepast. ◇

> **!** Let op dat u steeds `Array` (met een hoofdletter) gebruikt, en *niet* `array` (kleine letter). Het type `array` bestaat namelijk óók, vandaar dat u geen foutmelding of waarschuwing krijgt na zoiets als `array(2..5, [1,2,3,4])`. Omdat de evaluatieregels van een `array` heel anders zijn dan die van een `Array`, geeft dat onverwachte vervelende complicaties.

8.6 Variabelen als a_i

Als we in Maple (na een `restart`) `a[i];` invoeren, dan reageert het met a_i. Dat maakt het verleidelijk om namen als `a[1]`, `a[2]`, enzovoort gewoon als variabelen te gebruiken. We noemen dat *geïndiceerde*

variabelen. Meestal kan dat geen kwaad, maar men moet zich ervan bewust zijn dat, door één keer de naam `a[1]` te gebruiken, er door Maple een object `a` wordt aangemaakt. Dit is een zogenaamde *tabel* (*table*), in grote trekken net zoiets als een lijst, zie §8.7 voor meer uitleg.

Voorbeeldsessie

```
> a[i], a[j] := 5,6;
```

$$a_i,\, a_j := 5,\, 6$$

```
> vgl := 2*a[1] - 3*a[2] = 4;
```

$$vgl := 2\,a_1 - 3\,a_2 = 4$$

```
> s := solve( {vgl}, {a[1]} );
```

$$s := \{a_1 = \frac{3}{2}\,a_2 + 2\}$$

```
> subs( s, vgl );
```

$$4 = 4$$

```
> assign(s);
> a[1];
```

$$\frac{3}{2}\,a_2 + 2$$

```
> eval(a);
```

$$\mathrm{table}([1 = \frac{3}{2}\,a_2 + 2,\, j = 6,\, i = 5])$$

```
> a := 2/3:  solve( {2*a[1] - 3*a[2] = 4}, {a[1]} );

Warning, solving for expressions other than names or functions
is not recommended.
```

$$\left\{\left(\frac{2}{3}\right)_1 = \frac{3}{2}\left(\frac{2}{3}\right)_2 + 2\right\}$$

Toelichting

We zien dat we a_i en a_j gewoon een waarde kunnen geven, en a_1 uit een vergelijking in a_1 en a_2 kunnen oplossen. Uit de reactie op het commando `eval(a)` blijkt dat op dat moment de variabele met de naam `a` een tabel is, met 'ingangen' 1, i en j, waarin $a_1 = \frac{3}{2}\,a_2$, $a_i = 5$ en $a_j = 6$. Als de variabele `a` (zonder index) een waarde heeft, dan kunnen er rare dingen gebeuren. ◇

8.7 Tables

`table` In allerlei opzichten is een `table`. flexibeler dan een lijst of een Array.

De index in een `table` hoeft namelijk niet van het type `integer`, dus geen geheel getal te zijn. In het volgende voorbeeld maken we een tabel met 'indices' `a`, `b`, `0.25` en `x^2`:

Voorbeeldsessie

```
> T[a] := 1:    T[b] := x^2:    T[0.25] := 1/4:
  T[x^2] := "een even functie":
```

```
> T[0.25];
```

$$\frac{1}{4}$$

```
> T[0.250];
```

$$T_{0.250}$$

```
> f := x^2:  T[f];
```

"een even functie"

```
> T[a] := 'T[a]':  T[z] := "toegevoegd":
```

```
> T;
```

$$T$$

```
> eval(T);
```

table([$0.25 = \frac{1}{4}$, x^2 = "een even functie", $b = x^2$, z = "toegevoegd"])

```
> convert(T,set):  S := %;
```

$$S := \{x^2, \text{ "een even functie"}, \text{ "toegevoegd"}, \frac{1}{4}\}$$

Toelichting

Een tabel kan worden gemaakt door voor elke 'index' aan te geven welke waarde erbij hoort. Men spreekt bij tables trouwens liever van *entries* dan van indices. Voor die entries kan (bijna) alles dienen, en er wordt niets vereenvoudigd: `T[0.25]` is blijkbaar niet hetzelfde als `T[0.250]`.
Merk op dat `T;` niet volstaat om de inhoud van `T` te zien te krijgen. Daarvoor is een uitdrukkelijke `eval`-opdracht nodig. Dit is typisch voor tables, en wordt in Module 25 uitgelegd.
In het voorbeeld is ook te zien hoe entries verwijderd kunnen worden; entries toevoegen is net zo gemakkelijk.
Met `convert` is van een tabel een verzameling of een lijst te maken. Ga zelf na dat `op(T)` niet werkt om de elementen van `T` in de vorm van een expressierij te krijgen. ◇

> **!** Men moet erop verdacht zijn dat zodra er een toekenning als `T[x] := ...` wordt gedaan, er automatisch een table met de naam `T` wordt gemaakt – tenminste als `T` niet eerder als array of iets dergelijks was gemaakt.

Tables spelen een belangrijke rol als zogenaamde *remember-tables* van procedures. Dit wordt besproken in Module 29.

Opgave 8.1

We zagen dat twee verzamelingen verenigd konden worden met `union`. Stel dat `A` en `B` *lijsten* zijn, hoe zou u dan de lijst maken die je krijgt door alle elementen van `A` en `B` achter elkaar te zetten? (zogenaamde *concatenatie* van `A` en `B`.)

Opgave 8.2

Stel `A` is de expressierij `1,2,3,4`. Wat gaat er fout in

```
nops(A);
```

en waarom? Hoe kunnen we wel het aantal elementen in de rij bepalen?

Opgave 8.3

Voeg, met behulp van een Maple-commando, het element `a` in de lijst `A := [1,5,7,2,9,8,11]` tussen het derde en het vierde element.

Opgave 8.4

Maak van de lijst

```
A :=[[a,b],[c,d],...  ]
```

(`A` heeft onbekende lengte; vul zelf op de puntjes nog een aantal paren in) de lijst `[a,b,c,d,...]`
Aanwijzing: Gebruik bijvoorbeeld `seq` en `op`; met `map` gaat het ook.

Opgave 8.5

Hoe zou de opdracht `map(int,A,x)`, met `A` een lijst van onbekende lengte, kunnen worden uitgevoerd met uitsluitend gebruik van de procedures `seq`, `nops` en `int`?

Opgave 8.6

Beschouw de sessie

```
i := 5; seq(i^2, i=1..5);
```

Waarom kunnen we hieraan zien dat de argumenten van `seq` niet volledig worden geëvalueerd? Vergelijk met

```
i := 5; sum(i^2, i=1..5);
```