Inleiding Magma

Wieb Bosma

December 2005

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Computeralgebra

Een *computeralgebra*systeem is een pakket van programma's waarmee je op een computer 'algebra' kunt doen. In de praktijk moet je het begrip algebra ruim opvatten, maar het gebruik in deze context is vooral bedoeld om te benadrukken dat we *algebraïsche* manipulaties willen doen op wiskundige objecten, en niet zozeer numerieke berekeningen. Als synoniem wordt wel de uitdrukking *symbolisch rekenen* gebruikt. Om de gedachten te bepalen: in computeralgebra ligt de nadruk meer op het 'formele' rekenen met *discrete* objecten (zoals gehele getallen, en breuken, maar vooral abstractere objecten als polynomen, elementen in abstracte groepen, ringen, enz.) dan op bijvoorbeeld het rekenen met reële of complexe getallen.

In de praktijk hebben de meeste computeralgebrapakketten *ook* faciliteiten om met (functies van) reële getallen te werken, simpelweg omdat dat in veel toepassingen onontbeerlijk is. Wij zullen die aspecten ook beslist niet willen negeren; aanvankelijk zullen we het pakket dat we gaan gebruiken vooral zien als een (zeer) veredelde calculator. Maar hoe verder je komt, hoe meer behoefte er ook gaat komen aan de meer symbolische kant.

1.2 Wat valt er te leren?

Wanneer je voor het eerst een computeralgebrapakket wilt gebruiken om een wiskundig probleem op te lossen, doet zich een aantal nieuwe problemen voor.

(i) de keuze van het pakket;
(ii) het opstarten (en afsluiten), en het invoeren van commando's en data;
(iii) het leren (gebruiken) van de taal van het pakket;
(iv) het vinden van de juiste commando's en datastrukturen;
(v) het vertalen van je probleem van wiskunde naar de programmeertaal.

We zullen op die vragen in deze inleiding in de aangegeven volgorde kort proberen in te gaan.

1.3 Vertalen

Het probleem om een wiskundige opgave te vertalen naar een voor de computer oplosbare vraag heeft natuurlijk vele facetten. Om te beginnen zou je eigenlijk al van de vraag af willen laten hangen welk pakket je kiest, omdat voor veel specifieke vragen speciale software is ontwikkeld. Is, zoals voor ons, die keuze eenmaal bepaald — wij zullen Magma gaan gebruiken, maar bekijk vooral ook Maple eens — dan moet je de wiskundige objecten in objecten omzetten bruikbaar binnen dat pakket, je afvragen op welke manier je probleem in principe opgelost zou kunnen worden, en dan de juiste commando's toepassen om dat door het systeem uit te laten voeren. Kortom alle andere vragen uit het rijtje komen aan bod. Bovendien moet je, als je een antwoord krijgt, dat antwoord weer weten te interpreteren in de context waarmee je begon. Een belangrijke regel is dat computeralgebrapakketten, vooral bij meer geavanceerde problemen, wel eens foute of incomplete antwoorden kunnen opleveren. Het is dus van belang om na te gaan dat het gevonden antwoord zinnig is. Bovendien komt het vaak voor dat je langer op een antwoord moet wachten dan je lief is. In dat geval verdient het aanbeveling te zoeken naar een manier om Magma te helpen, door een eenvoudigere methode te kiezen, of het probleem in stapjes op te lossen zodat je kunt zien waar Magma moeite mee heeft. Soms zal dan blijken dat je vraag niet goed (genoeg) gesteld was. Het kan ook zijn dat je beter een numerieke dan een symbolische methode kunt proberen, enzovoorts. Natuurlijk weet je in het begin voor veel van de moeilijkheden die zich voordoen geen oplossing.

1.4 Doel

De bedoeling is dat je na deze inleidende werkcolleges een aantal eenvoudige taken met Magma kunt uitvoeren, zoals het doen van sommige standaardberekeningen die in de colleges Lineaire Algebra of Getallen voorkomen. Bovendien is het de bedoeling dat je tegen die tijd zo goed de weg kent dat je zonder begeleiding in staat bent steeds ingewikkeldere programma's te schrijven.

In de volgende hoofdstukken worden heel veel commando's alleen maar kort genoemd, zonder precieze uitleg over het gebruik. De bedoeling is dat de naam je voldoende houvast geeft om met de online help verder uit te vinden hoe de functie van dienst kan zijn.

1.4.1 Voorbeeld

In het tentamen Lineaire Algebra 1 kwam als opgave voor het oplossen van het lineaire stelsel vergelijkingen:

$$\begin{cases} x_1 + x_2 + x_3 = 2 \\ -x_1 + x_2 + 2x_3 = -1 \\ x_1 + x_3 = 0. \end{cases}$$

Dat is met Magma ook eenvoudig; het probleem is om de juiste commando's te vinden.

```
> K := RationalField();
> m := Transpose(Matrix(K, 3, 3, [1, 1, 1,  -1, 1, 2,  1, 0, 1]));
> v := Vector(K, [2, -1, 0]);
> s, N := Solution(m, v);
> s;
    ( 1  2 -1)
> N;
    Vector space of degree 3, dimension 0 over Rational Field
> s*m;
    ( 2 -1  0)
```

Vervolgens werd gevraagd voor welke waarden van α het stelsel

$$\begin{cases} x_1 + x_2 + x_3 = 2 \\ -x_1 + x_2 + 2x_3 = -1 \\ x_1 + \alpha x_2 + x_3 = 0. \end{cases}$$

geen oplossingen heeft. Ook daarmee heeft Magma geen moeite:

```
> F<a> := FunctionField(Rationals());
> m := Transpose(Matrix(F, 3, 3, [1, 1, 1,  -1, 1, 2,  1, a, 1]));
> v := Vector(F, [2, -1, 0]);
> s, N := Solution(m, v);
> s;
((5/3*a - 1)/(a - 1)   -2/(a - 1)   (1/3*a + 1)/(a - 1))
```

waarbij uit de output duidelijk blijkt dat er alleen geen oplossing is als $\alpha = 1$.

1.4.2 Voorbeeld

In het Analyseboek wordt bij Hoofdstuk 4 gevraagd een N te bepalen zodat voor alle $n \geq N$ geldt

$$2^{-n} + 3^{-n} + 4^{-n} < \frac{1}{365}.$$

In Magma kan dat bijvoorbeeld zo:

```
> n := 0;
> while (2^(-n)+3^(-n)+4^(-n) ge 1/365) do
>     n := n+1;
> end while;
> n;
      9
```

1.5 Gebruik

Een computeralgebrasysteem bestaat meestal uit meerdere componenten en kan op verschillende manieren gebruikt worden. We zullen hier beginnen met het interactieve gebruik, waarbij je commando's aan Magma geeft (door ze in te typen) en antwoorden terugkrijgt. Later, wanneer wat uitgebreidere programma's geschreven gaan worden, wil je ook wel met een teksteditor een programma in een bestand schrijven en Magma op dat bestand loslaten. Dat kan ook wel interactief (door het bestand in te lezen, zoals we binnenkort zien) of de tekst met knippen-en-plakken in de interactieve sessie in te voeren.

Voor de omgang met de ingebouwde functies (de *intrinsics*) en het bewerken van de variabelen beschikt een computeralgebrasysteem over een *programmeertaal* waarin je programma's schrijft. De taal van Magma is voornamelijk imperatief, hetgeen ongeveer betekent dat je waarden uitrekent (met een *intrinsic* of een zelf geschreven *user* functie) en die waarden dan met `:=` toekent aan een variabele (of *identifier*). In bovenstaand voorbeeld zag je dat gebeuren met de variabele n. De taal kent allerhande controlestructuren (zoals `while ... end while` boven), waarvan we er een aantal zullen leren gebruiken. Bovendien moeten we leren allerhande wiskundige objecten (specifieke *datastructuren*) in Magma te creëren. De taal en de datastructuren van Magma zijn met opzet zeer wiskundig van aard. Magma is veel minder dan bijvoorbeeld MAPLE geschikt voor gebruik door niet-wiskundigen.

In een interactieve sessie (na het opstarten van Magma, in het volgende hoofdstuk uitgebreider beschreven, maar gewoonlijk gedaan door `magma` te typen in een shell), geeft Magma je steeds een *prompt* om aan te geven dat je input kunt geven. Meestal is dat het symbool `>`. Als Magma aan het rekenen is kan het even duren voor de prompt weer verschijnt; eventuele input die je dan geeft wordt niet gelezen (behoudens de *interrupt* `<control C>`) tot de prompt weer verschijnt.

1.6 Versies, Omgeving

Magma is beschikbaar op verschillende 'soorten computers' (onder besturingssystemen als UNIX en LINUX, WINDOWS, en MACOS. In deze handleiding gaan we uit van een UNIX-achtige omgeving; interactie met de omgeving speelt voornamelijk een rol bij het opstarten, bij het creëren en gebruiken van bestanden voor Magma, en bij de lokatie en toegankelijkheid van hulpbronnen. Een enkele keer verwijzen we naar omstandigheden die specifiek zijn voor het gebruik van Magma in Nijmegen. Ook verschijnen er regelmatig nieuwe versies; deze tekst is gebaseerd op versie 2.12. Je ziet de versie bij opstarten in de 'banier'.

Hoofdstuk 2

Eerste stappen

2.1 Begin en eind

2.1.1 Begin

Om Magma te activeren kun je vanuit een shell het commando `magma` uitvoeren, of je kunt onder de *math* tools (die je window-manager vast ergens vertoont) Magma selecteren.

Eén manier om Magma te gebruiken, is als een veredelde calculator, waar je het antwoord op een vraag direkt hoopt terug te krijgen — dat is *interactief* gebruik. De kunst is dan de juiste commando's te vinden.

Bij niet-interactief gebruik schrijf je een kort of lang programma in een file en lees je dat in. Daartoe kun je het commando `load "`*filenaam*`"` gebruiken. Met de quotes kun je Magma (onder Unix) niet alleen een naam van een file in de huidige directory geven, maar ook een heel pad, op de gebruikelijke manier onder Unix.

Bij het opstarten kun je door middel van `magma -s` *filenaam* in plaats van `magma` direct een 'start-up' file in laten lezen (met bijvoorbeeld persoonlijk instellingen, definities, etc.).

2.1.2 Algemeenheden

De meeste programma's in Magma bestaan uit commando's die de toekenning van een waarde aan een variabele inhouden; dit gebeurt door middel van `:=` als in `x := 4;` elke stap wordt door een `;` afgesloten. Stappen mogen meerdere regels beslaan, maar er mogen ook meerdere stappen op 1 regel staan.

Een ander veel voorkomende stap is om Magma te vragen de waarde van een variabele te tonen; dat kan (voor de variabele 'x') simpelweg met `x;` of met `print x;`.

In een interactieve sessie kun je met de 'pijltjes' toetsen over de regel lopen en veranderingen aanbrengen; met een pijltje omhoog 'lopen' haalt eerdere regels terug. Als je dit midden in een regel doet loop je terug door regels die net zo beginnen als de huidige regel.

Met `SetLogFile` kun je alle commando's en uitvoer in een interactieve sessie naar een bestand wegschrijven. Dus met `SetLogFile("mijnLog")` schrijf je alles weg naar het einde van een bestand 'mijnLog' dat zonodig eerst aangemaakt wordt. Met `UnsetLogFile()` beëindig je het wegschrijven.

2.1.3 Eind

Je kunt de interactieve Magma-sessie afsluiten door `quit;` te typen, of `exit;`. Een alternatief is <Ctrl>-D, of, wanneer Magma in het midden van een berekening is, tweemaal snel achter elkaar <Ctrl>-C. In het uiterste geval kan het Magma-proces ook extern ge'kill'd worden.

Met een enkele <Ctrl>-C onderbreek je de berekening en komt er weer een prompt.

2.2 De filosofie van Magma

Belangrijk element van de filosofie in Magma is de getypeerdheid van objecten: elk object heeft een uniek bepaald *type* (dat je vrijwel altijd te zien kunt krijgen met de functie `Type`), dat bepaald wordt bij de creatie van het object, en slechts in uitzonderlijke omstandigheden kan veranderen. Zo kun je een veelterm $3 \cdot x^2 - 1$ in Magma niet zonder meer letterlijk intypen, omdat je tevoren eerst moet aangeven wat x eigenlijk is, en waar het object 'veelterm' leeft.

---

magma -s := print exit quit load SetLogFile UnsetLogFile <Ctrl>-C <Ctrl>-D

Heel vaak moet je daarom eerst een wiskundige structuur definiëren waarin het object zijn plaats kan krijgen; in dit voorbeeld zul je eerst de ring moeten definiëren waarin de veelterm leeft; dit moederobject (meestal met `Parent`) terug te vinden), bepaald ook welke operaties op het element toegestaan zijn.

```
> P<x> := PolynomialRing( Integers() );
> f := 3*x^2 - 1;
> Parent(f);
Univariate Polynomial Ring in x over Integer Ring
```

Een belangrijke uitzondering op de onveranderlijkheid van het type van een object bestaat in *typecoërcie*, waar een gedwongen verandering van type (en moederobject) plaatsvindt. Soms gebeurt dit automatisch in Magma (bijvoorbeeld bij de creatie van 'verzamelstructuren'), soms wil de gebruiker het zelf doen. Voor dat laatste is de operator `!` ingevoerd. Als simpel voorbeeld verandert in `Integers() ! (4/2);` het type van het resultaat van de deling van 4 door 2 van 'rationaal getal' in 'geheel getal'.

2.3 Documentatie

Alvorens te beginnen met de verdere beschrijving van het systeem is het wellicht nuttig een aantal tips te geven voor het verkrijgen van verdere hulp buiten deze handleiding om.

Er is een *Handbook of* Magma*-functions* beschikbaar bij Magma waarin (bijna) alle functionaliteit van Magma beschreven staat. Dit is meer een encyclopedisch werk (ongeveer 3800 pagina's en een index) dan een tekst waaruit je het gebruik van Magma kunt leren. Er zijn diverse manieren om toegang te krijgen tot dit *Handbook*.

(1) De tekstbestanden zijn in diverse vormen beschikbaar in subdirectories van de directory waar Magma zich bevindt. In Nijmegen is dat in `/vol/magma/pdf` bijvoorbeeld, voor `hb.pdf` (er zijn ook `.dvi`, en `.ps` versies), die je met de acrobat lezer met `acroread` (of met `xdvi`, en `ghostview` voor de andere versies) kunt lezen. Omdat de file zo groot is is er ook een versie (`hb1.pdf` tot en met `hb8.pdf`) die in stukken is gesplitst; in die versie staat het meest relevante materiaal voor deze Handleiding in `hb1` en `hb4`.

(2) Dezelfde bestanden kunnen ook met een web-browser bekeken worden; met het commande `magmahelp` in een shell start je een browser op die begint op de introductiepagina van deze faciliteit. De `.html` bestanden zijn te vinden in `/vol/magma/htmlhelp/`.

(3) Wanneer je eenmaal in Magma bezig bent kun je, wanneer je de 'prompt' ziet `??` typen om in de Magma *help-browser* terecht te komen. Dit laat je binnen de Magma-sessie zelf navigeren in de tekstbestanden waaruit ook de `.pdf` files en *htmlhelp* zijn opgebouwd. De *help-browser* legt zelf uit hoe dat moet (de eerste stap is het verzoek om `??` te laten volgen door `help`. Je kunt in de bestanden ongeveer navigeren als in een UNIX omgeving.). Je verlaat de *help-browser* door `quit` te typen (zonder `;`).

```
> ??
Magma Help Browser
Type help for more information
??> help
====================================================================
PATH: /browser
KIND: Overview
====================================================================
Introduction
------------
```

Type Parent !

```
You are reading about the Magma Help Browser.
To read this help page when you are in the Browser, type

        help

The Browser is entered by typing

        ??

and it lets you browse through the Magma Help tree.
When you are in the Browser, the prompt is  ??> .
To get from the Browser back to Magma, type

        quit

or

        <Ctrl>-D

at the Browser prompt.

Magma has a simpler help system, which you can access without entering
the Browser.  To find out about it, leave the Browser (as explained
above) and then type

        ?

While you are still in the Browser, you can type the single word
"walk" (without the quotes) followed by <Return> to view the next
help screen.  Keep typing "walk" (or "w") to see further screens.

[If you are in the Browser, type the single word "w" (without quotes)
followed by <Return> to view the next help screen.]
========================================================================
??> quit
>
```

(4) Er zijn ook enkele mogelijkheden om wat gerichter naar informatie te zoeken, vooral wanneer je al ongeveer weet wat je zoekt, binnen een lopende Magma-sessie.

(i) Wanneer je van het bestaan van een functie in Magma op de hoogte bent, maar bijvoorbeeld informatie over het preciese gebruik of de argumenten wilt hebben, kunnen we de naam typen gevolgd door `;`. Zo zal

```
Factorization;
```

je (misschien niet direct heel begrijpelijk) uitleggen welke argumenten precies zijn toegestaan en wat je als resultaat krijgt. De eerste regel

```
 (<RngIntElt> n) -> RngIntEltFact, RngIntElt, SeqEnum
```

zegt bijvoorbeeld dat je een `RngIntElt` (dat is een geheel getal)

?? ?

n als argument mag geven, en als resultaat drie objecten terugkrijgt. De uitleg

```
Attempt to find the prime factorization of the value of n; the
factorization is returned, together with the appropriate unit and
the sequence of composite factors which could not be factored.
```

vertelt wat die drie objecten zijn.

(ii) Soortgelijke informatie, maar geordend volgens de inhoud van de *help-browser*, krijg je door ?Factorization te typen; je krijgt dan een lijst van (in dit geval 32) plaatsen in de documentatie waar uitleg over de functie Factorization te vinden is, en ook of het om een Sectie uit het *Handbook* gaat, om de beschrijving van een Intrinsic functie, of een Example: zo krijg je na

```
?Factorization
```

onder 30 andere

```
18   I   /magma/ring-field-algebra/integer/creation/conversion/\
                FactorizationToInteger
```

te zien, hetgeen betekent dat je nu met

```
?18
```

uitleg over het gebruik van de gerelateerde functie FactorizationToInteger kunt krijgen.

(iii) Voor beide voorgaande opties moet je al weten naar welke naam van een functie je zoekt. Om je daarbij te helpen kun je als volgt zoeken naar het bestaan van namen voor functies in Magma. Wanneer je het begin van een mogelijke functienaam typt (die namen beginnen meestal met een Hoofdletter) en vervolgens de Tabulator toets <Tab> dan geeft Magma een lijst van alle functienamen die zo beginnen. Als er maar precies 1 mogelijkheid is om een bestaande functienaam te krijgen vult Magma het vervolg aan en wacht dan eventueel op een nieuwe <Tab> om alle mogelijke verlengingen te tonen. Zo leidt

```
> Fac<Tab>
```

tot

```
> Factor
```

en vervolgens

```
> Factor<Tab>
```

tot een lijst van ongeveer 20 functienamen die met Factor beginnen. Je wordt geacht vervolgens 1 toegestane mogelijkheid te typen (of een deel, weer gevolgd door <Tab>).

(iv) Een andere manier om mogelijke functie te vinden is door ListSignatures te gebruiken. Zo geeft ListSignatures(RngIntElt); je een (gigantische!) lijst van alle functies in Magma waar een geheel getal als (één van de) argument(en) is toegestaan.

Opgave 1. *W*elke andere functies beginnen met List? Wat doen ze?

Hoofdstuk 3

Eenvoudige datatypen

3.1 Getallen

Diverse soorten getallen zijn in Magma uitzonderlijke objecten omdat je ze letterlijk kunt creëren zonder eerst hun 'moederstructuur' aan te geven. Dat is niet alleen omdat je ergens moet kunnen beginnen met objecten te creëren, maar ook omdat er bij getallen meestal geen enkel misverstand bestaat over wat er bedoeld wordt. Het is dan ook gemakkelijk om gehele getallen, breuken, reële, en complexe getallen te maken en ermee te rekenen, met de operatoren `+, -, *, /, ^` die de gebruikelijke betekenis hebben.

3.1.1 Gehele getallen

Gehele getallen worden simpelweg letterlijk ingetypt, zoals bijvoorbeeld `-13`. Er is geen speciaal type van natuurlijke getallen. Soms is het nodig om $\mathbb{Z}$ (als ring van gehele getallen) expliciet in handen te hebben; daarvoor is `Integers()` geschikt.

Met `mod` kun je rekenen modulo een getal m.

Belangrijk voor latere constructies: de rijtjes gehele getallen van a tot b met stapjes ter grootte c maak je met `[ a..b by c]`. Als je de stapgrootte weglaat wordt deze op 1 gezet.

Opgave 2. *Creëer de gehele getallen -11, 0, 11 in* Magma *en wijs ze aan de variabelen x, y, z toe. Wat is de naam die* Magma *voor het type van deze objecten gebruikt, en wat is de naam van de moederstructuur?*

Opgave 3. *Op hoeveel verschillende manieren kun je legale uitdrukkingen van de vorm $a \circ b \diamond c$ als $a, b, c \in \{2, 3, 4\}$ en $\circ, \diamond$ beide één van bovenstaande operatoren mogen zijn? Geef twee antwoorden:*

(i) *wanneer a, b, c en $\circ, \diamond$ verschillend moeten zijn;*

(ii) *wanneer getallen en operaties hetzelfde mogen zijn.*

(iii) *Probeer een flink aantal van de mogelijkheden uit de vorige opgave uit in* Magma, *en zie of de uitkomsten altijd zijn wat je ervan verwacht (bijvoorbeeld, wat is 2^{3^4}?*

Opgave 4. *Probeer nogmaals een aantal mogelijkheden uit, en voorspel van welk type het resultaat zal zijn; gebruik positieve en negatieve getallen (en 0) in teller en exponent).*

Opgave 5. *Wat is het resultaat van $-11 \bmod 4$ in* Magma, *en van welk type is het?*

3.1.2 Rationale getallen

Rationale getallen maak je eenvoudig met `/`, als in `4/8`. Teller en noemer (na simplificatie!) vind je terug met `Numerator` en `Denominator`. Het lichaam der rationale getallen maak je met `RationalField()`.

Opgave 6. *Wat is het type van* `-12/2` *in* Magma*? Hoe maak je er een geheel getal van? Wat doet* `-12 div 2` *en wat doet* `-13 div 2`*?*

`+ - * / ^   Integers   mod   [ ..  by ]   RationalField   Numerator Denominator`

3.1.3 Reële getallen

Reële getallen kun je invoeren door decimalen `0-9` en een decimale `.` te gebruiken; je krijgt ze natuurlijk ook als resultaat van bepaalde functies, zoals logaritmen, trigonometrische functies, enzovoorts. Met `RealField()` wordt het lichaam van de reële getallen expliciet gemaakt.

Sommige constanten hebben hun eigen creatiefunctie, zoals π, met `Pi`.

Er wordt getracht op een intelligente manier om te gaan met het aantal decimalen dat correct is voor een reëel getal; zo zal worden onthouden dat een 'geheel reëel getal' met 'oneindige' precisie bekend is; in het algemeen zijn natuurlijk maar eindig veel (vaak 28) decimalen correct bekend. Het vereist enige oefening om met de precisie van reële getallen om te gaan; de functie `Precision` geeft het aantal decimalen.

Opgave 7. *Creëer het reële getal -3 op drie verschillende manieren: letterlijk, door coërcie van een geheel getal, en door coërcie van een rationaal getal.*

Opgave 8. *Vergelijk de typen van bovenstaande reële getallen en hun precisie.*

Opgave 9. *Maak $\pi^2/4$.*

Opgave 10. *Raad de naam voor de cosinus functie en haar inverse in* Magma*, en bereken de cosinus van 45 graden en $\cos^{-1}(1/2)$. Hoe zit het met de precisie van de uitkomsten?*

Opgave 11. *Creëer het getal e, de basis van de natuurlijke logaritmen.*

Opgave 12. *Bereken de logaritmen van e met grondtal e en met grondtal* 10.

3.1.4 Complexe getallen

De complexe getallen vormen het eerste voorbeeld waar we pas elementen kunnen definiëren nadat de moederstructuur bestaat. Dat is omdat in Magma niet de letter i gereserveerd is voor $\sqrt{-1}$. De manier om dit te doen is door middel van `C<ii> := ComplexField()`. Dit creëert tegelijkertijd het lichaam der complexe getallen (en kent dit object toe als waarde aan de variabele `C`) maar ook de voortbrenger $\sqrt{-1}$, die hier aan de variabele `ii` wordt toegekend.

Opgave 13. *Creëer de complexe getallen zoals hierboven, en maak de complexe getallen $3+\sqrt{-1}$ en $\pi/(1+\sqrt{-1})$.*

Opgave 14. *Bepaal de waarde, het type, en de moederstructuur van $\sqrt{-1}^2$ wanneer die als in de vorige opgave is gemaakt.*

Opgave 15. *Laat zien dat het niet werkelijk noodzakelijk is om het lichaam $\mathbb{C}$ eerst te creëren: je kunt de wortel uit het gehele getal -1 trekken, en de waarde aan de variabele i toekennen. Nu kun je complexe getallen hiermee maken. Laat dat zien. Wat valt je op als je zo'n complex getal afdrukt?*

Opgave 16. *Er zijn in* Magma *tenminste 4 functies waarmee je de wortel uit een getal kunt berekenen:* `Root`, `Sqrt`, `Iroot`, *en* `Isqrt`. *Vind uit wat de vereiste input is om $\sqrt{m}$ voor verschillende waarden en typen van m uit te rekenen en bekijk ook het type van de output.*

`RealField` `Pi` `Precision` `ComplexField` `Root` `Iroot` `Sqrt` `Isqrt`

3.2 Vergelijkingen en Boolese waarden

Gelijkheden en ongelijkheden maak je met `eq`, `ne`, `gt`, `ge`, `lt`, `le`, voor 'is gelijk', 'is ongelijk', 'is groter dan', 'is groter dan of gelijk aan', 'is kleiner dan', en 'is kleiner dan of gelijk aan'.

Het resultaat van zo'n vergelijking is een Boolese waarde, die je ook met `true` of `false` kunt maken. De operatoren `and`, `or`, `not`, kunnen op voor de hand liggende wijze worden gebruikt.

De structuur waarin Boolese waarden thuishoren is `Booleans()`.

Opgave 17. *Vraag* Magma *of de waarheidswaarden van* $5 < 4$ *en de ontkenning van 'waar' hetzelfde zijn.*

Opgave 18. *Wat is het type van* `true`*?*

3.3 Karakters, woorden, commentaar

Soms is het nuttig over 'strings' van letters te kunnen beschikken. De structuur `Strings()` herbergt deze objecten; de elementen kunnen gemaakt worden door de karakters tussen dubbele aanhalingstekens te plaatsen, dus bijvoorbeeld `w := "woord";`. Met `w[`i`]` kan (voor een natuurlijk getal i) dan de i-de letter verkregen worden. Met `*` kunnen strings acter elkaar geplakt worden; met ˆ4 kunnen (bijvoorbeeld) 4 copiën aan elkaar geregen worden.

Als speciale symbolen moeten de dubbele quotes en de backslash voorafgegaan worden door een backslash om ze als karakter te krijgen, dus `\"` en `\\`. Verder betekent `\n` een 'nieuwe regel' en `\t` een 'tab'.

Regels beginnend met een dubbele slash `//` worden door Magma als commentaar opgevat, en verder genegeerd.

Opgave 19. *Laat* Magma *de regel*

```
welkom bij Magma!
```

afdrukken.

Opgave 20. *Laat* Magma *nu de regels*

```
"Hartelijk welkom"
   nieuwe Magmagebruiker
```

afdrukken.

Opgave 21. *Verklaar het verschil in uitkomst tussen de twee volgende* Magma *statements:*

```
("73" * "9" * "42" eq "7" * "3942") eq (73 * 9 * 42 eq 7* 3942);

("73" * "9" * "41" eq "7" * "3941") eq (73 * 9 * 41 eq 7* 3941);
```

3.4 Coërcie

Een belangrijk karakter in Magma is het uitroepteken `!`. Het wordt vaak gebruikt om het type van een object te veranderen: met `R ! x` wordt gepoogd om het element x op te vatten als element van de structuur R. Zo zal `t := ComplexField() !  -2;` leiden tot de creatie van een complex getal t dat gelijk is aan -2.

Opgave 22. *Coërcie wordt vaak gebruikt wanneer een operatie niet van toepassing op een bepaald type object, maar wel wanneer we het interpreteren als een ander type. Laat zien hoe je de priemgetalontbinding van* $1155/7$ *kan vinden (met* `Factorization`*).*

Opgave 23. *Voor het uitrekenen van een* faculteit *bestaat de functie* `Factorial` *en kan* `!` *niet gebruikt worden. Bepaal* $5!$ *als reëel getal.*

```
Booleans()      true false       eq ne gt lt ge le       and or not
Strings()           "            !             //
```

Hoofdstuk 4

Verzamelstructuren

De ‘verzamelstructuren’ in Magma nemen een belangrijke plaats in onder de datastructuren omdat ze de gebruiker in staat stellen een aantal veel voorkomende constructies uit te voeren in één stap die anders omslachtiger in een aantal stappen zouden moeten gebeuren. In het bijzonder is het mogelijk met verzamelingen en rijen in één stap dezelfde bewerking een groot aantal keren uit te voeren en het resultaat op te slaan.

Een heel belangrijk principe bij de belangrijkste verzamelstructuren, die van *verzamelingen* en *rijtjes*, is dat deze slechts objecten kunnen bevatten die element zijn van dezelfde wiskundige structuur (groep, ring, enz.) en die dus van hetzelfde type zijn.

4.1 Verzamelingen

Verzamelingen in Magma zijn ongeordende collecties objecten uit één structuur zonder herhalingen.

De eenvoudigste manier om een verzameling te maken is door de elementen ervan op te sommen en tussen accoladen { } te plaatsen: `S := { 3, 7, 15 };` bijvoorbeeld. Wanneer elementen nogmaals, of in een andere volgorde, in de verzameling worden gestopt verandert het resultaat niet: `{ 15, 3, 15, 7, 3 }` is niet te onderscheiden van de voorgaande verzameling S. De elementen van de verzameling zijn dan ook niet geordend, en het is daarom niet mogelijk te vragen om het i-de element van de verzameling. Wel kan met `Representative` of `Random` een representatief of een willekeurig element uit de verzameling gehaald worden.

Voor gehele getallen in arithmetische progressie is de constructie `{ a .. b by c }` nuttig, die de getallen $a, a+c, a+2\cdot c$ enzovoorts tot (eventueel met) b in de verzameling stopt. De getallen a, b, c mogen negatief zijn (maar $b \neq 0$).

De vereiste dat de elementen allemaal uit eenzelfde structuur moeten komen is een aanzienlijke beperking; deels wordt die opgevangen door Magma’s automatische coërcie, een mechanisme dat probeert voor elementen uit verschillende structuren een gemeenschappelijk structuur te vinden waar beiden toe behoren; die structuur heet het universum, en is beschikbaar met `Universe`. Dit gebeurt alleen wanneer dat universum welgedefinieerd en eenduidig is. Zo zal Magma er geen moeite mee hebben om voor `{ 1, 1/2, 1.0 }` achtereenvolgens als universum $\mathbb{Z}$, $\mathbb{Q}$ en uiteindelijk $\mathbb{R}$ te kiezen (na het invoegen van het gehele getal 1, de breuk 1/2 en het reëele getal 1).

Opgave 24. *Hoeveel elementen denk je dat* `{ 1, 1/2, 1.0 }` *zal hebben? Verifieer dat!*

In zijn meest algemene vorm ziet de constructie van een verzameling er zo uit:
{ `U | e(x) :  x in S | P(x)` },
hetgeen het volgende betekent: creëer de verzameling van elementen $e(x)$, waar e een uitdrukking in (functie van) x is, voor alle x die in de eindige structuur S zitten en voldoen aan het 'predicaat' (de Boolese functie) $P(x)$; stop tenslotte de resultaten $e(x)$ allemaal in de structuur U, die dan ook het universum van de verzameling wordt. Hier is een eenvoudig voorbeeld:
{ `RealField() | (x-1)/(x+1) :  x in [1..100 by 2] | IsPrime(x)` }.
Merk op dat we hier de `a..b by c` constructie voor een rijtje `[ ]` (zie hieronder) hebben gebruikt.

Een aantal ingrediënten mag weggelaten worden: wanneer `U |` er niet is zoekt Magma het 'kleinste' universum dat voldoet; het universum moet de eigenschap hebben dat alle elementen $e(x)$ er in ieder geval met `U ! e(x)` in gestopt kunnen worden. In het voorbeeld wordt
{ `(x-1)/(x+1) :  x in [1..100 by 2] | IsPrime(x)` }
een verzameling rationale getallen. Ook kan het predicaat weggelaten worden (met de `|`), hetgeen in het voorbeeld natuurlijk een grotere verzameling oplevert:
{ `(x-1)/(x+1) :  x in [1..100 by 2]` }.
Vaak zal de uitdrukking $e(x)$ gewoon x zijn; in het voorbeeld komen we dan weer terug op de oorspronkelijke constructie:
{ `x :  x in [1..100 by 2]` }
en
{ `1..100 by 2` }.
zijn hetzelfde. De combinatie
{ `RealField() | x :  x in [1..100 by 2]` }
is ook toegestaan.

Wordt alles weggelaten dan krijgen we de constructie
{ }
voor de lege verzameling; hiervan is het universum niet bepaald zolang we geen elementen in de verzameling stoppen — het blijft de zogenaamde *null*-verzameling. Om een lege verzameling met een gedefinieerd universum te maken kan
{ `U |` }
gebruikt worden. Tenslotte moet nog opgemerkt worden dat de uitdrukking e ook meer dan één argument mag hebben; zo definieert

```
{ x1^2 + x2^2 : x1 in [0..9 by 2], x2 in [1..10 by 2] }
```

de verzameling van sommen van een kwadraat van een oneven getal en van een even getal onder de tien.

{ `U | e(x) :  x in S | P(x)` }   `[ U | e(x) :  x in S | P(x) ]`

4.2 Rijtjes

Een rijtje (sequence) in Magma is een geordende collectie objecten uit één wiskundige structuur, waaronder herhalingen mogen voortkomen. Met `S[i]` kan het i-de element uit de rij gehaald worden (waar de telling begint bij 1 en eindigt bij het aantal elementen van S). Het aantal elementen van een rijtje (of verzameling) S is met `#S` te verkrijgen.

De constructie van rijtjes is analoog aan die van verzamelingen, zij het dat `[ ]` gebruikt wordt in plaats van { }. Dus, in zijn meest algemene vorm

`[ U | e(x) : x in S | P(x) ]`,

waar de objecten dezelfde betekenis hebben en restrikties kennen als boven.

Opgave 25. *Creëer de verzameling van alle priemgetallen kleiner dan of gelijk aan 1003 die 5 modulo 8 zijn.*

Opgave 26. *Hoeveel priemgetallen kleiner dan* 10000 *zijn er? En kleiner dan* 10^6*?*

Opgave 27. *Bepaal de verzameling van alle getallen tussen 100 en 200 die door 11 deelbaar zijn op minstens 2 verschillende manieren.*

Opgave 28. *Creëer de verzameling van getallen tussen 500 en 600 die niet deelbaar zijn door 2 en ook niet door 7, maar wel door 11 of door 13.*

Opgave 29. *Hoeveel getallen zijn som van een kwadraat van een oneven getal en van een even getal onder de tien? En hoeveel van die getallen worden meer dan eens gerepresenteerd in die vorm?*

Opgave 30. *De functie* `PrimeBasis` *geeft van een positief geheel getal het rijtje verschillende priemdelers. Wat is het gemiddelde aantal priemdelers van de getallen tot 1000?*

4.3 Operaties op Verzamelingen en Rijtjes

Belangrijke binaire operaties op verzamelingen zijn `join` en `meet` voor vereniging en doorsnede, en `diff` en `sdiff` voor verschil en symmetrisch verschil. Om zulke (en sommige andere operaties) uit te voeren is het soms nodig dat Magma een gezamenlijk universum bepaalt dat de universa van de argumenten omvat. Met `eq` en `ne` kan vergeleken worden of twee verzamelingen gelijk of verschillend zijn, met `U subset V` of $U \subset V$ en met `u in U` of $u \in U$. Ook `notin` en `notsubset` bestaan. Voor kleine verzamelingen kan met `Subsets` de collectie van alle deeelverzamelingen, of via `Subsets(S, k)` de collectie van alle deelverzamelingen van S met precies k elementen, bepaald worden.

De voornaamste binaire operatie op rijtjes is die van concatenatie: `s cat t` plakt rijtje t achter s mits er een gezamelijk uniersum gevonden wordt. Met `in` en `notin` kan weer getest worden of een element al dan niet in een rijtje zit.

De predicaten `IsEmpty` en `IsNull` gaan na of een rij of verzameling leeg, danwel leeg zonder universum, is. Met `#` wordt het aantal elementen gevonden. Het paar `Seqset` en `Setseq` (voor `SequenceToSet` en `SetToSequence` maakt van een rij een verzameling en omgekeerd.

Wanneer de elementen van een rij of verzameling uit een geordende structuur komen (waarop een $<$ relatie is gedefinieerd), dan kan met de functies `Max` en `Min` het maximum en minimum van de rij of verzameling bepaald worden, met `Sort` kan de rij gesorteerd worden. Van `Sort` zijn twee versies beschikbaar, namelijk een *functie* en een *procedure*. Het verschil tussen beide in Magma is dat een functie output levert in de vorm van een object dat wordt afgeleverd en een procedure doet dat niet; bovendien kan een procedure sommige elementen die als input worden meegeleverd wijzigen, terwijl een functie nooit argumenten kan veranderen.

`[ ]` `[ U | e(x) : x in S | P(x) ]` `#`
`join meet diff sdiff eq ne in notin subset Subsets Seqset Setseq Sort Max Min ~`

In dit geval, als S een rijtje is kun je met `T := Sort(S);` de gesorteerde versie van S in T terugkrijgen; als procedure kun je `Sort(~S);` typen waarop Magma het object S verandert in een gesorteerde versie. Voor hele grote rijen kan het belangrijk zijn dat voor de procedurele versie maar 1 rij blijft bestaan, terwijl bij de functie-aanroep zowel de oorspronkelijke S als de gesorteerde variant aanwezig zijn.

Om rijtjes en verzamelingen uit te breiden zijn er functies en prcocedures als `Append` (plakt een element achter een bestaande rij) en `Include` (stopt een element in een bestaande verzameling). Van `Append` zijn er weer een procedurele versie, te gebruiken met `Append(~S, x);` en een functionele versie `T := Append(S, x);`. Net zo voor `Include` voor een verzameling. Ook bestaat `Include` voor rijtjes, waar alleen het element achter de rij geplakt wordt wanneer het niet al in de rij zit. Om het i-de element uit een rij te verwijderen gebruike men `Remove(~S, i)` of `Remove(S, i)`. Met `Exclude(~S, x)` wordt (de eerste) x uit de verzameling (of rij) S verwijderd; een rij wordt daar korter van (omdat de andere elementen opschuiven).

Het is ook mogelijk de waarde van het i-de element van een rijtje s te veranderen met de assignment S[i] `:= y;` De inhoud wordt dan overschreven. Omdat i groter mag zijn dan de lengte van het rijtje, kan de rij zo ook verlengd worden; bovendien is het mogelijk ‘gaten’ in de rij te laten vallen.

Opgave 31. *Hoe ziet s er uit na* `s := [1,2]; Append(~s,3);` *en* `s[#s+2] := 5;` *en* `s[#s+2] := 8;`*?*

Opgave 32. *Laat zien dat de verzamelingen $\{1, 2, 3, 4\}$ en $\{1, 3, 2, 4\}$ in* Magma *hetzelfde zijn, maar dat de rijtjes $[1, 2, 3, 4]$ en $[1, 3, 2, 4]$ verschillend zijn.*

Opgave 33. *Creëer van de priemgetallen P tot 1000 de deelrij Q van elementen waarvoor $p-4$ of $p+4$ weer een priemgetal is. Is $R = \{863, 911\}$ een deelverzameling van Q?*

Opgave 34. *Bepaal de rij van laatste cijfers van de getallen in Q (uit de vorige opgave); bepaal ook een rijtje paren $[c, f]$ waar c door de voorkomende laatste loopt en f het aantal malen dat het optreedt in Q.*

Opgave 35. *Bepaal de collectie van priemgetallen die optreden als deler van een getal van de vorm $q - 1$, met q zelf ook een priemgetal.*

Opgave 36. *De functie σ_i bepaalt van een natuurlijk getal de som van de i-de machten van de delers van n. Je vind het resultaat in* Magma *met* `DivisorSigma(i, n)`*. Gebruik dat om een verzameling S van getallen onder de 10000 te vinden met precies 48 delers. Bepaal ook de doorsnede hiervan met de getallen die precies 3 priemdelers hebben.*

Opgave 37. *Vind een voorbeeld van een rijtje waar R waar* `Setseq(Seqset(R))` *niet gelijk is aan R. Kun je ook een voorbeeld van een verzameling V vinden waarvoor* `Seqset(Setseq(V))` *niet gelijk is aan V?*

4.4 Overige Verzamelstructuren

Van de overige verzamelstructuren noemen we hier alleen maar de *geïndiceerde verzamelingen* waar de elementen wél geordend zijn, te maken met `{@ @}`; de multiverzamelingen waar elementen met multipliciteiten kunnen voorkomen, te maken met `{* *}`; de *formele verzamelingen* die weinig meer zijn dan predicaten, te maken met `{! !}`; de tuples (elementen van een Cartesisch product), te maken met `< >` en de lists, met `[* *]`. Voor de laatste twee gelden niet de strenge eisen op het universum, zodat er geheel verschillendsoortige objecten in mogen zitten.

Opgave 38. *Gebruik een Cartesisch product om alle paren (i, j) van gehele getallen die in absolute waarde kleiner dan 10 zijn te vinden waarvoor het quotient i/j ongelijk aan 1 is.*

`IsEmpty` `IsNull` `Append` `Include` `Exclude` `Remove`
`{@ @}` `{* *}` `{! !}` `[* *]` `< >`

Hoofdstuk 5

Input/Output

Voordat we meer elementen van de Magma taal bekijken, eerst iets over de manieren om input en output te behandelen.

Om de waarde van een variabele of het resultaat van de aanroep van een functie op het scherm te zien, is het commando `print` beschikbaar. Dus bijvoorbeeld

```
> x := 12;
> print x;
12
> print Gcd(12, 14), Gcd(15, 21);
2 3
```

Overigens geeft het weglaten van `print` precies hetzelfde effect. Meerdere waarden kunnen afgedrukt worden door ze via een komma te scheiden, zoals het voorbeeld laat zien.

Het is ook nuttig om te weten dat het mogelijk is om het resultaat van de voorgaande berekening (die reeds op het scherm is afgedrukt) nogmaals te verkrijgen — bijvoorbeeld om de waarde alsnog aan een variabele toe te kennen — middels `$1`.

```
> print Gcd(12, 14), Gcd(15, 21);
2 3
> g, h := $1;
> g, h;
2 3
```

Het belang hiervan is vooral dat de (mogelijk langdurige) berekening niet herhaald hoeft te worden. Het is mogelijk om meer dan één stap (maar niet teveel stappen) terug te gaan met `$2` enzovoorts. Ook is hier zichtbaar gemaakt dat wanneer meerdere waarden werden afgedrukt, ze allemaal teruggehaald kunnen worden.

Om de afgedrukte tekst er beter uit te laten zien is het mogelijk nieuwe regels in te voegen, met `print "\n"`, en tabs met `print "\t"`.

Iets meer controle dan met `print` kan worden verkregen met `printf`, bijvoorbeeld omdat in tegenstelling tot `print` bij `printf` niet achter elk Magma-object een nieuwe regel wordt geplakt. Het is met een extra argument mogelijk om `printf` een Magma-object een aantal karakterplaatsen toe te kunnen:

```
> for x in [ p : p in [50..150] | IsPrime(p) ] do
>     printf "%4o", x;
> end for;
  53  59  61  67  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149
```

Hier specificeert het getal tussen de `%` en de `o` het aantal karakters dat het af te drukken priemgetal in beslag moet nemen (naar rechts uitgelijnd).

Opgave 39. *Wat gebeurt er wanneer je* `%4o` *vervangt door* `%3o`*; en door* `%2o`*?*

`print` `printf` `$1`

Soms is het nodig (of handig) resultaten naar een bestand weg te schrijven (in plaats van op het scherm). Daarvoor is `PrintFile("mijnfile", x)`, waar het bestand met de naam '`mijnfile`' reeds mag bestaan (in de directory waar Magma loopt), in welk geval de waarde van `x` aan het einde van het bestand wordt bij geschreven, of het bestand met die naam wordt gecreeerd en `x` erin weggeschreven. Het kan nodig zijn (teneinde het bestand niet al te groot te laten worden bijvoorbeeld) om de inhoud van het bestand te overschrijven; dit kan met `PrintFile("anderfile", x: Overwrite := true)`. Deze constructie (een extra parameter, na de dubbele punt te specificeren) komt ook bij sommige andere functies voor, bijvoorbeeld om een keuze van de te gebruiken methode op de functie vast te leggen.

Met `SetOutputFile("opslaan");` is het mogelijk *alle* output van een sessie naar een bestand, bijvoorbeeld `opslaan`, weg te laten schrijven. Ook hier is het mogelijk steeds alleen de laatste waarde te bewaren door `: Overwrite := true` te specificeren. Om dit wegschrijven te beëindigen gebruikt men `UnsetOutputFile("opslaan");` Om uit te vinden of er naar een bestand (en zo ja, naar welk) wordt weggeschreven is `HasOutputFile()`.

Met `SetLogFile("log");` worden zowel input als output weggeschreven naar een file (hier: `log` geheten). Dit wordt beëindigd met `UnsetLogFile();`.

Bij serieuzer programmeerwerk is het verstandig het programma in een bestand op te slaan, en dit in te laten lezen door Magma. Dat inlezen kan via `load "naam";` waar `naam` natuurlijk de naam van een bestaand bestand moet zijn. Dit mag de naam van een bestand zijn in de directory waar Magma is opgestart (in dat geval zijn de `"` niet nodig), of in een andere directory, wanneer de naam dan als in Unix gebruikelijk met het pad wordt aangegeven, dus bijvoorbeeld `load "../new"` voor het bestand `new` in de bovengelegen directory. Een nuttige variant (bijvoorbeeld voor presentaties, is `iload`, waarbij de regels uit het bestand stuk voor stuk worden in gelezen en de gebruiker telkens de optie heeft de regel direct uit te voeren (door op `<enter>` te drukken), over te slaan (door met een pijltje naar beneden te gaan), of de regel eerst aan te passen (door er in te navigeren met de horizontale pijltjes etc.) alvorens deze uit te voeren.

Let op dat `load` niet in een lus of in een functie/procedure kan worden aangeroepen.

Met `SetEchoInput(true)` kan ervoor gezorgd worden dat van een ingelezen bestand niet alleen de output wordt afgedrukt, maar ook de inhoud van het bestand zelf.

Met `read` is het mogelijk om het programma een door de gebruiker te kiezen waarde in te laten lezen:

```
> read variabele;
1234
> print variabele;
1234
```

De eerste `1234` werden hier ingetypt (gevolgd door `<enter>`) terwijl Magma wachtte op input. Het is mogelijk om als tweede argument aan `read` een string mee te geven die als 'prompt' dient, bijvoorbeeld om Magma een instructie te laten afdrukken:

```
> read variabele, "type een integer:";
type een integer:
234
> print variabele;
234
```

`PrintFile` `SetOutputFile` `SetLogFile` `load` `read`

Hoofdstuk 6

Controlestructuren

6.1 Toekenning

We hebben al diverse malen de gewone toekenning van waarden aan variabelen gezien zoals in: `x := Gcd(4,6);` enzovoorts. Er zijn nog een paar varianten die we hier kort bespreken. In de eerste plaats zijn er functies (zoals de uitgebreide versie `Xgcd` van de grootste gemene deler) die meer dan een enkele waarde opleveren; de meervoudige toekenning ('multiple assignment') ziet er gewoon zo uit:

```
> g, x, y := Xgcd(6789,2345);
> print g, x, y;
1 734 -2125
```

Daarnaast is er nog de zogenaamde 'mutatie assignment', waarbij de waarde van een variabele als het ware ter plekke wordt veranderd:

```
> x := 12;
> x *:= 3;
> x;
36
> m := "Mag";
> m cat:= "ma";
> m;
Magma
```

Dit is mogelijk voor een groot aantal operaties, zoals `+ - * / ^ cat join meet` in diverse domeinen.

In rijtjes is het mogelijk de waarde op positie k toe te kennen door `s[4] := 7;` bijvoorbeeld, voor de rij `s` en $k = 4$. Hierbij hoeft niet aan alle plaatsen vóór k al een waarde te zijn toegekend:

```
> t := [];
> t[3] := 7;
> t;
[ undef, undef, 7 ]
```

en het resultaat is dus een rij waarin nog onbepaalde elementen staan (de `undef`s).

Opgave 40. *Wat is nu* `t[1]`*? En wat* `#t`*?*

In de volgende paragraaf laten we zien hoe je kunt itereren. Zoals al eerder opgemerkt zijn heel veel iteraties in Magma al mogelijk bij het creëren van rijtjes (en verzamelingen). Bovendien is mogelijk op alle elementen van een gedefinieerde rij een zelfde operatie los te laten, zoals bijvoorbeeld vermenigvuldiging, die dan resulteert in het product van alle elementen van die rij. Zulke reductie-operaties bestaan uit de ampersand gevolgd door de operatie, dus bijvoorbeeld:

```
> s := [1,2,3];
> &*s;
6
```

`:=` `+:=` `undef` `&+`

of

```
1060
> &+[ p : p in [1..100] | IsPrime(p) ];
```

de som van de priemgetallen onder de 100. Zo is `&*` dus de Magma versie van $\prod$.

Opgave 41. *Bepaal het product over $(n^3-1)/(n^2-1)$ van de natuurlijke getallen tussen 1 en honderd die product van precies twee priemgetallen zijn.*

6.2 Iteratie

Er zijn drie manieren in Magma om stappen herhaald uit te voeren: met `for`, met `while` en met `repeat`

De `for` lus komt in twee varianten. In de eerste wordt geïtereerd over een eindig aantal gehele getallen:

```
> for k := 5 to -(3^2 + 1) by -2 do
>   ...
> end for;
```

Hier wordt met stapjes van -2 over de gehele getallen van 5 tot -10 gelopen. Als stapjes ter grootte 1 nodig zijn mag de `by ...` gewoon weggelaten worden. Ondergrens, bovengrens en stapgrootte mogen allemaal het resultaat van een uitdrukking zijn die gehele getallen oplevert (zoals $-(3^2+2)$ hier).

In de tweede variant wordt over een eindige wiskundige structuur geïtereerd; die structuur kan een verzameling zijn, of een rijtje, of een ingewikkelder structuur, bijvoorbeeld een eindig lichaam:

```
> for e in FiniteField(17) do
>   ...
> end for;
```

Hierbij mag ook over meerdere structuren tegelijk geïtereerd worden:

```
> for e in FiniteField(17), k in [0..16] do
>   print e^k;
> end for;
```

De `while` en `repeat` constructies lijken erg op elkaar. Ze worden gebruikt om stappen te herhalen zolang aan een zekere voorwaarde is voldaan. De twee vormen verschillen voornamelijk in het moment waarop de voorwaarde wordt getest: vóór elke iteratie (met `while`, of erna (met `repeat`). Laten we de derde machten afdrukken van getallen van de vorm $\frac{(-1)^n}{n}$ met n positief geheel en n^2 ten hoogste 50:

```
> n := 1;
> while  n^2 le 50 do
>   print (-1)^n/n;
>   n +:= 1;
> end while;
```

Vergeet niet de beginwaarde van de iteratie vast te leggen èn in de lus de variabele op te hogen!

`for ... to ... by ... end for`   `while ... end while`

Hetzelfde met `repeat`:

```
> n := 1;
> repeat
>   print (-1)^n/n;
>   n +:= 1;
> until n^2 gt 50;
```

Het is heel goed mogelijk de verschillende soorten lussen te nesten.

Opgave 42. *Herhaal de vorige opgave, maar nu met gebruikmaking van een* `for` *lus; idem voor* `while` *en* `repeat`.

Er zijn twee speciale woorden gereserveerd in de taal om op een speciale manier uit een lus te springen: `continue` en `break`. De eerste is vooral bedoeld om ervoor te zorgen dat (soms, in combinatie met het gebruik van `if ... then ... end if`, zie onder) de stappen in de iteratie worden overgeslagen, en met de volgende iteratie wordt doorgegaan.

Bij het bereiken van een `break` statement in de lus wordt er onmiddellijk uit de lus gesprongen. Lussen kunnen genest zijn; gewoonlijk slaan `continue` of `break` op de binnenste lus waarin ze gebruikt worden. Is het nodig direct uit een verder naar buiten gelegen iteratie te springen, kan `break x` gebruikt worden voor het springen uit de lus met lusvariabele `x`.

```
> n := 0;
> repeat
>   n +:= 1;
>   if IsEven(n) then
>     continue;
>   end if;
>   print (-1)^n/n;
> until n^2 gt 50;
```

En met een geneste `for` lus:

```
> p := 10037;
> for x in [1..100] do
>    for y in [1..100] do
>       if x^2+y^2 eq p then
>          print x, y;
>          break x;
>       end if;
>    end for;
> end for;
```

`repeat ... until`   `break`   `continue`

6.3 Voorwaarden

We zagen hierboven al een geval van het testen van een voorwaarde met `if`. In zijn meest algemene vorm ziet dit er als volgt uit, waar de `elif` handig is om teveel geneste condities te voorkomen:

```
> for n in [0..50] do
>   if n mod 4 eq 1 then
>     print n;
>   elif n mod 4 eq 3 then
>     print -n;
>   elif n mod 8 in {2,6} then
>     print n div 2;
>   else
>     print n div 4;
>   end if;
> end for;
```

Opgave 43. *Schrijf dezelfde lus zonder de* `elif` *te gebruiken (maar met geneste* `if`*s).*

Er is ook een statement waarin meer gevallen kunnen worden onderscheiden: `case`. In het volgende eenvoudige voorbeeld wordt dit gebruikt om af te drukken wat het teken van een gegeven getal is:

```
> x := 73;
> case Sign(x):
>    when 1:
>       print x, "is positive";
>    when -1:
>       print x, "is negative";
>    else
>       print x, "is zero";
> end case;
73 is positive
```

Let op de dubbele punten! Wat hier gebeurt is dat de waarde van de uitdrukking na `case` wordt bepaald en achtereenvolgens vergeleken met de waarden van de uitdrukkingen na de `when`s. Van het eerste geval waarin de twee overeenkomen worden de stappen uitgevoerd. Als geen enkel geval overeenkomt wordt het `else` geval uitgevoerd; als dit weggeleten wordt gebeurt er dan niets.

Van `case` is ook een expressie-vorm beschikbaar, dat wil zeggen, een uitdrukking die het geval dat van toepassing is uitvoert en het resultaat teruggeeft:

```
> for x in [-3..3] do
>   y := case< x^3-x | 0 : x, default: 10 >;
>   printf "%5o", y;
> end for;
   10   10   -1    0    1   10   10
```

`case ... when ... else ... end case`   `case< | default: >`

Er is nog een derde manier om een voorwaarde te testen, die speciaal bedoeld is wanneer je bij het toekennen van een waarde aan een variabele een keuze uit twee alternatieven wilt maken. De vorm is `X select A else B`, waar `X` de waarde `true` of `false` moet opleveren; in het eerste geval wordt de waarde van de uitdrukking `A` opgeleverd, in het tweede geavl die van `B`.

```
> x := Sqrt(11);
> s := (x gt 0) select 1 else -1;
```

Opgave 44. *In het voorbeeld hierboven is een primitieve vorm van de teken-functie gemaakt. Vind de intrinsic die je het juiste antwoord geeft op de vraag of een gegeven x kleinder dan, gelijk aan, of groter dan* 0 *is.*

Opgave 45. *Het voorbeeld geeft niet precies de teken-functie, want daarin wil je drie gevallen onderscheiden ($\leq$, $=$, ≥ 0). Laat zien hoe je dat voor elkaar kunt krijgen door de* `select .. else ..` *te nesten.*

Hoofdstuk 7

Functies en Procedures

Om uitgebreidere programma's te kunnen maken, zal het nodig zijn je eigen functies te definiëren. Hoe deze 'user functions' (om ze te onderscheiden van de ingebouwde 'intrinsic functions') te maken zijn zullen we hier kort beschrijven. Naast functies zijn er ook 'user procedures', zoals ook intrinsics in functievorm en in procedurevorm bestaan (soms van dezelfde naam, zoals `Sort` of `Append`. Het kenmerkende verschil tussen functies en procedures ligt hem erin dat functies altijd een waarde teruggeven maar nooit de waarden van hun argumenten mogen veranderen, terwijl het effect van procedures juist is dat ze van sommige argumenten onderweg de waarde wijzigen maar ze nooit een waarde teruggeven.

7.1 Functies

Van de gewone vorm van de functies zijn er twee smaken met maar een enkel verschil:

```
> eenfunc := function(a, b, c)
>   ... statements ...
> end function;
```

De voorgaande vorm benadrukt dat de functie gewoon aan een variabele (hier `eenfunc`) wordt toegekend, zoals alle andere waardes. Natuurlijk mogen er meer of minder argumenten voor de functie zijn, en mogen de namen net als die van alle andere variabelen gekozen worden.

```
> function tweedefunc(x, y, z)
>   ... statements ...
> end function;
```

De tweede vorm wijkt enigszins af van de gebruikelijke toewijzing van namen. Dit dient slechts 1 doel, en dat is dat met deze vorm de naam van de functie (hier dus `tweedefunc`) gewoon gebruikt mag worden (recursief, zie ook verderop) binnen de functie zelf: de functie mag zichzelf aanroepen (zoals elke andere functie). In de eerste vorm mag dat ook wel maar kan de naam (zoals `eenfunc`) niet gebruikt worden in de functie zelf (omdat de definitie van die functie dan nog niet af is). In dat geval moet de functie zichzelf aanroepen met `$$`.

De 'statement' die in de functie moeten worden uitgevoerd mogen alle intrinsics natuurlijk gebruiken, evenals al eerder gedefinieerde user functions en procedures.

Uiteindelijk zal de functie een waarden moeten afleveren, hetgeen gebeurt met een `return` statement, waar de `return` gevolgd moet worden door 1 of meerdere uitdrukkingen die 'uitgerekend' worden. De functie kan dus meer dan 1 waarde opleveren. Het aanroepen van de functie gebeurt dan zo (in het geval dat er twee waarden worden teruggegeven, en bij het uitvoeren op de argumenten 10, 11, 12):

```
> res1, res2 := eenfunc(10,11,12);
```

In feite zal er in elke tak (afgesplitst met `if`s en dergelijke) zo'n return statement moeten zijn. Omdat het handig is dat elk return statement hetzelfde aantal functiewaarden teruggeeft maar niet altijd evenveel waarden worden uitgerekend is het mogelijk een 'ongedefinieerde' waarde terug te geven, met behulp van de underscore, dus bijvoorbeeld `return 3, _, 5`.

`function ... end function`   `$$`   `return`

Een verkorte expressie-vorm van de functie declaratie ziet er zo uit:

```
> f := func< a, b |  expressie(a, b) >;
```

waar de 'expressie' weer een waarde uitrekent voor gegeven argumenten. Ook hier mogen meerdere waarden (maar minstens 1 voor elke input) teruggegeven worden.

Opgave 46. *Maak een functie die het teken van een reëel getal teruggeeft.*

Opgave 47. *Schrijf een functie die recursief het n-de Fibonaccigetal oplevert.*

Tenslotte nog manier om in de constructie van een rijtje recursief naar die rij zelf te verwijzen: met `Self` wordt de rij onder constructie aangeduid.

Opgave 48. *Gebruik* `Self` *en* `select` *om in één statement recursief de rij van de eerste 100 Fibonaccigetallen te maken.*

7.2 Procedures

Ook hier zijn weer twee gewone en een verkorte expressie-versie voor handen:

```
> P := procedure(~a, b)
>   ... statements ...
> end procedure;
```

Dezelfde opmerkingen met betrekking tot recursie en namen gelden hier als voor functies.

```
> procedure Q(x, ~y)
>   ... statements ...
> end procedure;
```

De waarden van de 'reference variables' met een tilde ˜ ervoor mogen door de procedure worden veranderd; in de aanroep (bij het gebruiken) van de procedure moeten die variabelen dan óók van een ˜ worden voorzien.

De korte versie is;

```
> f := proc< ~a, b |  expressie(~a, b) >;
```

Hierin moet de expressie een aanroep van een andere procedure zijn, bijvoorbeeld:

```
> b := [ 1, 5, 3, 8, 1, 2, 0 ];
> s := proc< ~b |  Sort(~b) >;
> s(~b);
> b;
[ 0, 1, 1, 2, 3, 5, 8 ]
```

Opgave 49. *Implementeer het uitgebreide algoritme van Euclides.*

Opgave 50. *Het $3x+1$ vermoeden zegt dat, met welk natuurlijk getal x je ook begint, je na herhaald toepassen van de bewerking: 'vervang x door $3x+1$ als x oneven is en door $x/2$ als x even is' je uiteindelijk altijd een keer bij $x=1$ uitkomt. Schrijf een functie die voor gegeven x uitrekent hoeveel stappen dat kost; schrijf ook een procedure-versie.*

7.3 Overig

Soms doet zich de noodzaak voor om een functie al aan te roepen voordat deze gedefinieerd is — dat kan met `forward` gevolgd door de naam van de functie. Dit is bijvoorbeeld het geval wanneer twee functies elkaar via 'wederzijdse recursie' aanroepen.

Tenslotte noemen we hier nog de mogelijkheid om packages te maken met door de gebruiker gedefinieerde 'intrinsics'. Zie hiervoor de uitgebreidere documentatie.

`func< >`   `Self`   `procedure ... end procedure`   `proc< >`   `forward`