Programmeerproject:

Een wiskundig kladblok

Versie van 23 februari 1999

1 Doelstelling

Bedoeling is een programma te ontwikkelen dat met een groot aantal soorten wiskundige objecten op een intelligente manier kan omgaan en courante wiskundige operaties kan doorvoeren op deze objecten. Gedacht wordt aan types als

- Getallen
- Grafen
- Formules (dingen van het soort $\int_0^\theta \sqrt{f(x)}dx$)
- Formele bewijzen
- Programma's (gebruik makend van een eigen nog te ontwerpen taaltje)
- Meetkundige objecten in een n-dimensionale ruimte (rechten, vlakken, kegelsneden, oppervlakken, n-dimensionale bitmaps, ...)
- Groepen

Mogelijke bewerkingen zouden dan kunnen zijn: maak een plot (type: meetkundig object) van een functie (type: formule), bepaal de permutatiegroep (type: groep) van een graaf (type: grafen), controleer een formeel bewijs. Later zouden we ook nog minder wiskundige types kunnen toevoegen zoals

- Tekstdocumenten (HTML, en het nec plus ultra zou natuurlijk een WYSIWYG TEX-editor zijn)
- Muziekstukken, geluidsfragmenten (samples)

Inspiratie halen we o.a. uit de HP-48 reeks van grafische rekenmachines, Maple, GnuPlot en de programmeertaal Prolog.

In dit document zullen de hierbij opduikende problemen en vragen worden onderzocht. De belangrijkste hiervan is wellicht: *Hoe stellen we deze wiskundige structuren het best voor als datastructuren?* waarbij het kriterium voor "goede voorstelling als datastructuur" dient rekening te houden met zaken als efficiëntie van opslag en manipuleerbaarheid.

Als programmeertaal kiezen we voor Java, omwille van de grote portabiliteit, de eenvoud (vergelijk bijvoorbeeld met de andere populaire C-met-klassen taal, C++, die alles veel ingewikkelder maakt) en het afwezig zijn van pointer-bugs

wat een veel snellere applicatie-ontwikkeling mogelijk maakt. Bovendien kunnen we dan een applet-versie maken, die als voordeel heeft dat de gebruiker niets hoeft te installeren om met het programma te kunnen werken.

In een later stadium moet het mogelijk zijn dat de gevorderde gebruiker zelf nieuwe soorten objecten kan toevoegen aan het programma door op de juiste plaats een aantal Java `.class` bestanden te plaatsen. Het is dan ook de bedoeling dat dit document op termijn uitgroeit tot een soort handleiding waarin het hoe en waarom van de programma-internalia beschreven staan voor zij die het programma willen uitbreiden. De handleiding voor de "gewone" gebruiker is in HTML formaat beschikbaar op `http://cage.rug.ac.be/~gvernaev/GveCalc/`.

2 Gebruikers-interface

Het basisidee van de gebruikers-interface is gebaseerd op de HP-48, namelijk een **stapel**-gebaseerde RPN interface. De gebruiker duwt objecten op de stapel en kan er bewerkingen mee uitvoeren.

Op de figuur zijn twee formules en een graaf te zien. Voor het systeem is alles eigenlijk een formule; een graaf kan overal in een formule staan waar identifiers (zoals x) of numerieke constanten (zoals 3.1415) kunnen staan.

Het uitvoeren van bewerkingen op objecten die op de stapel staan gebeurt door ofwel een commando in te tikken, ofwel door op het object met de rechtermuisknop te klikken, waardoor een popup-menu met mogelijke acties verschijnt.

Het ingetikt commando kan een ingebouwd commando zijn (bijvoorbeeld **dup2**, dat de twee onderste elementen kopieert op de stapel, of **isisomorph**, dat controleert of twee grafen isomorf zijn) of de naam van een variabele. Als die variabele niet gedefinieerd is, dan wordt de naam als formule op de stapel bewaard; anders wordt de inhoud van die variabele op de stapel gezet, tenzij het een programma is; dan wordt het uitgevoerd. (Programma-objecten zijn in de huidige versie nog niet geïmplementeerd).

2.1 Het why-object

Als de gebruiker een operatie heeft doorgevoerd, bijvoorbeeld met **isisomorph** bepaald of twee grafen isomorf zijn, kan ze door het **why** commando te geven hier uitleg over krijgen (bijvoorbeeld "de grafen bevatten een verschillend aantal vertices", of een animatie die toont dat beide grafen inderdaad isomorf zijn). Uit de vorige paragraaf volgt dat we dit kunnen implementeren door in de **why** variabele een object op te slaan met de uitleg. Als we later ook HTML-objecten implementeren, zou de uitleg ook links naar verdere informatie kunnen bevatten (bijvoorbeeld over een in de verklaring gebruikte stelling).

3 Formules

Alle objecten kunnen deel uitmaken van een formule. De gebruiker moet een formule op een zo intuïtief mogelijke manier kunnen invoeren. We willen toestanden vermijden waarbij de gebruiker een regel à la

```
int(0,theta,sqrt(f(x)),x)
```

moet ingeven om dan pas

$$\int_0^\theta \sqrt{f(x)}dx$$

te bekomen, als hij tenminste nergens haakjes of komma's vergeten is.

3.1 Historiek

Eind 1992 kwam schrijver dezes voor het eerst in aanraking met de HP-48 serie grafische rekenmachines. Die waren wel leuk, maar net niet leuk genoeg, zodat de verleiding groot werd een computerprogramma met dezelfde functionaliteit te ontwikkelen en te draaien op een voldoende geminiaturiseerde computer (in die tijd circuleerden er 808x-compatibele handheld 'Portfolio' computers van Atari ...) als rekenmachine-vervanger. Dit resulteerde in een Turbo Pascal-programma (porteerbaar tussen Amiga en msdos) van ongeveer 5000 lijnen, uitgesmeerd over een tiental units, dat op het eind aardig tegen de beperkingen van Pascal begon te botsen. Intern werd een formule door een tekststring in computernotatie voorgesteld, die dan on-the-fly naar RPN strings converteerde, die dan makkelijk in mensen-formulevorm op het scherm kon omgezet worden, of geëvalueerd worden in een punt voor functieplots.

Ongeveer een jaar later werd C mijn favoriete programmeertaal, en een C-versie die grosso modo intern op dezelfde manier werkte was snel geboren (inclusief eigen window manager voor de msdos versie). Deze versie telde een 7500 regels C code, verspreid over een twintigtal modules.

Midden 1996 kwam Java op, en een Java versie liet niet al te lang op zich wachten. Intern is de aanpak fundamenteel verschillend doordat formules nu als syntaxboom voorgesteld worden. Syntaxbomen zijn veel flexibeler en handiger te manipuleren dan string-voorstellingen; zo is het bijvoorbeeld in de string-voorstelling niet zo triviaal complexere objecten als matrices te verwerken (zoals bepalen naar welke positie in de infix-string de cursor heen moet als de gebruiker door de elementen van de matrix heenwandelt met de pijltjestoetsen). Ook vermijdt de boom-aanpak dat er grote stukken tekst moeten verschoven worden als de gebruiker voorin een lange formule een letter toevoegt of verwijdert. Ook worden syntaxbomen top-down geëvalueerd, wat voordelen heeft tegenover de bottom-up evaluatie van RPN. In een formule als `int(0,1,f(x),x)` wordt bij de syntaxboom-aanpak `int` eerst geëvalueerd en dan pas de vier argumenten, zodat

`int` ervoor kan zorgen dat de `x` in `f(x)` goed—namelijk als integratievariabele—behandeld wordt. Bij de RPN methode moeten we eerst `f(x)` evalueren, en het is niet meteen duidelijk dat er niet een getal (de waarde van de functie `f` in het punt `x`) maar wel degelijk een functie als resultaat moet komen. Het object-georiënteerde karakter van Java maakt deze taal ideaal voor het gebruik van syntaxbomen.

We zouden dit programma ook echt als rekenmachine-vervanger kunnen inzetten als we maar ergens een van die handheld-computers (Palm en consoorten) er zouden toe kunnen krijgen Java te draaien ...

Begin 1999 heb ik het programma uitgebreid met ondersteuning voor grafentheorie. Het resultaat werd gepresenteerd als mijn project voor het vak Discrete Wiskunde.

Midden 1999 kwam de ondersteuning voor formele logica.

3.2 Syntaxbomen

Daartoe slaan we formules op als een syntaxboom. De knopen van zo'n boom zijn bewerkingen en de bladen argumenten van de bewerkingen. Een voorbeeld: de formule

$$\frac{x+y}{2*x}$$

wordt voorgesteld door de boom

De syntaxboom verandert terwijl de gebruiker de formule intikt.

3.2.1 Prioriteiten van operatoren; rotaties

In principe zou de formule '$a*b+c$' op de volgende twee manieren voorgesteld kunnen worden:

Om uit te maken welke van de twee vormen gebruikt wordt, hechten we aan elke operator een **prioriteit**. In het voorbeeld moet de formule als $(a*b)+c$ geïnterpreteerd worden; we zeggen dat '$*$' een hogere prioriteit heeft dan '$+$'. De regel is dat operatoren met lagere prioriteit dichter bij de top van de boom komen te staan. Als deze regel overtreden is, moet de positie van de operatoren in de boom aangepast worden. Dat gebeurt door de ene vorm in de andere om te zetten; men noemt dit een **rotatie**. Er zijn twee soorten rotaties:

Hier kunnen a, b en c eventueel zelf ook kinderen hebben.

3.2.2 Unaire operatoren

Er zijn unaire prefix- en postfix-operatoren, waarbij de operator vóór respectievelijk achter zijn argument komt. We kunnen unaire operatoren analoog als binaire operatoren afhandelen door de prefix-operatoren een denkbeeldig leeg linkerargument mee te geven en de postfix-operatoren een rechterargument.

3.2.3 Associativiteit van operatoren

Als voorbeeld beschouwen we de := operator. De uitdrukking $x := f$ waarbij x een naam van een variabele is en f een uitdrukking heeft als effect dat f in de variabele x bewaard wordt. Het resultaat van de uitdrukking $x := f$ bij evaluatie is weer f. We kunnen dus $y := (x := f)$ schrijven om f in de twee variabelen x en y te bewaren. Om dit makkelijker te laten verlopen, heeft de := operator **rechts-naar-links associativiteit**, hetgeen betekent dat $y := x := f$ als $y := (x := f)$ wordt geïnterpreteerd (en dus niet als $(y := x) := f$, dat zou links-naar-rechts associativiteit zijn).

De implementatie is eenvoudig: als de operatoren verkeerd geassocieerd staan, worden ze geroteerd:

De implementatie van links-naar-rechts associativiteit is analoog.

3.3 Interactief opbouwen van de syntaxboom

Zoals eerder vermeld wordt de syntaxboom opgebouwd terwijl de gebruiker de formule bewerkt. Initieel wordt maar één soort knopen aan de boom toegevoegd, namelijk spatie-operatorknopen (de spatie zullen we ook $\sqcup$ noteren), die twee kinderen hebben, en één soort bladen, namelijk 'identifier'-bladen, die een getal of een (eventueel leeg) woord kunnen bevatten. Wat we precies onder "woord" verstaan zal verderop blijken. Dus de formule '$a * b + c$' zou als volgt voorgesteld worden:

Nu zal in werkelijkheid een formule nooit als deze boom opgeslagen worden. Van zodra de gebruiker '$a * b$' ingetikt heeft, wordt dit herkend als een operator (namelijk '$*$') omgeven door twee argumenten (a en b) en als volgt omgezet:

(Merk op dat vóór de omzetting '*' nog als identifier verwerkt wordt.) Hiertoe is het nodig dat het systeem op voorhand weet dat '$*$' een binaire infix-operator is. In dit geval is het voor de hand liggend dat '$*$' een ingebouwde operator van het systeem zal zijn, maar als de gebruiker zelf operatoren wil bijmaken, impliceert dit dat hij die op een of andere manier op voorhand moet aankondigen.

Merk op dat tot vóór de operatorherkenning het systeem '*' als een gewone identifier behandelt. Het is pas als '*' een linker- en rechterbuur krijgt, dat '*' "promoveert" tot operator.

Als de gebruiker de formule vervolledigt tot '$a * b + c$' dan ziet de boom er als volgt uit:

Er wordt weer een operator herkend. Bovendien worden de prioriteiten gecontroleerd, waaruit blijkt dat een rotatie nodig is.

Soms zijn meerdere rotaties nodig, bijvoorbeeld als de formule

$$a + b - c * 2$$

ingetikt is en de gebruiker breidt ze uit tot "$a + b - c * 2, d$". De komma heeft een heel lage prioriteit, lager dan "$*$", dus een rotatie is noodzakelijk. Maar na die rotatie heeft de "," een lagere prioriteit dan "$-$", dus moet weer geroteerd worden, en zo voort, totdat de "," uiteindelijk helemaal bovenaan de syntaxboom is terechtgekomen:

Alle operatoren beginnen hun leven als identifier, dus als een blad in de syntaxboom. Wanneer ze door spatie-operatoren omgeven zijn, worden ze effectief tot operator omgevormd. Omdat de spatie-operator een heel hoge prioriteit heeft, verschijnt een nieuwe operator altijd onderaan de syntaxboom. Interessant hieraan is dat een operator dus enkel naar omhoog moet kunnen roteren om op zijn juiste plaats te geraken.

3.4 De komma-operator

De komma wordt in de klassieke wiskunde voor een aantal dingen gebruikt:

- als decimale komma in getallen: 3,14159265
- als scheiding tussen functie-argumenten: $f(x, y, z)$
- als scheiding tussen de componenten van een n-tal: (x, y, z).

Het decimale-komma probleem vangen we op door een decimale punt te gebruiken. Het probleem onderscheid te maken tussen de laatste twee gevallen is ernstiger. Bovendien moeten samengestelde n-tallen, zoals

$$((x, y, z), (a, b, c))$$

correct verwerkt worden.

De aanpak die we zullen volgen is als volgt (de huidige implementatie wijkt hier van af). We maken van de komma een gewone operator. De uitdrukking $f(x, y, z)$ wordt dan als volgt ontleed:

Deze uitdrukking bestaat dus uit een identifier f naast een tripel (x, y, z), waarbij de spatie-operator gebruikt wordt om twee stukken formule naast elkaar te plaatsen. We gebruiken de spatie-operator dus om een functor uit te drukken.

Bij het evalueren van formules moeten haakjes dan van een uitdrukking met als wortel een komma-operator (bijvoorbeeld x, y, z) een n-tupel maken. Op die manier wordt $((x, y, z), (a, b, c))$ niet als het 6-tupel x, y, z, a, b, c geëvalueerd—hetgeen zou gebeuren als we de ()-operator alleen zouden beschouwen als een handige operator voor infix-uitdrukkingen die verder niets doet bij het evalueren. Dit is de enige plaats waar een ()-operator effectief iets *doet* bij het evalueren. In alle andere gevallen (bijvoorbeeld in $(1+2)*3$) geeft deze operator zijn argument onveranderd terug.

3.5 Haakjes als rotatie-barrière

Het is duidelijk dat rotaties niet mogen plaatsvinden met haakjes-objecten.

3.5.1 De breuk-operator

Als we de breuk-operator analoog als de vermenigvuldiging-operator zouden behandelen, dan zou het volgende gebeuren als de gebruiker op + drukt:

$$\frac{1}{2\Box} \quad \rightarrow \quad \frac{1}{2} + \Box$$

terwijl we eerder $\frac{1}{2+\Box}$ verwachten. Met andere woorden, het is alsof de teller en noemer impliciet door haakjes omgeven zijn. In de praktijk komt dit erop neer dat in de syntaxboom operatoren die zichzelf naar de top van de boom toe roteren tegengehouden moeten worden bij de breuk.

3.5.2 Rotaties bij machtsverheffingen

Als we de gewone regels zouden toepassen, dan zou het volgende gebeuren als de gebruiker op / drukt:

$$a^{b\square} \quad \rightarrow \quad \frac{a^b}{\square}$$

omdat de machtsverheffing-operator een hogere prioriteit heeft dan de deling. Wat we eerder verwachten is dat er $a^{\frac{b}{\square}}$ verschijnt. De exponent heeft m.a.w. impliciete haakjes. Ook het grondtal heeft impliciete haakjes, want anders zou

$$a\square^b \quad \rightarrow \quad \frac{a}{\square^b}$$

kunnen gebeuren als de gebruiker op / drukt. Wat we verwachten is natuurlijk $\frac{a}{\square}^b$. In dit geval zou $\left(\frac{a}{\square}\right)^b$ beter zijn. Voorlopig laten we het zo en voegen geen expliciete haakjes in.

3.6 Ternaire operatoren

We zullen eerst onderzoeken welke ternaire operatoren we zouden kunnen gebruiken.

In C is er de `?:` operator (zie ook verderop), waarbij de uitdrukking

$$vw\texttt{?}dit\texttt{:}dat$$

als volgt wordt geëvalueerd: evalueer eerst *vw*; als het resultaat "waar" is, dan is het resultaat van de `?:` operator *dit*; is het resultaat van *vw* "vals", dan is het resultaat van de `?:` operator *dat*. Een uitdrukking als $x = y?z = u : u$ wordt correct verwerkt, hoewel "=" in C een lagere prioriteit heeft dan "? :". Rond het middelste argument van "? :" staan dus impliciete haakjes, en dit argument kan niet roteren. De linkse en rechtse argumenten kunnen wel roteren (zo wordt $x = y?z = u : y$ als $x = (y?z = u : y)$ geïnterpreteerd). Een ternaire operator zouden we dus kunnen beschouwen als een binaire operator waar nog iets tussen hangt (dat niet in aanmerking kan komen voor rotaties).

De `?:` operator is ook rechts-naar-links associatief, dus een constructie als

$$vw_1\texttt{?}dit_1\texttt{:}$$
$$vw_2\texttt{?}dit_2\texttt{:}dat$$

wordt als

$$vw_1\texttt{?}dit_1\texttt{:(}vw_2\texttt{?}dit_2\texttt{:}dat\texttt{)}$$

verwerkt.

Voorlopig kunnen we vermijden ternaire operatoren te moeten invoeren door een binaire '?' en een binaire ':' operator te maken. Het nadeel hiervan is dat er geen impliciete haakjes rond het "middelste" argument kunnen geïmplementeerd worden.

3.7 De Prolog `op` operator declaratie

De programmeertaal **Prolog** laat toe zelf eigen operatoren in te voeren. Het is interessant de manier waarop dit aangepakt wordt eens nader te bekijken. In prolog wordt een operator gedefinieerd met

`op(`*precedentie, type, operator*`).`

waarbij de *precedentie* een getal tussen 1 en 1200 is, de *operator* de naam van de operator aangeeft, en *type* één van de volgende mogelijkheden is:

- `xfx` Infix-operator
- `xfy` Rechts-naar-links associatieve infix-operator
- `yfx` Links-naar-rechts associatieve infix-operator
- `fx` Prefix-operator
- `fy` Prefix-operator, links-naar-rechts associatief
- `xf` Postfix-operator
- `yf` Postfix-operator, rechts-naar-links associatief

Operatoren mogen ook in functie-notatie genoteerd worden in Prolog; zo is `<(x,y)` equivalent met `x<y`. We kunnen dit misschien overnemen door het patroon "␣op␣" (een identifier omgeven door twee spatie-operatoren) enkel als infix-operator te beschouwen als het rechter operand geen "()" operator is, en omgekeerd, een identifier met als rechteroperand een "()" operator te identificeren als een prefix-operator.

Niet-associatieve operatoren mogen zonder haakjes niet twee keer naast elkaar voorkomen. Dus als "plus" een `xfx` operator zou zijn, dan geeft "4 plus 5 plus 6" een foutmelding.

Rond operatoren met precedentie groter dan 1000 (de prioriteit van ',') moeten altijd haakjes staan. ***pending: voorbeeld, waarom, en wat als er geen staan?***

Typische standaardoperatoren:

Precedentie	Type	Operator(en)
200	xfy	`^`
300	xfx	`mod`
400	yfx	`* / // << >>`
500	fx	`+ -`
500	yfx	`+ - # /\ \/`
550	xfy	`:`
700	xfx	`= is =.. == \== @< @> @=< @>= =:= =\= < > =< >=`
900	fy	`\+ spy nospy`
1000	xfy	`,`
1050	xfy	`->`
1100	xfy	`;`
1150	fx	`mode public dynamic multifile block meta_predicate parallel sequential`
1200	fx	`:- ?-`
1200	xfx	`:- -->`

De precieze betekenis van al deze operatoren is op dit punt niet zo belangrijk.

Merk op dat wat we klassiek "operatoren met hoge prioriteit" noemen, hier overeenkomt met *kleine* getalwaarden! Om verwarring te vermijden, zullen we deze getallen maar **precedentie** noemen. Een hoge precedentie komt dus overeen met een lage prioriteit en omgekeerd.

3.8 De C operatoren

De belangrijkste operatoren van de taal **C**, in volgorde van hoge naar lage prioriteit (operatoren met zelfde prioriteit staan gegroepeerd tussen horizontale lijnen):

`()`	functieaanroep	→
`[]`	array	
`.`	element van een struct	
`->`	element van een pointer naar een struct	
`!`	logisch niet (unair prefix)	←
`~`	binair niet (unair prefix)	
`-`	tegengestelde van een getal (unair prefix)	
`++ --`	incrementeer/decrementeer (unair post- en prefix)	
`&`	adres (unair prefix)	
`*`	indirectie (unair prefix)	
`*`	vermenigvuldiging	→
`/`	deling	
`%`	rest na deling	
`+ -`	som/verschil van twee getallen	→
`<<`	binair naar links schuiven	→
`>>`	binair naar rechts schuiven	
`<`	kleiner dan	→
`<=`	kleiner dan of gelijk	
`>`	groter dan	
`>=`	groter dan of gelijk	
`==`	gelijk	→
`!=`	ongelijk	
`&`	binair en	→
`^`	binair exclusieve of	→
`|`	binair of	→
`&&`	logisch en	→
`||`	logisch of	→
`? :`		←
`=`	toekenning met varianten:	←
	`*= /= %= += -= <<= >>= &= ^= |=`	
`,`		→

← betekent: rechts-naar-links evaluatievolgorde
→ betekent: links-naar-rechts evaluatievolgorde

We zien dat in C sommige operatoren zowel unair als binair gebruikt kunnen worden (bv. de `*` en `-` operatoren), en andere zowel prefix als postfix (`++` en `--`). We zullen later moeten onderzoeken of en hoe we deze eigenschappen overnemen.

Een operator zowel unair als binair gebruiken, kan als volgt. Laten we als voorbeeld de "$-$" operator nemen. Een uitdrukking als "$a+-b$" heeft de volgende syntaxboom:

We kunnen dus niet goed afleiden of hier $+$ als unaire postfix-operator of $-$ als unaire prefix-operator gebruikt is. We kunnen het probleem in elk geval oplossen door $+$ en $-$ een leeg argument te doen behandelen als een 0.

3.9 Magma operatoren

Magma gebruikt de '!' operator als een soort cast-operator:

```
VS4![2*w,8,w+2,25]
```

maakt duidelijk dat `[2*w,8,w+2,25]` behoort tot `VS4` (wat bijvoorbeeld een vierdimensionale vectorruimte zou kunnen zijn).

Ook interessant is de where~... is operator:

uitdrukking1 **where** *var* **is** *uitdrukking2*

heeft als effect dat bij het evalueren alle voorkomens van *var* in *uitdrukking1* door *uitdrukking2* worden vervangen.

De C operator `? :` heet in Magma select~...else.

3.10 MIKE operatoren

Hier wordt '@' als cast-operator gebruikt:

```
gcd @ Integer ()
```

geeft aan dat `gcd` van het type `Integer ()` is.

3.11 Prioriteit van de operatoren

3.11.1 Klassieke operatoren

We beginnen met de operatoren uit de klassieke wiskunde:

Type	Operator(en)	Betekenis
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$*$ /	
bin $\rightarrow$	$+$ $-$	

waarbij in de eerste kolom $\leftarrow$ betekent dat de operator links-naar-rechts associativiteit heeft, en $\rightarrow$ een rechts-naar-links associatieve operator aanduidt.

nog: prefix -

3.11.2 Vergelijkingsoperatoren

We willen dat '$a < b = c > d$' als '$(a < b) = (c > d)$' wordt verwerkt; de operatoren $<>\leq\geq$ moeten dus hogere prioriteit hebben dan $=$ en $\neq$. Verder willen we dat '$a + b < c$' als '$(a + b) < c$' wordt geïnterpreteerd; de prioriteit van de vergelijkingsoperatoren moet dus lager zijn dan die van $+$. Dit geeft:

Type	Operator(en)	Betekenis
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$* \ /$	
bin $\rightarrow$	$+ \ -$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking

Merk tenslotte op dat een uitdrukking die klassiek als '$a > b > c$' genoteerd wordt, geassocieerd wordt tot '$(a > b) > c$', terwijl eerder '$a > b$ en $b > c$' bedoeld is. Een ingewikkelder voorbeeld is '$a \geq b = c > d$' wat weer niet als '$a \geq b$ en $b = c$ en $c > d$' verwerkt wordt. Om dit wel correct te verwerken zouden we van de vergelijkingsoperatoren een soort van n-aire operator moeten maken. Voorlopig zullen we dat voor de eenvoud nog niet doen.

3.12 Komma-operator

De komma geven we de allerlaagste prioriteit, zodat alles van de vorm $f_1, f_2, \ldots, f_n$ als een n-tupel wordt beschouwd (waarbij de f_i willekeurige formules zijn die geen spatie-operator bevatten). Een uitdrukking als

$$a + b, c = e, f$$

wordt dus gelezen als '$(a + b), (c = e), f$' en niet als '$(a + b, c) = (e, f)$'.

Om een n-tupel makkelijk van links naar rechts (d.i. van het eerste element naar het laatste) te verwerken, maken we de komma-operator rechts-naar-links associatief.

(Als we rechts-naar-links hadden gekozen, dan was het laatste element uit een n-tupel het makkelijkst bereikbaar.)

De operatorentabel wordt:

Type	Operator(en)	Betekenis
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$* \ /$	
bin $\rightarrow$	$+ \ -$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking
bin $\leftarrow$	,	

3.13 Spatie-operator

De spatie-operator laat ons toe ook uitdrukkingen als '$\sin 10$' te verwerken; de gebruiker kan dus een functie ingeven zonder dat er noodzakelijk haakjes moeten achter volgen.

Als voorbeeld lichten we toe op welke manieren “sinus van 10” geschreven kan worden:

Uit de laatste twee gevallen blijkt dat we ook formules van de vorm $(\frac{\sin}{\cos})x$ correct kunnen verwerken.

We zouden zo ook functies met meerdere argumenten kunnen verwerken, zodat ‘int x^2 x 0 θ’ als $\int_0^\theta x^2 \mathrm{d}x$ wordt geïnterpreteerd. Dit komt erop neer dat we een n-aire functie als n-aire prefix-operator beschouwen. Het alternatief is komma’s eisen, dus een invoer als ‘int x^2,x,0,θ’, hetgeen een stuk minder intuïtief is voor de gebruiker. Voorlopig zullen we komma’s eisen, totdat we de problemen met prioriteit en associativiteit van n-aire operatoren beter bestudeerd hebben.

3.13.1 Associativiteit van de spatie-operator

Om de associativiteit en prioriteit van de spatie-operator te bepalen, zullen we onderzoeken wat er gebeurt met een aantal uitdrukkingen.

We beginnen met ‘$x = y_\sqcup \mathrm{mod}_\sqcup 10$’. We kunnen de prioriteit van de spatie-operator groter of kleiner dan die van ‘=’ kiezen, evenals de associativiteitsrichting. We krijgen de volgende vier gevallen:

L→R, hoge pri	R→L, hoge pri	L→R, lage pri	R→L, lage pri
$x = ((y_\sqcup \mathrm{mod})_\sqcup 10)$	$x = (y_\sqcup (\mathrm{mod}_\sqcup 10))$	$((x = y)_\sqcup \mathrm{mod})_\sqcup 10$	$(x = y)_\sqcup (\mathrm{mod}_\sqcup 10)$

Na het herkennen van de operator ‘mod’ worden de syntaxbomen:

L→R, hoge pri, R→L, hoge pri	L→R, lage pri, R→L, lage pri

Afhankelijk van de prioriteit van ‘mod’ t.o.v. ‘=’ gebeurt eventueel nog een rotatie naar de andere vorm. We kunnen besluiten dat prioriteit en associativiteitsrichting van de spatie-operator voor dit voorbeeld er niet toe doen.

Vervolgens beschouwen we ‘not$_\sqcup$x+y’. Afhankelijk van de prioriteit van de spatie-operator krijgen we één van de volgende twee vormen:

Afhankelijk van de prioriteit van ‘not’ t.o.v. ‘+’ gebeurt eventueel nog een rotatie, die de ene vorm in de andere omzet. Ook hier doet de prioriteit van de spatie-operator er niet toe.

Als f en g twee functies zijn, dan willen we dat f g x als $f(g(x))$ wordt geïnterpreteerd:

We zouden in principe de associativiteit vrij kunnen laten en dan ook $(f(g))(x)$ kunnen hebben. Deze vorm zullen we wel correct verwerken, maar zonder haakjes verkiezen we toch de rechts-naar-links geassocieerde vorm. Vandaar maken we de spatie-operator rechts-naar-links associërend.

We zitten nog altijd met het probleem welke prioriteit we aan de spatie-operator moeten geven. Hiertoe beschouwen we de uitdrukking

$$x * f\_y$$

waarbij '_' een spatie-operator voorstelt. We willen dat dit geassocieerd wordt tot

$$x * (f\_y).$$

Hiertoe dienen we de spatie-operator een prioriteit hoger dan die van '*' te geven.

Een andere reden om de spatie-operator een hoge prioriteit te geven is dat hierdoor operatoren hun leven beginnen onderaan de syntaxboom en we dus enkel maar naar boven moeten kunnen roteren. Als de spatie-operator een lage prioriteit zou hebben, dan zouden de spaties onderaan de boom beginnen en dan hun weg naar boven moeten roteren. Wanneer dan een operator herkend wordt moet die weer terug naar beneden roteren. Dit eerst naar boven en dan naar beneden roteren is duidelijk minder efficiënt dan in één keer naar de juiste plaats toe gaan. Bovendien kan het zijn dat een operator naar beneden kan roteren op twee manieren, namelijk met zijn linkerkind of met zijn rechterkind. Het is dan niet duidelijk welke van de twee keuzes te verkiezen is. Dit heeft enkel belang voor de efficiëntie van het programma; het uiteindelijke resultaat is hetzelfde ongeacht met welk kind we roteren.

Een voorbeeld: stel dat de gebruiker de formule '$a + b * c$' wil uitbreiden tot '$a + b^2 * c$'. Eerst wordt een '^' ingevoegd:

Daarna een 2:

Nu het hele geval naar boven geroteerd is, wordt de '^' operator herkend en naar beneden geroteerd. We kunnen met het linkerkind of met het rechterkind roteren.

3.14 Booleaanse operatoren

De operatoren $\neg$, & en $\vee$ geven we dezelfde relatieve prioriteiten als de klassieke unaire $-$, binaire $*$ en $+$ operatoren. Voor de prioriteit van de exclusieve of $\oplus$ baseren we ons op de bitsgewijze operatoren van C en kiezen een prioriteit tussen & en $\vee$. De operatoren $\Rightarrow$ en $\Leftrightarrow$ krijgen een lage prioriteit, omdat we willen dat

ze als 'scheiders' werken, zodat formules als $(A\&B) \Rightarrow (C \vee D)$ zonder haakjes geschreven kunnen worden. De prioriteit van $\neg$ moet lager zijn dan die van $=$ als we willen dat $\neg(a + b = c)$ zonder haakjes mag geschreven worden.

Dit geeft:

Type	Operator(en)	Betekenis
bin	$\sqcup$	Spatie
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$* /$	
bin $\rightarrow$	$+ -$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking
un	$\neg$	Negatie
bin $\rightarrow$	&	Logisch en
bin $\rightarrow$	$\oplus$	Logisch exclusief of
bin $\rightarrow$	$\vee$	Logisch inclusief of
bin $\rightarrow$	$\Rightarrow \Leftrightarrow$	
bin $\leftarrow$	,	

3.15 Kwantoren

We zullen naast de prefix-operatoren $\exists$ en $\forall$ ook nog een binaire hulp-operator ':' definiëren. De bedoeling is dat we komen tot formules als

$$\forall x \in R : \exists y \in Q : x = y$$

De invoering van de $\in$ operator zullen we verderop gedetailleerder behandelen. We willen dat de vorige formule wordt behandeld als

$$\forall(x \in R) : (\exists(y \in Q) : (x = y))$$

De prioriteit van de kwantoren moet dus lager zijn dan die van $=$. In plaats van $x = y$ zou er ook een formule kunnen gestaan hebben met $\Rightarrow$ erin, zodat we de prioriteit van de kwantoren lager kiezen dan die van $\Rightarrow$.

Vervolgens moeten we vermijden dat de kwantoren verkeerd geassocieerd worden. Er zijn vier mogelijke associaties, die in elkaar omgezet kunnen worden door rotatie van de ':' operatoren:

De linkse boom is de juiste. We kunnen de andere drie gevallen naar dit geval herleiden door een extra rotatieregel in te voeren:

waarbij Q voor een kwantor ($\exists$ of $\forall$) staat, en a en b deelbomen zijn.

Type	Operator(en)	Betekenis
bin	$\sqcup$	Spatie
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$*$ /	
bin $\rightarrow$	$+$ $-$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking
un	$\neg$	Negatie
bin $\rightarrow$	&	Logisch en
bin $\rightarrow$	$\oplus$	Logisch exclusief of
bin $\rightarrow$	$\vee$	Logisch inclusief of
bin $\rightarrow$	$\Rightarrow \Leftrightarrow$	
bin/un $\rightsquigarrow$	$\exists \forall$:	Kwantoren
bin $\leftarrow$	,	

waarbij het symbool $\rightsquigarrow$ duidt op de speciale rotatieregel.

3.16 Verzamelingen

We willen verzamelingen kunnen omschrijven in de aard van

$$\{x \in R : \exists y \in Z : x = y\}$$

We willen dat het enige argument van $\{\}$ een ':' operator is met als linkerargument $x \in R$ en als rechterargument de rest van de formule. Hiertoe moeten we de ':' operator rechts-naar-links associatief maken.

In Magma kunnen we verzamelingen ook door opsomming definieren:

$$\{Z : 10, 3, 5\}$$

geeft aan dat de opgesomde elementen tot Z behoren. In dit eenvoudige geval lijkt het een beetje overbodig expliciet Z te moeten vermelden (Magma kan in dit geval inderdaad zelf uitmaken dat $10 \in Z$ enzovoort), maar soms is het niet a priori duidelijk tot welke verzameling een element behoort: $(4, 6, 1)$ kan een element zijn van $Sym(7)$ maar ook van $Sym(8)$. Het linkerargument van ':' kan dus ook een verzameling zijn.

3.17 Functie-operator

We willen functies schrijven als

$$(x) \mapsto (y + z^2)$$

waarbij we de prioriteit van '$\mapsto$' dus lager moeten kiezen dan die van $+$ en $-$ om de haakjes hier te mogen weglaten. We willen de volgende uitdrukking ook

zonder haakjes mogen schrijven:

$$x \mapsto (y + z = 0)$$

waarbij het type van het beeld $y+z=0$ booleaans is (waar of vals). We willen in het beeld dus alle booleaanse operatoren, tot $\exists$ toe, kunnen kiezen. Dus moeten we '$\mapsto$' een lage prioriteit geven.

Type	Operator(en)	Betekenis
bin	$\sqcup$	Spatie
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$* \ /$	
bin $\rightarrow$	$+ \ -$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking
un	$\neg$	Negatie
bin $\rightarrow$	$\&$	Logisch en
bin $\rightarrow$	$\oplus$	Logisch exclusief of
bin $\rightarrow$	$\vee$	Logisch inclusief of
bin $\rightarrow$	$\Rightarrow \Leftrightarrow$	
bin $\leftarrow$/un $\rightsquigarrow$	$\exists \ \forall :$	Kwantoren
bin	$\mapsto$	functie
bin $\leftarrow$	,	

3.18 Logica-connectieven

De connectieven $\vdash$, $\models$ en $\rhd$ moeten in elk geval een prioriteit hebben lager dan de kwantoren, omdat ze als scheiders voor formules moeten kunnen werken:

$$(\exists x \in R : blabla) \rhd (\forall y \in Q : blabla)$$

moet ook zonder haakjes kunnen geschreven worden. De prioriteit moet zelfs lager zijn dan die van de komma, omdat we ook

$$(x \& y, z) \models x$$

zonder haakjes willen schrijven.

3.19 Toekenningsoperatoren

De toekenningsoperator $:=$ moet in elk geval een lagere prioriteit krijgen dan $\mapsto$ omdat we de volgende uitdrukking zonder haakjes willen noteren:

$$f := (x \mapsto x + 1)$$

We krijgen dus:

Type	Operator(en)	Betekenis
bin	$\sqcup$	Spatie
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$*$ /	
bin $\rightarrow$	$+$ $-$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking
un	$\neg$	Negatie
bin $\rightarrow$	&	Logisch en
bin $\rightarrow$	$\oplus$	Logisch exclusief of
bin $\rightarrow$	$\vee$	Logisch inclusief of
bin $\rightarrow$	$\Rightarrow$ $\Leftrightarrow$	
bin $\leftarrow$/un $\rightsquigarrow$	$\exists$ $\forall$:	Kwantoren
bin	$\mapsto$	functie
bin $\leftarrow$	$:= += -=$	toekenning
bin $\leftarrow$	,	
bin	$\vdash \models \rhd$	Logica-connectieven

Naast de drie toekenningsoperatoren uit de tabel hebben we ook nog $\wedge=$, $*=$, $/=$, $\&=$, $\oplus=$, $\vee=$, $\Rightarrow=$ en $\Leftrightarrow=$ die we weglaten om te tabel niet te overladen.

3.20 Verzamelingsoperatoren

Omdat we

$$(a \cup b) \subseteq (d \setminus f)$$

haakjesloos willen schrijven, kiezen we voor $\cup$, $\cap$ en $\setminus$ een hogere prioriteit dan $\in$, $\subseteq$, $\notin$, $\not\subseteq$, $\subsetneq$, $\ni$, $\supseteq$, $\not\ni$, $\not\supseteq$ en $\supsetneq$.

We willen ook

$$\neg(a \subseteq b)$$

zonder haakjes schrijven, dus geven we al de verzamelingsoperatoren een hogere prioriteit dan $\neg$.

Type	Operator(en)	Betekenis
bin	$\sqcup$	Spatie
bin $\leftarrow$	$\hat{}$	Machtsverheffing
bin $\rightarrow$	$* /$	
bin $\rightarrow$	$+ -$	
bin $\rightarrow$	$< > \leq \geq$	Vergelijking
bin $\rightarrow$	$= \neq$	Vergelijking
bin	$\cup \cap \setminus$	Verzameling
bin	$\in \subseteq \cdots$	Verzameling
un	$\neg$	Negatie
bin $\rightarrow$	&	Logisch en
bin $\rightarrow$	$\oplus$	Logisch exclusief of
bin $\rightarrow$	$\vee$	Logisch inclusief of
bin $\rightarrow$	$\Rightarrow \Leftrightarrow$	
bin $\leftarrow$/un $\rightsquigarrow$	$\exists \forall :$	Kwantoren
bin	$\mapsto$	functie
bin $\leftarrow$	$:= += -= \cdots$	toekenning
bin $\leftarrow$	,	
bin	$\vdash \models \rhd$	Logica-connectieven

3.21 Prioriteiten: implementatie

We geven elke binaire operator *drie* prioriteiten. Een unaire operator krijgt er twee.

Als een operator de prioriteit van (één van) zijn kinderen vraagt, dan wordt `getPri()` gebruikt. Als een linkerkind de prioriteit van de vader-operator vraagt, dan wordt `getLeftPri()` gebruikt; analoog voor de andere gevallen. Visueel betekent dit dat bij elk verbindingsstreepje tussen twee operatoren de prioriteiten "goed" moeten staan, d.i. de hoogste prioriteit onderaan.

Als voorbeeld bekijken we hoe we hiermee rechts-naar-links associativiteit van de ':=' operator kunnen implementeren. Als `getPri()` prioriteit kiezen we 600, als linker prioriteit 610 (`getLeftPri()`) en als rechter prioriteit 590 (`getRightPri()`). Het effect hiervan is dat het rechterkind van een ':=' operator deze operator ziet met een prioriteit van 590, het linkerkind ziet een prioriteit van 610 en de ouder van een ':=' operator ziet prioriteit 600.

In de linkervorm staat 610 bovenaan en 600 onderaan een verbindingsstreepje. De hoogste prioriteiten moeten echter onderaan staan; daarom is een rotatie noodzakelijk. Als we de prioriteiten van alle andere operatoren kleiner dan 590 en groter dan 610 nemen, dan gedraagt ':=' zich voor de andere operatoren

alsof ':=' prioriteit 600 heeft. Dit is het gedrag dat we van een rechts-naar-links associatie verwachten: het beïnvloedt de associatie van twee naast elkaar gelegen ':=' operatoren en laat de associatie van andere operatoren ongewijzigd.

Ook de speciale rotatieregel voor ':' kan hiermee afgehandeld worden. We geven ':' een prioriteit van 810 en '$\forall$' en '$\exists$' een prioriteit van 800.

De prioriteiten leggen we als volgt vast:

Type	Operator(en)	Prioriteit
bin	$\sqcup$	10000
bin $\leftarrow$	$\hat{}$	2000
bin $\rightarrow$	$* \ /$	1900
bin $\rightarrow$	$+ \ -$	1800
bin $\rightarrow$	$< > \leq \geq$	1700
bin $\rightarrow$	$= \neq$	1600
bin	$\cup \cap \setminus$	1500
bin	$\in \subseteq \cdots$	1400
un	$\neg$	1300
bin $\rightarrow$	$\&$	1200
bin $\rightarrow$	$\oplus$	1100
bin $\rightarrow$	$\vee$	1000
bin $\rightarrow$	$\Rightarrow \Leftrightarrow$	900
bin $\leftarrow$/un $\rightsquigarrow$	$\exists \ \forall :$	800
bin	$\mapsto$	700
bin $\leftarrow$	$:= \ += \ -= \cdots$	600
bin $\leftarrow$	,	500
bin	$\vdash \models \rhd$	400

Om een nieuwe operator aan te maken, moet een nieuwe subklasse van `InfixBinaryOp` of `PrefixUnaryOp` geprogrammeerd worden, samen met een `InfixBinaryOpFactory` of `UnaryOpFactory`. In de klasse `StandardOperators` worden alle operatoren die standaard aanwezig zijn, geregistreerd in twee centrale tabellen. Deze tabellen zitten in de klassen `InfixBinaryOp` of `PrefixUnaryOp`. Ze bevatten (string, factory) paren, waarbij de string de naam van een operator is (bijvoorbeeld `"union"` of `"+"`) en de factory een object is dat zo'n operator kan aanmaken (bijvoorbeeld `OperatorUnionFactory` resp. `OperatorPlusFactory`). De formule-editor raadpleegt deze tabellen tijdens het intikken van een formule. Meer bepaald is het de spatie-operator die probeert van identifiers operatoren te maken (zie `OperatorSpace.regocnizeOp()`). Als zo'n identifier in één van de operatorentabellen staat, dan maakt `recognizeOp()` er een operator van met behulp van de overeenkomstige `Factory`.

3.22 Types

Als types hebben we voorlopig `Matrix`, `Interval`, `Boolean` en `Real` (dat eigenlijk beter Double zo heten). Bij het evalueren van een uitdrukking kunnen identifiers in `Boolean` of `Real` omgezet worden. Intervallen kan de gebruiker voorlopig zelf nog niet ingeven.

We willen dat de gebruiker zelf types en operatoren kan bijmaken. Hiertoe is het nodig dat de operatoren overloading ondersteunen: we maken verschillende versies van een operator met eenzelfde naam. Zo zullen we van de '+' operator varianten hebben die twee Reals optellen, twee intervallen, twee matrices, ... Voorlopig hebben we één versie van alle operatoren, die argumenten van gelijk welk type aanvaarden.

Nieuwe types moeten op een of andere manier uit reeds standaard aanwezige types afgeleid worden. Welke standaardtypes hebben we nodig, hoe maken we daarmee nieuwe types, en wat voor informatie moeten we per type eigenlijk precies bijhouden?

3.22.1 Standaardtypes

Enkelvoudige types:

- Boolean
- Natural $\subset$ Integer $\subset$ Rational $\subset$ Real $\subset$ Complex
  waarbij '$\subset$' aanduidt dat alle elementen van het linkse type ook behoren tot het rechtse. We moeten dus ook een notie 'subtype' kunnen verwerken. Als we een operator definieren op een supertype (bijvoorbeeld Complex), dan moeten we die in principe niet implementeren op de subtypes; de operator wordt overgeërfd door de subtypes. Als we dus de '+' operator alleen definiëren voor twee complexe argumenten, dan kunnen we ook twee Naturals bij elkaar optellen (omdat Natural $\subset$ Complex). Van welk type zou het resultaat dan moeten zijn? Moet de optelling slim genoeg zijn om bijvoorbeeld te zien dat het resultaat van $(6 - i) + i$, de som van twee Complex objecten, een Natural is? Wat als de gebruiker een type Even maakt met als objectenverzameling alle even getallen? Moet het resultaattype van de optelling dan 'Even' worden (daar dit type specifieker is dan Natural)? Maar als we ook nog een type Drievoud zouden maken, dan zou het resultaattype net zo goed Drievoud moeten zijn. Er bestaat dus geen 'meest specifieke' type voor het object '6'. Daarom lijkt het beter om '6' zowel tot al de types Even, Drievoud, Natural, ..., Real en Complex te laten horen. Vraag is welke variant van '+' we moeten kiezen als deze operator zowel Even- als Drievoud-versies heeft. We stellen volgende oplossing voor: van elk object wordt behalve de waarde (bijvoorbeeld '6') ook het type (bijvoorbeeld 'Natural') bijgehouden. Een ingetikt getal wordt bij het evalueren in

/'e/'en van de types Natural, Integer of Real ingedeeld. Objecten van andere types onstaan door bewerkingen op deze drie types (bijvoorbeeld, deel twee Integers om een Rational te verkrijgen). Om objecten van niet standaard ingebouwde types (Even, Drievoud) te maken, moet een of andere cast-operator gemaakt worden.

Samengestelde types:

- Verzamelingen; zowel door opsomming (bv. $\{1, 2, 3\}$ als door omschrijving. (bv. $\{x \in \mathbb{N} : x > 0\}$). Om verzameling-paradoxen te vermijden moeten we wel opletten dat 'verzamelingen door omschrijving' ook echt wel verzamelingen voorstellen. Een mogelijke eis is dat de rang van een verzameling eindig moet zijn; er mogen met andere woorden geen oneindige ketens deelverzamelingen ($a \in b \in c \cdots \in X$) in een verzameling X voorkomen.

- Matrices. We zouden hier een 'geparametriseerd type' van kunnen maken door speciale notaties uit te vinden voor 'matrix van Integers' of '$n \times m$-matrix' enzovoort, maar het is misschien handiger en algemener om dit te laten oplossen door een 'verzameling' type: $\{m \in Matrix : m(i, j) \in Integer\}$ of $\{mat \in Matrix : rows(mat) = n \& cols(mat) = m\}$. In elk geval zien we dat verzameling-types ook parameters (m en n in het laatste voorbeeld) kunnen bevatten.

- Lijsten. We hebben hiervoor de komma-operator tot onze beschikking: a, b maakt een lijst met twee elementen. Probleem is dat a, b, c geassocieerd wordt als $a, (b, c)$. We willen liever niet dat de uitdrukking 'a, b, c' een lijst van twee elementen (resp. a en de lijst 'b, c') voorstelt. Daarom eisen we dat komma-operatoren kind moeten zijn van ofwel een andere komma-operator, ofwel van een Brackets-knoop. De Brackets maken van een $a, b, c, \ldots$ uitdrukking een lijst bij het evalueren; de komma-operatoren zelf doen niets bij het evaluaren. Met de verzameling-types kunnen we lijsttypes maken: $\{(a, b) \in List : a \in \mathbb{R}, b \in \mathbb{N}\}$ stelt een type voor met als elementen koppels met als eerste component een Real en als tweede een Natural. Misschien is het handig voor deze verzameling de afkorting $\{(a \in \mathbb{R}, b \in \mathbb{N})\}$ toe te staan.

- Functies. Hiertoe kunnen we de $\mapsto$ operator gebruiken:

$$f := x \in \mathbb{R} \mapsto x^2$$

stelt een object van het type 'functie' voor. Om 'de verzameling van alle $\mathbb{R} \to \mathbb{R}$ functies' uit te drukken, zouden we '$\mathbb{R} \mapsto \mathbb{R}$' kunnen proberen te schrijven. Dit zou ook kunnen geïnterpreteerd worden als "de functie die $\mathbb{R}$ afbeeldt op $\mathbb{R}$", dus niet als een verzameling van functies, maar als één welbepaalde functie waarvan zowel domein als beeld uit slechts één element

('$\mathbb{R}$') bestaan (net als '$1 \mapsto 1$'). We moeten dus een andere notatie kiezen, bijvoorbeeld

$$\{f \in Function : dom(f) = \mathbb{R} \& img(f) = \mathbb{R}\}$$

We zien dat voor alle types een algemene naam nodig is ('Matrix', 'List', 'Function') die de verzameling van alle objecten van dat type aangeeft.

- Groepen.
- Grafen.
- Programma's.

Blijkbaar is het voldoende dat we per type enkel de verzameling bijhouden van objecten die er toe behoren. Met andere woorden, een type is gewoon maar een verzameling-object. Geparametriseerde types zijn gewoon maar functies die een verzameling als resultaattype hebben; zo kunnen we het type '$n \times m - matrix$' noteren als $Mat(n, m)$ nadat we 'Mat' gedefinieerd hebben als de functie

$$Mat := (n \in \mathbb{N}, m \in \mathbb{N}) \mapsto \{mat \in Matrix : rows(mat) = n \& cols(mat) = m\}$$

We hebben ook opgemerkt dat de basistypes een eigen identifier moeten krijgen ('Matrix', 'List', 'Integer'). We zouden deze identifiers ook weer objecten kunnen laten worden, ditmaal van het type 'Type'.

3.23 Interne werking

(Noot: Deze paragraaf is behoorlijk verwarrend en verouderd.)

Alle klassen die met formules te maken hebben, zitten in het package **formula**. De klasse **formula.Formula** kapselt de interne details in en zorgt voor het afbeelden van de componenten, het doorsturen van toetsdrukken naar de actieve component, enz. De elementen van de syntaxboom zelf (dit zijn allemaal **formula.Part** objecten) weten niets over de plaats waar ze op dat ogenblik afgebeeld staan en de manier waarop ze grafisch gepresenteerd worden. De bedoeling is enkel de syntaxboom als datastructuur voor te stellen. Zo weet een **formula.Graph** object niet op welke positie een bepaalde top getoond wordt. De gebruikersinterface-informatie zit in het **formula.Formula** object, en de koppeling tussen een **Part** en de grafische info gebeurt met de **Formula.getCustomData()** en **Formula.putCustomData()** methoden. Zo zal een **Graph** object als custom data een **GraphView** object laten registreren bij de **Formula** waar het deel uit van maakt. In dit **GraphView** object worden dan de schermposities van de toppen van de graaf bewaard. De syntaxboom is dubbelgelinkt; elk Part heeft een **parent** veld, en Parts die één of meerdere kinderen bevatten, hebben velden waar die kinderen in zitten.

De **Formula** houdt ook bij in welk van zijn **Part**s de cursor zich op dat moment bevindt (in **Formula.cursorPart**) en op welke positie in dat **Part** (in de int **Formula.cursorPos**). Wat dit getal betekent voor de positie moet het **Part** zelf uitmaken; bij **Identifier** Parts bijvoorbeeld betekent 0 dat de cursor volledig links staat, 1 dat de cursor na de eerste letter van de identifier staat, enz; bij **Fraction** Parts (die breuken voorstellen) betekent -1 dat de cursor links van de breuk staat, 1 dat de cursor rechts ervan staat, en 0 dat de cursor op de breukstreep zelf staat. Een extra afspraak hier is dat cursorpositie 0 door alle Parts op een zinvolle "default" manier moet geïnterpreteerd worden.

Een **Part** weet hoe het zichzelf moet tekenen (met de **Part.paint()** methode). Dit is niet in tegenspraak met wat we eerder vermeldden, nl. dat een **Part** niets weet over de grafische voorstelling; het **Part** krijgt deze informatie immers pas via de argumenten van **paint()** binnen.

pending: meer uitleg over de internalia (bv welke objecten moeten de cursor links/rechts van zichzelf kunnen geplaatst hebben)

4 Formele logica

4.1 Bewijsregels

4.1.1 Testgevallen

Voor A:

$$1 \quad a, b, c \vdash a$$

$$1 \quad a, b, c \vdash b$$

$$1 \quad a, b, c \vdash c$$

(OK op 23-nov-1999)

Voor C:

1	$a \vdash a$	$prem$
2	$b \vdash c$	$prem$
3	$a, b \vdash a \& c$	$C(1, 2)$

(OK op 24-nov-1999)

Voor C1 en C2:

1	$a \vdash b \& c$	$prem$
2	$a \vdash b$	$C1(1)$
3	$a \vdash c$	$C2(1)$

(OK op 7-dec-1999)

Voor R:

1	$a, b, c \vdash d$	$prem$
2	$a, \neg b, c \vdash d$	$prem$
3	$a, c, x \vdash d$	$R(1, 2)$

(OK op 5-dec-1999)

Voor X:

$$\begin{array}{lll} 1 & a \vdash b & prem \\ 2 & a \vdash \neg b & prem \\ 3 & a \vdash c & X(1,2) \end{array}$$

(OK op 5-dec-1999)

5 Functieplotter

De meeste klassieke functieplotters tekenen de grafiek van een functie $y = f(x)$ door in een aantal x-waarden de functie f te evalueren en vervolgens de gevonden punten te verbinden. Problemen treden op wanneer de functie sterk schommelt en te weinig punten worden bekeken, of wanneer de functie een discontinuïteit heeft, omdat dan de punten horend bij twee naast elkaar gelegen x-waarden *niet* met elkaar verbonden mogen worden.

Hierdoor kan soms een totaal verkeerd beeld van de functie onstaan, zoals zal blijken uit de volgende voorbeelden. Met de opkomst van de grafische rekenmachines is dit probleem ook van didaktische aard geworden; de uitleg "de studenten moeten maar leren de apparatuur niet blindelings te vertrouwen" klinkt aannemelijk als een aanvaardbaar algemeen principe, maar wordt hier toch wel een beetje misbruikt om software-gebreken goed te praten. Per slot van rekening is de computer uitgevonden om routineklussen aangenamer te maken, en niet om extra hoofdpijn te introduceren :-)

Een mogelijke oplossing voor dit probleem is de volgende. De klassieke functieplotter evalueert de functie in een aantal punten $x_0, x_1, \ldots, x_n$. Wij gaan de functie evalueren in een aantal *intervallen* $[x_0, x_1[, [x_1, x_2[, \ldots, [x_{n-2}, x_{n-1}[, [x_n - 1, x_n]$. Hoe we dat technisch aanpakken (en de hierbij opduikende problemen) zullen we verderop behandelen. Het voordeel van deze aanpak zal blijken uit de volgende voorbeelden.

5.1 De functie $f(x) := \sqrt{1 - x^2}$

Als we in Gnuplot deze functie plotten met

```
plot [x=-2:2] sqrt(1-x**2)
```

dan krijgen we het volgende te zien:

De plotter mist de linker- en rechterkant van de grafiek. Ook Maple V Rel 3 vertoont hetzelfde gebrek. De verklaring is eenvoudig: stel dat de plotter punten plot met x-waarden die telkens Δx verschillen, stel bv. $\Delta x = 0.1$. Op een gegeven

moment komt de plotter aan bv. $x = -1.05$; de functie heeft daar geen waarde, en er wordt geen punt geplot. Het volgende punt is $x = -0.95$. Daar heeft de functie de waarde 0.0975, en dit wordt het eerste geplotte punt van de grafiek.

Als we daarentegen met de interval-aanpak werken, geeft $f([-1.05, -0.095[)$ het interval $[0, 0.975[$ terug. Er wordt dus geen enkel punt gemist.

5.2 De functie $\frac{1}{x}$

Deze functie vertoont een discontinuïteit in $x = 0$. Veel wiskundeprogramma's hebben problemen met dit soort discontinuïteiten; Maple kan deze functie bijvoorbeeld niet integreren tussen -1 en 1 ("`unable to handle singularity`"), Derive geeft (denk ik) een fout resultaat (in de manual staat dat de gebruiker zelf voor zo'n situaties moet opletten ***controleren***).

Als we in Gnuplot deze functie plotten met

```
plot [x=-1:1] 1/x
```

dan krijgen we het volgende te zien:

De plotter verbindt hier de twee takken van de grafiek met elkaar.

Maple geraakt nog meer in de war van zulke grote functiewaarden rond $x = 0$ en geeft het volgende curiosum:

Hier heeft de functieplotter ergens een punt $(-2 \cdot 10^{-10}, -5 \cdot 10^9)$ geplot; het volgende punt is iets in de aard van $(0.1, 10)$ ($\Delta x = 0.1$ maar bij het bereiken van $x = 0$ is de afrondingsfout $x \approx -2 \cdot 10^{-10}$ gebeurd). Hierdoor is het gevonden y-bereik $-5 \cdot 10^9..100$ geworden, met alle gevolgen van dien. Wie goed kijkt, ziet dat de plotter bovendien beide takken met elkaar heeft verbonden.

Het gebruik van de interval-methode lost bovengenoemde problemen op. Ten eerste is er het verbinden van de twee takken. Dit probleem ontstaat doordat de plotter het laatste punt van de linkertak (bv. $(-0.1, -10)$) met het eerste punt van de rechtertak (bv. $(0.1, 10)$ met $\Delta x = 0.2$) verbindt. Alleen als de plotter toevallig het punt $x = 0$ beschouwt, kan de singulariteit gedetecteerd worden. De interval-methode levert: $\frac{1}{[-0.1,0.1[} = [-10, -\infty[\cup]\infty, 10[$ en toont dus twee disjuncte takken, die bovendien reiken tot aan ∞.

We merken op dat om deze situatie correct af te handelen rekening moet gehouden worden met twee problemen:

- De plotter moet overweg kunnen met de waarde ∞, en onderscheid maken tussen $+\infty$ en $-\infty$

- Het resultaat van een functie die op een interval inwerkt, is soms niet meer een interval maar een aantal disjuncte intervallen

Ten tweede is er het probleem dat sommige functiewaarden rond een asymptoot zeer groot in absolute waarden kunnen worden. De puntsgewijze plotters proberen het Y-bereik automatisch te vinden door alle gevonden punten te tonen, met alle problemen van dien. De intervalplotter vindt rond een asymptoot echt de waarde ∞ als uiteinde van een interval; als we met deze intervallen geen rekening houden bij het automatisch zoeken naar een Y-bereik, dan worden alleen de "interessante" stukken getoond.

5.3 De functie $\sin\frac{1}{x}$

Als we in Gnuplot deze functie plotten met

```
set samples 1000
plot [x=-1:1] sin(1/x)
```

dan krijgen we het volgende te zien:

Hier blijven een hoop pixels rond $x = 0$ wit, terwijl een correcte plot een volledig zwarte balk van de vorm $[-\delta, \delta] \times [-1, 1]$ zou moeten bevatten. Ook Maple V heeft hier last van. Dit probleem wordt duidelijk veroorzaakt doordat te weinig punten bemonsterd worden. Het probleem is dat de plotter nooit kan weten wanneer er genoeg punten bekeken zijn.

De interval-methode heeft hier geen problemen mee: $f([0.01, 0.02[) = [-1, 1]$; er worden geen pixels wit gelaten.

5.4 De interval-methode

Hiervoor is gebleken dat een interval-functieplotter voordelen heeft tegenover de klassieke puntsgewijze functieplotters. De vraag is natuurlijk of en hoe we zo'n interval-functieplotter kunnen maken.

De eenvoudigste aanpak is alle operatoren niet alleen als input een getal te laten nemen, maar ook een interval. Als voorbeeld bekijken we hoe de operator $\sqrt{}$ geïmplementeerd moet worden (voor het gemak werken we in de reële getallen zodat we geen complicaties krijgen door complexe getallen):

- Als het argument een getal is, dan is het resultaat een symbool "ongedefinieerd" wanneer het getal negatief is; in het andere geval berekenen we de wortel van het getal en geven dat als resultaat terug.

- Als het argument een interval van de vorm $[a, b]$ is (we veronderstellen $a < b$), dan onderscheiden we de volgende getallen:

  - het symbool "ongedefinieerd" als $b < 0$
  - het getal 0 als $b = 0$
  - het interval $[0, \sqrt{b}]$ als $b > 0$ en $a < 0$
  - het interval $[\sqrt{a}, \sqrt{b}]$ als $b > 0$ en $a \geq 0$.
- Voor argumenten van de vorm $]a, b]$, $[a, b[$ of $]a, b[$ is het algoritme analoog.

We zien dus dat het werken met intervallen als argument conceptueel niet veel ingewikkelder is dan het werken met getallen. Toch schuilt er een addertje onder het gras. Stel dat we $f(x) = x - x$ willen plotten. We moeten dan $f([a, b[)$ voor een aantal a- en b-waarden ($a < b$) berekenen. Bij het uitrekenen van deze uitdrukking vinden we $f([a, b[) = [a, b[-[a, b[=]a - b, b - a[$ en niet 0. Met andere woorden, waar puntsgewijs plotten te weinig punten oplevert (als we de gevonden punten tenminste niet met lijnstukken verbinden), vinden we hier te veel punten.

Een mogelijke oplossing is de volgende. We proberen beide methoden als volgt te verzoenen: bereken eerst $f([a, b[)$ met de intervalmethode. Probeer vervolgens in een aantal x-waarden met $x \in [a, b[$ de functie f te evalueren. Als we in elke pixel van $f([a, b[)$ een $f(x)$ vinden, dan zijn we zeker dat we niet teveel punten gevonden hebben met de intervalmethode. Blijken er na proberen van een bepaald aantal x-waarden pixels van $f([a, b[)$ niet te bedekken met $f(x)$-waarden, dan kiezen we een $c \in]a, b[$ (bijvoorbeeld halfweg tussen a en b) en beschouwen nu $f([a, c[) \cup f([c, b[)$ in plaats van $f([a, b[)$. Dit gebied is mogelijks kleiner dan $f([a, b[)$. We herhalen de procedure van het zoeken van $f(x)$ waarden om alle pixels van $f([a, c[) \cup f([c, b[)$ te bedekken. Het is natuurlijk mogelijk dat we na herhaald invoeren van extra verdelingen van het interval $[a, b[$ nog altijd punten te veel blijven vinden. Na een bepaald aantal verdelingen zouden we kunnen besluiten er de brui aan te geven en alle pixels waarvoor we een $f(x)$ gevonden hebben in het zwart te kleuren, en alle overige punten van $f([a, c[) \cup \cdots \cup f([c, b[)$ in het grijs.

6 Grafen

6.1 Doelstelling

Om te beginnen moeten we een datastructuur zoeken om grafen voor te kunnen stellen. Voor het gemak zullen we geen multibogen ondersteunen. We wensen ook op een of andere manier eigenschappen te kunnen hechten aan toppen (zoals kleuren) en bogen (zoals gewichten).

We wensen ook dat de gebruiker deze grafen makkelijk kan manipuleren. De gebruikers-interface is dus een niet te verwaarlozen aspect. De graaf wordt nu getoond als een vlakke figuur; een toekomstige uitbreiding is het gebruik van drie dimensies.

Tenslotte willen we natuurlijk ook een aantal interessante operaties op grafen ondersteuenen. Een aantal hiervan maken gebruik van noties uit andere onderdelen van het programma, zoals verzamelingen, groepen, ...

6.1.1 Gewenste operaties

Om te beginnen wensen we uit een graaf informatie te kunnen halen; zo zullen we een functie willen om de **adjacentiematrix** van een graaf te bepalen (het resultaat is een matrix) en om de verzameling van bogen te bepalen (het resultaat is een verzameling).

Een functie om te bepalen of twee grafen **isomorf** zijn. Bovendien moet de gebruiker ook kunnen zien **waarom** het antwoord ja of nee is. Als ze isomorf zijn, laat het programma een animatie zien waarin de ene graaf in de andere verandert; als ze niet isomorf zijn kan bijvoorbeeld een top in één van de twee grafen getoond worden die een valentie heeft die verschilt van de valenties van alle toppen in de andere graaf.

Bij gekleurde grafen moet een isomorfisme ook een bijectie tussen de kleuren van beide grafen bepalen; in dit kader is een functie die van een gekleurde graaf een kleurloze graaf maakt wenselijk.

Een functie om de automorfismengroep van een graaf te bepalen.

Functies om grafen te genereren: een aantal voorgedefinieerde grafen (zoals de Petersen graaf), een functie om een complete graaf van n toppen, een polygon van orde n, de graaf die overeenkomt met één van de vijf regelmatige veelvlakken te genereren. Voor het laatste geval zou het interessant zijn de graaf in drie dimensies voor te kunnen stellen.

Een functie om de graad van een top te bepalen (bij gerichte grafen: ingraad en uitgraad). Dit veronderstelt dat de toppen op een of andere manier gelabeld zijn.

Een functie om te bepalen of een graaf een opspannende deelgraaf is van een andere graaf.

Een functie om de samenhangende componenten van een graaf te vinden. Het resultaat is een verzameling van deelgrafen van de oorspronkelijke graaf.

Functies om met wandelingen te werken (lengte, bepalen of een wandeling een (gesloten / enkelvoudig) pad is) en om te testen of een graaf een boom is.

Een functie om te bepalen of een pad een Eulerpad is van een graaf, en een functie om te bepalen of een graaf Euleriaans is (de **waarom** functie moet dan bijvoorbeeld een gevonden Eulerpad tonen).

Een functie om bruggen te zoeken in een graaf, een functie om te bepalen of een graaf bruggen bevat (de **waarom** functie kan dan een gevonden brug tonen).

Functies om te bepalen of een pad Hamiltoniaans is, om Hamiltoncykels in een graaf te zoeken, om te bepalen of een graaf Hamiltoniaans is.

Een functie om het handelsreizigerprobleem op te lossen voor een gegeven graaf, een functie om een benaderde oplossing te vinden.

Enumeratorfuncties: deze functies geven een object van het type **Enumerator** terug. Enumerator objecten bevatten een methode **nextElement()** die bij herhaald aanroepen telkens een element uit de gevraagde opsomming oplevert. Er moeten functies zijn om een graaf depth-first en breadth-first te doorzoeken vanaf een gegeven top. Hiervan gebruik makend: functies om opspannende bomen te genereren gegeven een graaf en een enumerator ervan.

Een functie om een minimale opspannende boom te vinden van een graaf. De **waarom** functie kan grafisch door een animatie tonen hoe het algoritme (naar keuze Kuskal of Prim) gevolgd wordt.

Een functie die het kortste gewogen pad tussen twee toppen bepaalt.

Functies om te testen of een graaf een (maximale/volledige/maximum) koppeling is van een andere graaf, en om zulke koppelingen te vinden in een graaf.

pending: blz 166 ev

6.2 Implementatie

Als basis-datastructuur kiezen we een **incidentiematrix** van **boolean**s die bepalen of er een boog loopt van een top naar een andere top. Als de graaf ongericht is, is deze matrix symmetrisch en kunnen we in principe volstaan met het opslaan van de helft ervan; dit is voorlopig nog niet geïmplemeteerd (om de implementatie in eerste instantie simpel te houden). We zouden later ook maatregelen kunnen nemen om grafen met weinig bogen efficiënter op te slaan (bijvoorbeeld door alleen een lijst van bogen op te slaan in plaats van een volledige matrix).

De hiervoor opgesomde bewerkingen op grafen gebruiken een aantal datastructuren uit andere domeinen, zoals verzamelingen, matrices, groepen, ... die nu nog niet geï mplementeerd zijn. We zullen ons zoveel mogelijk proberen te behelpen met wat we hebben; zo kunnen we een functie die een verzameling zou moeten teruggeven voorlopig de elementen van die verzameling één voor één op de stapel laten zetten, gevolgd door een getal dat het aantal elementen aangeeft. Op dezelfde manier zullen we paden teruggeven (die eigenlijk n-tupels zijn).

We wensen ook **late evaluatie**: als de gebruiker bijvoorbeeld wil weten of een graaf als deelgraaf een 5-gon bevat, dan kan ze dat doen door de commando's **genereer_n_gon(5)** en **is_deelgraaf_van()** achtereenvolgens uit te voeren; het is wenselijk dat het genereren van een 5-gon niet direkt gebeurt. De routine **is_deelgraaf_van()** zou dan kunnen zien dat het argument een 5-gon is en een speciale routine geoptimaliseerd voor polygons kunnen uitvoeren. Dit vermijdt ook het genereren van grote datastructuren (vervang in het vorige voorbeeld 5 door 1000). Dit komt overeen met de manier waarop mensen het probleem aanpakken; niemand tekent eerst een 1000-gon om het probleem op te lossen.

pending: hiervoor ergens uitleg over het stapel mechanisme

*** pending: probleem: wat als user een object wijzigt: nieuw object maken, of: voor het wijzigen een nieuw object maken? Inspiratie: java.lang.String***

7 Uitbreidbaarheid en programma-objecten

We kunnen onmogelijk alle soorten wiskundige objecten voorzien die de gebruiker van het programma ooit nodig zal hebben (al was het maar omdat er altijd nieuwe bijkomen). Het moet dus mogelijk zijn de functionaliteit van het programma uit te breiden.

Een eerste mogelijkheid is via standaard Java klassen. We documenteren (in dit document bijvoorbeeld) de nodige klassen en methoden van het programma die nodig zijn om "in te kunnen pluggen".

Een tweede mogelijkheid is interactief in het programma zelf stukjes code aanmaken, via programma-objecten (die net als alle andere types op de stapel bewerkt kunnen worden). Zoals de meeste computer-algebra systemen zouden we een eigen programmeertaaltje kunnen uitvinden. Het handigst is misschien wel hiervoor een superset van Java te kiezen, en hieruit **.class** bestanden te genereren, zodat uiteindelijk zelf toegevoegde klassen op precies dezelfde manier behandeld worden als de standaard ingebouwde klassen. Een ander voordeel is dat we formele bewijs-infrastructuur op programma-objecten zullen laten inwerken, en we dus het programma correctheidsbewijzen over zichzelf zullen kunnen laten afleveren.

We zouden ook eens moeten uitzoeken in hoeverre het mogelijk/interessant is JavaBeans op de stapel te kunnen plaatsen en omgekeerd om van formules, grafen, ... beans te maken.

8 Toekomstplannen

- Ondersteuning voor 3D-beeldbewerking, door grote delen van ST-DIPP, mijn Simple Three-Dimensional Image Processing Program (`http://zeus.rug.ac.be/g̃eert/image/`) te recycleren.
- Ondersteuning voor de driewaardige logica voor partiële functies
- Semi-automatische bewijsgenerator
- Hoare-calculus op Java-byte of source code (we zullen zien wat het best gaat): het programma moet dus eigenschappen kunnen bewijzen van stukken Java code, en uiteindelijk dus ook over zichzelf.
- Uitwisseling met bestaande pakketten:
  - LaTeX
  - Computeralgebra pakketten: Maple
  - OpenMath standaard (`http://www.openmath.org/`)
  - meer misschien op `http://slashdot.org/askslashdot/00/02/27/1343255.shtml`

Index

Inhoudsopgave