Stageverhandeling:

SICECAS

Ontwikkeling van een struktuur-editor

Geert Vernaeve

UNIVERSITEIT GENT

15 januari 2001

1 Situering

Het onderwerp van dit document, SICECAS (simplistisch calculator- en computer-algebra-systeem), is een poging om een vrij algemeen tool te construeren om symbolische strukturen te manipuleren op een voor de gebruiker intuïtieve en visuele manier. Zo'n struktuur kan een wiskundige formule zijn, een formeel logisch bewijs, een HTML-document, ... Een dergelijk hulpmiddel kan onder meer ingezet worden ter ondersteuning van de oefeningenlessen bij een cursus logica (zie §6.2). We zullen in dit document proberen uiteen te zetten wat er al bereikt is en in welke richting we verder wensen te bouwen, want het zal blijken dat een zo algemeen systeem zich voor zovele toepassingen kan lenen dat één enkele ontwikkelaar er jarenlang aan bezig zou kunnen zijn. Dit werk dient dan ook meer gezien te worden als het leggen van een fundament voor toekomstige uitbreidingen dan het afleveren van een kant-en-klare toepassing—wat niet wegneemt dat er met de huidige partiële (en experimentele) implementatie al enkele interessante resultaten bereikt zijn.

Het programma zelf, handleidingen, broncode, documentatie en dergelijke zijn op `http://cage.rug.ac.be/~gvernaev/sicecas/` te raadplegen.

1.1 Naam

Tot eind 2000 is het programma door het leven gegaan onder de weinig inspiratievolle naam *GveCalc*. De naamswissel naar SICECAS (spreek uit ongeveer als "*ziekenkas*") zal gebeuren op 12 februari 2001, bij de aanvang van het tweede semester van dit academiejaar.

1.2 Doelpubliek

Dit document probeert

- voor geïnteresseerden uit te leggen wat SICECAS is en waarom het zo is
- voor eventuele medewerkers aan de verdere uitbreiding van het basissysteem (dit project is *Open Source*; zie §4) de interne werking van SICECAS uit te leggen, en schetst de richting waarin verdere ontwikkelingen zullen lopen
- voor zij die SICECAS wensen uit te breiden uit te leggen hoe de machinerie aangeboden door SICECAS werkt en hoe ze gebruikt moet worden.

Het is minder bedoeld als handleiding voor de gebruiker van SICECAS; daarvoor verwijzen we naar de webpagina. Ook zullen we enige wiskundige begrippen die al eens opduiken in aangehaalde voorbeelden niet nader verduidelijken: het zou ons te ver voeren een gedetailleerde uitleg te geven en over het algemeen is deze detailkennis ook niet echt nodig om de grote lijnen van het verhaal te kunnen begrijpen.

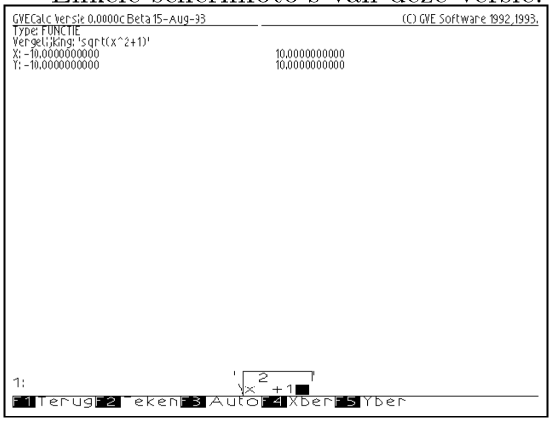
2 Historiek

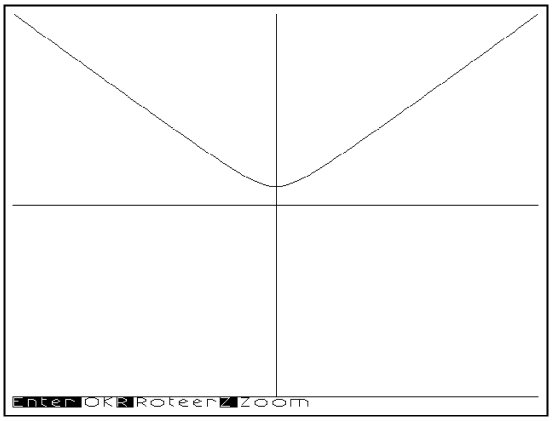
2.1 De Pascal-versie

Projecten als deze komen zelden zomaar uit de lucht vallen. De aanleiding gaat terug tot de tweede helft van 1992, toen de HP-48 rekenmachines hun intrede deden bij een medestudent—en in een mum van tijd de halve klas er zo eentje had. Zelf waren we natuurlijk ook zeer geïnteresseerd in deze toestellen, maar omdat de prijs nogal fiks was (de bijnaam *High Price* voor HP is niet volledig uit de lucht gegrepen) begon ik aan een programma, toen nog voorlopig *GveCalc* gedoopt, dat dezelfde dingen zou moeten kunnen (matrixrekenen, symbolische bewerkingen als afleiden en integreren, ...) in Pascal. Midden 1993 was dat programma uitgegroeid tot een 5000-tal regels programmacode verspreid over een tiental units. Symbolische bewerkingen zijn er nooit van gekomen, maar de kenmerkende eigenschappen van alle opvolgers waren toen al aanwezig:

- Formules worden in "klassieke" notatie getoond tijdens het intikken zelf. Er verschijnt dus nooit iets in "computer-notatie" zoals `x^2/4` maar $\frac{x^2}{4}$.
- Eenvoudige numerieke bewerkingen, zodat het programma als rudimentair zakrekenmachientje zou kunnen gebruikt worden.
- Functieplotter.
- Het programma presenteert zich als een *stapel* aan de gebruiker, waar hij formules kan op plaatsen en bewerkingen mee doen.

Enkele schermfoto's van deze versie:

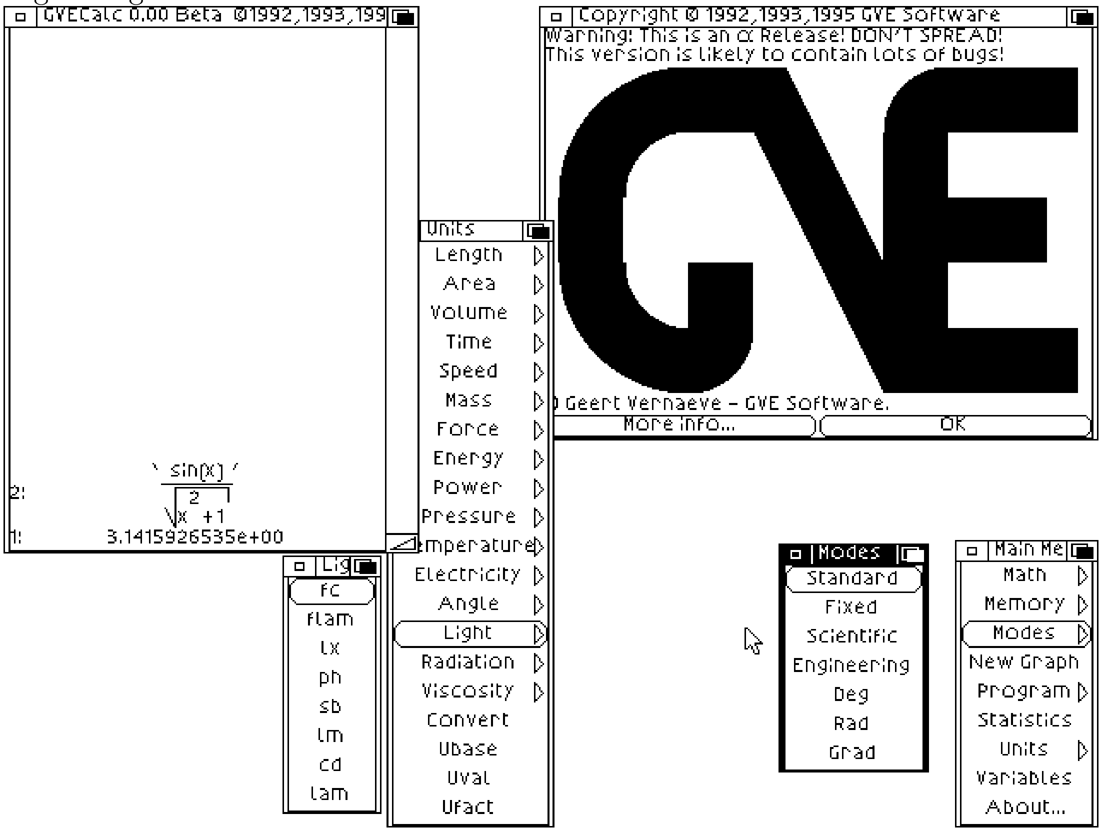




Vóór de andere doelstellingen, zoals het felbegeerde symbolisch integreren en afleiden, gehaald konden worden, begonnen de beperkingen van de toenmalige Pascal-compilers (de broncode compileerde op msdos- en Amiga-systemen) parten te spelen. De overstap naar C was dan ook snel gemaakt.

2.2 De C-versie

Eind 1993 begon het werk aan de C-versie. Ze draaide onder Amiga en msdos. In tegenstelling tot de Pascal-versie, werkte de C-versie met een vensteromgeving (onder msdos moesten we dus onze eigen window manager programmeren!) en zag er ongeveer zo uit:



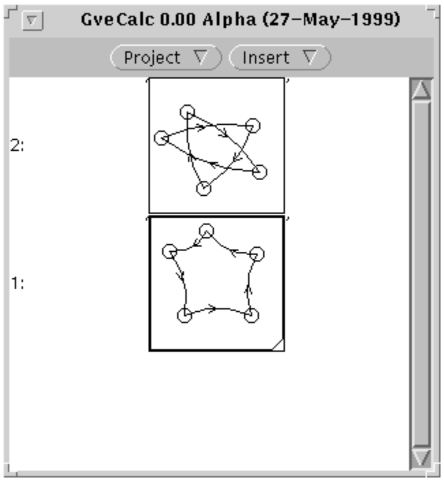
De C-versie had uiteindelijk ongeveer dezelfde functionaliteit als de Pascal-versie.

2.3 De Java-versie

Midden 1995 kwam de taal Java op de wereld, en na enkele maanden hadden we al in de gaten dat dit platform heel wat voordelen biedt. Een onmiddellijke motivatie om over te stappen op Java is de grote platform-onafhankelijkheid. Met de C-versie hadden we al ondervonden dat het niet eenvoudig is om programma's die met een grafische muis- en vensteromgeving werken overdraagbaar te houden tussen zelfs maar twee platformen. In Java gaat het openen van een venster onder windows, Unix of Macintosh op precies dezelfde manier.

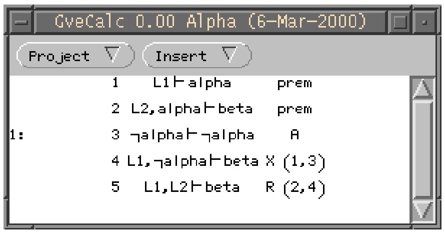
Bovendien gaf Java ons de kans te leren werken met object-georiënteerd programmeren. De Pascal- en C-versies behandelden de formule die de gebruiker aan het intikken is steeds als een tekststring, hetgeen nogal onhandige constructies tot gevolg had—zo is het bijvoorbeeld niet evident de juiste positie in de tekststring te vinden als de gebruiker midden in een matrix één rij naar boven wil bewegen. De Java-versie maakt volop gebruik van het object-georiënteerde paradigma en stelt de formule, ook tijdens het intikken al, voor als een zogenaamde *syntaxboom*, waarover verder meer.

2.4 Project Discrete Wiskunde



In 1998–1999 volgde ik het *eenjarig brugprogramma kandidaturen informatica voor houders van een diploma van kandidaat wiskunde of natuurkunde* waarin het vak *Discrete wiskunde* van prof. De Clerck voorkwam. Aangezien houders van voornoemd diploma verondersteld worden reeds genoeg discrete wiskunde meester te zijn, werd hen de mogelijkheid geboden een programmeerproject als examen af te leveren. Het idee om het bestaande rekenmachine-programma uit te breiden met bewerkingen op grafen lag voor de hand. We kwamen tot een erg visuele implementatie van het graaf-begrip, waarin ook voor het eerst het *why*-commando opdook. De gebruiker kan een bewerking uitvoeren (bijvoorbeeld: laten bepalen of twee grafen isomorf zijn, waarop het programma simpelweg met *true* of *false* antwoordt), en achteraf meer uitleg vragen door het *why*-commando in te geven (waarop het programma bijvoorbeeld de melding *beide grafen hebben een verschillend aantal toppen* geeft, of, als ze wel isomorf waren, een animatie toont waarin de ene graaf in de andere overvloeit).

2.5 Formele logica



Tijdens datzelfde jaar kwamen we tijdens de lessen *Formele logica* van prof. Hoogewijs in kontakt met *Flobes*, een programma dat in de lessen gebruikt werd om gemaakte bewijzen te controleren. Flobes is geschreven in een Prolog-versie die onder msdos draait in een zuivere tekst-omgeving, wat soms nogal onhandig noteert (symbolen als $\exists$ of $\forall$ zijn absoluut niet beschikbaar in deze omgeving). Het zal de aandachtige lezer wel niet verbazen dat we nogmaals het idee kregen het bestaande programma uit te breiden, ditmaal met ondersteuning voor formele logica.

Op 8 maart 2000 ging de eerste alpha-versie in première voor een halve tweede kandidatuur informatica. Dit was de eerste keer dat het programma op min of meer grote schaal verspreid werd (vorige versies circuleerden enkel onder enkele geïnteresseerde intimi), wat niet zonder gevolgen bleef: de volgende dag was er voor de andere helft van de tweede kandidatuur al een bijgewerkte versie, met een *undo*-functie, ondersteuning voor *knippen en plakken* van stukken formule en nog enkele soortgelijke zaken die men als programmeur nog al eens over het hoofd durft te zien.

We kunnen dan ook stellen dat de interactie met de gebruikers erg motiverend werkt, gelukkig ook in de andere richting: we stelden vast dat de studenten zelf nieuwe bewijsregels begonnen toe te voegen aan het systeem, iets wat met Flobes ook wel mogelijk was maar toen uiterst zelden bleek te gebeuren.

Het programma is intussen ongeveer 11500 regels Java code (350 KB) groot geworden.

3 Waarom nóg een computeralgebra-programma?

Heden ten dage zijn er computeralgebra-pakketten bij de vleet (op `http://www.cs.kun.nl/~freek/digimath/` is een lijst te vinden van meer dan 200 dergelijke pakketten!). De vraag waarom we niet een bestaand pakket proberen uit te breiden naar onze wensen is dus ten zeerste gerechtvaardigd. Hiervoor zijn een aantal uiteenlopende redenen:

- Uit *academische interesse*: de beste manier om te leren hoe een computeralgebra-systeem in elkaar zit, is er zelf een te ontwerpen. We hebben overigens een lange persoonlijke traditie in dat verband (gaande van een tekstverwerker, een multipass assembler geschreven in BASIC, een (nooit afgeraakte) C compiler in C, een teksteditor/Java IDE, een eigen implementatie van de Java Virtuele Machine in C, enz). Het spreekt vanzelf dat zo'n experimenteel systeem op termijn ook thesisstudenten kan aantrekken.

- *Overdraagbaarheid*: doordat we Java gebruiken, kan het programma in principe zonder enige verandering draaien op bijna alle populaire computerplatforms. Bij de meeste andere talen is één en ander (werken in een grafische muis- en venster-omgeving bijvoorbeeld) niet zo gestandaardiseerd en moeten we voor elk platform aparte modules maken, hetgeen voor aanzienlijke complicaties zorgt.

  Bij de verdere ontwikkeling zullen we proberen zo dicht mogelijk bij de programmeertaal Java te blijven, zodat we niet het zoveelste nieuwe programmeertaaltje voor het zoveelste nieuwe computeralgebrapakket moeten uitvinden. Dat dit overigens geen alleenstaande evolutie is, moge blijken uit de in de loop van 2000 geïntroduceerde *J/Link* voorziening binnen het computeralgebra-pakket *Mathematica* die ervoor zorgt dat Java en Mathematica van elkaars functies gebruik kunnen maken.

- *Kostprijs*: het is voor vele onderwijsinstellingen en studenten niet altijd mogelijk budget vrij te maken voor de aanschaf van (voldoende licenties van) een commercieel pakket, waarvan men dan maar moet ondervinden of het voldoet na aankoop.

- Beschikbaarheid van de broncode (*Open Source*, zie ook §4): van de meeste commerciële pakketten is het niet mogelijk deze broncode te bekijken. De broncode vormt nochtans een belangrijk hulpmiddel bij het begrijpen van de opbouw van de werking van een computeralgebra-systeem (hier komen we dus deels terug op de reeds eerder aangehaalde academische interesse in de technische werking van een computeralgebra-systeem). Ook kunnen we het programma naar onze eigen wensen aanpassen als we de broncode hebben—men denke bijvoorbeeld aan het verbeteren van een functieplotter-routine in een bestaand pakket.

- Nadruk op de *gebruikersvriendelijkheid*: de meeste computeralgebra-pakketten hechten hier wat minder belang aan. Functionaliteit is natuurlijk ook heel belangrijk, maar we willen de instap-drempel voor nieuwe gebruikers toch een stuk lager houden dan de klassieke omgevingen. Het is onze bescheiden mening dat een computer nog altijd in de eerste plaats uitgevonden is om ons het werk aangenamer te maken, en niet omgekeerd. Dit vertaalt zich in het niet gebruiken van de "computer-notatie"[1], een betere functieplotter dan de meeste andere pakketten, een *why*-commando, …

- *Omdat het zo zoetjesaan tijd is*:[2] In *het wis- en natuurkundig tijdschrift Simon Stevin* uit 1951/52 (jg. 29 aflevering IV) lezen we in een artikel van J.G. van der Corput getiteld *Moderne rekenmachines*:

  > *Niet alleen cijferen zullen de machines, maar ze zullen in hun geheugen een enorm groot aantal stellingen opbergen; ze zullen in speciale gevallen kunnen verifiëren of de voorwaarden van zulk een stelling vervuld zijn of niet en, indien dat het geval is, het door de stelling opgeleverde resultaat noteren, enz. Ze zullen wellicht een eind maken aan de chaos, waarin de huidige mathematicus leeft, die niet alle stellingen kan vinden, geldig in het gebied waarop hij werkt, want ze zullen hem die allemaal onmiddellijk leveren.*

[1] de gebruiker krijgt dus nooit te maken met voorstellingen als `x^2/4` of `\forall x: \alpha` maar ziet onmiddellijk $\forall x : \alpha$ en $\frac{x^2}{4}$

[2] Interessante lectuur in dit verband is het *QED Manifesto*, een project om een systeem op te bouwen dat alle gekende wiskunde kan omvatten met alle wetenschappelijke, technologische en educatieve toepassingen van dien. Zie `http://www.cs.kun.nl/~freek/qed/qed.html` en "The QED Manifesto" in "Automated Deduction—CADE 12", Springer-Verlag, Lecture Notes in Artificial Intelligence, Vol. 814, pp. 238–251, 1994.

> *(…) [De machine] zal de haar meegedeelde stellingen combineren en verifiëren en nagaan of in bepaalde gevallen de voorwaarden van een theorema al dan niet vervuld zijn, hetgeen een aanmerkelijke verlichting van de taak van de denker kan betekenen.*

Zij die onder dit publiek al eens een wiskundig artikel plegen te schrijven en hiervoor een programma gebruiken dat hen op de zoëven geciteerde wijze helpt, steken nu hun hand op. Ja, we hebben wel mooie *equation editor*s, of LaTeX, maar die zorgen enkel voor het layout-gedeelte, en helpen doorgaans niet met bijvoorbeeld een bepaalde rekenregel toepassen op een formule. En ja, computeralgebra-systemen kunnen wel hun resultaten bewaren in formaten als LaTeX, maar het blijft achteraf een knip- en plakboel om alle bekomen formules in de tekst in te plakken. (Zie §8.2.11.)

Vanzelfsprekend maken we ons, al deze redenen ten spijt, niet de illusie dat we op ons eentje hét volgende populaire "all-singing-all-dancing" computeralgebra-pakket zullen in elkaar knutselen.

4 Open Source, Copyright en dies meer

I decry the current tendency to seek patents on algorithms.
There are better ways to earn a living
than to prevent other people from making use
of one's contributions to computer science.
DONALD E. KNUTH—The Art of Computer Programming, vol. 3

In de situering vermeldden we al dat het één enkele ontwikkelaar jaren zou kosten om alle in dit document opgesomde uitbreidingen en toepassingen te implementeren. Het spreekt dan ook voor zich dat, eens we een basissysteem hebben geconstrueerd dat voldoende stabiel en flexibel is, de tijd rijp is om andere geïnteresseerde programmeurs mee te laten ontwikkelen. Ons inziens is de beste methode hiervoor het publiekelijk beschikbaar stellen van de broncode. We hopen zo veel meer geïnteresseerden te kunnen bereiken dan wanneer we het programma niet open source hadden gemaakt.

Dat is dan ook de reden waarom we het programma onder de *GNU Public License* (GPL) licenseren. De kerngedachte van deze licentie is dat het iedereen kosteloos toegestaan is de broncode te bekijken en aan te passen, maar dat aanpassingen ook weer onder de GPL moeten gelicenseerd worden.

5 Een struktuur-editor?

We omschreven SICECAS op de titelpagina al als een *struktuur-editor* en wensen de lezer hieromtrent niet langer in spanning te houden. Om het begrip struktuur-editor uit te leggen, zullen we eerst een kijkje nemen naar de manier waarop het traditionele computeralgebra-pakket werkt:

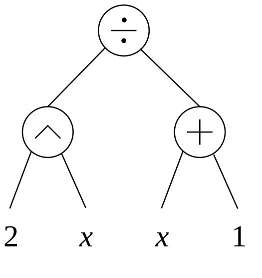
- De gebruiker tikt een formule in of de naam van een commando dat op eerder ingetikte formules bewerkingen uitvoert. In dit stadium wordt de ingetikte invoer als een rijtje letters behandeld. Als we geen geluk hebben, zijn we ergens een haakje of iets dergelijks vergeten, en krijgen we een min of meer duidelijke omschrijving van de aard en de plaats van de fout.

- De ingetikte rij letters wordt geïnterpreteerd als formule. Het resultaat wordt op het scherm getoond. Als we geluk hebben, krijgen we geen computernotatie maar een mooi geformatteerd resultaat.

SICECAS werkt op een minder traditionele manier: terwijl de gebruiker een formule aan het intikken is, wordt ze al syntactisch geanalyseerd. Daar waar in de klassieke systemen de gebruiker tijdens het intikken van een formule een rijtje letters manipuleert, is de gebruiker bij SICECAS bezig met het manipuleren van een syntax-boom, wat ons inziens een meer natuurlijke voorstelling is van een formule (zo is het onmogelijk om haakjes vergeten te sluiten). Voor de gebruiker is het voordeel van deze manier van werken ook onmiddellijk zichtbaar: van zodra een deel van een formule ingetikt is, wordt ze netjes op het scherm getoond. Wie bijvoorbeeld `1/2+x` aan het intikken is, krijgt na de eerste drie toetsaanslagen al $\frac{1}{2}$ op het scherm te zien.

5.1 Syntaxbomen

We schetsen kort het idee van de werking aan de hand van een voorbeeldje: onderstel dat de gebruiker de formule $\frac{2^x}{x+1}$ intikt. Dan wordt deze formule intern als de volgende boomstruktuur voorgesteld:

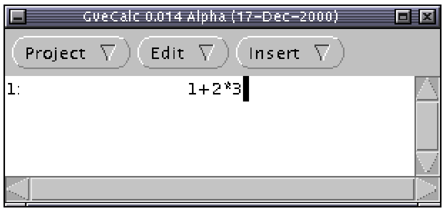


Deze boomstructuur wordt de *syntaxboom* van de formule genoemd. De boom bestaat uit een aantal *knopen* (de '÷', '∧', '+', '2', 'x' en '1' in de figuur). Sommige ervan hebben *kinderen*; zo heeft de '÷' knoop de twee knopen '∧' en '+' als kinderen. Het tegengestelde begrip is *ouder*: de ouder van '∧' is '÷', net als die van '+'. Knopen die geen kinderen hebben, noemen we de *bladen* van de boom. De boom heeft precies één *top*, een knoop zonder ouder, waarvan alle andere knopen afstammen (in ons voorbeeld de '÷' knoop).

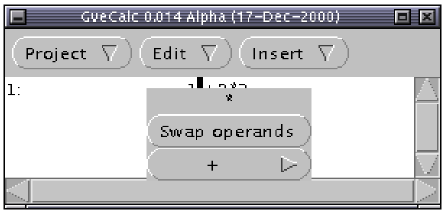
Zo'n syntaxboom is een natuurlijker voorstelling van de formule dan de reeks letters `2^x / x+1` (strikt genomen zouden we ergens wat haakjes moeten bijschrijven, die we voor het gemak even wegmoffelen): de syntaxboom weerspiegelt duidelijk de struktuur van de formule, terwijl in een reeks letters weinig meer struktuur zit dan 'letter A staat links van letter B'. Het zal de lezer wellicht niet verbazen dat deze aanpak ook enige complicaties met zich meebrengt: tijdens het intikken moet de syntaxboom correct opgebouwd worden; de gebruiker moet met de cursortoetsen door de syntaxboom kunnen wandelen (en één positie naar links of rechts bewegen kan soms overeenkomen met een verre sprong in de syntaxboom). Deze problematiek wordt uitgebreider behandeld in bijlage A.

5.2 De struktuur-editor in de praktijk

De gebruiker kan snel de struktuur van de ingetikte formule achterhalen. We zullen dit demonstreren met een eenvoudig voorbeeld: we tikken `1+2*3` in. De formule verschijnt dan op het scherm:

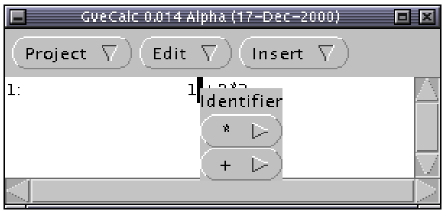


We zouden ons nu kunnen afvragen of deze formule nu $1 + (2 * 3)$ of $(1 + 2) * 3$ betekent. We kunnen dat te weten komen door met de muis boven de $*$ te gaan staan en op de rechter muisknop te klikken:



Het menu dat nu verschijnt, toont bovenaan de bewerkingen die we kunnen uitvoeren op de $*$. In ons geval is dat alleen *swap operands* (waardoor de formule zou veranderen in $1 + 3 * 2$). Daaronder zien we een menu-knop '+' genaamd, die aangeeft dat de $*$ een kind is van $+$. In ons geval is dus $2 * 3$ een kind van $+$ (het andere kind is 1). We kennen dus nu de struktuur van deze formule: er wordt $1 + (2 * 3)$ mee bedoeld.

Soms is de struktuur dieper; als we op de 3 met de rechter-muisknop klikken krijgen we:



Op de identifier 3 zelf kunnen we geen operaties uitvoeren, dus staan er alleen maar menu-knoppen die aangeven hoe de 3 struktureel in de formule ingebed is: het is een kind van een $*$-operator, die op zijn beurt weer een kind is van een $+$-operator.

Op deze manier kan de gebruiker snel een inzicht krijgen in de opbouw van een formule.

6 Gerealiseerde toepassingen

6.1 Functieplotter

Net als de Pascal- en de C-versies, heeft SICECAS een functieplotter aan boord, met dit verschil dat de huidige plotter geavanceerder is dan de vorige, en zelfs dan veel andere commerciële pakketten. In het kort komt de verbetering erop neer dat we ons realiseren dat de klassieke manier van functies plotten—bereken een aantal punten van de grafiek en verbind ze met lijnen—wel werkt voor 'brave' functies (die niet al te hard schommelen), maar in het algemeen een vrij pover resultaat geeft.

We kunnen met de klassieke aanpak zelfs niet weten wat de grafiek juist voorstelt! We zullen deze uitspraak iets preciseren. Onderstel dat het medium waarop de plot gemaakt wordt uit een aantal beeldelementen (pixels) bestaat, die in het begin allemaal wit zijn, en waarop in het zwart volgens de klassieke methode een functie geplot wordt. Een ideale functieplotter maakt enkel de pixels zwart waar de functie wel door passeert, en laat alle andere pixels wit. Nochtans kan het bij de klassieke functieplotter gebeuren dat

- een pixel zwart is waar de functie niet passeert.
- een pixel wit is waar de functie wel passeert.

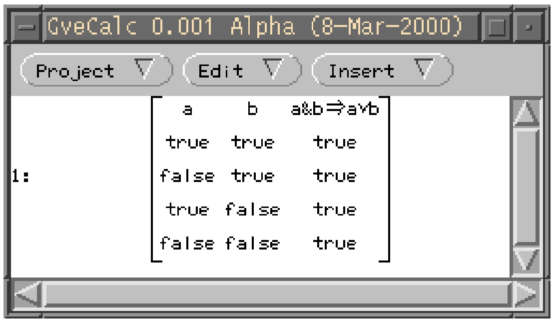
Dus: zo'n grafiek is eigenlijk waardeloos, omdat we van geen enkele geproduceerde pixel zeker kunnen weten wat er gebeurt met de functie.

In bijlage B leggen we uit op welke manier we aan dit euvel verhelpen, en geven expliciete voorbeelden van de problemen met de klassieke methode.

6.2 Logica voor de tweede kandidatuur informatica en de aanvullende studies informatica

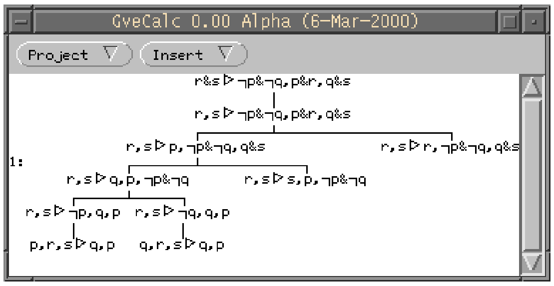
We hebben enkele functies ingebouwd naar aanleiding van het gebruik tijdens de lessen formele logica.

6.2.1 Waarheidstabellen



De eerste toepassing is het genereren van waarheidstabellen (op de figuur de waarheidstabel van de formule $a\&b \Rightarrow a \vee b$). We maken handig gebruik van de reeds geïmplementeerde ondersteuning voor matrices—een waarheidstabel is gewoon een matrix.

6.2.2 Semantische tableaus



In de cursus formele logica wordt een methode behandeld om na te gaan of een uitspraak al dan niet waar is aan de hand van een tableau-methode. Ons basissysteem bleek voldoende flexibel om zulke tableau's te tonen en door de gebruiker te laten bewerken. Het programma laat de gebruiker toe stap voor stap het tableau te construeren. In de huidige versie is het niet mogelijk zelf eigen regels voor de tableau-constructie toe te voegen, en het is ook nog niet mogelijk een ingegeven tableau te controleren.

6.2.3 Formele bewijzen

De grootste nadruk ligt evenwel op de ondersteuning voor het automatisch controleren van formele bewijzen. De studenten kunnen hun bewijzen ingeven in een notatie die weinig verschilt van die op papier, hetgeen met de voorloper *Flobes* duidelijk niet het geval was: iets als

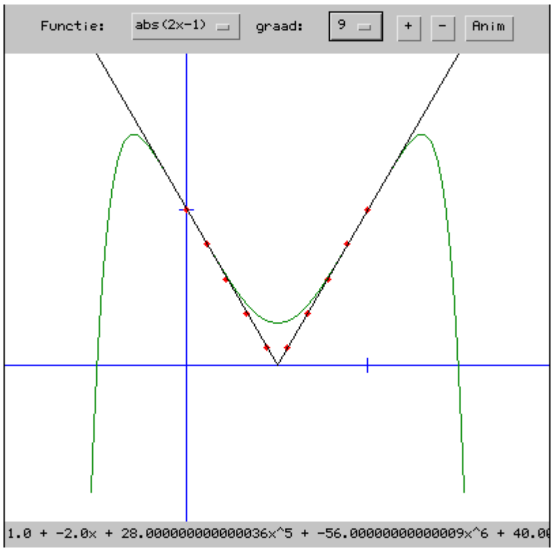
```
s(1,
[alle(v(x),pred(p,[v(x),c(a)])),
        alle(v(x),i(pred(p,[v(x),c(a)]),pred(q,[v(x)])))],
alle(v(x),pred(p,[v(x),c(a)])),regel("A")).
s(2,
[alle(v(x),pred(p,[v(x),c(a)])),
        alle(v(x),i(pred(p,[v(x),c(a)]),pred(q,[v(x)])))],
alle(v(x),i(pred(p,[v(x),c(a)]),pred(q,[v(x)]))),regel("A")).
```

(een excerpt uit mijn eigen examenoplossing uit de tijd van toen) is duidelijk niet wat men intuïtief geneigd is op een blad papier neer te schrijven.

De ondersteuning voor propositierekening is zo goed als volledig; predikaatrekening (met de $\forall$ en $\exists$ kwantoren) was nog niet operationeel ten tijde van de lessen van 2000 maar intussen wel.

Net als bij de waarheidstabellen hebben we op het allerlaagste niveau kunnen gebruik maken van de reeds ingebouwde ondersteuning voor matrices: een bewijsobject is gewoon een bijzonder soort matrix (zonder haken errond, met steeds drie kolommen, enzovoort).

6.3 Bernstein-veeltermen applet voor de eerste kandidatuur wiskunde en natuurkunde



Aan het begin van het akademiejaar 2000–2001 kregen we de vraag van prof. Impens of we voor een applet konden zorgen dat benaderingen van functies door middel van Bernstein-veeltermen toont. We hebben hier uiteraard gebruik gemaakt van de functieplotter-module van SICECAS, waardoor het eigenlijke programmeerwerk van de applet een kwestie was van slechts enkele uren.

7 Management van het project

Zoals al eerder opgemerkt, is dit een project van een potentiële omvang die de capaciteit van één enkele persoon ruimschoots overschrijdt. Men zou zich dan ook terecht de vraag kunnen stellen waarom het al die jaren een strikte éénman-show is gebleven.

We hebben er doelbewust voor gekozen het project binnenshuis te houden totdat de kern min of meer stabiel is. We zijn er immers van overtuigd dat dergelijke systemen het best door één of twee personen ontworpen worden. De beste voorbeelden hiervan zijn misschien wel het *Unix* besturingssysteem en de programmeertaal *C*, die door twee man ineen zijn gezet. Eén van de meest aangehaalde redenen om dit te verklaren is dat werken met een heel klein team de ontwerpers verplicht het systeem in zijn geheel tot in de details te begrijpen, daar waar bij grote teams het probleem opduikt dat het systeem te veel wensen moet vervullen van te veel ontwikkelaars. Het basissysteem wordt daardoor misschien iets minder uitgebreid (wat op zich geen nadeel hoeft te zijn) maar wel duidelijk consistenter. En natuurlijk is het ook niet makkelijk om iemand warm te krijgen voor een project dat nog in de kinderschoenen staat …

Eens het basissysteem klaar, wensen we uiteraard de boeg volledig om te gooien en zoveel mogelijk externe hulp te gebruiken bij het verder ontwikkelen (zie §8.2).

7.1 Doelplatform

We gebruiken voor het ogenblik enkel functionaliteit die in Java 1.1 beschikbaar is, hoewel Java intussen al gevorderd is tot versie 1.3. De belangrijkste reden hiervoor is dat de meeste hedendaagse browsers enkel Java 1.1 ondersteunen, en we wensen het programma ook bruikbaar te houden als applet binnen een browser. Om dezelfde reden gebruiken we nog de AWT (Abstract Window Toolkit) als grafische interface.

We voorzien dat we in de toekomst op Java 1.3 en *Swing* (de opvolger van de AWT) zullen overschakelen. We dienen er wel over te waken dat alles nog overdraagbaar is naar platforms met relatief weinig geheugen en processorkracht (zie §8.3); vooral Swing zou hier op beide gebieden voor problemen kunnen zorgen. Het zou dus interessant zijn ervoor te zorgen dat, indien mogelijk, SICECAS zowel onder Swing als onder de AWT gedraaid kan worden.

7.2 Versies

Tijdens het ontwikkelen van het pakket moet de interne opbouw soms grondig gewijzigd worden. Zo hebben we rond begin juni 2000 het programma omgebouwd om het Model-View-Controller paradigma (zie §A.4) te ondersteunen, een karweitje dat drie weken heeft geduurd. Tijdens zulke ingrijpende veranderingen compileert het programma vaker niet dan wel.

We werken daarom steeds met twee broncode-versies: één *stabiele*, waar enkel opgemerkte bugs in worden verbeterd, en één *ontwikkelings*-versie, waar het programma significante wijzigingen kan ondergaan. Wanneer we snel een fout in het programma moeten verbeteren of een kleine uitbreiding moeten aanbrengen, kunnen we de stabiele versie aanpassen en zo snel inspelen op de vragen en noden van de gebruikers. Eens de ontwikkelings-versie afgewerkt is, wordt die versie de nieuwe stabiele versie. Dient er later zich weer een nieuwe ingrijpende verandering aan, dan beginnen we een nieuwe ontwikkelings-versie.

Deze aanpak voldoet min of meer voor de huidige situatie waarbij het programma onderhouden wordt door één enkele ontwikkelaar. In de toekomst zullen we moeten overschakelen op een wat geavanceerder versie-systeem; we denken in de richting van een CVS-server.

Een aanverwante problematiek is het vermijden dat eerder gemaakte programmeerfouten opnieuw opduiken in het programma. Hiervoor maken we gebruik van regressietesten (zie bijlage C).

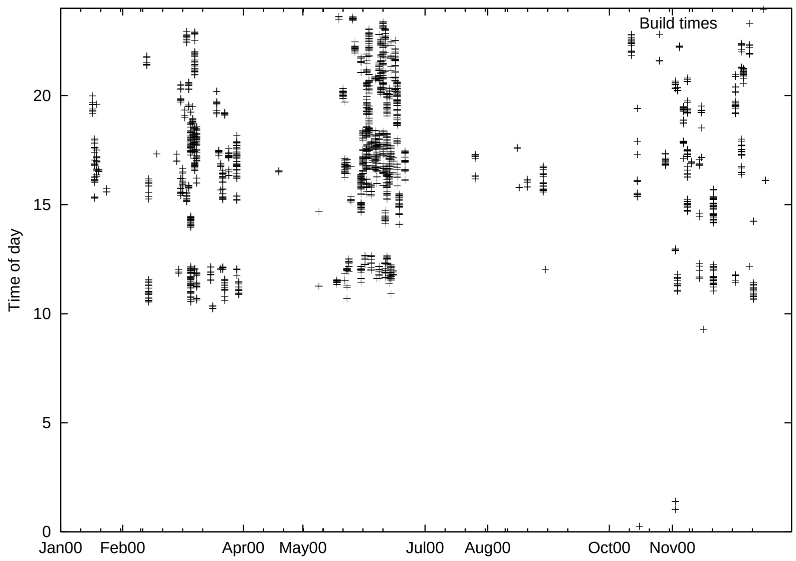
7.3 Tijdsmanagement van onszelf

We hebben bij wijze van experiment onze ontwikkelomgeving zo geïnstrumenteerd dat in een logboekbestand statistieken worden bijgehouden telkens als we een nieuwe compilatie van SICECAS doen. Vanuit tijdsmanagement-oogpunt zijn er twee mogelijke bedreigingen voor het goede verloop van het project:

- Andere taken slorpen te veel tijd op en het project verwatert.
- Het project slorpt te veel tijd op waardoor onze andere taken verwaarloosd worden.

Als we onze eigen functieclassificatie mogen geloven, stoppen we 15% van onze tijd in de taak *Ondersteuning van het onderwijs*.

Uit de verzamelde gegevens van het logboekbestand blijkt dat we spontaan tot de volgende methode kwamen om beide tendensen te verzoenen: het ontwikkelen gebeurt in 'vlagen' waarbij in relatief korte periodes (typisch één tot hooguit twee weken) veel geconcentreerde activiteit merkbaar is, afgewisseld door zeer kalme tussenruimtes (typisch enkele weken tot een paar maanden) waarin onze andere taken weer meer van onze aandacht opeisen.



Figuur: voor elke compilatie is een kruisje geplaatst. De *Time of day*-as geeft het uur van de dag aan (we kunnen bijvoorbeeld afleiden dat middagpauzes ongeveer tussen 12 en 2 vallen).

8 Toekomstplannen

8.1 Basissysteem

In deze paragraaf zullen we enkele gewenste uitbreidingen aan het basissysteem opsommen. In tegenstelling tot de volgende paragraaf, *Toepassingen*, beschouwen we de hier opgesomde uitbreidingen als behorend tot het fundament van het programma. Het is dus de bedoeling dat deze uitbreidingen geïmplementeerd worden vóór het programma door een wat groter programmeerteam ontwikkeld wordt.

8.1.1 Voorstelling loskoppelen van inhoud

In de huidige implementatie wordt een type object (bijvoorbeeld een getal, een graaf, een +-operator) altijd op dezelfde manier getoond op het scherm. Bedoeling is dat we de gebruiker toelaten zijn eigen 'stijl' te kiezen, een beetje zoals we tegenwoordig *Cascading Style Sheets* kennen waarmee de gebruiker kan bepalen in welke stijl een webpagina getoond wordt. Voorbeelden:

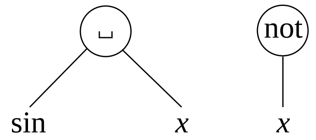
- Een bepaalde operator kan in functie-notatie ($plus(1, 2)$), in infix-notatie ($1 + 2$), in prefix-notatie ($+\ 1\ 2$), in postfix-notatie ($1\ 2\ +$), ... getoond worden. We moeten wel uitzoeken of het doenbaar is om bijvoorbeeld een +-operator in postfix-notatie tegelijkertijd toe te laten met een $*$-operator in prefix-notatie.

- We kunnen de namen van functies loskoppelen van de functie zelf; zo kunnen we *internationalisatie* (zie §8.2.1) van het programma bereiken door bijvoorbeeld de *isregular* functie op verschillende manieren te tonen naargelang te taal van de gebruiker. Een Nederlandstalige gebruiker zou die functie dan te zien krijgen als *isregulier*. Dit zal de leercurve van SICECAS aanzienlijk vlakker maken voor nieuwe gebruikers van computeralgebra-systemen.

8.1.2 Algemene parser

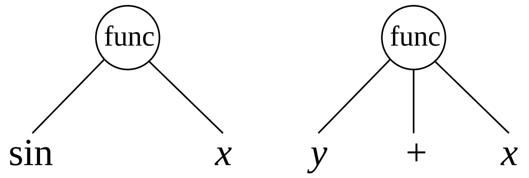
De *grammatica*, die bepaalt welke formules welgevormd zijn (zoals $1 + 3 * x$) en welke niet (zoals $1 + *x$). In de huidige versie van SICECAS zit de grammatica vast ingebakken; in de toekomst zal die door de gebruiker bepaald kunnen worden. Grammatica's van populaire programmeertalen en bestandsformaten zijn courant te vinden. Zulke grammatica's worden vaak gebruikt om automatisch een zogenaamde *parser* voor een programmeertaal te maken, die ervoor zorgt dat het programma in tekstvorm omgezet wordt naar een syntaxboom, en vaak de eerste stap is van een compiler voor die programmeertaal.

Ook behandelen we nu functies fundamenteel anders als operatoren. De uitdrukkingen '$\sin x$' (functie) en 'not x' (operator) krijgen de volgende syntaxbomen:



Bij '$\sin x$' maken we gebruik van de 'spatie-operator', die we ook al (zie bijlage A) gebruikten om van half ingetikte formules toch een syntaxboom te kunnen maken.

Mede in het kader van het loskoppelen van voorstelling en inhoud zullen we beide op dezelfde manier behandelen, waarschijnlijk ongeveer als volgt (we geven een voorbeeld van een unaire en een binaire operator):



We gebruiken hier een echte 'functie'-knoop in de syntaxboom om zowel operatoren als functies aan te geven—in feite maken we geen onderscheid meer tussen beide. We zijn zo ook af van het lelijke dubbelgebruik van de spatie-operator (als 'functie'-knoop en als aaneenplak-operator voor half ingetikte formules).

Het algemeen maken van de parser is waarschijnlijk de meest ingrijpende van de nog aan het basissysteem uit te voeren werkzaamheden.

8.1.3 Dichtere integratie met Java

De manier waarop het programma nu omgaat met objecten is nogal ad-hoc. Stel om de gedachten te vestigen dat de gebruiker bijvoorbeeld het object $a\&b \Rightarrow a \vee b$ intikt en er het commando *truthtab* op los laat om een waarheidstabel te genereren (het resultaat is te zien in §6.2.1). SICECAS bevat nu één centrale routine die beslist wat er moet gebeuren als de gebruiker zo'n commando intikt. In dit geval ziet die routine dus dat het woord *truthtab* is ingetikt en beslist de juiste handelingen te ondernemen (in dit geval: een waarheidstabel construeren). Het spreekt vanzelf dat deze manier van werken niet vriendelijk is voor zij die SICECAS willen uitbreiden met eigen types objecten en eigen commando's. In de toekomst zal SICECAS in staat moeten zijn externe types objecten (concreet: externe Java-klassen) te herkennen en gebruiken.

Hiermee samenhangend is het gebruik van *packages*. Als er nu twee gebruikers elk een verschillende bewijsregel aan het systeem toevoegen met dezelfde naam (zeg *MijnRegel*), dan komen ze in de problemen als ze bewijzen uitwisselen die van die regel gebruik maken. We willen ertoe komen dat (net als in Java overigens) alle namen van objecten binnen een bepaald *package* opgenomen zitten, waardoor ze van gelijknamige objecten in een ander package onderscheiden kunnen worden. In ons voorbeeld zouden er dan twee regels bestaan met de namen *jan.MijnRegel* en *rug.cage.computeralgebra.geert.MijnRegel*. Een elegante toepassing hiervan zou ook kunnen zijn dat iemand die in plaats van de in de lessen gebruikte logica, alle regels aanpast voor een andere logica (bijvoorbeeld een tweewaardige in plaats van een driewaardige logica). Veel regels uit de ene logica hebben een analogon in de andere, en we zouden dus naast de standaard regel *gve.calc.logic.propc.A* ook de regel *jan.driewaardig.A* kunnen hebben.

8.1.4 Bestandsformaten

Tot nog toe worden bewaarde objecten in een eigen ad-hoc formaat bewaard. Het is de bedoeling zo snel mogelijk over te stappen op een meer standaard formaat; *MathML* lijkt ons een zeer interessante kandidaat. Meer algemeen wensen we in het basissysteem een mechanisme in te bouwen dat de algemene definitie van import- en export-formaten toelaat. Formules bewaren in LaTeX-formaat zou ook erg handig zijn (zie ook §8.2.11).

8.2 Toepassingen

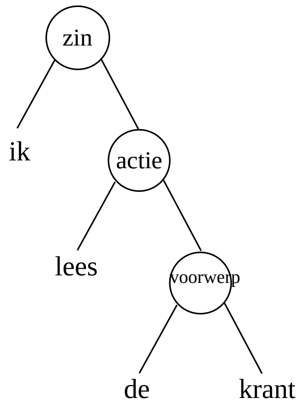
De hier voorgestelde onderwerpen bouwen verder op het basissysteem. Het is de bedoeling dat dat basissysteem relatief snel (we mikken op zomer 2001) voldoende flexibel uitgebouwd zal zijn om deze toepassingen te kunnen ondersteunen. In die zin is deze paragraaf op te vatten als een lijst van mogelijke uitbreidingen waar we bij de opbouw van het basissysteem op voorzien moeten zijn; sommige van de hier voorgestelde uitbreidingen zullen wellicht nooit of in een zeer verre toekomst pas daadwerkelijk geïmplementeerd worden, maar het gezegde luidt dan ook niet voor niets *plan big but start small* ... Alles zal een beetje afhangen van hoeveel en welke geïnteresseerden we voor ons project zullen kunnen warm maken—thesisstudenten kunnen een dankbare bron van arbeidskrachten vormen. Onze ervaringen met SICECAS tijdens de lessen formele logica in de tweede kandidatuur informatica leren overigens dat de studenten vrij enthousiast zijn, inzoverre dat er (tot onze grote maar aangename verrassing) al spontaan vragen zijn gerezen of er thesissen over te maken vallen. We hebben dan ook niet geaarzeld en een web-pagina gemaakt met mogelijke thesis- en project-voorstellen (zie `http://cage.rug.ac.be/~gvernaev/sicecas/thesis.html`).

8.2.1 Internationalisatie

Voor het ogenblik bestaan de meldingen die het programma aan de gebruiker geeft een ad-hoc mengeling van Nederlandse en Engelse tekst. We moeten hier eens orde in brengen en een algemeen mechanisme voorzien om de gebruiker zijn taal te laten kiezen. Java heeft hiervoor een aantal ingebouwde voorzieningen.

Anderzijds zijn er de wiskundige functies die het programma zelf biedt. Die kunnen we ook aanpassen aan de taal van de gebruiker door middel van de ontkoppeling van voorstelling en inhoud (zie §8.1.1).

Als extreem doorgetrokken internationalisatie zouden we misschien kunnen proberen een grammatica voor (eenvoudige) natuurlijke zinnen op te stellen, waarbij een zin als 'ik lees de krant' een syntaxboom krijgt in de trant van



Taalkundigen zullen wel betere namen hebben voor de operatoren *zin*, *actie*, ... Deze operatoren kunnen dan afhankelijk van de gekozen taal hun kinderen in de juiste woordvolgorde laten zien. Vanzelfsprekend heeft dit systeem zijn beperkingen: we moeten zelf de struktuur van de zin aangeven, we kunnen enkel standaard-zinsbouw gebruiken, ... maar misschien kan het ooit zijn nut bewijzen als we een helpsysteem willen opzetten waarbij de gebruiker uitleg krijgt over de werking van SICECAS zelf, over de gebruikte wiskundige operatoren, ... dat voor het grootste deel bestaat uit standaard-proza.

8.2.2 Afstappen van de stapel als standaard-interface

De huidige versie van het programma presenteert zich voor de gebruiker als een *stapel* waar hij objecten op kan gooien en vervolgens bewerkingen op uitvoeren. We hebben deze manier van werken overgenomen van de HP-48 rekenmachines en eens de gebruiker eraan went is het een handig systeem. De leercurve is echter nogal steil voor beginnende gebruikers: in vele toepassingen wordt toch maar met één object tegelijkertijd gewerkt en een stapel is dus een beetje teveel van het goede. Bij het gebruik in de lessen formele logica bijvoorbeeld is de gebruiker meestal bezig met het bewerken van één enkel bewijs-object (er zijn tenslotte weinig zinvolle operaties die op twee bewijzen inwerken).

We zullen het programma dus ombouwen zodat er maar één object tegelijkertijd bewerkt kan worden. We zullen het stapel-principe van haar bevoorrechte plaats halen en degraderen tot een gewoon object (net als bijvoorbeeld een matrix) zodat de gevorderde gebruiker toch met een stapel kan werken. Deze methode is ook veel eleganter: net zoals we nu bijvoorbeeld een matrix van formules kunnen maken (of een matrix van matrices van bewijzen, als het moet), zal de gebruiker dan ook stapels van stapels, stapels van matrices, ... kunnen bewerken.

8.2.3 SICECAS als vriendelijke interface

We zouden SICECAS achter de schermen kunnen laten communiceren met andere computeralgebra-pakketten. Als rampscenario (stel dat de ontwikkeling van SICECAS onverhoopt in de vergeethoek geraakt) is dat een redelijk veilige optie. Natuurlijk hopen we dat het zo'n vaart niet zal lopen, maar ook dan blijft communiceren met andere pakketten interessant, bijvoorbeeld als aanvulling op de functionaliteit die we zelf bieden (zeker in het begin heeft het minder zin functies van bestaande gespecialiseerde pakketten te kopiëren). In het formele bewijsvoerings-wereldje is dat overigens geen ongebruikelijke praktijk: er zijn toepassingen bekend[3] die formele bewijzen genereren voor bepaalde wiskundig gerichte vraagstukken en onderweg een pakket als Maple gebruiken als "orakel"— het is veel makkelijker te bewijzen dat bijvoorbeeld een door Maple bekomen integraal correct is dan zelf een formeel integratie-algoritme op te stellen.

[3] Op de vijfde Proof Tool Day die op 26 mei 2000 door onze vakgroep gehouden werd, demonstreerden O. Caprotti en M. Oostdijk een systeem dat het pakket GAP gebruikte als orakel binnen de Coq bewijsomgeving. Zie `http://cage.rug.ac.be/prooftool/oostdijk.html` en `http://crystal.win.tue.nl/~olga/openmath/pocklington/`.

8.2.4 Programma's als object

Net zoals we nu met een getal, een bewijs of een graaf dingen kunnen doen binnen SICECAS, zouden we, eens de algemene parser (zie §8.1.2) klaar is, ook *programma*'s als objecten kunnen gebruiken. We zouden dan SICECAS als intelligente editor kunnen gebruiken: aangezien SICECAS de syntaxboom opstelt tijdens het tikken, kan het onmiddellijk waarschuwen bij fouten (bijvoorbeeld het gebruiken van een ongedeclareerde variabele, iets dat we bij de klassieke systemen pas te weten komen bij het compileren van het programma). Om dezelfde reden kunnen we operaties doorvoeren die bij een klassieke, louter tekst-gebaseerde editor quasi onmogelijk zijn, zoals de naam van een variabele of functie overal binnen het programma-object vervangen door een andere naam. Dat zoiets niet altijd even triviaal is, moge blijken uit onderstaand voorbeeld:

```
class Foo {
    Bar x;

    Foo xyzzy() {
        return x.x;
    }
}

class Bar {
    Foo x;

    Bar xyzzy() {
        return x.x;
    }
}
```

Als we hier één van de vette **x**'en in iets anders veranderen, moeten alleen die vette **x**'en mee veranderen en de gewone x'en moeten blijven staan, en vice versa.

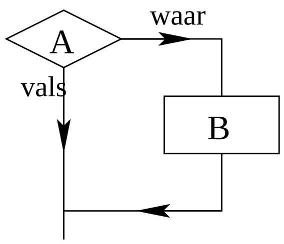
Een andere mogelijke toepassing zou kunnen zijn dat wanneer er bij het wijzigen van het programma-object een nieuwe variabele wordt toegevoegd, automatisch een variabelendeclaratie wordt gegenereerd. We denken hierbij bijvoorbeeld aan Fortran-programmeurs die het lastig vinden in een taal als C al hun integers expliciet te moeten declareren.

In combinatie met regressie-testen (zie bijlage C) zouden we SICECAS kunnen laten nagaan welke gedeelten van een programma door een regressie-test worden bereikt. Een mogelijke werkwijze is op strategische plaatsen in dat programma een registratie-functie aan te roepen die bijhoudt welke delen van dat programma gedraaid hebben. Hiertoe kunnen we ofwel aangepaste broncode genereren, die extra instrumentatie-regels bevat, ofwel (vooral in als het gaat om Java-programma's) een eigen compiler maken die direkt geïnstrumenteerde code maakt. Aangezien we toch het programma al in syntaxboom-vorm hebben, is een compiler schrijven dan niet zo'n grote moeite meer.

In combinatie met het loskoppelen van voorstelling en inhoud zouden we ook interessante dingen kunnen doen zoals de *if A then B* operator naar keuze op één van de volgende manieren voorstellen:

```
if A then B
```

```
if (A) {
    B
}
```



"Pascal−stijl" *"C−stijl"* *"Stroomdiagram−stijl"*

Deze mogelijkheid is interessant vanuit educatief oogpunt; ook zou iemand die gewoon is in Pascal te programmeren en die even een C-programma wil aanpassen maar geen zin of tijd heeft veel C te leren, het C-programma op een Pascaliaanse wijze kunnen bekijken en editeren.

Een soortgelijke toepassing is functies met obscure namen, zoals het Fortran-fragment Y = A02ABF(XR,XI) dat een functie uit de NAG-bibliotheek gebruikt, op een aangenamer manier als Y = ComplexModulus(XR,XI) aan de gebruiker tonen. Deze manier van voorstellen is waarschijnlijk intuïtiever voor gebruikers die niet vertrouwd zijn met de NAG-bibliotheek. Men zou kunnen opwerpen dat met een of ander preprocessor-programma we hetzelfde effect zouden kunnen bereiken: we geven de preprocessor de opdracht vlak vóórdat de Fortran-compiler ons programma te zien krijgt, alle voorkomens van ComplexModulus achter de schermen te vervangen door A02ABF. Het nadeel van deze techniek is dat we om bestaande Fortran-programma's leesbaarder te maken, zelf alle A02ABF's in de Fortran-bestanden moeten beginnen vervangen door ComplexModulus. Onze aanpak verandert de Fortran-code zoals ze op schijf bewaard wordt niet; we tonen ze enkel op een vriendelijker manier op het scherm.

Ook Java programma's kunnen we op deze manier natuurlijker weergeven. Het is welbekend dat de programmeertaal Java niet toestaat operatoren te overladen. Als we zelf een gegevenstype Complex maken met bewerkingen telop en vermenigv, dan moeten we in Java a.telop(b.vermenigv(c)) schrijven om de wiskundige formule $a + b * c$ uit te drukken. Als we echter de programma-object-voorstelling zelf kunnen configureren, zouden we dat stukje programma daadwerkelijk op het scherm kunnen zien verschijnen als a+b*c.

8.2.5 WWW-browser

We zagen vroeger al dat SICECAS de eigenschap heeft dat ingetikte tekst al geparsed wordt tijdens het intikken. We zouden de grammatica van HTML in SICECAS kunnen stoppen en zo met relatief weinig moeite een WWW-browser kunnen maken die pagina's toont en opmaakt van zodra de informatie binnenkomt, op precies dezelfde manier als we nu al formules netjes opgemaakt tonen van zodra de gebruiker ze begint in te tikken.

Van die browser zouden we dan weer gebruik kunnen maken om een interactief help-systeem op te zetten dat HTML gebruikt om de hulppagina's aan de gebruiker te tonen. Het zou interessant zijn om aan elke operator een help-tekst te verbinden. Om het voorbeeld over de Fortran-NAG-bibliotheek uit vorige paragraaf weer aan te halen zouden we dan ervoor kunnen zorgen dat wanneer de gebruiker met de rechtermuisknop op A02ABF(...) klikt, een *Help*-knop in het struktuur-menu verschijnt die uitlegt wat die A02ABF ook alweer deed.

8.2.6 Struktuur-zoekmachine

Met de opkomst van het WWW kwamen ook de web-zoekmachines op. Omdat de tekst op webpagina's vrij ongestruktureerd wordt bewaard, is de manier van zoeken dan ook volledig tekst-gebaseerd. Vooral bij wiskundige formules of programma's is het bijna een noodzaak om ook strukturele zoekvragen te kunnen stellen. Zo zullen we een document dat de formule $\sin(x + y)$ bevat even graag willen vinden als een document dat over de formule $\sin(a + b)$ gaat, maar willen liefst geen documenten die $\sin(x * y)$ behandelen. Bij programma's zouden we bijvoorbeeld graag willen weten welke subklassen van een klasse een bepaalde methode herdefiniëren of gebruiken, ..., allemaal zoekvragen die een klassieke tekst-gebaseerde zoekmachine niet of nauwelijks aankan. Het spreekt dan ook voor zich dat we een struktuur-georiënteerde zoekmachine aan SICECAS willen toevoegen.

8.2.7 Systeembeheer?

Sommige aspecten van systeembeheer komen eigenlijk neer op verkapte wiskundeproblemen. Een *makefile* is eigenlijk een afhankelijkheids-graaf. Een lijst van mail-aliassen, waarbij bijvoorbeeld mail die naar (nep-)gebruiker *allusers* wordt gestuurd, naar *staff, students* en *guests* wordt gestuurd, en al wat naar *staff* moet gaan, naar *vakzwc* en *vakzwa* moet gestuurd worden, enz. is ook weer een graaf. We zouden ons bijvoorbeeld kunnen afvragen welke gebruikers in geen enkele mail-alias zitten—of dus zoeken naar toppen die niet met de rest van de graaf verbonden zijn, ...

8.2.8 Hypertekst prettyprinter voor programma's

We stellen de broncode van SICECAS zelf ter beschikking op de homepage van het project. Hiervoor hebben we een eenvoudig programma gemaakt dat de Java broncode-bestanden omzet in wat aangenaam ogender HTML-bestanden en een aantal hypertekst-verwijzingen aanbrengt wanneer het ene bestand verwijst naar het andere. Het omzettingsprogramma is uiterst simpel gehouden en begrijpt dan ook maar tot op een zeer beperkt niveau de Java-syntax. Eens SICECAS in staat is Java-programma's te parsen (zie §8.2.4), kunnen we een veel krachtiger HTML-prettyprinter maken omdat SICECAS de Java-broncode nu in syntaxboom-vorm heeft. Het zal dan bijvoorbeeld erg makkelijk worden om, telkens een variabele in de tekst voorkomt, te verwijzen naar de plaats waar ze gedeclareerd werd, iets waarvoor de huidige prettyprinter veel te weinig benul heeft van de volledige Java-syntax. Vanzelfsprekend hopen we deze faciliteit algemeen genoeg te maken opdat ook programma's in andere talen door onze prettyprinter behandeld kunnen worden.

8.2.9 Formele logica-module herwerken

In de huidige versie is het niet mogelijk om van eerder gemaakte bewijzen gebruik te maken in een nieuw bewijs. Het is wel mogelijk om zelf een bewijsregel bij te maken. Als we een bewijs willen hergebruiken in latere bewijzen, moeten we er dus eigenlijk een nieuwe bewijsregel (axioma) van maken. Deze aanpak is duidelijk verre van ideaal; we zullen dan ook komen tot een systeem waar veel minder onderscheid is tussen bewijzen en bewijsregels.

8.2.10 Herschrijfregels

Om het programma ook bruikbaar te maken als een soort intelligent wiskundig kladblok, willen we de gebruiker de mogelijkheid bieden eigen *herschrijfregels* in te voeren. Een voorbeeld van zo'n regel zou kunnen zijn:

$$\sin(x+y) \longrightarrow \sin(x) * \cos(y) + \cos(x) * \sin(y)$$

Eens het systeem van deze regel kennis heeft genomen, kan de gebruiker naar gelijk welk deel van de formule wijzen en vragen dat die herschrijfregel wordt toegepast, of een overzicht vragen van welke herschrijfregels zouden kunnen toegepast worden op dat deel van de formule.

Vele andere computeralgebra-systemen hebben soortgelijke functies zoals *simplify*, *collect*, *expand* maar het is niet altijd even makkelijk precies aan te duiden welke regel er op welk deel van de formule moet toegepast worden.

We zouden de herschrijfregel-faciliteit kunnen combineren met formele bewijzen om zo de gebruiker toe te laten zijn herschrijfregels formeel te verifiëren. Misschien kunnen we zelfs een herschrijfregel een bijzonder geval maken van een bewijs; in het voorbeeld van daarnet zou elk bewijs van de formule '$\sin(x+y) = \sin(x) * \cos(y) + \cos(x) * \sin(y)$' automatisch de status van herschrijfregel kunnen krijgen.

8.2.11 Hulpmiddel bij het schrijven van wiskundige teksten

Zoals aangekondigd in §3 zou het interessant zijn het systeem te gebruiken als hulpmiddel bij het schrijven van wiskundige teksten. Zo'n document bestaat meestal uit een aantal formules die afgeleid worden van elkaar, met daartussenin stukken tekst. Met de infrastructuur die we hierboven beschreven hebben (herschrijfregels en eventueel ook formele bewijsvoering) is het vanzelfsprekend mogelijk, net zoals in andere computeralgebra-systemen, om de ene formule uit de andere af te leiden. De voorziening om die formules te mengen met stukken tekst is in de meeste pakketten veel zwakker: de formule bewaren in een populair tekstverwerker-formaat is meestal al wat we krijgen. We stellen voor één of ander mechanisme te ontwerpen dat de afgeleide formules automatisch in een stuk LaTeX-tekst plakt (in afwachting dat SICECAS ooit zelf een tekstverwerkermodule bezit ...); de gebruiker zou de plaats waar de formules moeten komen in de tekst kunnen aangeven door een of andere gestandardiseerde commentaarlijn. Een hypothetisch voorbeeld: stel dat we de volgende formules bewezen hebben:

$$\begin{aligned} \sin(x+y+z) &= \sin(x+y) * \cos(z) + \cos(x+y) * \sin(z) \\ &= (\sin(x) * \cos(y) + \cos(x) * \sin(y)) * \cos(z) + \cos(x+y) * \sin(z) \end{aligned}$$

dan zou het LaTeX-fragment

```
Uit
%%% formule 1
leiden we gemakkelijk af dat
%%% formule 3
waaruit blablabla
```

door SICECAS omgezet worden in iets in de trant van

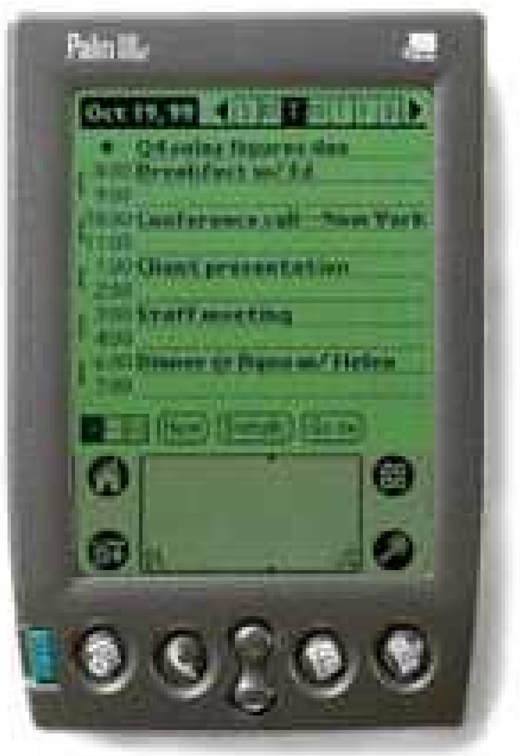
```
Uit
%%% formule 1
%%% begin formule 1
$\sin(x+y+z)$
%%% eind formule 1
leiden we gemakkelijk af dat
%%% formule 3
%%% begin formule 3
$(\sin(x)*\cos(y)+\cos(x)*\sin(y))*\cos(z) + \cos(x+y)*\sin(z)$
%%% eind formule 3
waaruit blablabla
```

wat uiteindelijk op papier komt als

> Uit $\sin(x+y+z)$ leiden we gemakkelijk af dat $(\sin(x)*\cos(y)+\cos(x)*\sin(y))*\cos(z)+\cos(x+y)*\sin(z)$ waaruit blablabla

Het is dan de bedoeling dat de gebruiker van de stukken LaTeX code tussen `%%% begin formule` en `%%%eind formule` afblijft, omdat SICECAS die stukken weer overschrijft als de gebruiker de formules in SICECAS zou wijzigen.

8.3 Hardware



SICECAS is van in het begin bedoeld om op een rekenmachine te draaien. Het draait uiteraard probleemloos op een laptop, maar dat is toch nogal een groot en duur ding voor een rekenmachine. Nu de prijs van de "handhelds" binnen het bereik van die van de rekenmachines begint te zakken, en ze voldoende krachtig worden (2 tot 8MB RAM en 15 à 100 MHz is geen uitzondering) wordt het interessant een poging te wagen SICECAS op zo'n toestel aan de praat te krijgen. Grootste obstakel hierbij is het vinden van een Java-implementatie die draait op zulke handhelds. Sun heeft zelf een afgeslankte versie van Java ontwikkeld die op de *PalmPilot* toestellen draait, maar helaas is die versie wel heel afgeslankt (ze is eigenlijk bedoeld voor nog compactere toestellen, zoals GSM's): er is geen ondersteuning voor kommagetallen en de grafische toolkit is ook anders en beperkter. Het gebrek aan kommagetallen is voor ons niet zo dramatisch als het mag lijken: de meeste computeralgebra-pakketten werken intern met eigen implementaties om een willekeurig hoge precisie te kunnen halen (de standaard kommagetallen hebben een vooraf bepaalde precisie). Het tweede bezwaar is ook al onderkend door een hoop andere ontwikkelaars, en ze zijn niet bij de pakken blijven zitten: het *kAWT-Project* zorgt ervoor dat we onze vertrouwde grafische omgeving kunnen behouden op de Palm. Een alternatief is een of andere Open Source implementatie van Java zelf (bijvoorbeeld Kaffe[4]) overdragen naar de Palm.

[4] zie `http://www.kaffe.org/`

9 Besluit

We menen met dit document aangetoond te hebben dat we in staat zijn

- Bestaande pakketten te analyseren op hun sterke en zwakke punten.
- De beperkingen van bestaande hard- en software te onderkennen en te omzeilen.
- De gebruikte methodiek en de beperkingen van onze eigen projecten kritisch trachten te onderzoeken en te verbeteren.
- Zelfstandig innovatieve oplossingen te bedenken en te implementeren voor opduikende problemen.
- Zelfstandig niet-triviale projecten (d.i. een project waarvan de ontwikkeling over meerdere jaren gespreid verloopt) op te zetten en aan de gang te houden.
- Uit de vele voorhanden zijnde mogelijkheden consistente en algemene tools te distilleren, die niet enkel een oplossing vormen voor het probleem dat zich toevallig op een bepaald ogenblik stelt, maar ook later hergebruikt kunnen worden in een veelheid van situaties.

Technisch gedeelte

In deze bijlagen zullen we enige meer technische aspecten van SICECAS nader toelichten. We hebben ervoor gekozen deze uitleg in bijlagen onder te brengen om de rode draad uit het eerste gedeelte van deze verhandeling niet te zeer te moeten onderbreken. Vanzelfsprekend is het onmogelijk alle aspecten van het meer dan 11500 regels tellende programma te behandelen zonder de omvang van dit document tot onredelijke proporties op te blazen.

A Werking van de struktuur-editor

A.1 Syntaxbomen

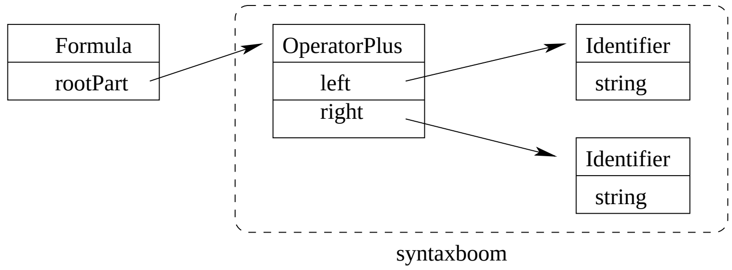
In de tekst gaven we al aan dat formules intern als syntaxboom worden behandeld. Hier gaan we wat meer in op enkele technische details die met deze aanpak gepaard gaan.

We zullen hier naar de broncode verwijzen, die geraadpleegd kan worden op `http://cage/~gvernaev/sicecas/source/gve/calc/`. De meeste klassen die we zullen beschrijven zitten in het gve.calc.formula package. (We zullen later op een package-naam overschakelen die minder doet denken aan de oude naam *GveCalc*, misschien wel gnu.calc, rug.sicecas of iets dergelijks.)

De syntaxboom zelf bestaat uit Part objecten. Elk Part object bevat een verwijzing parent naar zijn ouder (als het Part object in kwestie de top van de boom is, dan bevat parent de waarde null).

Part zelf is een abstracte klasse; de syntax-boom wordt dus bevolkt met subklassen ervan. Sommige van deze subklassen zijn eindpunten ('bladeren') van de boom, zoals Identifier. Andere bevatten kind-knopen; zo bevat een OperatorPlus object twee kinderen, left en right.

Manipulaties van de structuur van een syntax-boom worden niet rechtstreeks op de Part objecten ervan doorgevoerd, maar door een Formula object. Qua gegevensstructuur is een Formula niets meer dan enkel maar een verwijzing naar het Part object dat overeenkomt met de top van de syntax-boom. Tijdens manipulaties van de syntaxboom houdt het Formula object bij welk Part de nieuwe top van de syntax-boom geworden is.

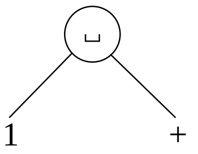


De enige manier waarop een syntaxboom gemanipuleerd mag worden is de replace() methode van Formula, die dan het werk achter de schermen doet.

A.2 Herkenning van operatoren

A.2.1 Extensie van de grammatica met een spatie-operator

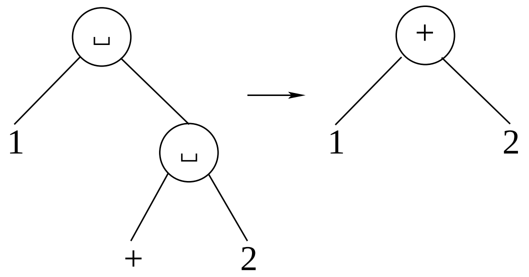
Terwijl de gebruiker een formule aan het intikken is, komt het vaak voor dat de gedeeltelijk ingetikte formule niet welgevormd is. Als we bijvoorbeeld 1+2 wensen in te tikken, dan is 1+ een niet welgevormde formule. Toch moeten we die op het scherm tonen—of de gebruiker zou nogal vreemd opkijken. De methode die we gebruiken is een "spatie-operator" (in de figuur voorgesteld door ␣) invoeren die ervoor zorgt dat alle formules grammaticaal worden:



Merk op dat het '+'-symbool hier nog niet als operator herkend is; het staat op dezelfde voet als identifiers als 'x' en '1'.

Vertaald naar de theorie van de formele talen komt onze aanpak erop neer dat we aan de originele grammatica (die formules herkent en hun structuur beschrijft) een *extensie* associëren waarvoor alle invoer een geldige formule is.

Op het ogenblik dat beide argumenten van een operator ingetikt zijn, wordt hij herkend. Als de gebruiker dus net '1+' heeft getikt en nu de afsluitende 2 intikt, dan gebeurt het volgende:

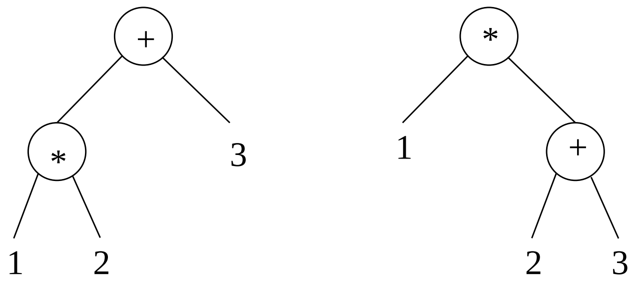


Eerst wordt dus met een extra spatie-operator de 2 aan de formule 'geplakt', waarna het systeem ziet dat de + (die voorkomt in een centrale lijst van binaire operatoren) door twee spaties omgeven is, en de operator wordt herkend.

Het toevoegen van spatie-operatoren wordt geregeld door de splitme() methode van de IdentifierView klasse. Het herkennen van een operator wordt afgehandeld door OperatorSpace.recognizeOp().

A.2.2 Prioriteit van operatoren

We zouden voor de formule '$1 * 2 + 3$' de volgende twee syntaxbomen kunnen bedenken:

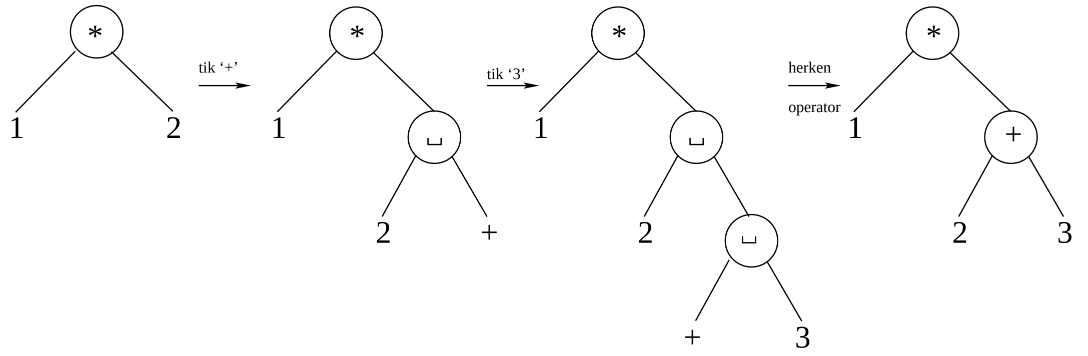


De linkse boom komt eigenlijk overeen met '$(1 * 2) + 3$' en de rechtse met '$1 * (2+3)$'. Deze twee formules betekenen dus iets heel verschillend. In de wiskunde is het gebruikelijk deze dubbelzinnigheid op te heffen door aan elke operator een *prioriteit* te hechten. We zeggen dat $*$ een *hogere prioriteit* heeft dan $+$, waarmee we bedoelen dat $*$ "harder plakt" dan $+$. Met '$1 * 2 + 3$' bedoelen we dan '$(1 * 2) + 3$'.

Implementatie: de getPri() methode van de Operator klasse moet de prioriteit van de operator in kwestie teruggeven.

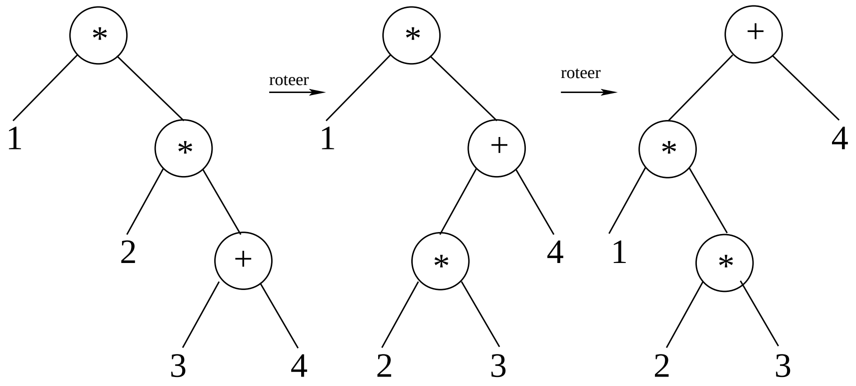
A.2.3 Rotatie

Tijdens het bewerken van een formule kan het gebeuren dat we toevallig de verkeerde van de twee mogelijke syntaxbomen hebben. Onderstel bijvoorbeeld dat de gebruiker de formule '$1 * 2$' ombouwt tot '$1 * 2 + 3$'. Het proces verloopt als volgt:



We moeten dus, nadat we een operator herkend hebben, controleren of de prioriteiten gerespecteerd worden, en zo nodig de andere van de twee mogelijke syntaxbomen kiezen. De regel is eenvoudig: operatoren met een lagere prioriteit moeten meer naar onder staan in de syntax-boom. Het wisselen tussen de twee vormen uit de figuur in §A.2.2 is een welbekende bewerking uit de wereld van de gegevensstructuren en wordt *roteren* genoemd.

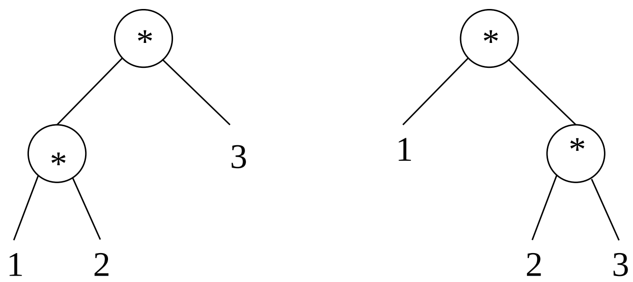
Soms zijn meerdere rotaties nodig, bijvoorbeeld wanneer de gebruiker de formule '$1 * 2 * 3$' aanvult tot '$1 * 2 * 3 + 4$' (we laten het herkennen van de operator met de spatie-operatoren even weg):



Roteren wordt afgehandeld door de rotate() methode van de InfixBinaryOp() en UnaryOp klassen, die zelf weer herhaaldelijk simpleRotate() aanroepen. De simpleRotate() methode doet één enkele rotatie of rapporteert dat de operator op de juiste plaats staat, en rotate() doet niet veel meer dan herhaaldelijk simpleRotate() blijven aanroepen totdat de operator op de juiste plek is aangekomen.

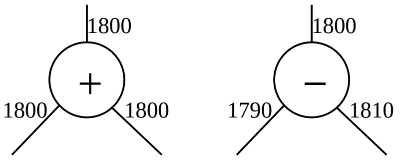
A.2.4 Associativiteit van operatoren

Merk op dat het bij de formule '$1 * 2 * 3$' eigenlijk niet uitmaakt op welke van beide manieren we de syntaxboom opbouwen:

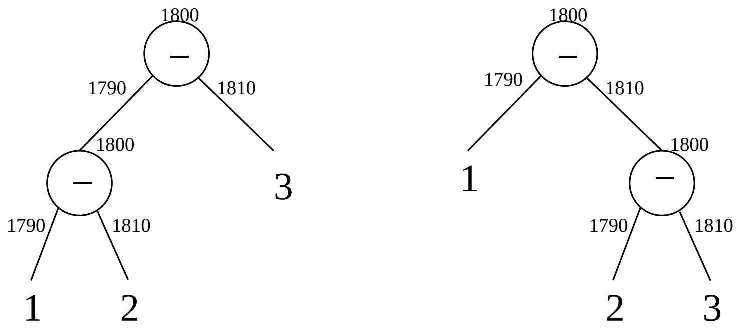


De reden hiervoor is dat de bewerking '$*$' commutatief is, of met andere woorden dat $(1*2)*3 = 1*(2*3)$. Niet alle operatoren zijn commutatief: zo is $(1-2)-3 \neq 1-(2-3)$. Voor de '$-$' operator moeten we dus een afspraak maken over wat we precies bedoelen met '$1 - 2 - 3$'. De meest gebruikelijke afspraak binnen de wiskunde is dat we bewerkingen met gelijke prioriteit van links naar rechts uitrekenen, dus in ons geval als '$(1 - 2) - 3$'.

We implementeren associativiteit in SICECAS door aan elke operator niet één prioriteit te hechten, maar *drie* prioriteiten:



Bij een commutatieve operator als '+' zijn alle drie de prioriteiten gelijk en het zal blijken dat er in dit geval niets verandert. Maar bij een links-naar-rechts associatieve operator zoals '−', verschillen de prioriteiten een klein beetje zoals aangegeven op de figuur. Als we nagaan of de syntax-boom geroteerd moet worden, kijken we nu naar beide uiteinden van elk verbindingsstreepje. Een voorbeeld, dat aantoont dat de '−' operator met deze aanpak wel degelijk goed geassocieerd wordt:



Er zijn hier in beide gevallen maar twee prioriteiten te vergelijken (omdat identifiers als 1 en 2 geen prioriteit hebben). In het eerste geval moeten we 1790 en 1800 vergelijken; de grootste prioriteit staat onderaan dus we doen niets. In het tweede geval staat 1810 meer naar boven in de boom dan 1800 en moeten we dus, de regel volgend, roteren (en we bekomen de linkse syntaxboom). Het eindresultaat van dit alles is dat de operator inderdaad rechts-naar-links geassocieerd wordt. Als we bij de keuze van de prioriteiten de 1790 en 1810 hadden omgewisseld, dan hadden we een links-naar-rechts geassocieerde operator.

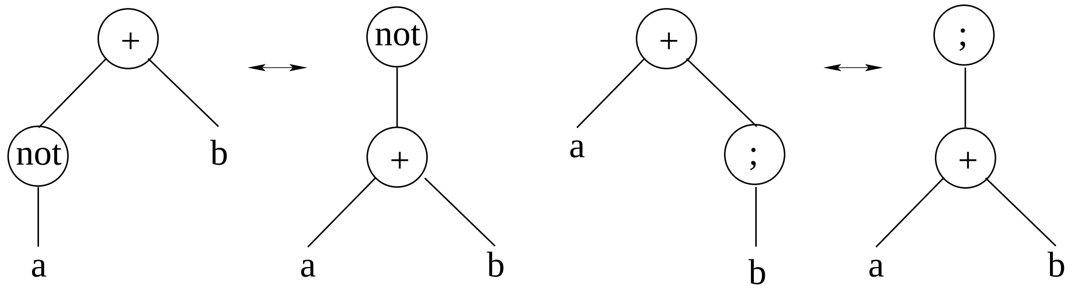
Het idee om meerdere prioriteiten per operator te gebruiken hebben we overgenomen van *Compiler Writing*, Tremblay & Sorenson, McGraw-Hill, 1985 waar een algoritme beschreven staat om een formule in infix-notatie naar postfix-notatie om te zetten, dat gebruikt maakt van twee prioriteiten per operator.

Implementatie: de `getLeftPri()` en `getRightPri()` methoden van `BinaryOp` geven de prioriteiten aan de twee onderste verbindingsstreepjes terug. De prioriteit van het bovenste streepje wordt zoals vroeger door `getPri()` teruggegeven. `getLeftPri()` en `getRightPri()` nemen standaard de waarde van `getPri()` over, dus als de operator commutatief is en de associativiteitsrichting er niet toe doet, moet de implementerende klasse enkel maar een `getPri()` methode hebben.

A.2.5 Unaire operatoren

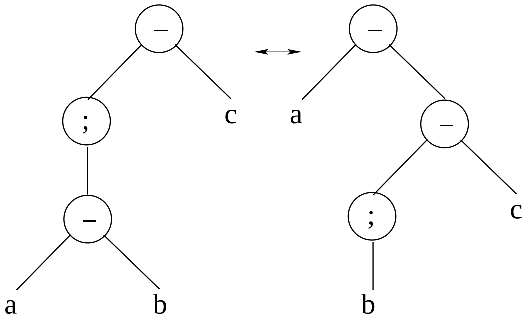
Unaire operatoren hebben maar twee prioriteiten nodig, en in plaats van getRight-Pri() en getLeftPri() is er nu de getDownPri() methode van de UnaryOp klasse.

De rotatieregels zijn hier soortgelijk, maar er is een extra onderscheid tussen unaire prefix- (zoals 'not') en postfix-operatoren (zoals ';'):



Figuur: de formules 'not $a + b$' en '$a + b$;' op twee manieren geassocieerd

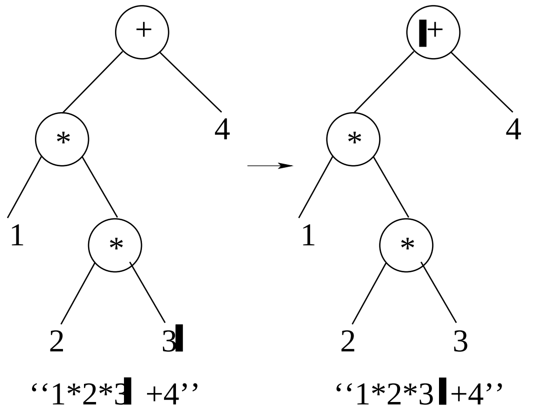
Bovendien duiken er ook rotatieregels op met *drie* operatoren—iets dat in de meeste klassieke boom-algoritmes niet voorkomt!



Figuur: de formule '$a - b; -c$' op twee manieren geassocieerd

A.3 Rondwandelen in een syntaxboom

Hoewel de formule intern als een syntaxboom voorgesteld wordt, mag de gebruiker daar niet al te veel van merken bij het bewerken ervan. We bekijken eens wat er gebeurt als de gebruiker op de 'cursor naar rechts' toets drukt binnen de formule '$1 * 2 * 3 + 4$':



In de syntaxboom heeft de cursor een grote sprong gemaakt, terwijl op het scherm de cursor maar een klein beetje is bewogen.

Het is instructief eens te bekijken wat er allemaal achter de schermen gebeurt als gevolg van zo'n toetsaanslag.

- Het systeem aanroept de keydn() methode van de IdentifierView waar de cursor op staat (de '3' identifier). Deze methode probeert de cursor naar rechts te verplaatsen binnen de identifier zelf, maar omdat de cursor al uiterst rechts staat, is dat onmogelijk. De methode geeft daarom als resultaat false terug, om aan te geven dat het verplaatsen van de cursor mislukt is.

- Het systeem gaat dan de keydn() methode van de ouder van de IdentifierView proberen, een InfixBinaryOpView (die overeenkomt met de rechtse '$*$' in de formule). Die InfixBinaryOpView ziet dat de cursor ergens in zijn rechterkind staat; de cursor kan niet verder naar rechts; het resultaat van de keydn() aanroep is weer false.

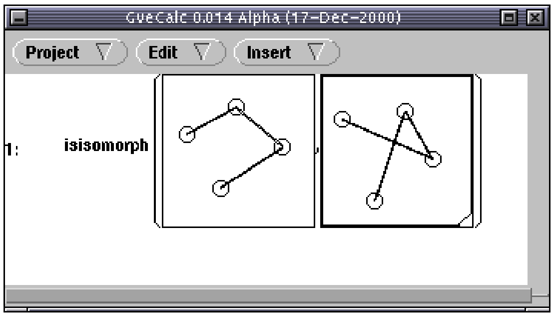
- Het systeem gaat weer een stapje naar omhoog in de syntaxboom; nu komen we bij de meest linkse '$*$' in de formule uit. Hier speelt zich weer hetzelfde scenario af.

- Tenslotte komen we terecht bij de InfixBinaryOpView die overeenkomt met de '+' operator. Deze view ziet dat de cursor ergens in zijn linkerkind staat. De cursor kan dus naar rechts bewogen worden door hem net links van de '+' te plaatsen. De keydn() methode van deze view geeft dus eindelijk true terug, en het systeem stopt zijn zoektocht.

A.3.1 Cursorpositie

In deze paragraaf lichten we toe op welke manier een formule-view bijhoudt waar de cursor ergens staat.

Elke FormulaView bevat een cursorComp veld dat de component aanduidt waarin de cursor ergens staat (als er geen cursor staat in de formule, dan bevat dit veld de waarde null). De precieze positie van de cursor binnen die component wordt door deze component zelf bijgehouden. Componenten waarbinnen de cursor kan bewegen, implementeren de HasCursorPos interface. Een IdentifierView is een voorbeeld van een view waarbinnen de cursor kan bewegen; binnen een GraphView daarentegen kan de cursor niet bewegen en GraphView implementeert HasCursorPos dan ook niet. Dat betekent overigens niet dat de cursor niet op een graaf kan staan; de cursor kan enkel niet *binnen* een graaf bewegen. In de volgende formule staat de cursor op de rechtse graaf:



De HasCursorPos interface ziet er als volgt uit:

```
public interface HasCursorPos extends ActivationListener {
        public static final int Somewhere_LEFT = -1;
        public static final int Somewhere_RIGHT = -2;
        public static final int Somewhere_TOP = -3;
        public static final int Somewhere_BOTTOM = -4;
        public static final int Somewhere = -5;
        public static final int Somewhere_NOWHERE = -6;
        /** MYRIGHT and MYLEFT differ from RIGHT and LEFT:
         * in the case of a binary operator (e.g. "a+b"),
         * LEFT means at the left of the leftmost argument
         * (so at the left of the "a"),
         * while MYLEFT means at the left of the operator itself
         * (so between "a" and "+") */
        public static final int Somewhere_MYLEFT = -7;
        public static final int Somewhere_MYRIGHT = -8;

        public void setCursorPos(int pos);

        public int getCursorPos();
}
```

Een component waarbinnen de cursor kan bewegen, houdt de cursorpositie dus bij door middel van een geheel getal (een int). Bij een IdentifierView bijvoorbeeld betekent 0 dat de cursor links van de eerste letter staat, 1 dat de cursor tussen de eerste en de tweede letter staat, enz. Met de setCursorPos() en getCursorPos() methoden kunnen we dit getal veranderen en opvragen. Waarden kleiner dan nul zijn bij afspraak enkel toegestaan voor setCursorPos(), zodat we de cursor ergens kunnen plaatsen binnen een component zonder dat we precies moeten weten op welke manier die zijn cursorpositie codeert. Zo kunnen we met setCursorPos(Somewhere_RIGHT) de cursor aan de rechterkant van een IdentifierView krijgen, zonder dat we moeten weten hoeveel letters die identifier lang is.

De component waarin de cursor zich bevindt (die bijgebouden wordt in het cursorComp veld van de FormulaView) kan enkel maar veranderd worden door methoden van de FormulaView als setCursorComp(), setCursorPart, ... Vlak vóór de cursor een component verlaat, wordt de deactivate() methode van die component aanroepen; vlak nadat de cursor op een component terecht is gekomen, wordt de activate() methode ervan aanroepen (deze beide methoden behoren tot de ActivationListener interface). Op die manier kan de component zijn voorstelling op het scherm aanpassen aan de situatie.

A.4 Model-View-Controller

A.4.1 Principe

Fundamenteel aan de basis van onze struktuur-editor ligt het zogenaamde *Model-View-Controller paradigma*, dat voor het eerst rond 1980 in Smalltalk werd toegepast. Het idee is dat we de interne voorstelling van gegevens (*Model*) loskoppelen van de manier waarop we die aan de gebruikers voorstellen (*View*). Het model kan bijvoorbeeld een rijtje getallen zijn, waarbij de view een staafdiagram, een taartdiagram, ... kan zijn. Eén model kan verschillende views tegelijkertijd hebben; zo zouden we dezelfde gegevens tegelijkertijd als taart- en als staaf-diagram kunnen laten zien. De derde speler, de controller, is verantwoordelijk voor het verwerken van invoer van de gebruiker en het overeenkomstig aanpassen van het model. In het voorbeeld zou de gebruiker aan één van de staven kunnen trekken met de muis. De controller vangt deze muisbeweging op en past het model aan. Het model stuurt dan een signaal naar alle views dat het model veranderd is, zodat ook de taartdiagram-view tegelijkertijd mee verandert. Merk op dat niet de controller zelf, maar wel het model diegene is die verantwoordelijk is voor het signaleren aan de views dat er iets aan het model is veranderd.

A.4.2 Implementatie

De klasse FormulaView beschrijft views van Formula objecten. De view bevat intern een verwijzing naar het model (de Formula), zodat de view snel kan uitvinden hoe het model eruit ziet. Modellen zelf weten niet welke views er allemaal met hen geassocieerd zijn; het is de klasse MVC die deze informatie bevat. Een verse view moet zich bij de MVC klasse registreren met de registerView() methode. Als een model zichzelf van toestand verandert, moet het de MVC.changed() methode aanroepen. Die methode kijkt dan welke views er allemaal voor dat model geregistreerd waren, en roept de updateView() methoden van al die views één voor één op. Optioneel kan een extra argument aan changed() worden geleverd, dat het doorstuurt als argument van updateView(). Op die manier kunnen we de views wat meer informatie geven over wat er precies aan het model is veranderd, zodat de views zich sneller kunnen aanpassen aan de verandering.

Een model kan gelijk welk Java object zijn, maar een view moet de View interface implementeren. Deze interface bevat maar twee methoden:

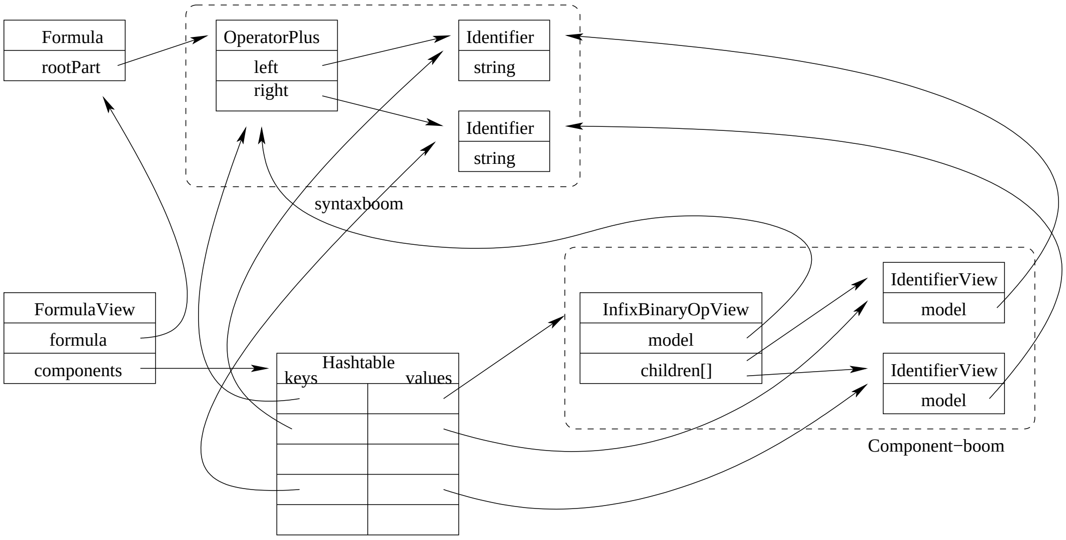
```
public interface View {
        /** Called by the model whenever the model has changed.
         * The View should change its appearance accordingly. */
        public void updateView(Object with);

        public Object getModel();
}
```

Een syntaxboom wordt op het scherm getoond als een boom van standaard Java Component objecten. De FormulaView is verantwoordelijk voor het manipuleren van die Component-boom, net zoals de Formula verantwoordelijk is voor de manipulatie van de Part boom. De FormulaView roept de createView() methode van een Part op om de Component-boom op te bouwen: Part.createView() wordt verondersteld een Component terug te geven die een visuele voorstelling is van het Part. Een FormulaView roept gegarandeerd maar één keer de createView() methode van een Part op: eens het Part zijn view heeft gegenereerd, houdt de FormulaView die bij in een interne hashtabel.

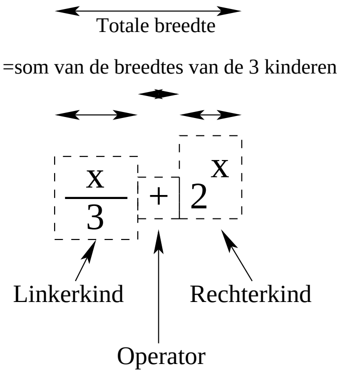
Wanneer de gebruiker een formule manipuleert, worden er dus eigenlijk twee bomen tegelijkertijd veranderd: de syntax-boom (die bestaat uit Part objecten) en de Component-boom.

Zoals al uitgelegd in §A.1 is de enige toegelaten manier om een syntaxboom te manipuleren de Formula.replace() methode. Deze methode zorgt ervoor (via MVC.changed()) dat alle geregistreerde FormulaViews een melding krijgen van welke manipulatie er juist doorgevoerd is.



Voor de eenvoud hebben we op deze figuur het MVC object weggelaten,
dat alle registraties van views met hun modellen bevat ...

In de figuur is ook duidelijk te zien dat de component-boom precies analoog wordt opgebouwd met de model-boom. Al deze componenten hebben zich bij de MVC klasse geregistreerd als views van de overeenkomende Part-objecten uit de syntaxboom. Een belangrijk aandachtspunt bij het ontwerpen van het programma is dat de algoritmen die deze bomen manipuleren niet onredelijk langzaam mogen draaien wanneer de bomen wat groter worden. Concreet moet een component bijvoorbeeld berekenen wat zijn breedte en hoogte (in pixels) op het scherm is. Laten we eens nader bestuderen hoe een InfixBinaryOpView-knoop zijn afmetingen berekent. Zo'n InfixBinaryOpView stelt een binaire operator voor (bijvoorbeeld '+'). De breedte ervan zouden we dan kunnen bepalen door de breedte van het linkerkind te berekenen, de breedte van de operator zelf, en de breedte van het rechterkind, en tenslotte deze drie getallen bij elkaar op te tellen.



Maar beide kinderen kunnen zelf weer grote syntaxbomen zijn. Om hun afmetingen te berekenen, vragen ze weer de afmetingen van hun kinderen op, enz.

Deze aanpak werkt uiteraard wel, maar is erg traag: meestal is er maar een klein stukje van de syntaxboom veranderd. Het is dan ook veel efficiënter de berekende afmetingen ergens bij te houden zodat we die volgende keer onmiddellijk kunnen kopiëren in plaats van opnieuw berekenen. Alleen de views die echt van grootte veranderd zijn, moeten dan even hun afmetingen herberekenen. De Java LayoutManager-klassen hebben hiervoor voorzieningen.

B Werking van de 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 :-)

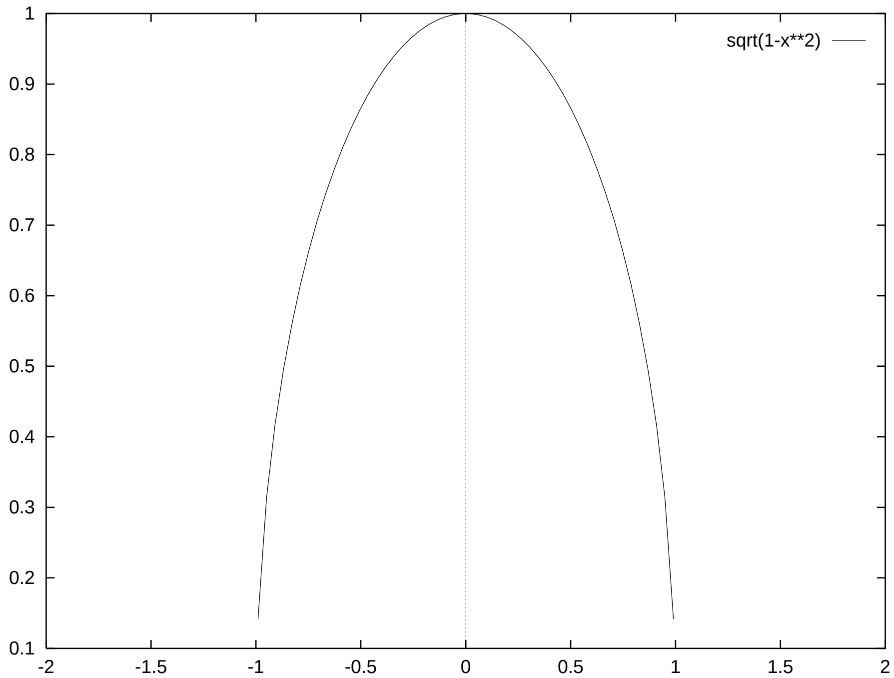
Een mogelijke oplossing voor dit probleem is de volgende. De klassieke functieplotter evalueert de functie in een aantal punten $x_0, x_1, \ldots, x_n$. Wij gaan de functie evalueren in een aantal *intervallen* $[x_0, x_1[, [x_1, x_2[, \ldots, [x_{n-2}, x_{n-1}[, [x_n - 1, x_n]$. Hoe we dat technisch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

B.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: normaal gezien moet de grafiek van de functie vertrekken vanaf de X-as (dus $y = 0$), terwijl de grafiek hier vertrekt vanaf ongeveer $y = 0,15$, een flink stuk boven de X-as. 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.

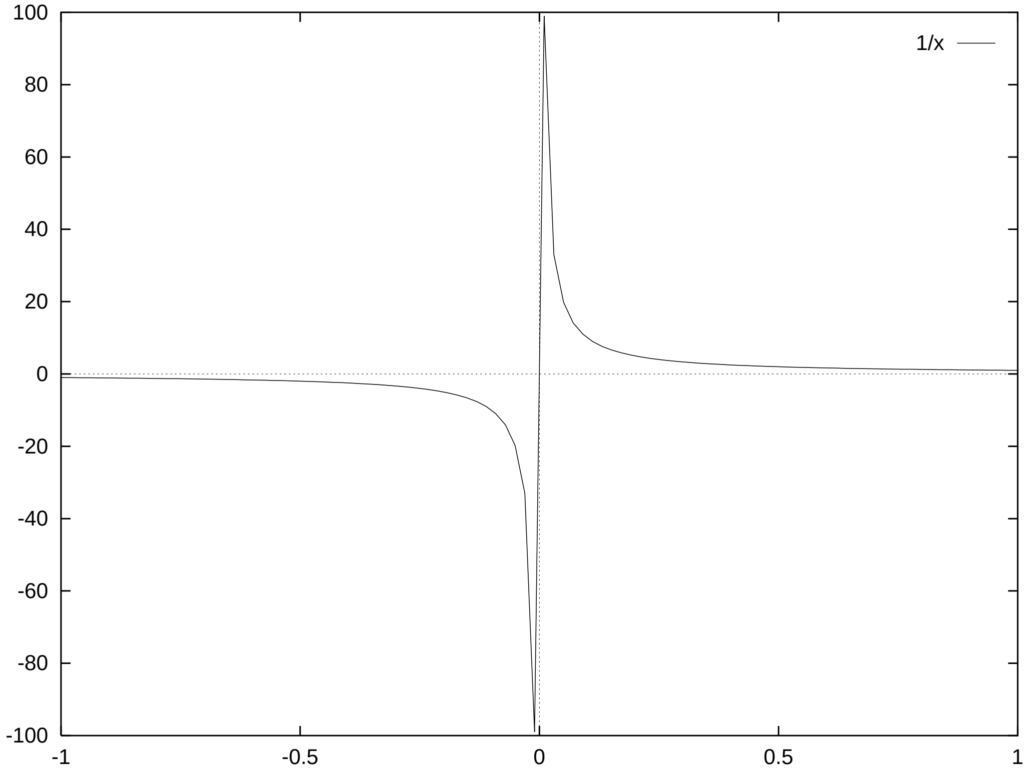
B.2 De functie $\dfrac{1}{x}$

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 een fout resultaat (in de handleiding staat dat de gebruiker zelf voor zo'n situaties moet opletten).

Als we in Gnuplot deze functie plotten met

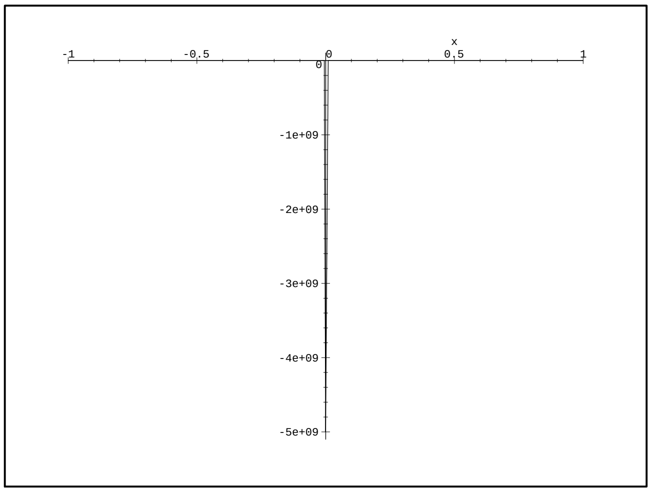
```
plot [x=-1:1] 1/x
```

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

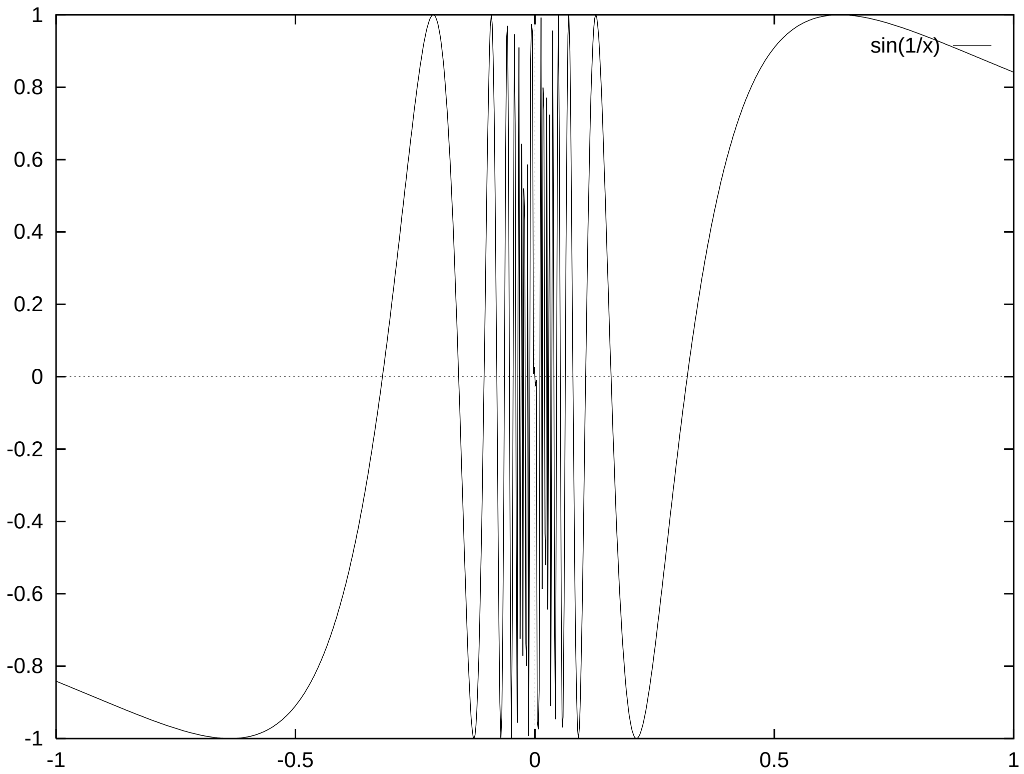
Ten tweede is er het probleem dat sommige functiewaarden rond een asymptoot zeer groot in absolute waarden kunnen worden. De puntsgewijze plotters proberen het Y-bereik automatisch te vinden door alle gevonden punten te tonen, met alle problemen van dien. De intervalplotter vindt rond een asymptoot echt 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.

B.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. Een vaak gesuggereerde schijn-oplossing is om het programma de grafiek van de functie pixel voor pixel te laten volgen. In het voorbeeld van $\sin \dfrac{1}{x}$ zouden we dan elke oscillatie van -1 tot $+1$ pixel voor pixel meevolgen (en dus steeds trager van links naar rechts moeten gaan bewegen). Het probleem met deze manier van werken is dat er oneindig veel oscillaties zijn in de grafiek van $\sin \dfrac{1}{x}$ en de plotter dus nooit bij $x = 0$ zou aankomen!

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

B.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.

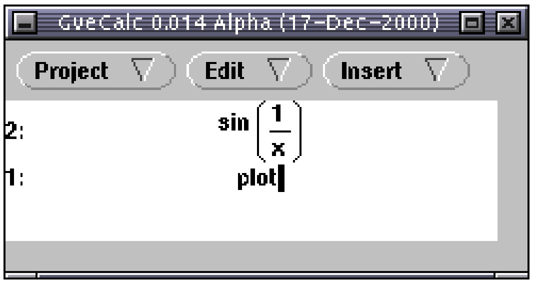
We zien dus dat het werken met intervallen als argument conceptueel niet veel ingewikkelder is dan het werken met getallen. Toch schuilt er een addertje onder het gras. Stel dat we $f(x) = x - x$ willen plotten. We moeten dan $f([a, b[)$ voor een aantal a- en b-waarden ($a < b$) berekenen. Bij het uitrekenen 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.

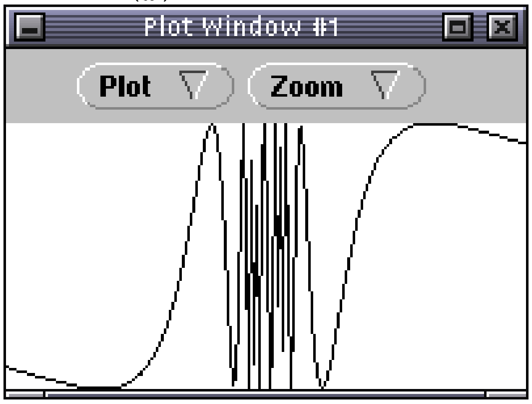
Een mogelijke oplossing is de volgende. We proberen beide methoden als volgt te verzoenen: bereken eerst $f([a, b[)$ met de intervalmethode. Probeer vervolgens in een aantal x-waarden met $x \in [a, b[$ de functie f te evalueren. Als we in elke pixel van $f([a, b[)$ een $f(x)$ vinden, dan zijn we zeker 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 $[a, b[$ (noem ze $[a, c_0[, [c_0, c_1[, \ldots, [c_n, b[$) 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 $f([a, c_0[) \cup \cdots \cup f([c_n, b[)$ in het grijs.

Deze methode levert ons dus weliswaar niet de ideale functieplotter op, maar we weten wel zeker dat de functie gaat door alle zwarte pixels en niet gaat door alle witte pixels.

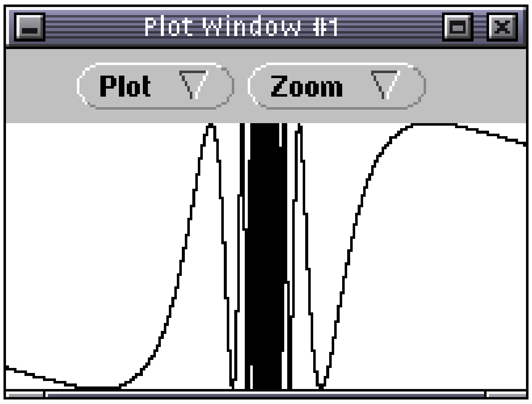
B.5 Huidige implementatie

Voor het ogenblik ondersteunt SICECAS de zuivere interval-methode; de verfijning met het proberen te bedekken van de gevonden intervallen met pixels hebben we nog niet geïmplementeerd. Niettemin kunnen we toch laten zien dat enige problemen verholpen zijn. We maken de grafiek van $sin\left(\frac{1}{x}\right)$ met het *plot*-commando:





De getoonde grafiek is geplot op de klassieke wijze, waarbij een aantal punten berekend worden en met lijntjes verbonden worden. In het *Plot* menu kiezen we de optie *Intervals* en we krijgen een goede grafiek te zien:



C Regressietesten

Wanneer een gedeelte van een programma gewijzigd wordt, kunnen hierdoor fouten ontstaan in andere gedeelten ervan. Als we een programmeerfout maken en ontdekken, is de kans groot dat we later, bij het wijzigen van het verbeterde stuk programma, dezelfde fout opnieuw zullen maken. De geijkte manier om dit soort problemen op te vangen is een *regressie-test*. In een regressie-test laten we het programma automatisch een aantal bewerkingen verrichten, en controleren of ze het correcte resultaat opleveren. Telkens we een verandering aan het programma aanbrengen, laten we de regressie-testsuite draaien om te zien of we niet 'hervallen' in oude bugs.

C.1 Codevoorbeeld

We zullen eens expliciet nagaan hoe zo'n regressietest eruit ziet: het testprogramma roept achtereenvolgens de testroutines test1, test2, ... aan, waarvan we er hier één nader belichten:

```
static boolean test1(StackCanvas canvas) {
        System.out.println("test 1 (2-mar-2000)");
        // Test: type "notx", split with a space in "not x"
        // and check that no superfluous space has been generated
        // (same as "forallx" in the debug report)
        Formula f;
        Part shouldbe = new OperatorNot(new Identifier("x"));
        canvas.push(f = new Formula("notx"));
        canvas.setActiveFormula(0);
        FormulaView view = (FormulaView)canvas.getComponent(0);
        view.keydn(KeydnListener.Key_RIGHT);
        view.keydn(KeydnListener.Key_RIGHT);
        view.keydn(KeydnListener.Key_RIGHT);
        view.keydn(' ');
        Part p = f.getRootPart();
        boolean result;
        if (p.same(shouldbe)) {
                result = true;
        } else {
                p.dump();
                shouldbe.dump();
                result = false;
        }
        canvas.pop();
        return result;
}
```

We gaan hier na wat er gebeurt als de gebruiker de identifier 'notx' zou intikken en daarna een spatie tussen de 'not' en de 'x' zou tikken. Met de view.keydn(KeydnListener.Key_RIGHT); wandelen we met de cursor naar de positie tussen 'not' en 'x', en met view.keydn(' '); bootsen we het intikken van een spatie na. Daarna vergelijken we de resulterende syntaxboom p met de syntaxboom die we verwachten (shouldbe). Met canvas.pop() verwijderen we de ingetikte formule weer van de stapel, zodat we de testfunctie met een schone stapel verlaten. De testfunctie wordt verwacht true terug te geven als de test geslaagd is en false in het andere geval (dan wordt ook wat extra debug informatie uitgeschreven door de `dump()` aanroepen).

Index

Inhoudsopgave