Functioneel Programmeren

U kunt de auteur bereiken op het volgende adres: Jeroen Fokker, Informatica-instituut,
Postbus 80089, 3508 TB Utrecht, e-mail `jeroen@cs.uu.nl`.

1e druk (informatica-versie) september 1992
2e druk (informatica-versie) februari 1993
3e druk (informatica-versie) september 1993
4e druk (informatica-versie) september 1994
5e druk (informatica-versie) september 1995
6e druk (CKI-versie) oktober 1995
7e druk (informatica-versie) september 1996
8e druk (CKI-versie) oktober 1996
9e druk (CKI-versie) oktober 1997
10e druk (CKI-versie) oktober 1998

Inhoudsopgave

Hoofdstuk 1

Functioneel Programmeren

1.1 Functionele talen

1.1.1 Functies

In de jaren veertig werden de eerste computers gebouwd. De allereerste modellen werden nog 'geprogrammeerd' met grote stekkerborden. Al snel werd het programma echter in het geheugen van de computer opgeslagen, waardoor de eerste *programmeertalen* de intrede deden.

Omdat destijds het gebruik van een computer vreselijk duur was, lag het voor de hand dat de programmeertaal zo veel mogelijk aansloot bij de architectuur van de computer. Een computer bestaat uit een besturingseenheid en een geheugen. Een programma bestond daarom uit instructies om het geheugen te veranderen, die door de besturingseenheid worden uitgevoerd. Daarmee was de *imperatieve programmeerstijl* ontstaan. Imperatieve programmeertalen, zoals Pascal en C, worden gekenmerkt door de aanwezigheid van toekenningsopdrachten (*assignments*), die na elkaar worden uitgevoerd.

Ook voordat er computers bestonden werden er natuurlijk al methoden bedacht om problemen op te lossen. Daarbij is eigenlijk nooit de behoefte opgekomen om te spreken in termen van een geheugen dat verandert door instructies in een programma. In de wiskunde wordt, in ieder geval de laatste vierhonderd jaar, een veel centralere rol gespeeld door *functies*. Functies leggen een verband tussen parameters (de 'invoer') en het resultaat (de 'uitvoer') van bepaalde processen.

Bij elke berekening hangt een resultaat op een of andere manier af van parameters. Daarom is een functie een goede manier om een berekening te specificeren. Dit is de basis van de *functionele programmeerstijl*. Een 'programma' bestaat uit de definitie van een of meer functies. Bij het 'uitvoeren' van een programma wordt een functie van parameters voorzien, en moet het resultaat berekend worden. Bij die berekening is nog een zekere mate van vrijheid aanwezig. Waarom zou een programmeur immers moeten voorschrijven in welke volgorde onafhankelijke deelberekeningen moeten worden uitgevoerd?

Met het goedkoper worden van computertijd en het duurder worden van programmeurs wordt het steeds belangrijker om een berekening te beschrijven in een taal die dichter bij de belevingswereld van de mens staat dan bij die van de computer. Functionele programmeertalen sluiten aan bij de wiskundige traditie, en zijn niet al te sterk beïnvloed door de concrete architectuur van de computer.

1.1.2 Talen

De theoretische basis voor het imperatief programmeren werd al in de jaren dertig gelegd door Alan Turing (in Engeland) en John von Neuman (in de USA). Ook de theorie van functies als berekeningsmodel stamt uit de twintiger en dertiger jaren. Grondleggers zijn onder andere M. Schönfinkel (in Duitsland en Rusland), Haskell Curry (in Engeland) en Alonzo Church (in de USA).

Het heeft tot het begin van de jaren vijftig geduurd voordat iemand op het idee is gekomen om deze theorie daadwerkelijk als basis voor een programmeertaal te gebruiken. De taal Lisp van John McCarthy was de eerste functionele programmeertaal, en is ook jarenlang de enige gebleven. Hoewel Lisp nog steeds wordt gebruikt, is dit niet de taal die aan de jongste eisen voldoet. Met het toenemen van de complexiteit van computerprogramma's deed zich namelijk steeds meer de behoefte voelen aan een sterkere controle van het programma door de computer. Het gebruik van

typering speelt daarbij een grote rol. In de jaren tachtig zijn dan ook een groot aantal getypeerde functionele programmeertalen ontstaan: ML, Scheme (een aanpassing van Lisp), Miranda, Gofer en Clean zijn een paar voorbeelden.

Op den duur ging elke onderzoeker zijn eigen taal ontwikkelen. Om deze wildgroei in talen enigszins te beteugelen, heeft een aantal onderzoekers besloten om samen een taal te ontwerpen. Dit heeft in 1992 geresulteerd in de taal *Haskell*. Er zijn inmiddels een drietal verschillende implementaties van deze taal. In dit diktaat gebruiken we een implementatie genaamd *hugs* (wat staat voor *Haskell User's Gofer System*, waarbij *Gofer* weer staat voor *good for equational reasoning*; Gofer was een van de voorlopers van Haskell.

De talen ML en Scheme hebben ook een grote aanhang; deze talen hebben echter enkele concessies gedaan in de richting van imperatieve talen. Als voorbeeld van een puur functionele taal zijn ze daarom minder geschikt. Miranda is wel een echte functionele taal, maar door de moeilijke verkrijgbaarheid niet erg gangbaar.

1.2 De Haskell-interpreter

1.2.1 Expressies uitrekenen

In een functionele programmeertaal kun je functies definiëren. Deze functies zijn bedoeld om te gebruiken in een expressie, waarvan de waarde uitgerekend moet worden. Om de waarde van een expressie met een computer te berekenen is een programma nodig dat de functiedefinities begrijpt. Zo'n programma heet een *interpreter*.

Voor de taal Haskell die in dit diktaat gebruikt wordt is een Haskell-interpreter beschikbaar genaamd *hugs*. Deze interpreter wordt gestart door de naam van het programma, `hugs`, in te tikken. Op het scherm verschijnt dan het volgende:

```
% hugs
   ___    ___   ___    ___   __________   __________
  /  /   /  /  /  /   /  /  /  _______/  /  _______/        Hugs 1.4
 /  /___/  /  /  /   /  /  /  / _____   /  /______
/  ____   /  /  /   /  /  /  / /_   /  /  /______   /  The Nottingham and Yale
/  /   /  /  /  /___/  /  /  /___/  /  /  _______/  /    Haskell User's System
/__/   /__/  /_________/  /_________/  /_________/         January 1998
  Copyright (c) The University of Nottingham and Yale University, 1994-1997.
   Bug reports: hugs-bugs@haskell.org.   Web: http://www.haskell.org/hugs.
Reading file "/sw/pkg/hugs/share/hugs/lib/Prelude.hs":
```

In de file `Prelude.hs` (die in het voorbeeld in de directory `/sw/pkg/hugs/share/hugs/lib` staat) staan de definities van standaardfuncties. Het eerste wat de Haskell-interpreter doet, is het analyseren van deze standaardfuncties. 'Prelude' betekent letterlijk 'voorspel': deze file wordt bekeken voordat eventuele nieuwe functies gedefinieerd worden. In dit geval zijn er nog geen nieuwe functies; de interpreter geeft daarom met

```
Hugs session for:
/sw/pkg/hugs/share/hugs/lib/Prelude.hs
```

aan dat alleen de definities die in de prelude staan gebruikt mogen worden. Tenslotte meldt de interpreter dat je `:?` kunt tikken voor een korte gebruiksaanwijzing:

```
Type :? for help
Prelude>
```

De prompt `Prelude>` geeft aan dat dat de laatst geladen file is, en dat de interpreter nu klaarstaat om de waarde van een expressie te berekenen. Doordat in de prelude onder andere rekenkundige functies gedefinieerd worden, kan de interpreter nu gebruikt worden als rekenmachine:

```
Prelude> 5+2*3
11
```

De interpreter berekent de waarde van de ingetikte expressie, waarbij `*` vermenigvuldiging aanduidt.

In de rest van de voorbeelden duiden we de prompt kortheidshalve aan met `?`.

De van de rekenmachine bekende functies kunnen ook gebruikt worden in een expressie:

```
? sqrt(2.0)
1.41421
```

De functie `sqrt` berekent de 'square root', oftewel de vierkantswortel van een getal. Omdat functies in een functionele taal zo veel gebruikt worden, mogen de haakjes worden weggelaten bij de aanroep (gebruik in een expressie) van een functie. In ingewikkelde expressies scheelt dat een heleboel haakjes, en dat maakt zo'n expressie een stuk overzichtelijker. Bovenstaande aanroep van `sqrt` kan dus ook zo geschreven worden:

```
? sqrt 2.0
1.41421
```

In wiskundeboeken is het gebruikelijk dat 'naast elkaar zetten' van expressies betekent dat die expressies vermenigvuldigd moeten worden. Bij aanroep van een functie moeten er dan haakjes worden gezet. In Haskell-expressies komt functie-aanroep echter veel vaker voor dan vermenigvuldigen. Daarom wordt 'naast elkaar zetten' in Haskell geïnterpreteerd als functie-aanroep, en moet vermenigvuldiging expliciet worden genoteerd (met een `*`):

```
? sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
1.0
```

Grote hoeveelheden getallen kunnen in Haskell in een *lijst* worden geplaatst. Lijsten worden genoteerd met behulp van vierkante haken. Er is een aantal standaardfuncties die op lijsten werken:

```
? sum [1..10]
55
```

In bovenstaand voorbeeld is `[1..10]` de Haskell-notatie voor de lijst getallen van 1 tot en met 10. De standaardfunctie `sum` kan op zo'n lijst worden toegepast om de som (55) van de getallen in de lijst te bepalen. Net als bij `sqrt` en `sin` zijn (ronde) haakjes overbodig bij de aanroep van de functie `sum`.

Een lijst is één van de manieren om gegevens samen te voegen, zodat functies op grote hoeveelheden gegevens tegelijk kunnen worden toegepast. Lijsten kunnen ook als resultaat van een functie voorkomen:

```
? reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

De standaardfunctie `reverse` zet de elementen van een lijst in omgekeerde volgorde.

Er zijn diverse standaardfuncties die lijsten manipuleren. De namen spreken vaak voor zich: `length` bepaalt de lengte van een lijst, en `sort` sorteert de elementen van klein naar groot.

```
? length [1,5,9,3]
4
? sort [1,6,2,9,2,7]
[1, 2, 2, 6, 7, 9]
```

In één expressie kunnen meerdere functies gecombineerd worden. Zo is het bijvoorbeeld mogelijk om een lijst eerst te sorteren en vervolgens om te draaien:

```
? reverse (sort [1,6,2,9,2,7])
[9, 7, 6, 2, 2, 1]
```

Zoals in wiskundeboeken ook gebruikelijk is, betekent $g\ (f\ x)$ dat de functie f op x moet worden toegepast, en dat g op het resultaat daarvan moet worden toegepast. De haakjes zijn in dit voorbeeld (zelfs in Haskell!) noodzakelijk, om aan te geven dat $(f\ x)$ in zijn geheel als parameter dient voor de functie g.

1.2.2 Functies definiëren

In een functionele programmeertaal is het mogelijk om zelf nieuwe functies te definiëren. De functies kunnen daarna, samen met de standaardfuncties in de prelude, gebruikt worden in expressies. Definities van een functie worden altijd opgeslagen in een file. Deze file kan worden gemaakt met een tekstverwerker naar keuze.

Het zou omslachtig zijn om voor elke verandering in een functiedefinitie de Haskell-interpreter te verlaten, de tekstverwerker te starten om de functie-definitie te veranderen, de tekstverwerker weer te verlaten, de Haskell-interpreter weer te starten, enzovoort. Daarom is het mogelijk gemaakt om

de tekstverwerker te starten *zonder* de Haskell-interpreter te verlaten; bij het verlaten van de tekstverwerker staat de interpreter dan meteen weer klaar om de nieuwe definitie te verwerken.

De tekstverwerker wordt gestart door '`:edit`' in te tikken, gevolgd door de naam van een file, bijvoorbeeld:

```
? :edit nieuw.hs
```

Door de dubbele punt aan het begin van de regel weet de interpreter dat `edit` geen functie is die uitgerekend moet worden, maar een huishoudelijke mededeling. De file-extensie `.hs` wordt meestal gebruikt om aan te geven dat het om een *Haskell script* gaat. Er wordt nu een tekstverwerker opgestart.

In de file '`nieuw.hs`' kan bijvoorbeeld de definitie worden gezet van de faculteit-functie. De faculteit van een getal n (vaak genoteerd als $n!$) is het product van de getallen van 1 tot en met n, bijvoorbeeld $4! = 1 * 2 * 3 * 4 = 24$. In Haskell kan de definitie van de functie `fac` er als volgt uitzien:

```
fac n = product [1..n]
```

Deze definitie maakt gebruik van de notatie voor 'lijst van getallen tussen twee waarden' en de standaardfunctie `product`.

Als de functie is ingetikt kun je deze bewaren met `File→Save`, en vervolgens de tekstverwerker verlaten met `File→Close`. Daarna is de interpreter weerk aktief. Voordat de nieuwe functie kan worden gebruikt, moet Haskell weten dat de nieuwe file functiedefinities bevat. Dat kan hem meegedeeld worden met de huishoudelijke mededeling `:load`, dus:

```
? :load nieuw.hs
Reading script file "nieuw.hs":
Hugs session for:
/sw/pkg/hugs/share/hugs/lib/Prelude.hs
nieuw.hs
```

Na het analyseren van de nieuwe file meldt Haskell dat naast de `prelude` nu ook de definitie in de file `nieuw` gebruikt kan worden:

```
? fac 6
720
```

Het is mogelijk om later definities aan een file toe te voegen. Het is dan voldoende om alleen `:edit` te tikken; de naam van de file hoeft dan niet meer genoemd te worden.

Een functie die bijvoorbeeld aan de file kan worden toegevoegd is die voor 'n boven k': het aantal manieren waarop k objecten uit een verzameling van n gekozen kunnen worden. Volgens de kansrekeningboeken is dat aantal

$$\binom{n}{k} = \frac{n!}{k!\ (n-k)!}$$

Deze definitie kan, net als die van `fac`, vrijwel letterlijk in Haskell worden opgeschreven:

```
boven n k  =  fac n / (fac k * fac (n-k))
```

De functiedefinities in een file mogen de andere definities aanroepen: `boven` maakt bijvoorbeeld gebruik van de functie `fac`. Daarnaast mogen natuurlijk ook standaardfuncties gebruikt worden.

Na het verlaten van de tekstverwerker wordt de veranderde file automatisch door Haskell bekeken; het is dus niet nodig om opnieuw een `:load`-opdracht te geven. Er kan meteen bepaald worden op hoeveel manieren uit tien mensen een commissie van drie samengesteld kan worden:

```
? boven 10 3
120
```

1.2.3 Opdrachten aan de interpreter

Naast `:edit` en `:load` is er nog een aantal opdrachten die direct voor de interpreter zijn bedoeld (en die dus niet als uit te rekenen expressie worden beschouwd). Al deze opdrachten beginnen met een dubbele punt.

De volgende opdrachten zijn mogelijk:

:? Dit is een opdracht om een lijstje te maken van de andere mogelijke opdrachten. Handig om te weten te komen hoe een opdracht ook al weer heet ('je hoeft niet alles te weten, als je maar weet hoe je het kan vinden').

:quit Met deze opdracht wordt een Haskell-sessie afgesloten.

:load *file(s)* Na deze opdracht kent Haskell de functies die in de gespecificeerde file(s) zijn gedefinieerd. Met `:load` zonder filenamen er achter vergeet Haskell alles behalve de prelude.

:also *file(s)* Met deze opdracht kunnen later nog files worden toegevoegd, zonder de eerder geladen files te vergeten.

:edit *file* Dit is een opdracht om de genoemde file te creëren of veranderen. Als de filenaam wordt weggelaten, wordt de file die het laatst is geladen geëdit, en na het verlaten van de tekstverwerker automatisch opnieuw geladen.

:reload Dit is een afkorting om de laatst geladen file opnieuw te lezen (bijvoorbeeld als die in een ander venster is veranderd).

:find *functienaam* Met deze opdracht wordt de tekstverwerker gestart, precies op de plaats waar de genoemde functie is gedefinieerd.

:type *expressie* Door deze opdracht wordt het type (zie sectie 1.5) van de gegeven expressie bepaald. blz. 14

:set ±*letter* Met deze opdracht kan een aantal opties aan- en uitgeschakeld worden. Normaliter wordt bijvoorbeeld na iedere berekening het aantal reducties en gebruikte cellen weergegeven. Met de opdracht `:set -s` wordt dit achterwege gelaten. Na `:set +s` wordt de melding voortaan weer wel gegeven.

De belangrijkste opties zijn `s`, `t`, `g` en `i`:

- **s**tatistische informatie (aantal reducties en cellen) na elke berekening;
- **t**ypeer elke uitkomst van een berekening (zie paragraaf 1.5.2); blz. 15
- **g***arbage collection* wordt gemeld (zie paragraaf 5.1.4) blz. 90
- **i**nteger-constanten worden speciaal behandeld (zie paragraaf 6.1.4). blz. 115

De overige opties zijn bij normaal gebruik van Haskell niet van belang.

1.3 Standaardfuncties

1.3.1 Ingebouwd/voorgedefinieerd

Behalve functie-definities kunnen in Haskell-programma's ook constanten en operatoren worden gedefinieerd. Een *constante* is een functie zonder parameters. Dit is een constante-definitie:

```
pi = 3.1415926
```

Een *operator* is een functie met twee parameters die *tussen* de parameters wordt geschreven in plaats van er voor. In Haskell is het mogelijk om zelf operatoren te definiëren. De functie `boven` uit paragraaf 1.2.2 had misschien beter als operator gedefinieerd kunnen worden, die bijvoorbeeld als `!^!` genoteerd kan worden: blz. 3

```
n !^! k  =  fac n / (fac k * fac (n-k))
```

In de prelude worden ruim tweehonderd standaardfuncties en -operatoren gedefinieerd. Het grootste deel van de prelude bestaat uit gewone functie-definities, zoals je die ook zelf kunt schrijven. De functie `sum` bijvoorbeeld zit alleen maar in de prelude omdat hij zo vaak gebruikt wordt; als hij er niet in had gezeten, dan had je er zelf een definitie voor kunnen schrijven. De definitie is ook gewoon te bekijken met de opdracht `:find`. Dit is een handige manier om te weten te komen wat een standaardfunctie doet. Voor `sum` luidt de definitie bijvoorbeeld

```
sum = foldl' (+) 0
```

Dan moet je natuurlijk wel weten wat de standaardfunctie `foldl'` doet, maar ook dat is op te zoeken...

Andere functies, zoals de functie `primPlusInt` die twee gehele getallen optelt, kun je niet zelf definiëren; ze zitten op een 'magische' manier ingebouwd in de interpreter. Dit soort functies

wordt *ingebouwde functies* genoemd; hun definitie zit ingebouwd in de interpreter (in het Engels heten ze *primitive functions*). Je kunt ze wel proberen op te zoeken, maar dat maakt je niet veel wijzer:

```
primitive primPlusInt "primPlusInt"
```

Het aantal ingebouwde functies in de prelude is zo klein mogelijk gehouden. De meeste standaardfuncties zijn gewoon in Haskell gedefinieerd. Deze functies worden *voorgedefinieerde* functies genoemd.

1.3.2 Namen van functies en operatoren

In de functiedefinitie

```
fac n  =  product [1..n]
```

is `fac` de naam van een functie die gedefinieerd wordt, en `n` de naam van zijn parameter.

Namen van functies en parameters moeten met een kleine letter beginnen. Daarna mogen nog meer letters volgen (zowel kleine letters als hoofdletters), maar ook cijfers, het apostrof-teken (`'`) en het onderstrepings-teken (`_`). Kleine letters en hoofdletters worden als verschillende letters beschouwd. Een paar voorbeelden van mogelijke functie- of parameter-namen zijn:

```
f    sum   x3   g'   tot_de_macht   langeNaam
```

Het onderstrepings-teken wordt vaak gebruikt om een lange naam gemakkelijk leesbaar te maken. Een andere manier daarvoor is om de woorden die samen één naam vormen (behalve het eerste woord) met een hoofdletter te laten beginnen. Ook in andere programmeertalen wordt dit vaak gedaan.

Cijfers en apostrofs in een naam kunnen gebruikt worden om te benadrukken dat een aantal functies of parameters met elkaar te maken hebben. Dit is echter alleen bedoeld voor de menselijke lezer; voor de interpreter heeft de naam `x3` even weinig met `x2` te maken als `qX'a_y`.

Namen die met een hoofdletter beginnen worden voor speciale functies en constanten gebruikt, de
blz. 57 zogenaamde *constructor-functies*. De definitie daarvan wordt beschreven in paragraaf 3.4.1.

Er zijn 16 namen die niet voor functies of variabelen gebruikt mogen worden. Deze *gereserveerde woorden* hebben een speciale betekenis voor de interpreter. Dit zijn de gereserveerde woorden in Haskell:

```
case      class    data     else
if        in       infix    infixl
infixr    instance let      of
primitive then     type     where
```

De betekenis van de gereserveerde woorden komt later in dit diktaat aan de orde.

Operatoren bestaan uit één of meer symbolen. Een operator kan uit één symbool bestaan (bijvoorbeeld `+`), maar ook uit twee (`&&`) of meer (`!^!`) symbolen. De symbolen waaruit een operator opgebouwd kan worden zijn de volgende:

```
:  #  $  %  &  *  +  -  =  .  /  \  <  >  ?  !  @  ^  |
```

Toegestane operatoren zijn bijvoorbeeld:

```
+   ++   &&   ||   <=   ==   /=   .  $  //
?   @@   -*-  \/   /\   ...  <+>  :->
```

De operatoren op de eerste van deze twee regels worden in de prelude gedefinieerd. Op de tweede regel staan operatoren die zelf gedefinieerd zouden kunnen worden. Operatoren die met een dubbele punt (`:`) beginnen zijn bestemd voor *constructor-operatoren* (net zoals namen die met een hoofdletter beginnen dat zijn voor constructor-functies).

Er zijn elf symbolen of symbolen-combinaties die niet als operator gebruikt mogen worden, omdat ze een speciale betekenis hebben in Haskell. Wel mogen ze deeluitmaken van een langere symbolencombinatie. Het gaat om de volgende combinaties:

```
::   =   ..   --  @   \   |   <-   ->   ~   =>
```

Er blijven er echter genoeg over om je creativiteit op bot te vieren...

1.3.3 Functies op getallen

Er zijn twee soorten getallen beschikbaar in Haskell:

- Gehele getallen, zoals `17`, `0` en `-3`;
- 'Floating-point getallen', zoals `2.5`, `-7.81`, `0.0`, `1.2e3` en `0.5e-2`.

De letter `e` in floating-point getallen betekent 'maal tien-tot-de'. Bijvoorbeeld `1.2e3` is het getal $1.2 \cdot 10^3 = 1200.0$. Het getal `0.5e-2` staat voor $0.5 \cdot 10^{-2} = 0.005$.

De vier rekenkundige operatoren optellen (`+`), aftrekken (`-`), vermenigvuldigen (`*`) en delen (`/`) werken op getallen:

```
? 5-12
-7
? 2.5*3.0
7.5
? 19/4
4.75
```

In de prelude wordt daarnaast een Haskell-definitie gegeven voor een aantal standaardfuncties op getallen. Deze functies zijn dus niet 'ingebouwd', maar slechts 'voorgedefinieerd' en hadden, als ze niet in de prelude zaten, eventueel ook zelf gedefinieerd kunnen worden. Enkele van deze voorgedefinieerde functies zijn:

`abs`	de absolute waarde van een getal
`signum`	-1 voor negatieve getallen, 0 voor nul, 1 voor positieve getallen
`gcd`	de grootste gemene deler van twee getallen
`^`	de 'machtsverheffen'-operator

Een aantal functies die echt ingebouwd zijn, zijn:

`sqrt`	de vierkanstswortel-functie
`sin`	de sinus-functie
`log`	de natuurlijke logaritme
`exp`	de exponentiële functie (e-tot-de-macht)

Er zijn twee ingebouwde functies die van gehele getallen floating-point getallen maken en andersom:

`fromInteger`	maakt een geheel getal tot een floating-point getal
`truncate`	gooit het deel achter de punt weg

Gehele getallen moeten kleiner blijven dan een bepaald maximum. Het hangt van de gebruikte computer af hoe groot dat maximum is. Meestal is dat maximum 2^{31} (ruim 2.1 miljard). Zou het resultaat van een berekening toch groter worden dan dit maximum, dan komt er een onzin-waarde uit, bijvoorbeeld:

```
? 3 * 100000000
300000000
? 3 * 1000000000
-1294967296
```

Ook floating-point getallen kennen een maximum waarde (afhankelijk van de computer 10^{38} of 10^{308}), en een kleinste positieve waarde (10^{-38} of 10^{-308}). Bovendien is de rekennauwkeurigheid beperkt tot 8, respectievelijk 15 significante cijfers:

```
? 123456789.0 - 123456780.0
8.0
```

Het is aan te raden om voor discrete grootheden, zoals aantallen, altijd gehele getallen te gebruiken. Floating-point getallen kunnen worden gebruikt voor continue grootheden zoals afstanden en gewichten.

Je kunt in Haskell ook met gehele getallen groter dan 2 miljard rekenen. Je moet dan echter aangeven dat het type van de getallen `Integer` is, in plaats van het gebruikelijke `Int`. In dat geval kunnen de getallen zo groot worden als maar nodig is:

```
? 123456789 * 123456789 :: Integer
15241578750190521
```

1.3.4 Boolese functies

De operator `<` kijkt of een getal kleiner is dan een ander getal. De uitkomst is de constante `True` (als dat inderdaad zo is) of de constante `False` (als dat niet het geval is):

```
? 1<2
True
? 2<1
False
```

De waarden `True` en `False` zijn de enige elementen van de verzameling *waarheidswaarden* of *Boolean values* (genoemd naar de Engelse wiskundige George Boole). Functies (en operatoren) die zo'n waarde opleveren heten *Boolean functions* of *Boolese functies*.

Behalve `<` is er ook een operator `>` (groter-dan), een operator `<=` (kleiner-of-gelijk), en een operator `>=` (groter-of-gelijk). Daarnaast is er een operator `==` (gelijk-aan) en een operator `/=` (ongelijk-aan). Voorbeelden:

```
? 2+3 > 1+4
False
? sqrt 2.0 <= 1.5
True
? 5 /= 1+4
False
```

Uitkomsten van Boolese functies kunnen gecombineerd worden met de operatoren `&&` ('en') en `||` ('of'). De operator `&&` geeft alleen `True` als resultaat als links èn rechts een ware uitspraak staat:

```
? 1<2 && 3<4
True
? 1<2 && 3>4
False
```

Voor de 'of'-operator hoeft maar één van de twee uitspraken waar te zijn (maar allebei mag ook):

```
? 1==1 || 2==3
True
```

Er is een functie `not` die `True` en `False` omwisselt. Verder is er een functie `even` die kijkt of een geheel getal een even getal is:

```
? not False
True
? not (1<2)
False
? even 7
False
? even 0
True
```

1.3.5 Functies op lijsten

In de prelude wordt een aantal functies en operatoren op lijsten gedefinieerd. Hiervan is er slechts één ingebouwd (de operator `:`), de rest is gedefinieerd met behulp van een Haskell-definitie.

Sommige functies op lijsten zijn al eerder besproken: `length` bepaalt de lengte van een lijst, `sum` de som van een lijst gehele getallen, en `reverse` de elementen van een lijst in omgekeerde volgorde.

De operator `:` zet een extra element op kop van een lijst. De operator `++` plakt twee lijsten aan elkaar. Bijvoorbeeld:

```
? 1 : [2,3,4]
[1, 2, 3, 4]
? [1,2] ++ [3,4,5]
[1, 2, 3, 4, 5]
```

De functie `null` is een Boolese functie op lijsten. Deze functie kijkt of een lijst leeg is (geen elementen bevat). De functie `and` werkt op een lijst waarvan de elementen Boolese waardes zijn; `and` controleert of alle elementen van de lijst `True` zijn:

```
? null [ ]
True
```

```
? and  [ 1<2, 2<3, 1==0]
False
```

Sommige functies hebben twee parameters. De functie `take` krijgt bijvoorbeeld een getal en een lijst als parameter. Als het getal n is, levert deze functie de eerste n elementen van de lijst op:

```
? take 3 [2..10]
[2, 3, 4]
```

1.3.6 Functies op functies

In de functies die tot nu toe besproken zijn zijn de parameters getallen, Boolese waarden of lijsten. De parameter van een functie kan echter zelf ook weer een functie zijn! Een voorbeeld daarvan is de functie `map`, die twee parameters heeft: een functie en een lijst. De functie `map` past de parameter-functie toe op alle elementen van de lijst. Bijvoorbeeld:

```
? map fac [1,2,3,4,5]
[1, 2, 6, 24, 120]
? map sqrt [1,2,3,4]
[1.0, 1.41421, 1.73205, 2.0]
? map even [1..8]
[False, True, False, True, False, True, False, True]
```

Functies met functies als parameter worden veel gebruikt in Haskell (het heet niet voor niets een 'functionele' taal!). In hoofdstuk 2 worden meer van dit soort functies besproken. blz. 21

1.4 Functie-definities

1.4.1 Definitie door combinatie

De eenvoudigste manier om functies te definiëren is door een aantal andere functies, bijvoorbeeld standaardfuncties uit de prelude, te combineren:

```
fac n  =  product [1..n]
oneven x  =  not (even x)
kwadraat x  =  x*x
som_van_kwadraten lijst  =  sum (map kwadraat lijst)
```

Functies kunnen ook meer dan één parameter krijgen:

```
boven n k  =  fac n / (fac k * fac (n-k))
abcFormule a b c  =  [ (-b+sqrt(b*b-4.0*a*c)) / (2.0*a)
                     , (-b-sqrt(b*b-4.0*a*c)) / (2.0*a)
                     ]
```

Functies met nul parameters worden meestal 'constanten' genoemd:

```
pi  =  3.1415926535
e   =  exp 1.0
```

Elke functie-definitie heeft dus de volgende vorm:

- de naam van de functie
- de naam van de eventuele parameters
- een =-teken
- een expressie waar de parameters, standaardfuncties en zelf-gedefinieerde functies in mogen voorkomen.

Bij een functie met als resultaat een Boolese waarde staat rechts van het =-teken een expressie met een Boolese waarde:

```
negatief x  =  x < 0
positief x  =  x > 0
isnul    x  =  x == 0
```

Let in de laatste definitie op het verschil tussen de = en de ==. Een enkel is-teken (=) scheidt in functiedefinities de linkerkant van de rechterkant. Een dubbel is-teken (==) is een operator, net zoals < en >.

In de definitie van de functie `abcFormule` komen de expressies `sqrt(b*b-4.0*a*c)` en `(2.0*a)` twee keer voor. Behalve dat dat veel tikwerk geeft, kost het uitrekenen van zo'n expressie onnodig

veel tijd: de identieke deel-expressies worden tweemaal uitgerekend. Om dat te voorkomen, is het in Haskell mogelijk om deel-expressies een naam te geven. De verbeterde definitie wordt dan als volgt:

```
abcFormule' a b c  =  [ (-b+d)/n
                      , (-b-d)/n
                      ]
                      where d = sqrt (b*b-4.0*a*c)
                            n = 2.0*a
```

Met de statistiek-optie van de interpreter aangeschakeld (door de opdracht `:set +s`) is het verschil goed te zien:

```
? abcFormule' 1.0 1.0 0.0
[0.0, -1.0]
(18 reductions, 66 cells)
? abcFormule 1.0 1.0 0.0
[0.0, -1.0]
(24 reductions, 84 cells)
```

Het woord `where` is niet de naam van een functie: het is één van de 'gereserveerde woorden' die
blz. 6 in paragraaf 1.3.2 opgesomd zijn. Achter '`where`' staan definities. In dit geval definities van de constanten `d` en `n`. Deze constanten mogen in de expressie waarachter `where` staat worden gebruikt. Ze kunnen daarbuiten niet gebruikt worden: het zijn *lokale definities*. Het lijkt misschien vreemd om `d` en `n` 'constanten' te noemen, omdat de waarde bij elke aanroep van `abcFormule'` verschillend kan zijn. Gedurende het berekenen van één aanroep van `abcFormule'`, bij een gegeven `a`, `b` en `c`, zijn ze echter constant.

1.4.2 Definitie door gevalsonderscheid

Soms is het nodig om in de definitie van een functie meerdere gevallen te onderscheiden. De absolute-waarde functie `abs` is hiervan een voorbeeld: voor een negatieve parameter is de definitie anders dan voor een positieve parameter. In Haskell wordt dat als volgt genoteerd:

```
abs x  |  x<0   =  -x
       |  x>=0  =  x
```

Er kunnen ook meer dan twee gevallen onderscheiden worden. Dat gebeurt bijvoorbeeld in de definitie van de functie `signum`:

```
signum x  |  x>0   =  1
          |  x==0  =  0
          |  x<0   =  -1
```

De definities voor de verschillende gevallen worden 'bewaakt' door Boolese expressies, die dan ook *guards* worden genoemd.

Als een functie die op deze manier is gedefinieerd wordt aangeroepen, worden de guards één voor één geprobeerd. Bij de eerste guard die de waarde `True` heeft, wordt de expressie rechts van het `=`-teken uitgerekend. De laatste guard kan dus desgewenst vervangen worden door `True` (of de constante `otherwise`).

De beschrijving van de vorm van een functiedefinitie is dus uitgebreider dan in de vorige paragraaf gesuggereerd werd. Een completere beschrijving van 'functiedefinitie' is:

- de naam van de functie;
- de naam van nul of meer parameters;
- een `=`-teken en een expressie, òf: één of meer 'guarded expressies';
- desgewenst het woord `where` gevolgd door lokale definities.

Daarbij bestaat elke 'guarded expressie' uit een `|`-teken, een Boolese expressie, een `=`-teken, en een expressie[1]. Deze beschrijving is echter ook nog niet volledig...

1.4.3 Definitie door patroonherkenning

De parameters van een functie in een functie-definitie, zoals `x` en `y` in

```
f x y  =  x * y
```

[1] Deze beschrijving lijkt zelf ook wel een definitie, met een lokale definitie voor 'guarded expressie'!

worden de *formele parameters* van die functie genoemd. Bij aanroep wordt de functie voorzien van *actuele parameters*. Bijvoorbeeld, in de aanroep

```
f 17 (1+g 6)
```

is `17` de actuele parameter die overeenkomt met `x`, en `(1+g 6)` de actuele parameter die overeenkomt met `y`. Bij aanroep van een functie worden de actuele parameters ingevuld op de plaats van de formele parameters in de definitie. Het resultaat van de aanroep hierboven is dus `17*(1+g 6)`.

Actuele parameters zijn dus *expressies*. Formele parameters zijn tot nu toe steeds *namen* geweest. In de meeste programmeertalen moet een formele parameter altijd een naam zijn. In Haskell zijn er echter andere mogelijkheden: een formele parameter mag ook een *patroon* zijn.

Een voorbeeld van een functie-definitie, waarin een patroon wordt gebruikt als formele parameter is:

```
f [1,x,y]  =  x+y
```

Deze functie werkt alleen op lijsten met precies drie elementen, waarvan het eerste element 1 moet zijn. Van zo'n lijst worden dan het tweede en derde element opgeteld. De functie is dus niet gedefinieerd op kortere of langere lijsten, of op lijsten waarvan het eerste element niet 1 is. (Het is heel normaal dat functies niet voor alle mogelijke actuele parameters gedefinieerd zijn. Zo is bijvoorbeeld de functie `sqrt` niet gedefinieerd voor negatieve parameters, en de operator `/` niet voor 0 als rechter parameter.)

Je kunt een functie definiëren met verschillende patronen als formele parameter:

```
som []      =  0
som [x]     =  x
som [x,y]   =  x+y
som [x,y,z] =  x+y+z
```

Deze functie kan worden toegepast op lijsten met nul, een, twee of drie elementen (in de volgende paragraaf wordt de functie gedefinieerd op willekeurig lange lijsten). In alle gevallen worden de elementen opgeteld. Bij aanroep van de functie kijkt de interpreter of de parameter 'past' op een van de definities; de aanroep `som [3,4]` past bijvoorbeeld op de derde regel van de definitie. De `3` komt daarbij overeen met de `x` in de definitie en de `4` met de `y`.

De volgende constructies zijn toegestaan als patroon:

- Getallen (bijvoorbeeld `3`);
- De constanten `True` en `False`;
- Namen (bijvoorbeeld `x`);
- Lijsten, waarvan de elementen ook weer patronen zijn (bijvoorbeeld `[1,x,y]`);
- De operator `:` met patronen links en rechts (bijvoorbeeld `a:b`);
- De operator `+` met een patroon links en een natuurlijk getal rechts (bijvoorbeeld `n+1`);
- De operator `*` met een patroon rechts en een natuurlijk getal links (bijvoorbeeld `2*x`).

Met behulp van patronen zijn een aantal belangrijke functies te definiëren. De operator `&&` uit de prelude kan bijvoorbeeld op deze manier gedefinieerd worden:

```
False && False  =  False
False && True   =  False
True  && False  =  False
True  && True   =  True
```

Met de operator `:` kunnen lijsten worden opgebouwd. De expressie `x:y` betekent immers 'zet element `x` op kop van de lijst `y`'. Door de operator `:` in een patroon te zetten, wordt het eerste element van een lijst juist afgesplitst. Daarmee kunnen twee nuttige standaardfuncties geschreven worden:

```
head (x:y)  =  x
tail (x:y)  =  y
```

De functie `head` levert het eerste element van een lijst op (de 'kop'); de functie `tail` levert alles behalve het eerste element op (de 'staart'). Gebruik van deze functies in een expressie kan bijvoorbeeld als volgt:

```
? head [3,4,5]
3
? tail [3,4,5]
[4, 5]
```

De functies `head` en `tail` kunnen op bijna alle lijsten worden toegepast; ze zijn alleen niet gedefinieerd op de lege lijst (een lijst zonder elementen): die heeft immers geen eerste element, laat staan een 'staart'.

In patronen waarin een `+` of een `*` voorkomt, wordt het 'passen' van parameters zó uitgevoerd, dat variabelen altijd een *natuurlijk* getal voorstellen. Er kan bijvoorbeeld een functie `even` gedefinieerd worden, die `True` oplevert als een getal even is:

```
even (2*n)   =  True
even (2*n+1) =  False
```

Bij de aanroep `even 5` 'past' alleen het tweede patroon (waarbij `n` het natuurlijke getal 2 is). Het eerste patroon past niet, want dan zou `n` het niet-natuurlijke getal 2.5 moeten zijn.

1.4.4 Definitie door recursie of inductie

In de definitie van een functie mogen standaardfuncties en zelf-gedefinieerde functies gebruikt worden. Maar ook de functie die gedefinieerd wordt mag in zijn eigen definitie gebruikt worden! Zo'n definitie heet een *recursieve definitie* (recursie betekent letterlijk 'terugkeer': de naam van de functie keert terug in zijn eigen definitie). De volgende functie is een recursieve functie:

```
f x  =  f x
```

De naam van de functie die gedefinieerd wordt (`f`) staat in de definiërende expressie rechts van het =-teken. Deze definitie is echter weinig zinvol; om bijvoorbeeld de waarde van `f 3` te bepalen, moet volgens de definitie eerst de waarde van `f 3` bepaald worden, en daarvoor moet eerst de waarde van `f 3` bepaald worden, enzovoort, enzovoort...

Recursieve functies zijn echter wèl zinvol onder de volgende twee voorwaarden:

- de parameter van de recursieve aanroep is *eenvoudiger* (bijvoorbeeld: numeriek kleiner, of een kortere lijst) dan de parameter van de te definiëren functie;
- voor een *basis-geval* is er een niet-recursieve definitie.

Een recursieve definitie van de faculteit-functie is de volgende:

```
fac n  |  n==0  =  1
       |  n>0   =  n * fac (n-1)
```

Het basisgeval is hier `n==0`; in dit geval kan het resultaat direct (zonder recursie) bepaald worden. In het geval `n>0` is er een recursieve aanroep, namelijk `fac (n-1)`. De parameter bij deze aanroep (`n-1`) is, zoals vereist, kleiner dan `n`.

Een andere manier om deze twee gevallen (het basis-geval en het recursieve geval) te onderscheiden, is gebruik te maken van patronen:

```
fac 0      =  1
fac (n+1)  =  (n+1) * fac n
```

Ook in dit geval is de parameter van de recursieve aanroep (`n`) kleiner dan de parameter van de te definiëren functie (`n+1`).

Het gebruik van patronen sluit nauw aan bij de wiskundige traditie van 'definiëren met inductie'. De wiskundige definitie van machtsverheffen kan bijvoorbeeld vrijwel letterlijk als Haskell-functie worden gebruikt:

```
x ^ 0      =  1
x ^ (n+1)  =  x * x^n
```

Een recursieve definitie waarin voor het gevalsonderscheid patronen worden gebruikt (in plaats van Boolese expressies) wordt daarom ook wel een *inductieve definitie* genoemd.

Functies op lijsten kunnen ook recursief zijn. Daarbij is een lijst 'kleiner' dan een andere als hij minder elementen heeft (korter is). De in de vorige paragraaf beloofde functie `som`, die de getallen in een lijst van willekeurige lengte optelt, kan op verschillende manieren worden gedefinieerd. Een gewone recursieve definitie (waarin het onderscheid tussen het recursieve en het niet-recursieve geval wordt gemaakt met Boolese expressies) luidt als volgt:

```
som lijst  |  lijst==[]  =  0
           |  otherwise  =  head lijst + som (tail lijst)
```

Maar ook hier is een inductieve versie mogelijk (waarin het gevalsonderscheid wordt gemaakt met patronen):

```
som []           =  0
som (kop:staart) =  kop + som staart
```

In de meeste gevallen is een definitie met patronen duidelijker, omdat de verschillende onderdelen in het patroon direct een naam kunnen krijgen (zoals `kop` en `staart` in de functie `som`). In de gewone recursieve versie van `som` zijn de standaardfuncties `head` en `tail` nodig om de onderdelen uit de `lijst` te peuteren. In die functies worden bovendien alsnog patronen gebruikt.

De standaardfunctie `length`, die het aantal elementen in een lijst bepaalt, kan ook inductief worden gedefinieerd:

```
length []           =  0
length (kop:staart) =  1 + length staart
```

Daarbij is de waarde van het `kop`-element niet van belang (alleen het feit dàt er een kop-element is).

In patronen is het toegestaan om in dit soort gevallen het teken '`_`' te gebruiken in plaats van een naam:

```
length []         =  0
length (_:staart) =  1 + length staart
```

1.4.5 Layout en commentaar

Op de meeste plaatsen in een programma mag extra witte ruimte staan, om het programma overzichtelijker te maken. In bovenstaande voorbeelden zijn bijvoorbeeld extra spaties toegevoegd, om de =-tekens van één functie-definitie netjes onder elkaar te zetten. Natuurlijk mogen er geen spaties worden toegevoegd midden in de naam van een functie of in een getal: `len gth` is iets anders dan `length`, en `1 7` iets anders dan `17`.

Ook regelovergangen mogen worden toegevoegd om het resultaat overzichtelijker te maken. In de definitie van `abcFormule` is dat bijvoorbeeld gedaan, omdat de regel anders wel erg lang zou worden.

Anders dan in andere programmeertalen is een regelovergang echter niet helemaal zonder betekenis. Bekijk bijvoorbeeld de volgende twee `where`-constructies:

```
where                    where
      a  = f x y               a   = f x
      b  = g z               y b = g z
```

De plaats van de regelovergang (tussen `x` en `y`, of tussen `y` en `b`) maakt nogal wat uit.

In een rij definities gebruikt Haskell de volgende methode om te bepalen wat bij elkaar hoort:

- een definitie die *precies evenver* is ingesprongen als de vorige, wordt als nieuwe definitie beschouwd;
- is de definitie *verder* ingesprongen, dan hoort hij bij de vorige regel;
- is de definitie *minder ver* ingesprongen, dan hoort hij niet meer bij de huidige lijst definities.

Dat laatste is van belang als een `where`-constructie binnen een andere `where`-constructie voorkomt. Bijvoorbeeld in

```
f x y = g (x+w)
        where g u = u + v
                    where v = u * u
              w   = 2 + y
```

is `w` een lokale declaratie van `f`, en niet van `g`. De definitie van `w` is immers minder ver ingesprongen dan die van `v`; hij hoort dus niet meer bij de `where`-constructie van `g`. Hij is evenver ingesprongen als de definitie van `g`, en hoort dus bij de `where`-constructie van `f`. Zou hij nog minder ver zijn ingesprongen, dan hoorde hij zelfs daar niet meer bij, en krijg je een foutmelding.

Het is allemaal misschien een beetje ingewikkeld, maar in de praktijk gaat alles vanzelf goed als je één ding in het oog houdt:

> *gelijkwaardige definities moeten evenver worden ingesprongen*

Dit betekent ook dat alle globale functiedefinities evenver moeten worden ingesprongen (bijvoorbeeld allemaal nul posities).

Commentaar

Op elke plaats waar spaties mogen staan (bijna overal dus) mag commentaar worden toegevoegd. Commentaar wordt door de interpreter genegeerd, en is bedoeld voor eventuele menselijke lezers van het programma. Er zijn in Haskell twee soorten commentaar:

- met de symbolen `--` begint commentaar dat tot het eind van de regel doorloopt;
- met de symbolen `{-` begint commentaar dat doorloopt tot de symbolen `-}`.

Uitzondering op de eerste regel is het geval dat `--` deel uitmaakt van een operator, bijvoorbeeld
blz. 6 `<-->`. Een losse `--` kan echter geen operator zijn: deze combinatie werd in paragraaf 1.3.2 gereserveerd.

Commentaar met `{-` en `-}` kan worden *genest*, dat wil zeggen weer paren van deze symbolen bevatten. Het commentaar is pas afgelopen bij het bijbehorende sluitsymbool. Bijvoorbeeld in

```
{-  {- hallo -}  f x = 3  -}
```

wordt géén functie `f` gedefinieerd; het geheel is één stuk commentaar.

1.5 Typering

1.5.1 Soorten fouten

Vergissen is menselijk, ook bij het schrijven of intikken van een functie. Gelukkig kan de interpreter waarschuwen voor sommige fouten. Als een functie-definitie niet aan de vorm-eisen voldoet, krijg je daarvan een melding zodra deze functie geanalyseerd wordt. De volgende definitie bevat een fout:

```
isNul x  =  x=0
```

De tweede `=` had een `==` moeten zijn (`=` betekent 'is gedefinieerd als', en `==` betekent 'is gelijk aan'). Bij de analyse van deze functie meldt de interpreter:

```
Reading script file "nieuw.hs":
ERROR "nieuw.hs" (line 18): Syntax error in input (unexpected `=')
```

De vormfouten in een programma (*syntax errors*) worden door de interpreter ontdekt tijdens de eerste fase van de analyse: het ontleden (*to parse*). Andere syntaxfouten zijn bijvoorbeeld openingshaakjes waar geen bijbehorende sluithaakjes bij zijn, of het gebruik van gereserveerde woorden (zoals `where`) op plaatsen waar dat niet mag.

Er zijn behalve syntax-fouten nog andere fouten waar de interpreter voor kan waarschuwen. Een mogelijke fout is het aanroepen van een functie die nergens is gedefinieerd. Vaak zijn dit soort fouten het gevolg van een tik-fout. Bij het analyseren van de definitie

```
fac x  =  produkt [1..x]
```

meldt de interpreter:

```
Reading script file "nieuw.hs":
ERROR "nieuw.hs" (line 19): Undefined variable "produkt"
```

Deze fouten worden pas opgespoord tijdens de tweede fase: de afhankelijkheids-analyse (*dependency analysis*).

Het volgende struikelblok voor een programma is de controle van de types (*type checking*). Functies die bedoeld zijn om op getallen te werken mogen bijvoorbeeld niet op Boolese waarden toegepast worden, en ook niet op lijsten. Functies op lijsten mogen weer niet op getallen worden gebruikt, enzovoort.

Staat er bijvoorbeeld in een functiedefinitie de expressie `1+True` dan meldt de interpreter:

```
Reading script file "nieuw.hs":
ERROR "nieuw.hs" (line 22): Type error in application
*** expression     : 1 + True
*** term           : 1
*** type           : Int
*** does not match : Bool
```

De deel-expressie (*term*) `1` heeft het *type* `Int` (een afkorting van *integer*, oftewel geheel getal). Zo'n integer-waarde kan niet worden opgeteld bij `True`, die van het type `Bool` is (een afkorting van 'Boolese waarde').

Andere typerings-fouten treden bijvoorbeeld op bij het toepassen van de functie `length` op iets anders dan een lijst, zoals in `length 3`:

```
ERROR: Type error in application
*** expression     : length 3
*** term           : 3
*** type           : Int
*** does not match : [a]
```

Pas als er geen typerings-fouten meer in een programma zitten, kan de vierde analyse-fase (genereren van code) worden uitgevoerd. Alleen dan kan de functie worden gebruikt.

Alle foutmeldingen worden al gegeven op het moment dat een functie wordt geanalyseerd. De bedoeling hiervan is dat er tijdens het gebruik van een functie geen onaangename verrassingen meer optreden. Een functie die de analyse doorstaat, bevat gegarandeerd geen typerings-fouten meer.

Sommige andere talen controleren de typering pas op het moment dat een functie wordt aangeroepen. In dat soort talen weet je nooit zeker of er ergens in een ongebruikte uithoek van het programma nog een typerings-fout verborgen ligt...

Het feit dat een functie de analyse doorstaat wil natuurlijk niet zeggen dat de functie correct is. Als in de functie `som` een minteken staat in plaats van een plusteken, dan zal de interpreter daar niet over klagen: hij kan immers niet weten dat het de bedoeling is dat `som` getallen optelt. Dit soort fouten, 'logische fouten' genaamd, zijn het moeilijkst te vinden, omdat de interpreter er niet voor waarschuwt.

1.5.2 Typering van expressies

Het type van een expressie kan bepaald worden met de interpreter-opdracht `:type`. Achter `:type` staat de expressie die getypeerd moet worden. Bijvoorbeeld:

```
?  :type True && False
True && False :: Bool
```

Het symbool `::` kan gelezen worden als 'heeft het type'. De expressie wordt met de `:type`-opdracht niet uitgerekend; alleen het type wordt bepaald.

Er zijn aantal basis-types:

- `Int`: het type van de gehele getallen (*integer numbers* of *integers*), tot een maximum van ruim 2 miljard;
- `Integer`: het type van gehele getallen, zonder praktische begrenzing
- `Float`: het type van de floating-point getallen;
- `Bool`: het type van de Boolese waardes `True` en `False`;
- `Char`: het type van letters, cijfers en symbolen op het toetsenbord (*characters*), dat in paragraaf 3.2.2 zal worden besproken. blz. 46

Let er op dat deze types met een hoofdletter geschreven worden.

Lijsten kunnen verschillende types hebben. Zo zijn er bijvoorbeeld lijsten van integers, lijsten van bools, en zelfs lijsten van lijsten van integers. Al deze lijsten hebben een verschillend type:

```
? :type ['a', 'b', 'c']
['a','b','c'] :: [Char]
? :type [True,False]
[True,False] :: [Bool]
? :type [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

Het type van een lijst wordt aangegeven door het type van de elementen van een lijst tussen vierkante haken te zetten: `[Int]` is het type van een lijst gehele getallen. Alle elementen van een lijst moeten van hetzelfde type zijn. Zo niet, dan verschijnt er een melding van een typerings-fout:

```
? [1,True]
ERROR: Type error in list
*** expression     : [1,True]
*** term           : True
*** type           : Bool
*** does not match : Int
```

Ook functies hebben een type. Het type van een functie wordt bepaald door het type van de parameter en het type van het resultaat. Het type van de functie `sum` is bijvoorbeeld als volgt:

```
? :type sum
sum :: [Int] -> Int
```

De functie `sum` werkt op lijsten integers en heeft als resultaat een enkele integer. Het symbool `->` in het type van de functie moet een pijltje (→) voorstellen. In handschrift kan dit gewoon als pijltje geschreven worden.

Andere voorbeelden van types van functies zijn:

```
sqrt :: Float -> Float
even :: Int   -> Bool
sums :: [Int] -> [Int]
```

Zo'n regel kun je uitspreken als '`even` heeft het type int naar bool' of '`even` is een functie van int naar bool'.

Omdat functies (net als getallen, Boolese waarden en lijsten) een type hebben, is het mogelijk om functies in een lijst op te nemen. De functies die in één lijst staan moeten dan wel precies hetzelfde type hebben, omdat de elementen van een lijst hetzelfde type moeten hebben. Een voorbeeld van een lijst functies is:

```
? :type [sin,cos,tan]
[sin,cos,tan] :: [Float -> Float]
```

De drie functies `sin`, `cos` en `tan` zijn allemaal functies 'van float naar float'; ze kunnen dus in een lijst gezet worden, die dan het type 'lijst van functies van float naar float' heeft.

De interpreter kan zelf het type van een expressie of een functie bepalen. Dit gebeurt dan ook bij het controleren van de typering van een programma. Desondanks is het toegestaan om het type van een functie in een programma erbij te schrijven. Een functiedefinitie ziet er dan bijvoorbeeld als volgt uit:

```
sum        ::  [Int] -> Int
sum []      =  0
sum (x:xs)  =  x + sum xs
```

Hoewel zo'n *type-declaratie* overbodig is, heeft hij twee voordelen:

- er wordt gecontroleerd of de functie inderdaad het type heeft dat je ervoor hebt gedeclareerd;
- de declaratie maakt het voor een menselijke lezer eenvoudiger om een functie te begrijpen.

De typedeclaratie hoeft niet direct voor de definitie te staan. Je zou bijvoorbeeld een programma kunnen beginnen met de declaraties van de types van alle functies die erin worden gedefinieerd. De declaraties dienen dan als een soort inhoudsopgave.

1.5.3 Polymorfie

Voor sommige functies op lijsten maakt het niet uit wat het type van de elementen van die lijst is. De standaardfunctie `length` bijvoorbeeld, kan de lengte bepalen van een lijst integers, maar ook van een lijst boolse waarden, en –waarom niet– van een lijst functies. Het type van de functie `length` wordt als volgt genoteerd:

```
length :: [a] -> Int
```

Dit type geeft aan dat de functie een lijst als parameter heeft, maar het type van de elementen van de lijst doet er niet toe. Het type van deze elementen wordt aangegeven door een *typevariabele*, in het voorbeeld `a`. Typevariabelen worden, in tegenstelling tot de vaste types als `Int` en `Bool`, met een kleine letter geschreven.

De functie `head`, die het eerste element van een lijst oplevert, heeft het volgende type:

```
head :: [a] -> a
```

Ook deze functie werkt op lijsten waarbij het type van de elementen niet belangrijk is. Het resultaat van de functie `head` heeft echter hetzelfde type als de elementen van de lijst (het is immers het eerste element van de lijst). Voor het type van het resultaat wordt dan ook dezelfde type-variabele gebruikt als voor het type van de elementen van de lijst.

Een type waar type-variabelen in voorkomen heet een *polymorf type* (letterlijk: 'veelvormig type'). Functies met een polymorf type heten polymorfe functies. Het verschijnsel zelf heet *polymorfie* of

polymorfisme.

Polymorfe functies, zoals `length` en `head`, hebben met elkaar gemeen dat ze alleen de *structuur* van de lijst gebruiken. Een niet-polymorfe functie, zoals `sum`, gebruikt ook eigenschappen van de *elementen* van de lijst, zoals 'optelbaarheid'.

Polymorfe functies zijn vaak algemeen bruikbaar; in veel programma's moet bijvoorbeeld wel eens de lengte van een lijst bepaald worden. Daarom zijn veel van de standaardfuncties in de prelude polymorfe functies.

Niet alleen functies op lijsten kunnen polymorf zijn. De eenvoudigste polymorfe functie is de identiteits-functie (de functie die zijn parameter onveranderd oplevert):

```
id   ::  a -> a
id x  =  x
```

De functie `id` kan op elementen van willekeurig type werken (en het resultaat is dan van hetzelfde type). Hij kan dus worden toegepast op een integer, bijvoorbeeld `id 3`, maar ook op een Boolese waarde, bijvoorbeeld `id True`. Ook kan de functie werken op lijsten van Booleans, bijvoorbeeld `id [True,False]` of op lijsten van lijsten van integers: `id [[1,2,3],[4,5]]`. De functie kan zelfs worden toegepast op functies van float naar float, bijvoorbeeld `id sqrt`, of op functies van lijsten van integers naar integers: `id sum`. Zoals het type al aangeeft kan de functie worden toegepast op parameters van een willekeurig type. De parameter mag dus ook het type `a->a` hebben, zodat de functie `id` ook op zichzelf kan worden toegepast: `id id`.

1.5.4 Functies met meer parameters

Ook functies met meer dan één parameter hebben een type. In het type staat tussen de parameters onderling, en tussen de laatste parameter en het resultaat, een pijltje. De functie `boven` uit paragraaf 1.2.2 heeft twee integer parameters en een integer resultaat. Het type is daarom: blz. 4

```
boven  ::  Int -> Int -> Int
```

De functie `abcFormule` uit paragraaf 1.4.1 heeft drie floating-point getallen als parameter en een lijst met floating-point getallen als resultaat. De type-declaratie luidt daarom: blz. 9

```
abcFormule  ::  Float -> Float -> Float -> [Float]
```

In paragraaf 1.3.6 werd de functie `map` besproken. Deze functie heeft twee parameters: een functie en een lijst. De functie wordt op alle elementen van de lijst toegepast, zodat het resultaat ook weer een lijst is. Het type van `map` is als volgt: blz. 9

```
map  ::  (a->b) -> [a] -> [b]
```

De eerste parameter van `map` is een functie tussen willekeurige types (`a` en `b`), die niet eens hetzelfde hoeven te zijn. De tweede parameter van `map` is een lijst, waarvan de elementen hetzelfde type (`a`) moeten hebben als de parameter van de functie-parameter (die functie moet er immers op toegepast kunnen worden). Het resultaat van `map` is een lijst, waarvan de elementen hetzelfde type (`b`) hebben als het resultaat van de functie-parameter.

In de type-declaratie van `map` moeten er haakjes staan om het type van de eerste parameter (`a->b`). Anders zou er staan dat `map` drie parameters heeft: een `a`, een `b`, een `[a]` en een `[b]` als resultaat. Dat is natuurlijk niet de bedoeling: `map` heeft twee parameters: een `(a->b)` en een `[a]`.

Ook operatoren hebben een type. Operatoren zijn tenslotte gewoon functies met twee parameters die op een afwijkende manier genoteerd worden (tussen de parameters in plaats van ervoor). Voor het type maakt dat niet uit. Er geldt dus bijvoorbeeld:

```
(&&)  ::  Bool -> Bool -> Bool
```

1.5.5 Overloading

De operator `+` kan gebruikt worden op twee gehele getallen (`Int`) of op twee floating point getallen (`Float`). Het resultaat is weer van datzelfde type. Het type van `+` kan dus zowel `Int->Int->Int` als `Float->Float->Float` zijn. Toch is `+` niet echt een polymorfe operator. Als het type `a->a->a` zou zijn, zou de operator namelijk ook op bijvoorbeeld `Bool` parameters moeten werken, hetgeen niet mogelijk is. Zo'n functie die 'een beetje' polymorf is, wordt een *overloaded* functie genoemd.

Om toch een type te kunnen geven aan een overloaded functie of operator, worden types ingedeeld

in klassen (*classes*). Een klasse is een groep types met een gemeenschappelijk kenmerk. In de prelude worden alvast een paar klassen gedefinieerd:

- `Num` is de klasse van types waarvan de elementen opgeteld, afgetrokken, vermenigvuldigd en gedeeld kunnen worden (numerieke types);
- `Ord` is de klasse van types waarvan de elementen geordend kunnen worden (ordenbare types);
- `Eq` is de klasse van types waarvan de elementen met elkaar vergeleken kunnen worden (equality types).
- `Integral` is de klasse van types waarvan de elementen gehele getallen zijn, dus `Int` en `Integer`

De operator `+` heeft nu het volgende type:

```
(+)  ::  Num a => a->a->a
```

Dit dient gelezen te worden als: '`+` heeft het type `a->a->a` mits `a` een type is in de klasse `Num`'.

Let op het gebruik van het pijltje met dubbele stok (`=>` of desgewenst ⇒). Dit heeft een heel andere betekenis dan een pijltje met enkele stok. Zo'n dubbel pijltje kan maar één keer in een type staan.

Andere voorbeelden van overloaded operatoren zijn:

```
(<)  ::  Ord a  =>  a -> a -> Bool
(==) ::  Eq  a  =>  a -> a -> Bool
```

Zelf-gedefinieerde functies kunnen ook overloaded zijn. Bijvoorbeeld de functie

```
kwadraat x  =  x * x
```

heeft het type

```
kwadraat  ::  Num a  =>  a -> a
```

doordat de operator `*` die erin gebruikt wordt overloaded is.

Ook constanten zijn overloaded. Bijvoorbeeld:

```
? :t 3
3 :: Num a => a
```

Dat betekent dat de constanten `3` overal gebruikt kan worden waar een numeriek type wordt verwacht. Deze overloading strekt zich uit tot lijsten:

```
:t [1,2,3]
[1,2,3] :: Num a => [a]
```

blz. 113 Het gebruik van klassen en de definitie ervan wordt uitgebreid besproken in hoofdstuk 6. Klassen werden hier alleen kort genoemd om de overloaded operatoren te kunnen typeren.

Opgaven

1.1 De taal Haskell is genoemd naar Haskell B. Curry. Wie was dat? (Zoek op op internet).

1.2 Als de onderstaande dingen in een programma staan, zijn ze dan:

- iets met een vaststaande betekenis (gereserveerd woord of symbool);
- naam van een functie of parameter;
- een operator;
- niets van dit alles?

Als het een functie of operator is, betreft het dan een 'constructor'-functie respectievelijk -operator?

```
=>    3a    a3a   ::    :=
:e    X_1   <=>   a'a   _X
***   'a'   A     in    :-<
```

1.3 Hoeveel is:

```
4.0e3 + 2.0e-2
4.0e3 * 2.0e-2
4.0e3 / 2.0e-2
```

1.4 Wat is het verschil in betekenis tussen `x=3` en `x==3` ?

1.5 Schrijf een functie `aantalOpl` die, gegeven a, b en c, het aantal oplossingen van de vergelijking $ax^2 + bx + c$ oplevert, in twee versies:
- met gevalsonderscheid
- door combinatie van standaardfuncties

1.6 Wat is het voordeel van 'genest' commentaar (zie paragraaf 1.4.5)? blz. 14

1.7 Wat is het type van de volgende functies: `tail`, `sqrt`, `pi`, `exp`, `(^)`, `(/=)` en `aantalOpl`? Hoe kan je aan de interpreter vragen om dat type te bepalen, en hoe kun je de types zelf specificeren in een programma?

1.8 Stel dat `x` de waarde 5 heeft. Wat is de waarde van de expressies `x==3` en `x/=3` ? (Voor wie de programmertaal C kent: Wat is de waarde van deze expressies in de taal C?)

1.9 Wat betekent '*syntax error*'? Wat is het verschil tussen een *syntax error* en een *type error*?

1.10 Bepaal de types van `3`, `even` en `even 3`. Hoe doe je dat laatste? Bepaal nu ook het type van `head`, `[1,2,3]` en `head [1,2,3]`. Wat gebeurt er bij het toepassen van een polymorfe functie op een actuele parameter?

1.11 Wat is het type van de volgende expressies (zie appendix E voor de types van de gebruikte standaardfuncties):
- `until even`
- `until or`
- `foldr (&&) True`
- `foldr (&&)`
- `foldr until`
- `map sqrt`
- `map filter`
- `map map`

1.12 Gebruik makend van patronen zou je kunnen proberen om de volgende functie te definiëren om te testen of een getal een priemgetal is:

```
is_priem ((x+2)*(y+2)) = False
is_priem x             = True
```

a. Waarom heeft de programmeur hier `x+2` en `y+2` geschreven, en niet gewoon `x` en `y`?

b. Helaas is deze definitie niet toegestaan. Tegen welke regel is gezondigd?

c. Waarom zou de ontwerper van de taal deze regel hebben ingesteld?

d. Als de regel niet bestond, hoe zou je dan de functie `sqrt` kunnen definiëren?

1.13 Als voorwaarde voor het zinvol-zijn van een recursieve definitie staat in paragraaf 1.4.4 de voorwaarde genoemd dat de parameter bij de recursieve aanroep eenvoudiger moet zijn, en dat er een niet-recursief basis-geval moet zijn. Beschouw nu de volgende definitie van de faculteit-functie: blz. 12

```
fac n  |  n==0       =  1
       |  otherwise  =  n * fac (n-1)
```

a. Wat gebeurt er bij de aanroep `fac (-3)` ?

b. Hoe kan je de voorwaarde waaronder een recursieve definitie zinvol is nauwkeuriger formuleren?

1.14 Wat is het verschil tussen een *lijst* en het wiskundige begrip *verzameling*?

1.15 In paragraaf 1.4.3 wordt de functie `even` gedefinieerd door een aparte definitie te geven voor even en voor oneven parameter. In paragraaf 1.4.4 wordt een recursieve definitie voor machtsverheffen gegeven. blz. 10 blz. 12

a. Geef nu een alternatieve definitie voor machtsverheffen, waarbij je de gevallen dat n even en oneven is apart behandelt. Je kunt daarbij gebruik maken van het feit dat $x^n = (x^{n/2})^2$.

b. Welke tussenresultaten worden er berekend bij het uitrekenen van 2^{10} bij de oude definitie en bij de nieuwe definitie?

1.16 Stel dat is gedefinieerd:

```
driekopie x  =  [x,x,x]
```

Wat is dan de waarde van de expressie

```
map driekopie (sums [1..4])
```

Hoofdstuk 2

Getallen en functies

2.1 Operatoren

2.1.1 Operatoren als functies en andersom

Een operator is een functie met twee parameters die tussen de parameters wordt geschreven in plaats van er voor. Namen van functies bestaan uit letters en cijfers, 'namen' van operatoren uit symbolen (zie paragraaf 1.3.2 voor de precieze regels voor naamgeving). blz. 6

Soms is het gewenst om een operator toch vóór de parameters te schrijven, of een functie er tussen. In Haskell zijn daar twee speciale notaties voor beschikbaar:

- een operator tussen haakjes gedraagt zich als de overeenkomstige functie;
- een functie tussen *back quotes* gedraagt zich als de overeenkomstige operator.

(Een 'back quote' is het symbool `‘`, vooral niet te verwarren met de `’`, de *apostrof*. Op de meeste toetsenborden zit de backquote-toets links van de 1-toets, en de apostrof links van de 'return'-toets.)

Het is dus toegestaan `(+) 1 2` te schrijven in plaats van `1+2`. Deze notatie is in paragraaf 1.5.5 ook gebruikt om het type van `+` te kunnen declareren: blz. 17

```
(+)  ::  Num a => a->a->a
```

Voor de `::` moet namelijk een expressie staan; een losse operator is geen expressie, maar een functie wel.

Andersom is het mogelijk om `1 ‘f‘ 2` te schrijven in plaats van `f 1 2`. Dit wordt vooral gebruikt om een expressie overzichtelijker te maken; de expressie `5 ‘boven‘ 3` leest nu eenmaal makkelijker dan `boven 5 3`. Dit kan natuurlijk alleen als de functie twee parameters heeft.

2.1.2 Prioriteiten

Iedereen heeft de regel 'vermenigvuldigen gaat voor optellen' geleerd, ook wel bekend als 'Meneer Van Dalen'[1]. Je kunt dit deftiger uitdrukken als: 'de prioriteit van vermenigvuldigen is hoger dan die van optellen'. Ook in Haskell zijn deze prioriteiten bekend: de expressie `2*3+4*5` heeft als waarde 26 en niet 50, 46 of 70.

Er zijn in Haskell nog meer prioriteits-nivo's. De vergelijkings-operatoren, zoals `<` en `==`, hebben een lagere prioriteit dan de rekenkundige. Zo heeft `3+4<8` de betekenis die je ervan zou verwachten: 3+4 wordt met 8 vergeleken (resultaat `False`), en niet: 3 wordt opgeteld bij het resultaat van `4<8` (dat zou een typerings-fout opleveren).

In totaal zijn er negen nivo's van prioriteiten. De operatoren in de prelude hebben de volgende prioriteit:

[1] Het zinnetje 'Meneer Van Dalen Wacht Op Antwoord' is een ezelsbruggetje voor deze regel: de beginletters komen overeen met die van Machtsverheffen, Vermenigvuldigen, Delen, Worteltrekken, Optellen en Aftrekken. In dit ezelsbruggetje zit niet verwerkt dat optellen en aftrekken gelijkwaardig zijn. Op mijn school prefereerde men daarom het rijmpje: 'vermenigvuldigen gaat altijd voor / daarna komt altijd delen door / daarna komt altijd min of plus / geen van die twee heeft voorrang dus'.

nivo 9	`.` en `!!`
nivo 8	`^`
nivo 7	`*`, `/`, ‘`div`‘, ‘`rem`‘ en ‘`mod`‘
nivo 6	`+` en `-`
nivo 5	`:`, `++` en `\\`
nivo 4	`==`, `/=`,`<`, `<=`, `>`, `>=`, ‘`elem`‘ en ‘`notElem`‘
nivo 3	`&&`
nivo 2	`||`
nivo 1	(niet gebruikt in de prelude)

(Nog niet al deze operatoren zijn aan de orde geweest; sommige worden in dit of een volgend hoofdstuk besproken.) Vermenigvuldigen en delen hebben dus dezelfde prioriteit: Nederland schijnt alleen te staan in de voorrang van vermenigvuldigen op delen.

Om af te wijken van de geldende prioriteiten kunnen in een expressie haakjes geplaatst worden rond de deel-expressies die eerst uitgerekend moeten worden: in `2*(3+4)*5` wordt wèl eerst `3+4` uitgerekend.

De allerhoogste prioriteit wordt gevormd door het aanroepen van functies (de ‘onzichtbare’ operator tussen `f` en `x` in `f x`). De expressie `kwadraat 3 + 4` berekent dus het kwadraat van 3, en telt daar 4 bij op. Zelfs als je schrijft `kwadraat 3+4` wordt eerst de functie aangeroepen, en dan pas de optelling uitgevoerd. Om het kwadraat van 7 te bepalen zijn haakjes nodig om de hoge prioriteit van functie-aanroep te doorbreken: `kwadraat (3+4)`.

blz. 10

Ook bij het definiëren van functies met gebruik van patronen (zie paragraaf 1.4.3) is het van belang te bedenken dat functie-aanroep altijd voor gaat. In de definitie

```
sum []     =  0
sum (x:xs) =  x + sum xs
```

zijn de haakjes rond `x:xs` essentieel; zonder haakjes zou dit immers opgevat worden als `(sum x):xs`, en dat is geen geldig patroon.

2.1.3 Associatie

Met de prioriteitsregels ligt nog steeds niet vast wat er moet gebeuren met operatoren van gelijke prioriteit. Voor optelling maakt dat niet uit, maar voor bijvoorbeeld aftrekken is dat wel belangrijk: is de uitkomst van `8-5-1` de waarde 2 (eerst 8 min 5, en dan min 1) of 4 (eerst 5 min 1, en dat aftrekken van 8)?

Voor elke operator wordt in Haskell vastgelegd in welke volgorde hij berekend moet worden. Voor een operator, laten we zeggen $\oplus$, zijn er vier mogelijkheden:

- de operator $\oplus$ *associeert naar links*, dat wil zeggen $a \oplus b \oplus c$ wordt uitgerekend als $(a \oplus b) \oplus c$;
- de operator $\oplus$ *associeert naar rechts*, dat wil zeggen $a \oplus b \oplus c$ wordt uitgerekend als $a \oplus (b \oplus c)$;
- de operator $\oplus$ is *associatief*, dat wil zeggen het maakt niet uit in welke volgorde $a \oplus b \oplus c$ wordt uitgerekend;
- de operator $\oplus$ is *non-associatief*, dat wil zeggen dat het verboden is om $a \oplus b \oplus c$ te schrijven; je moet altijd met haakjes aangeven wat de bedoeling is.

Voor de operatoren in de prelude is de keuze overeenkomstig de wiskundige traditie gemaakt. In geval van twijfel zijn de prelude-operatoren non-associatief gemaakt. Voor de associatieve operatoren is toch een keuze gemaakt voor links- of rechts-associatief. (Daarbij is de keuze gevallen op de meest efficiënte volgorde. Je hoeft daar geen rekening mee te houden, want voor het eindresultaat maakt het toch niet uit.)

De volgende operatoren associëren naar **links**:

- de ‘onzichtbare’ operator functie-applicatie, dus `f x y` betekent `(f x) y` (de reden hiervoor wordt besproken in sectie 2.2);

blz. 23

- de operator `!!` (zie paragraaf 3.1.2);

blz. 41

- de operator `-`, dus de waarde van `8-5-1` is 2 (zoals gebruikelijk in de wiskunde) en niet 4.

De volgende operatoren associëren naar **rechts**:

- de operator `^` (machtsverheffen), dus de waarde van `2^2^3` is $2^8 = 256$ (zoals gebruikelijk in de wiskunde) en niet $4^3 = 64$;
- de operator `:` (‘zet op kop van’), zodat de waarde van `1:2:3:x` een lijst is die begint met de waarden 1, 2 en 3.

De volgende operatoren zijn **non**-associatief:

- de operator `/` en de verwante operatoren `div`, `rem` en `mod`. Uit de expressie `64/8/2` komt dus geen 4 en ook geen 16, maar de foutmelding

  ```
  ERROR: Ambiguous use of operator "/" with "/"
  ```

  (*ambiguous* betekent: 'dubbelzinnig', 'voor meerdere uitleg vatbaar');
- de operator `\\` (zie opgave 3.6); blz. 65
- de vergelijkings-operatoren `==`, `<` enzovoort: het heeft meestal toch geen zin om `a==b==c` te schrijven. Wil je testen of `x` tussen 2 en 8 ligt, schrijf dan niet `2<x<8` maar `2<x && x<8`.

De volgende operatoren zijn associatief:

- de operatoren `*` en `+` (deze operatoren worden overeenkomstig wiskundige traditie links-associërend uitgerekend);
- de operatoren `++`, `&&` en `||` (deze operatoren worden rechts-associërend uitgerekend omdat dat efficiënter is);
- de operator `.` (zie paragraaf 2.3.3). blz. 27

2.1.4 Definitie van operatoren

Wie zelf een operator definieert, moet daarbij aangeven wat de prioriteit is, en op welke manier de associatie plaatsvindt. In de prelude staat als volgt gespecificeerd dat `^` prioriteitsnivo 8 heeft en naar rechts associeert:

```
infixr  8  ^
```

Voor operatoren die naar links associëren dient het gereserveerde woord `infixl`, en voor non-associatieve operatoren het woord `infix`:

```
infixl  6  +, -
infix   4  ==, /=, `elem`
```

Door een slimme keuze voor de prioriteit te maken, kunnen haakjes in expressies zo veel mogelijk worden vermeden. Bekijk nog eens de operator 'n boven k' uit paragraaf 1.2.2: blz. 4

```
n `boven` k  =  fac n / (fac k * fac (n-k))
```

of met een zelfbedacht symbool:

```
n !^! k  =  fac n / (fac k * fac (n-k))
```

Omdat je misschien wel eens $\binom{a+b}{c}$ wilt berekenen, is het handig om `` `boven` `` een lagere prioriteit te geven dan `+`; je kunt dan `a+b`boven`c` schrijven zonder haakjes. Aan de andere kant zijn expressies als $\binom{a}{b} < \binom{c}{d}$ gewenst. Door `` `boven` `` een hogere prioriteit te geven dan `<`, zijn ook hierbij geen haakjes nodig.

Voor de prioriteit van `` `boven` `` kan dus het beste 5 gekozen worden (lager dan `+` (6), maar hoger dan `<` (4)). Wat betreft de associatie: omdat het weinig gebruikelijk is om `a`boven`b`boven`c` uit te rekenen, kan de operator het beste non-associatief gemaakt worden. De prioriteits-definitie luidt al met al:

```
infix  5  !^!, `boven`
```

2.2 Currying

2.2.1 Partieel parametriseren

Stel dat `plus` een functie is die twee gehele getallen optelt. In een expressie kan deze functie twee parameters krijgen, bijvoorbeeld `plus 3 5`.

In Haskell mag je ook *minder* parameters aan een functie meegeven. Als `plus` maar één parameter krijgt, bijvoorbeeld `plus 1`, dan houd je een functie over die nog een parameter verwacht. Deze functie kan bijvoorbeeld gebruikt worden om een andere functie te definiëren:

```
opvolger ::  Int -> Int
opvolger  =  plus 1
```

Het aanroepen van een functie met minder parameters dan deze verwacht heet *partieel parametriseren*.

Een tweede toepassing van een partieel geparametriseerde functie is dat deze als parameter kan dienen voor een andere functie. De functie-parameter van de functie `map` (die een functie toepast op alle elementen van een lijst) is bijvoorbeeld vaak een partieel geparametriseerde functie:

```
? map (plus 5) [1,2,3]
[6, 7, 8]
```

De expressie `plus 5` kun je beschouwen als 'de functie die 5 ergens bij optelt'. Deze functie wordt in het voorbeeld door `map` op alle elementen van de lijst `[1,2,3]` toegepast.

De mogelijkheid van partiële parametrisatie werpt een nieuw licht op het type van `plus`. Als `plus 1`, net als `opvolger`, het type `Int->Int` heeft, dan is `plus` zelf blijkbaar een functie van `Int` (het type van `1`) naar dat type:

```
plus  ::  Int -> (Int->Int)
```

Door af te spreken dat `->` naar rechts associeert, zijn de haakjes hierin overbodig:

```
plus  ::  Int -> Int -> Int
```

blz. 17 Dit is precies de notatie voor het type van een functie met twee parameters, die in paragraaf 1.5.4 werd besproken.

Eigenlijk bestaan er helemaal geen 'functies met twee parameters'. Er zijn alleen maar functies met één parameter, die desgewenst een functie op kunnen leveren. Die functie heeft op zijn beurt een parameter, zodat het lijkt alsof de oorspronkelijke functie twee parameters heeft.

Deze truc, het simuleren van functies met meer parameters door een functie met één parameter die een functie oplevert, wordt *Currying* genoemd, naar de Engelse wiskundige Haskell Curry. De functie zelf heet een *gecurryde* functie. (Dit eerbetoon is niet helemaal terecht, want de methode werd eerder gebruikt door M. Schönfinkel).

2.2.2 Haakjes

De 'onzichtbare operator' functie-toepassing associeert naar links. Dat wil zeggen: de expressie `plus 1 2` wordt door de interpreter opgevat als `(plus 1) 2`. Dat klopt precies met het type van `plus`: dit is immers een functie die een integer verwacht (1 in het voorbeeld) en dan een functie oplevert, die op zijn beurt een integer kan verwerken (2 in het voorbeeld).

Associatie van functie-toepassing naar rechts zou onzin zijn: in `plus (1 2)` zou eerst 1 op 2 worden toegepast (??) en vervolgens `plus` op het resultaat.

Staan er in een expressie een hele rij letters op een rij, dan moet de eerste daarvan een functie zijn die de andere achtereenvolgens als parameter opneemt:

```
f a b c d
```

wordt opgevat als

```
((((f a) b) c) d)
```

Als `a` type `A` heeft, `b` type `B` enzovoort, dan is het type van `f`:

```
f  ::  A -> B -> C -> D -> E
```

of, als je alle haakjes zou schrijven:

```
f  ::  A -> (B -> (C -> (D -> E)))
```

Zonder haakjes is dit alles natuurlijk veel overzichtelijker. De associatie van `->` en functie-applicatie is daarom zó gekozen, dat Currying 'geruisloos' verloopt: functie-applicatie associeert naar links, en `->` associeert naar rechts. In een makkelijk te onthouden slagzin:

als er geen haakjes staan,
staan ze zó, dat Currying werkt.

Haakjes zijn alleen nodig, als je hiervan wilt afwijken. Dat gebeurt bijvoorbeeld in de volgende gevallen:

- In het type als een functie een functie als *parameter* krijgt (bij Currying heeft een functie een functie als *resultaat*). Het type van `map` is bijvoorbeeld

  ```
  map  ::  (a->b) -> [a] -> [b]
  ```

  De haakjes in `(a->b)` zijn essentieel, anders zou `map` een functie met drie parameters lijken.

- In een expressie als het *resultaat* van een functie aan een andere functie wordt meegegeven, en niet de functie zelf. Bijvoorbeeld, als je het kwadraat van de sinus van een getal wilt uitrekenen:

```
kwadraat (sin pi)
```

  Zouden hier de haakjes ontbreken, dan lijkt het alsof `kwadraat` eerst op `sin` wordt toegepast (??) en het resultaat daarvan vervolgens op `pi`.

2.2.3 Operator-secties

Voor het partieel parametriseren van operatoren zijn twee speciale notaties beschikbaar:
- met $(\oplus x)$ wordt de operator $\oplus$ partieel geparametriseerd met x als *rechter* parameter;
- met $(x\oplus)$ wordt de operator $\oplus$ partieel geparametriseerd met x als *linker* parameter.

Deze notaties heten *operator-secties*.

Met operator-secties kunnen een aantal functies gedefinieerd worden:

```
opvolger    =  (+1)
verdubbel   =  (2*)
halveer     =  (/2.0)
omgekeerde  =  (1.0/)
kwadraat    =  (^2)
tweeTotDe   =  (2^)
eencijferig =  (<=9)
isNul       =  (==0)
```

De belangrijkste toepassing van operator-secties is echter het meegeven van zo'n partieel geparametriseerde operator aan een functie:

```
? map (2*) [1,2,3]
[2, 4, 6]
```

2.3 Functies als parameter

2.3.1 Functies op lijsten

In een functionele programmeertaal gedragen functies zich in veel opzichten hetzelfde als andere waarden, zoals getallen en lijsten. Bijvoorbeeld:
- functies hebben een *type*;
- functies kunnen door andere functies worden opgeleverd als *resultaat* (waarvan met Currying veel gebruik wordt gemaakt);
- functies kunnen als *parameter* van andere functies worden gebruikt.

Met deze laatste mogelijkheid is het mogelijk om algemeen bruikbare functies te schrijven, waarvan het detail-gedrag wordt bepaald door een functie die als parameter wordt meegegeven.

Functies met functies als parameter worden soms *hogere-orde functies* genoemd, om ze te onderscheiden van 'laag-bij-de-grondse' numerieke functies.

De functie `map` is een voorbeeld van een hogere-orde functie. Deze functie verzorgt het algemene principe 'alle elementen van een lijst langsgaan'. Wat er met de elementen van de lijst moet gebeuren, wordt aangegeven door een functie die, naast de lijst, als parameter aan `map` wordt meegegeven.

De functie `map` kan als volgt worden gedefinieerd:

```
map          ::   (a->b) -> [a] -> [b]
map f []      =   []
map f (x:xs)  =   f x : map f xs
```

De definitie maakt gebruik van patronen: de functie wordt apart gedefinieerd voor het geval de tweede parameter een lijst zonder elementen is, en voor het geval dat de lijst bestaat uit een eerste element `x` en een rest `xs`. De functie is recursief: in het geval van een niet-lege lijst wordt de functie `map` opnieuw aangeroepen. De parameter is daarbij korter (`xs` is korter dan `x:xs`) zodat uiteindelijk het niet-recursieve deel van de functie gebruikt zal kunnen worden.

De functie wordt in de prelude overigens op een andere manier gedefinieerd. Dat merk je als je het type van `map` opvraagt:

```
? :t map
map :: Functor c => (a->b) -> c a -> c b
```

De tweede parameter van `map` is blijkbaar niet een lijst-van-a's, maar een c-van-a's, waarbij c een 'Functor' moet zijn. In dit diktaat zullen we echter geen andere functoren dan lijsten tegenkomen; voor de praktijk is het dus gemakkelijker om j evoor te stellen dat `map` het type

```
map          ::   (a->b) -> [a] -> [b]
```

heeft. Kom je in een foutmelding het woord `Functor` tegen, interpreteer dat dan als 'lijst-achtig'.

Een andere veel gebruikte hogere-orde functie op lijsten is `filter`. Deze functie levert díe elementen uit een lijst[2], die aan een bepaalde eigenschap voldoen. Welke eigenschap dat is, wordt bepaald door een functie die als parameter aan `filter` wordt meegeven. Voorbeelden van het gebruik van `filter` zijn:

```
? filter even [1..10]
[2, 4, 6, 8, 10]
? filter (>10) [2,17,8,12,5]
[17, 12]
```

Als de lijstelementen van type `a` zijn, heeft de functie-parameter van `filter` het type `a->Bool`. Ook de definitie van `filter` is recursief:

```
filter                      ::  (a->Bool) -> [a] -> [a]
filter p []                  =  []
filter p (x:xs) | p x        =  x : filter p xs
                | otherwise  =  filter p xs
```

In het geval dat de lijst niet leeg is (dus de vorm `x:xs` heeft), worden de gevallen onderscheiden dat het eerste element `x` aan de eigenschap `p` voldoet, of niet. Zo ja, dan wordt dit element in ieder geval in het resultaat gezet; de andere elementen worden (met een recursieve aanroep) 'door het filter gehaald'.

Bruikbare hogere-orde functies kun je op het spoor komen door de overeenkomst in functiedefinities op te sporen. Bekijk bijvoorbeeld de definities van de functies `sum` (die de som van een lijst getallen berekent), `product` (die het product van een lijst getallen berekent) en `and` (die kijkt of een lijst Boolese waarden allemaal `True` zijn):

```
sum     []      =  0
sum     (x:xs)  =  x +  sum xs
product []      =  1
product (x:xs)  =  x *  product xs
and     []      =  True
and     (x:xs)  =  x && and xs
```

De structuur van deze drie definities is hetzelfde. Het enige wat verschilt, is de waarde die er bij een lege lijst uitkomt (`0`, `1` of `True`), en de operator die gebruikt wordt om het eerste element te koppelen aan het resultaat van de recursieve aanroep (`+`, `*` of `&&`).

Door deze twee veranderlijken als parameter mee te geven, ontstaat een algemeen bruikbare hogere-orde functie:

```
foldr op e []      =  e
foldr op e (x:xs)  =  x  `op`  foldr op e xs
```

Gegeven deze functie, kunnen de andere drie functies gedefinieerd worden door de algemene functie partieel te parametriseren:

```
sum      =  foldr (+)  0
product  =  foldr (*)  1
and      =  foldr (&&) True
```

De functie `foldr` is in veel meer gevallen bruikbaar; daarom is hij als standaardfunctie in de prelude gedefinieerd.

De naam van `foldr` laat zich als volgt verklaren. De waarde van

[2] Nou ja, eigenlijk een `MonadZero`, wat weer een bijzonder soort `Functor` is, maar we gaan hier niet moeilijk zitten doen...

```
foldr (+) e [w,x,y,z]
```

is gelijk aan de waarde van de expressie

```
(w + (x + (y + (z + e))))
```

De functie `foldr` 'vouwt' de lijst ineen tot één waarde, door tussen alle elementen de gegeven operator te zetten, daarbij beginnend aan de rechterkant met de gegeven startwaarde. (Er is ook een functie `foldl` die aan de linkerkant begint).

Hogere-orde functies, zoals `map` en `foldr`, spelen in functionele talen de rol die controlestructuren (zoals `for` en `while`) in imperatieve talen spelen. Die controlestructuren zijn echter 'ingebouwd', terwijl de functies zelf gedefinieerd kunnen worden. Dit maakt functionele talen flexibel: er is weinig ingebouwd, maar je kunt alles zelf maken.

2.3.2 Iteratie

In de wiskunde wordt vaak *iteratie* gebruikt. Dit houdt in: neem een startwaarde, en pas daarop net zolang een functie toe, tot het resultaat aan een bepaalde eigenschap voldoet.

Iteratie is goed te beschrijven met een hogere-orde functie. In de prelude wordt deze functie `until` genoemd. Het type is:

```
until  ::  (a->Bool) -> (a->a) -> a -> a
```

De functie heeft drie parameters: de eigenschap waar het eindresultaat aan moet voldoen (een functie `a->Bool`), de functie die steeds wordt toegepast (een functie `a->a`), en de startwaarde (van type `a`). Het eindresultaat is ook van type `a`. De aanroep `until p f x` kan gelezen worden als: 'pas net zo lang `f` toe op `x` totdat het resultaat voldoet aan `p`'.

De definitie van `until` is recursief. Het recursieve en het niet-recursieve geval worden ditmaal niet onderscheiden door patronen, maar door gevalsonderscheid met 'verticale streep/Boolese expressie':

```
until p f x  |  p x        =  x
             |  otherwise  =  until p f (f x)
```

Als de startwaarde `x` meteen al aan de eigenschap `p` voldoet, dan is de startwaarde tevens de eindwaarde. Anders wordt de functie `f` éénmaal op `x` toegepast. Het resultaat, `(f x)`, wordt gebruikt als nieuwe startwaarde in de recursieve aanroep van `until`.

Zoals alle hogere-orde functies kan `until` goed aangeroepen worden met partieel geparametriseerde functies. Onderstaande expressie berekent bijvoorbeeld de eerste macht van twee die groter is dan 1000 (begin met 1 en verdubbel net zo lang, tot het resultaat groter is dan 1000):

```
? until (>1000) (2*) 1
1024
```

Anders dan bij eerder besproken recursieve functies, is de parameter van de recursieve aanroep van `until` niet 'kleiner' dan de formele parameter. Daarom levert `until` niet altijd een resultaat op. Bij de aanroep `until (<0) (+1) 1` wordt aan de voorwaarde nooit voldaan; de functie `until` zal dus tot in de eeuwigheid blijven doortellen, en dus nooit met een resultaat komen.

Als de computer steeds maar geen antwoord geeft omdat hij in zo'n oneindige recursie terecht is gekomen, kan de berekening worden afgebroken door tegelijkertijd de 'ctrl'-toets en de C-toets in te drukken:

```
? until (<0) (+1) 1
ctrl-C
{Interrupted!}
?
```

2.3.3 Samenstelling

Als f en g functies zijn, dan is $g \circ f$ de wiskundige notatie voor 'g na f': de functie die eerst f toepast, en daarna g op het resultaat. Ook in Haskell komt de operator die twee functies samenstelt goed van pas. Als er zo'n operator `` `na` `` is, dan is het bijvoorbeeld mogelijk om te definiëren:

```
oneven       =  not   `na` even
dichtbijNul  =  (<10) `na` abs
```

De operator `‘na‘` kan als hogere-orde operator worden gedefinieerd:

```
infixr  8 ‘na‘
g ‘na‘ f  =  h
      where  h x = g (f x)
```

Niet alle functies kunnen zomaar worden samengesteld. Het bereik van f moet hetzelfde zijn als het domein van g. Als f dus een functies $a \to b$ is, kan g een functie $b \to c$ zijn. De samenstelling van de twee functies is een functie die van a direct naar c gaat. Dit komt ook tot uiting in het type van `na`:

```
na  ::  (b->c) -> (a->b) -> (a->c)
```

Omdat `->` naar rechts associeert, is het derde paar haakjes overbodig. Het type van `na` kan dus ook geschreven worden als

```
na  ::  (b->c) -> (a->b) -> a -> c
```

De functie `na` kan dus beschouwd worden als functie met drie parameters; door het Currying-mechanisme is dit immers hetzelfde als een functie met twee parameters die een functie oplevert (en hetzelfde als een functie met één parameter die een functie oplevert met één parameter die een functie oplevert). Inderdaad kan `na` worden gedefinieerd als functie met drie parameters:

```
na g f x  =  g (f x)
```

Het is dus niet nodig om de functie `h` apart een naam te geven met een `where`-constructie (al mag dat natuurlijk wel). In de definitie van `oneven` hierboven wordt `na` dus in feite partieel geparametriseerd met `not` en `even`. De derde parameter is nog niet gegeven: deze wordt pas ingevuld als `oneven` wordt aangeroepen.

Het nut van de operator `na` lijkt misschien beperkt, omdat functies als `oneven` ook gedefinieerd kunnen worden door

```
oneven x  =  not (even x)
```

Een samenstelling van twee functies kan echter als parameter dienen van een andere hogere-orde functie, en dan is het handig dat hij geen naam hoeft te krijgen. De volgende expressie geeft een lijst met de oneven getallen tussen 1 en 100:

```
? filter (not ‘na‘ even) [1..100]
```

In de prelude wordt de functiesamenstellings-operator gedefinieerd. Hij wordt genoteerd als punt (omdat het teken ∘ nu eenmaal niet op het toetsenbord zit). Je kunt dus schrijven:

```
? filter (not.even) [1..100]
```

Deze operator komt vooral goed tot zijn recht als er veel functies samengesteld worden. Het programmeren kan dan geheel op functie-nivo plaatsvinden (zie ook de titel van dit diktaat). Laag-bij-de-grondse dingen als getallen en lijsten zijn uit het gezicht verdwenen. Is het niet veel mooier om `f=g.h.i.j.k` te kunnen schrijven in plaats van `f x=g(h(i(j(k x))))`?

2.3.4 De lambda-notatie

blz. 23

In paragraaf 2.2.1 werd opgemerkt dat de functie die je als parameter meegeeft aan een andere functie vaak ontstaat door partiële parametrisatie, al dan niet met behulp van de operator-sectie notatie:

```
map (plus 5) [1..10]
map (*2) [1..10]
```

In andere gevallen kan de functie die als parameter wordt meegegeven geconstrueerd worden door andere functies samen te stellen:

```
filter (not.even) [1..10]
```

Maar soms is het te ingewikkeld om de functie op die manier te maken, bijvoorbeeld als we $x^2 + 3x + 1$ willen uitrekenen voor alle x in een lijst. Het is dan altijd mogelijk om de functie apart te definiëren in een `where`-clausule:

```
ys  =  map f [1..100]
  where f x = x*x + 3*x + 1
```

Als dit veel voorkomt is het echter een beetje vervelend dat je steeds een naam moet verzinnen voor de functie, en die dan achteraf definiëren.

Voor dit soort situaties is er een speciale notatie beschikbaar, waarmee functies kunnen worden gecreëerd zonder die een naam te geven. Dit is dus vooral van belang als de functie alleen maar nodig is om als parameter meegegeven te worden aan een andere functie. De notatie is als volgt:

\ *patroon* -> *expressie*

Deze notatie staat bekend als de *lambda-notatie* (naar de Griekse letter λ; het symbool \ is de beste benadering voor die letter die op het toetsenbord beschikbaar is...)

Een voorbeeld van de lambda-notatie is de functie `\x -> x*x+3*x+1`. Dit kun je lezen als: 'de functie die bij parameter x de waarde x^2+3x+1 oplevert'. De lambda-notatie wordt veel gebruikt bij het meegeven van functies als parameter aan andere functies, bijvoorbeeld:

```
ys = map (\x->x*x+3*x+1) [1..100]
```

2.4 Numerieke functies

2.4.1 Rekenen met gehele getallen

Bij deling van gehele getallen (`Int`) gaat het gedeelte achter de komma verloren: `10/3` is 3. Toch is het niet nodig bij delingen dan maar altijd `Float` getallen te gebruiken. Integendeel: vaak is de *rest* van de deling interessanter dan de decimale breuk. De rest van een deling is het getal dat op de laatste regel van een staartdeling staat. Bijvoorbeeld in de deling `345/12`

```
1 2 / 3 4 5 \ 2 8
      2 4
      -----
      1 0 5
        9 6
        ---
          9
```

is het quotiënt 28 en de rest 9.

De rest van een deling kan bepaald worden met de standaardfunctie `rem` (*remainder*):

```
? 345 `rem` 12
9
```

De rest van een deling is bijvoorbeeld in de volgende gevallen van nut:

- Rekenen met tijden. Als het nu bijvoorbeeld 9 uur is, dan is het 33 uur later `(9+33)`rem`24` = 20 uur.
- Rekenen met weekdagen. Codeer de dagen als 0=zondag, 1=maandag, ..., 6=zaterdag. Als het nu dag 3 is (woensdag), dan is het over 40 dagen `(3+40)`rem`7` = 1 (maandag).
- Bepalen van deelbaarheid. Een getal is deelbaar door n als de rest bij deling door n gelijk aan nul is.
- Bepalen van losse cijfers. Het laatste cijfer van een getal `x` is `x`rem`10`. Het op één na laatste getal is `(x/10)`rem`10`. Het op twee na laatste `(x/100)`rem`10`, enzovoort.

Als een wat uitgebreider voorbeeld van het rekenen met gehele getallen volgen hier twee toepassingen: het berekenen van een lijst priemgetallen, en het bepalen van de dag van de week gegeven de datum.

Berekenen van een lijst priemgetallen

Een getal is deelbaar door een ander getal als de rest bij deling door dat getal gelijk aan nul is. De functie `deelbaar` test twee getallen op deelbaarheid:

```
deelbaar     ::   Int -> Int -> Bool
deelbaar t n  =   t `rem` n == 0
```

De delers van een getal zijn de getallen waardoor een getal deelbaar is. De functie `delers` bepaalt de lijst delers van een getal:

```
delers   ::   Int -> [Int]
delers x  =   filter (deelbaar x) [1..x]
```

De functie `deelbaar` wordt hierin partieel geparametriseerd met `x`; door de aanroep van `filter` worden díe elementen uit `[1..x]` ge'filter'd, waardoor `x` deelbaar is.

Een getal is een priemgetal als het precies twee delers heeft: 1 en zichzelf. De functie `priem` kijkt of de lijst delers inderdaad uit deze twee elementen bestaat:

```
priem   ::   Int -> Bool
priem x  =   delers x == [1,x]
```

De functie `priemgetallen` tenslotte bepaalt alle priemgetallen tot een gegeven bovengrens:

```
priemgetallen   ::   Int -> [Int]
priemgetallen x  =   filter priem [1..x]
```

Hoewel dit misschien niet de meest efficiënte manier is om priemgetallen te berekenen, is het qua programmeerwerk wel de makkelijkste: de functies zijn een directe vertaling van de wiskundige definities.

Bepalen van de dag van de week

Op welke dag valt het laatste oudjaar van deze eeuw?

```
? dag 31 12 1999
vrijdag
```

Als het nummer van de dag bekend is (volgens de hierboven genoemde codering 0=zondag enz.) dan is de functie `dag` vrij eenvoudig te schrijven:

```
dag d m j  =  weekdag (dagnummer d m j)
weekdag 0  =  "zondag"
weekdag 1  =  "maandag"
weekdag 2  =  "dinsdag"
weekdag 3  =  "woensdag"
weekdag 4  =  "donderdag"
weekdag 5  =  "vrijdag"
weekdag 6  =  "zaterdag"
```

De functie `weekdag` gebruikt zeven patronen om de juiste tekst te selecteren (een woord tussen aanhalingstekens (") is een tekst; voor de details zie paragraaf 3.2.1).

blz. 45

De functie `dagnummer` kiest een zondag in een ver verleden en telt vervolgens op:

- het aantal sindsdien verstreken jaren maal 365;
- een correctie voor de verstreken schrikkeljaren;
- de lengtes van de dit jaar al verstreken maanden;
- het aantal dagen in de lopende maand.

Van het resulterende (grote) getal wordt de rest bij deling door 7 bepaald: dat is het gevraagde dagnummer.

Sinds de kalenderhervorming van paus Gregorius in 1582 (die in het anti-paapse Nederland en Engeland overigens pas in 1752 werd geaccepteerd) geldt de volgende regel voor schrikkeljaren (jaren met 366 dagen):

- een jaartal deelbaar door 4 is een schrikkeljaar (bijv. 1972);
- uitzondering: als het deelbaar is door 100 is het geen schrikkeljaar (bijv. 1900);
- uitzondering op de uitzondering: als het deelbaar is door 400 is het tòch een schrikkeljaar (bijv. 2000).

Als nulpunt van de dagnummers zouden we de dag van de kalenderhervorming kunnen kiezen, maar het is eenvoudiger om terug te extrapoleren tot het fictieve jaar 0. De functie `dagnummer` is dan namelijk simpeler: de 1e januari van het jaar 0 zou op een zondag zijn gevallen.

```
dagnummer d m j  =  ( (j-1)*365
                    + (j-1)/4
                    - (j-1)/100
                    + (j-1)/400
                    + sum (take (m-1) (maanden j))
                    + d
                    ) `rem` 7
```

De aanroep `take n xs` geeft de eerste `n` elementen van de lijst `xs`. De functie `take` kan gedefinieerd worden door:

```
take 0     xs      =  []
take (n+1) (x:xs)  =  x : take n xs
```

De functie `maanden` moet de lengtes van de maanden in een gegeven jaar opleveren:

```
maanden j  =  [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
       where  feb  |  schrikkel j  =  29
                   |  otherwise    =  28
```

De hierin gebruikte functie `schrikkel` wordt gedefinieerd volgens de eerder genoemde regels:

```
schrikkel j  = deelbaar j 4 && (not(deelbaar j 100) || deelbaar j 400)
```

Een andere manier om dit te definiëren is:

```
schrikkel j  |  deelbaar j 100 =  deelbaar j 400
             |  otherwise      =  deelbaar j 4
```

Hiermee zijn de functie `dag` en alle benodigde hulpfuncties voltooid. Het is misschien nog verstandig om in de functie `dag` op te nemen dat hij alleen gebruikt kan worden voor jaartallen na de kalenderhervorming:

```
dag d m j  |  j>1752  =  weekdag (dagnummer d m j)
```

aanroep van `dag` met een kleiner jaartal geeft dan automatisch een foutmelding.

(Einde voorbeelden).

Bij de opzet van de twee programma's in de voorbeelden is een verschillende strategie gevolgd. In het tweede voorbeeld werd met de gevraagde functie `dag` begonnen. Daarvoor waren de hulpfuncties `weekdag` en `dagnummer` nodig. Voor `dagnummer` was een functie `maanden` nodig, en voor `maanden` een functie `schrikkel`. Deze benadering heet *top-down*: beginnen met het belangrijkste, en dan steeds 'lagere' details invullen.

Het eerste voorbeeld gebruikte de *bottom-up* benadering: eerst werd een functie `deelbaar` geschreven, met behulp daarvan een functie `delers`, daarmee een functie `priem`, en tenslotte de gevraagde `priemgetallen`.

Voor het eindresultaat maakt het niet uit (het maakt voor de interpreter niet uit in welke volgorde de functies staan). Bij het programmeren is het echter handig om te bedenken of je een top-down of een bottom-up strategie volgt, of dat deze twee strategieën wellicht afwisselend gebruikt kunnen worden (totdat de 'top' de 'bottom' raakt).

2.4.2 Numeriek differentiëren

Bij het rekenen met `Float` getallen is een exact antwoord meestal niet haalbaar. De uitkomst van een deling wordt bijvoorbeeld afgerond op een bepaald aantal decimalen (afhankelijk van de rekennauwkeurigheid van de computer):

```
? 10.0/6.0
1.6666667
```

Voor de berekening van een aantal wiskundige operaties, zoals `sqrt`, wordt ook een benadering gebruikt. Bij het schrijven van eigen functies die op `Float` getallen werken is het dan ook acceptabel dat het resultaat een benadering is van de 'werkelijke' waarde.

Een voorbeeld hiervan is de berekening van de afgeleide functie. De wiskundige definitie van de afgeleide f' van de functie f is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

De precieze waarde van de limiet kan de computer niet berekenen. Een benadering kan echter worden verkregen door voor h een zeer kleine waarde in te vullen (niet té klein, want dan geeft de deling onacceptabele afrondfouten).

De operatie 'afgeleide' is een hogere-orde functie: er gaat een functie in en er komt een functie uit. De definitie in Haskell kan luiden:

```
diff   ::  (Float->Float) -> (Float->Float)
diff f  =  f'
    where  f' x  =  (f (x+h) - f x) / h
           h     =  0.0001
```

Er zijn andere definities van `diff` mogelijk die een nauwkeurigere benadering geven, maar op deze manier lijkt de definitie van de benaderingsfunctie het meest op de wiskundige definitie.

Door het Currying-mechanisme kan het tweede paar haakjes in het type worden weggelaten, omdat `->` naar rechts associeert.

```
diff  :: (Float->Float) -> Float -> Float
```

De functie `diff` kan dus ook beschouwd worden als functie met twee parameters: de functie waarvan de afgeleide genomen moet worden, en het punt waarin de afgeleide functie berekend moet worden. Vanuit dit gezichtspunt had de definitie kunnen luiden:

```
diff f x  =  (f (x+h) - f x) / h
      where  h = 0.0001
```

De twee definities zijn volkomen equivalent. Voor de duidelijkheid van het programma verdient de tweede versie misschien de voorkeur omdat hij eenvoudiger is (het is niet nodig om de functie `f'` een naam te geven en hem vervolgens te definiëren). Aan de andere kant benadrukt de eerste definitie dat `diff` beschouwd kan worden als functie-transformatie.

De functie `diff` leent zich goed voor partiële parametrisatie, zoals in de definitie:

```
afgeleide_van_sinus_kwadraat  =  diff (kwadraat.sin)
```

De waarde `h` is in beide definities van `diff` in een `where` clausule gezet. Daardoor is hij gemakkelijk te wijzigen, als het programma later nog eens veranderd zou moeten worden (dat kan natuurlijk ook in de expressie zelf, maar dan moet het twee keer gebeuren, met het gevaar dat je er een vergeet).

Nog flexibeler is het, om de waarde van `h` als parameter van `diff` te gebruiken:

```
flexDiff h f x  =  (f (x+h) - f x) / h
```

Door `h` als eerste parameter van `flexDiff` te definiëren, kan deze functie weer partieel geparametriseerd worden om verschillende versies van `diff` te maken:

```
grofDiff  =  flexDiff 0.01
fijnDiff  =  flexDiff 0.0001
superDiff =  flexDiff 0.000001
```

2.4.3 Zelfgemaakte wortel

In Haskell is de functie `sqrt` ingebouwd om de vierkantswortel (*square root*) van een getal uit te rekenen. In deze paragraaf wordt een methode besproken hoe je zelf een wortel-functie kunt maken, als deze niet ingebouwd zou zijn. Het demonstreert een techniek die veel gebruikt wordt bij het rekenen met `Float` getallen. De functie wordt in paragraaf 2.4.5 gegeneraliseerd naar inverses van andere functies dan de kwadraat-functie. Daar wordt ook verklaard waarom de hier geschetste methode werkt.

blz. 34

Voor de vierkantswortel van een getal x geldt de volgende eigenschap:

> als y een goede benadering is voor $\sqrt{x}$
> dan is $\frac{1}{2}(y + \frac{x}{y})$ een betere benadering.

Deze eigenschap kan gebruikt worden om de wortel van een getal x uit te rekenen: neem 1 als eerste benadering, en bereken net zolang betere benaderingen, totdat het resultaat goed genoeg is. De waarde y is goed genoeg als benadering voor $\sqrt{x}$ als y^2 niet te veel meer afwijkt van x.

Voor de waarde van $\sqrt{3}$ zijn de benaderingen y_0, y_1, enz. als volgt:

$$\begin{array}{lcll} y_0 & = & & = & 1 \\ y_1 & = & 0.5 * (y_0 + 3/y_0) & = & 2 \\ y_2 & = & 0.5 * (y_1 + 3/y_1) & = & 1.75 \\ y_3 & = & 0.5 * (y_2 + 3/y_2) & = & 1.732142857 \\ y_4 & = & 0.5 * (y_3 + 3/y_3) & = & 1.732050810 \\ y_5 & = & 0.5 * (y_4 + 3/y_4) & = & 1.732050807 \end{array}$$

Het kwadraat van deze laatste benadering wijkt nog maar 10^{-18} af van 3.

Voor het proces 'een startwaarde verbeteren totdat het goed genoeg is' kan de functie `until` uit paragraaf 2.3.2 gebruikt worden:

blz. 27

```
wortel x  =  until goedGenoeg verbeter 1.0
```

```
        where  verbeter y    =  0.5*(y+x/y)
               goedGenoeg y  =  y*y ~= x
```

De operator `~=` is de 'ongeveer gelijk aan' operator, die als volgt gedefinieerd kan worden:

```
infix 5 ~=
a ~= b   =  a-b<h && b-a<h
    where  h = 0.000001
```

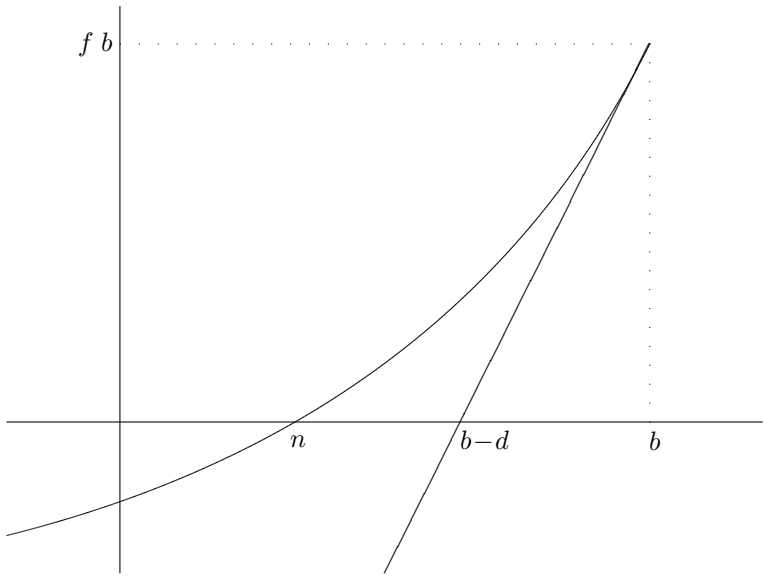
De hogere-orde functie `until` werkt op de *functies* `verbeter` en `goedGenoeg` en op de startwaarde `1.0`. Hoewel `verbeter` naast `1.0` staat, wordt de functie `verbeter` dus niet onmiddellijk op `1.0` toegepast; in plaats daarvan worden beiden aan `until` meegegeven. Door het Currying-mechanisme is het immers alsof de haakjes geplaatst stonden als `(((until goedGenoeg) verbeter) 1.0)`. Pas bij de uitwerking van `until` blijkt dat `verbeter` alsnog op `1.0` wordt toegepast.

Iets anders dat opvalt aan de definitie van `verbeter` en `goedGenoeg` is dat deze functies, behalve van hun parameter `y`, ook gebruik kunnen maken van `x`. Voor deze functies is het dus alsof `x` een constante is. (Vergelijk de definities van de 'constanten' `d` en `n` in de definitie van `abcFormule'` in paragraaf 1.4.1). blz. 9

2.4.4 Nulpunt van een functie

Een ander numeriek probleem dat opgelost kan worden met iteratie door middel van `until`, is het bepalen van het nulpunt van een functie.

Beschouw een functie f waarvan het nulpunt n gezocht wordt. Stel dat b een benadering is voor het nulpunt. Dan is het snijpunt van de raaklijn aan f in b met de x-as een betere benadering voor het nulpunt (zie figuur).



Het gezochte snijpunt ligt op afstand d van de eerste benadering b. De waarde van d kan als volgt bepaald worden. De richtingscoëfficiënt van de raaklijn aan f in b is gelijk aan $f'(b)$. Anderzijds is deze richtingscoëfficiënt gelijk aan $f(b)/d$. Dus $d = f(b)/f'(b)$.

Hiermee is een verbeter-functie gevonden: als b een benadering is voor het nulpunt van f, dan is $b - f(b)/f'(b)$ een betere benadering. Dit staat bekend als de 'methode van Newton'. (De methode werkt niet altijd; bijvoorbeeld voor functies met lokale extremen: je kunt dan 'heen en weer blijven slingeren'. Daar gaan we hier niet verder op in.)

Net als bij `wortel` kan de Newton's verbeter-functie gebruikt worden als parameter van `until`. Als 'goed genoeg'-functie kan ditmaal gecontroleerd worden of de f-waarde in het benaderde nulpunt al klein genoeg is.

```
nulpunt f y0 =  until goedGenoeg verbeter y0
         where  verbeter b   =  b - f b / diff f b
                goedGenoeg b =  f b ~= 0.0
```

De eerste benadering die gebruikt kan worden, is als extra parameter aan de functie meegegeven.
blz. 31 De differentieer-functie uit paragraaf 2.4.2 komt ook goed van pas.

2.4.5 Inverse van een functie

Het nulpunt van de functie f met $f\ x = x^2 - a$ is $\sqrt{a}$. De wortel van a kan dus bepaald worden door het nulpunt van f te zoeken. Nu de functie `nulpunt` beschikbaar is, kan `wortel` dus ook zo geschreven worden:

```
wortel a  =  nulpunt f 1.0
     where  f x  =  x*x-a
```

Ook de derdemachtswortel kan op deze manier berekend worden:

```
derdemachtswortel a  =  nulpunt f 1.0
                  where  f x  =  x*x*x-a
```

In feite kan de inverse van elke gewenste functie bepaald worden, door deze functie te gebruiken in de definitie van f, bijvoorbeeld:

```
arcsin a  =  nulpunt f 0.0
     where  f x  =  sin x - a
arccos a  =  nulpunt f 0.0
     where  f x  =  cos x - a
```

Er begint zich een patroon af te tekenen in al deze definities. Dat is altijd een signaal om een
blz. 26 hogere-orde functie te definiëren, die de generalisatie ervan is (zie paragraaf 2.3.1, waar `foldr` werd gedefinieerd als generalisatie van `sum`, `product` en `and`). De hogere-orde functie is in dit geval `inverse`, die als extra parameter een functie g meekrijgt waarvan de inverse berekend moet worden:

```
inverse g a  =  nulpunt f 0.0
         where  f x  =  g x - a
```

Als je het patroon eenmaal ziet, is zo'n hogere-orde functie niet moeilijker meer dan de andere definities. Die andere definities zijn speciale gevallen van de hogere-orde functie, en kunnen nu ook geschreven worden als partiële parametrisatie:

```
arcsin  =  inverse sin
arccos  =  inverse cos
ln      =  inverse exp
```

De functie `inverse` kan naar believen gebruikt worden als functie met twee parameters (een functie en een `Float`) en `Float` resultaat, of als functie met één parameter (een functie) en een functie als resultaat. Het type van `inverse` is namelijk

```
inverse :: (Float->Float) -> Float -> Float
```

wat ook geschreven kan worden als

```
inverse :: (Float->Float) -> (Float->Float)
```

omdat `->` naar rechts associeert.

blz. 32 De wortel-functie uit paragraaf 2.4.3 maakt in feite ook gebruik van de Newton-methode. Dit blijkt door in de definitie van `wortel` hierboven:

```
wortel a  =  nulpunt f 1.0
     where  f x = x*x-a
```

de aanroep `nulpunt f 1.0` te vervangen door de definitie daarvan:

```
wortel a  =  until goedGenoeg verbeter 1.0
     where  verbeter b   =  b - f b / diff f b
            goedGenoeg b =  f b ~= 0.0
            f x = x*x-a
```

In dit specifieke geval hoef je `diff f` niet numeriek uit te rekenen: de afgeleide van de hier gebruikte `f` is immers de functie `(2*)`. De formule voor `verbeter b` is dus te vereenvoudigen:

$$\begin{aligned} & b - \frac{f\ b}{f'\ b} \\ = \quad & b - \frac{b^2 - a}{2b} \\ = \quad & b - \frac{b^2}{2b} + \frac{a}{2b} \\ = \quad & \frac{b}{2} + \frac{a/b}{2} \\ = \quad & 0.5 * (b + a/b) \end{aligned}$$

Dit is precies de verbeter-formule die in paragraaf 2.4.3 werd gebruikt. blz. 32

Opgaven

2.1 Verklaar de plaatsing van de haakjes in de expressie `(f (x+h) - f x) / h`.

2.2 Welke haakjes zijn overbodig in de volgende expressies?
- `(plus 3) (plus 4 5)`
- `sqrt(3.0) + (sqrt 4.0)`
- `(+) (3) (4)`
- `(2*3)+(4*5)`
- `(2+3)*(4+5)`
- `(a->b)->(c->d)`

2.3 Waarom is het in de wiskunde gebruikelijk om machtsverheffen naar rechts te laten associëren?

2.4 Is de operator `na` (`.`) associatief?

2.5 In de taal Pascal heeft `&&` dezelfde prioriteit als `*`, en `||` dezelfde prioriteit als `+`. Waarom is dat niet handig?

2.6 Geef een voorbeeld van een functie met de volgende types:
- `(Float -> Float) -> Float`
- `Float -> (Float -> Float)`
- `(Float -> Float) -> (Float -> Float)`

2.7 In paragraaf 2.3.3 wordt beweerd dat `na` ook beschouwd kan worden als functie met één parameter. Hoe kun je dat zien aan het type? Geef een definitie van `na` in de vorm `na y =` ... blz. 27

2.8 Schrijf een functie die bepaalt hoeveel jaar een bedrag op de bank moet staan om, bij een gegeven rente, een gegeven eindkapitaal te kunnen incasseren.

2.9 Geef een definitie van `derdemachtswortel` die geen gebruik maakt van numeriek differentiëren (ook niet indirect via `nulpunt`).

2.10 Definiëer de functie `inverse` uit paragraaf 2.4.5 met gebruikmaking van de lambda-notatie. blz. 34

2.11 Waarom kunnen de functies `wortel` en `derdemachtswortel` wel worden geschreven zonder gebruik te maken van de functie `diff`, maar is het niet mogelijk om zo een algemene functie `inverse` te schrijven?

2.12 Schrijf een functie `integraal`, die de integraal van een functie tussen twee grenzen berekent. De functie werkt door het integratiegebied in een (te specificeren) aantal gebiedjes te verdelen, en op elk gebiedje de functie te benaderen door een lineaire functie. In welke volgorde kunnen de parameters het beste worden meegegeven, om de functie handig partieel te kunnen parameteriseren?

Hoofdstuk 3

Datastructuren

3.1 Lijsten

3.1.1 Opbouw van een lijst

Lijsten worden gebruikt om een aantal elementen te groeperen. Die elementen moeten van *hetzelfde type* zijn. Voor elk type is er een type 'lijst-van-dat-type'. Er bestaan dus bijvoorbeeld lijsten-van-integers, lijsten-van-floats, en lijsten-van-functies-van-int-naar-int. Maar ook een aantal lijsten van hetzelfde type kunnen weer in een lijst worden opgenomen; zo ontstaan lijsten-van-lijsten-van-integers, lijsten-van-lijsten-van-lijsten-van-booleans, enzovoort.

Het type van een lijst wordt aangegeven door het type van de elementen tussen vierkante haken. De hierboven genoemde types kunnen dus korter worden aangegeven door `[Int]`, `[Float]`, `[Int->Float]`, `[[Int]]` en `[[[Bool]]]`.

Er zijn verschillende manieren om een lijst te maken: opsomming, opbouw met `:`, en numerieke intervallen.

Opsomming

Opsomming van de elementen is vaak de eenvoudigste manier om een lijst te maken. De elementen moeten van hetzelfde type zijn. Enkele voorbeelden van lijst-opsommingen met hun type zijn:

```
[1, 2, 3]           :: [Int]
[1, 3, 7, 2, 8]     :: [Int]
[True, False, True] :: [Bool]
[sin, cos, tan]     :: [Float->Float]
[ [1,2,3], [1,2] ]  :: [[Int]]
```

Het maakt voor het type van de lijst niet uit hoeveel elementen er zijn. Een lijst met drie integers en een lijst met twee integers hebben allebei het type `[Int]`. Daarom mogen de lijsten `[1,2,3]` en `[1,2]` in het vijfde voorbeeld op hun beurt elementen zijn van één lijst-van-lijsten.

De elementen van de lijst hoeven geen constanten te zijn; ze mogen bepaald worden door een berekening:

```
[ 1+2, 3*x, length [1,2] ] :: [Int]
[ 3<4, a==5, p && q ]      :: [Bool]
[ diff sin, inverse cos ]  :: [Float->Float]
```

De gebruikte functies moeten dan wel als resultaat het gewenste type hebben.

Het aantal elementen van een lijst is vrij. Een lijst kan dus ook bestaan uit maar één element:

```
[True]    :: [Bool]
[[1,2,3]] :: [[Int]]
```

Een lijst met één element wordt ook wel een *singleton-lijst* genoemd. De lijst `[[1,2,3]]` is ook een singleton-lijst: het is immers een lijst van lijsten, die één element (de lijst `[1,2,3]`) heeft.

Let op het verschil tussen een *expressie* en een *type*. Als er tussen de vierkante haken een type staat, is er sprake van een type (bijvoorbeeld `[Bool]` of `[[Int]]`). Als er tussen de vierkante haken een expressie staat, is het geheel ook een expressie (een singleton-lijst, bijvoorbeeld `[True]` of `[3]`).

Het aantal elementen van een lijst kan ook nul zijn. Een lijst met nul elementen heet de *lege lijst*. De lege lijst heeft een polymorf type: het is een 'lijst-van-maakt-niet-uit-wat'. Op plaatsen

in een polymorf type waar een willekeurig type ingevuld mag worden, staan type-variabelen (zie
blz. 16 paragraaf 1.5.3), dus het type van de lege lijst is `[a]`:

```
[]  ::  [a]
```

De lege lijst mag op elke plaats in een expressie gebruikt worden waar een lijst nodig is. Het type wordt daarbij door de context bepaald:

```
sum []                [] is een lege lijst getallen
and []                [] is een lege lijst Booleans
[ [], [1,2], [3] ]    [] is een lege lijst getallen
[ [1<2,True], [] ]    [] is een lege lijst Booleans
[ [[1]], [] ]         [] is een lege lijst lijsten-van-getallen
length []             [] is een lege lijst (doet er niet toe waarvan)
```

Opbouw met :

Een andere manier om een lijst te maken is het gebruik van de operator `:`. Deze operator zet een element op kop van een lijst, en maakt zo een langere lijst.

```
(:)  ::  a -> [a] -> [a]
```

Als bijvoorbeeld `xs` de lijst `[3,4,5]` is, dan is `1:xs` de lijst `[1,3,4,5]`. Gebruik makend van de lege lijst en de `:`-operator is elke lijst te construeren. Zo is bijvoorbeeld `1:(2:(3:[]))` de lijst `[1,2,3]`. De `:`-operator associeert naar rechts, dus je kunt kortweg `1:2:3:[]` schrijven.

In feite is deze manier van opbouw de enige 'echte' manier om een lijst te maken. Een opsomming van een lijst is vaak overzichtelijker in programma's, maar heeft precies dezelfde betekenis als de overeenkomstige expressie met `:`-operatoren. Daarom kost een opsomming ook tijd:

```
? [1,2,3]
[1, 2, 3]
(7 reductions, 29 cells)
```

Elke aanroep van `:` (die je niet ziet, maar die er dus wel staat) kost 2 reducties.

Numerieke intervallen

Een derde manier om een lijst te maken is de interval-notatie: twee numerieke expressies met twee punten ertussen en vierkante haken eromheen:

```
? [1..5]
[1, 2, 3, 4, 5]
? [2.5 .. 6.0]
[2.5, 3.5, 4.5, 5.5]
```

(Hoewel de punt gebruikt mag worden als symbool in operatoren, is `..` geen operator. Het is
blz. 6 namelijk één van symbolen-combinaties die in paragraaf 1.3.2 werden gereserveerd voor speciaal gebruik.)

De waarde van de expressie `[x..y]` wordt berekend door `enumFromTo x y` aan te roepen. De functie `enumFromTo` is als volgt gedefinieerd:

```
enumFromTo x y  |  y<x       =  []
                |  otherwise =  x : enumFromTo (x+1) y
```

Als `y` kleiner is dan `x` is de lijst dus leeg; anders is `x` het eerste element, is het volgende element één groter (tenzij `y` is gepasseerd), enzovoort.

De notatie voor numerieke intervallen is niet meer dan een aardigheidje die het gebruik van de taal iets makkelijker maakt; het zou geen groot gemis zijn als deze constructie niet mogelijk was, want dan kan altijd nog de functie `enumFromTo` gebruikt worden.

3.1.2 Functies op lijsten

Functies op lijsten worden vaak gedefinieerd door gebruik te maken van *patronen*: de functie wordt apart gedefinieerd voor de lege lijst, en voor een lijst die de vorm `x:xs` heeft. Elke lijst is immers òf leeg, òf heeft een eerste element `x`, dat op kop staat van een (mogelijk lege) lijst `xs`.

Een aantal definities van functies op lijsten zijn al ter sprake gekomen: `head` en `tail` in para-
blz. 10 graaf 1.4.3, `som` en `length` in paragraaf 1.4.4, en `map`, `filter` en `foldr` in paragraaf 2.3.1. Hoewel
blz. 12
blz. 26

dit allemaal standaardfuncties zijn die in de prelude worden gedefinieerd, en ze dus niet zelf gedefinieerd hoeven te worden, is het toch belangrijk om hun definitie te bekijken. Ten eerste omdat het goede voorbeelden zijn van definities van functies op lijsten, ten tweede omdat de definitie vaak de duidelijkste beschrijving geeft wat een standaardfunctie doet.

In deze paragraaf volgen nog meer definities van functies op lijsten. Veel van deze functies zijn recursief, dat wil zeggen dat ze, voor het patroon `x:xs`, zichzelf aanroepen met de (kleinere) parameter `xs`.

Lijsten vergelijken en ordenen

Twee lijsten zijn gelijk als ze precies dezelfde elementen hebben, die in dezelfde volgorde staan. Dit is een definitie van de functie `eq` waarmee de gelijkheid van lijsten getest kan worden:

```
[]     `eq` []      =  True
[]     `eq` (y:ys)  =  False
(x:xs) `eq` []      =  False
(x:xs) `eq` (y:ys)  =  x==y && xs `eq` ys
```

In deze definitie kan zowel de eerste als de tweede parameter leeg of niet-leeg zijn; voor alle vier de combinaties is er een definitie. In het vierde geval worden de overeenkomstige elementen met elkaar vergeleken (`x==y`), en wordt de operator recursief aangeroepen op de rest-lijsten (`xs`eq`ys`).

Omdat in deze functie de overloaded operator `==` op de lijst-elementen wordt gebruikt, is ook `eq` een overloaded functie. Het type is:

```
eq  ::  Eq a => [a] -> [a] -> Bool
```

De functie `eq` wordt niet in de prelude gedefinieerd. In plaats daarvan is er voor gezorgd dat lijsten lid zijn van de klasse `Eq`, zodat de operator `==` ook op lijsten gebruikt mag worden. Daarbij geldt wel weer als voorwaarde dat het type van de lijst-elementen ook in de klasse `Eq` zit. Lijsten van functies zijn dus niet vergelijkbaar, omdat functies dat niet zijn. Lijsten van lijsten van integers zijn echter wel vergelijkbaar, omdat lijsten van integers vergelijkbaar zijn (omdat integers vergelijkbaar zijn). In paragraaf 6.2.3 wordt besproken hoe de operator `==` in de prelude voor lijsten gedefinieerd wordt. Die definitie verloopt geheel analoog aan die van de functie `eq` hierboven. blz. 118

Als de elementen van een lijst geordend kunnen worden met $<$, $\leq$ enz., dan kunnen ook lijsten geordend worden. Dit gebeurt volgens de *lexicografische ordening* ('woordenboek-volgorde'): het eerste element van de lijsten is bepalend, tenzij het eerste element van beide lijsten gelijk is; in dat geval beslist het tweede element, tenzij dat ook gelijk is, enzovoort. Er geldt dus bijvoorbeeld `[2,3]<[3,1]` en `[2,1]<[2,2]`. Als een van de twee lijsten een beginstuk is van de ander, dan is de kortste het 'kleinste', bijvoorbeeld `[2,3]<[2,3,4]`. Dat in deze beschrijving het woord 'enzovoort' nodig is, is een aanwijzing dat er recursie nodig is in de definitie van de functie `kg` (kleiner dan of gelijk aan):

```
kg                ::  Ord a => [a] -> [a] -> Bool
[]     `kg` ys     =  True
(x:xs) `kg` []     =  False
(x:xs) `kg` (y:ys) =  x<y || (x==y && xs`kg`ys)
```

Nu de functies `eq` en `kg` gedefinieerd zijn, kunnen andere vergelijkings-functies eenvoudig gedefinieerd worden: `ng` (niet gelijk), `gg` (groter dan of gelijk aan), `kd` (kleiner dan) en `gd` (groter dan):

```
xs `ng` ys  =  not (xs `eq` ys)
xs `gg` ys  =  ys `kg` xs
xs `kd` ys  =  xs `kg` ys && xs `ng` ys
xs `gd` ys  =  ys `kd` xs
```

Deze functies hadden natuurlijk ook direct met recursie gedefinieerd kunnen worden. In de prelude worden deze functies overigens niet gedefinieerd; in plaats daarvan wordt er voor gezorgd dat de bekende vergelijkings-operatoren (`<=` enz.) ook op lijsten gebruikt kunnen worden.

Lijsten samenvoegen

Twee lijsten van hetzelfde type kunnen worden samengevoegd tot één lijst met de operator `++`. Deze operatie wordt ook wel *concatenatie* ('samen-ketening') genoemd. Bijvoorbeeld: `[1,2,3]++[4,5]` geeft de lijst `[1,2,3,4,5]`. Concatenatie met de lege lijst (zowel aan de voorkant als aan de

achterkant) laat een lijst onveranderd: `[1,2]++[]` geeft `[1,2]`.

De operator `++` is een standaardfunctie, maar hij kan gewoon in Haskell gedefinieerd worden (dat gebeurt ook in de prelude). Het is dus geen 'ingebouwde' operator zoals `:`. De definitie luidt:

```
(++)         ::  [a] -> [a] -> [a]
[]     ++ ys  =  ys
(x:xs) ++ ys  =  x : (xs++ys)
```

In de definitie wordt de linker parameter onderworpen aan patroon-analyse. In het niet-lege geval wordt de operator recursief aangeroepen met de kortere lijst `xs` als parameter.

Er is nog een functie die lijsten samenvoegt. Deze functie, `concat`, werkt op een *lijst* van lijsten. Alle lijsten in de lijst van lijsten worden samengevoegd tot één lange lijst. Dus bijvoorbeeld

```
? concat [ [1,2,3], [4,5], [], [6] ]
[1, 2, 3, 4, 5, 6]
```

De definitie van `concat` is als volgt:

```
concat          ::  [[a]] -> [a]
concat []        =  []
concat (xs:xss)  =  xs ++ concat xss
```

Het eerste patroon, `[]` is de lege lijst; een lege lijst lijsten wel te verstaan. Het resultaat is dan een lege lijst: een lijst zonder elementen. In het tweede geval van de definitie is de lijst lijsten niet leeg. Er staan dus één lijst, `xs`, op kop, en er is een rest-lijst-van-lijsten `xss`. Eerst worden alle lijsten in de rest samengevoegd door recursieve aanroep van `concat`; tenslotte wordt de eerste lijst `xs` daar ook nog voor gezet.

Let op het verschil tussen `++` en `concat`: de operator `++` werkt op *twee* lijsten, de functie `concat` werkt op *een lijst* van lijsten. Beide worden in de wandeling 'concatenatie' genoemd. (Vergelijk de situatie met de operator `&&`, die kijkt of twee Booleans `True`, en de functie `and` die kijkt of een hele lijst van Booleans allemaal `True` zijn).

Delen van een lijst selecteren

In de prelude worden een aantal functies gedefinieerd die delen van een lijst selecteren. Bij sommige functies is het resultaat een (kortere) lijst, bij andere is het één element.

Aangezien een lijst wordt opgebouwd uit een kop en een staart, is het eenvoudig om de kop en staart van een lijst weer terug te krijgen:

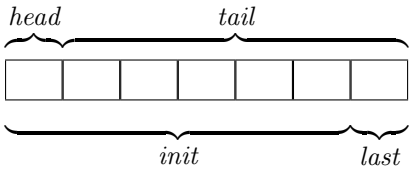
```
head        ::   [a] -> a
head (x:xs)  =   x
tail        ::   [a] -> [a]
tail (x:xs)  =   xs
```

Deze functies doen patroon-analyse op de parameter, maar er is geen aparte definitie voor het patroon `[]`. Als deze functies worden aangeroepen op een lege lijst volgt er dan ook een foutmelding.

Minder eenvoudig is het om een functie te schrijven die het *laatste* element uit een lijst selecteert. Daarvoor is recursie nodig:

```
last        ::  [a] -> a
last (x:[])  =  x
last (x:xs)  =  last xs
```

Ook deze functie is niet gedefinieerd voor de lege lijst, omdat die met geen van de twee patronen overeenkomt. Zoals er bij `head` een functie `tail` hoort, zo hoort er bij `last` een functie `init`. Een schematisch overzicht van deze vier functies:

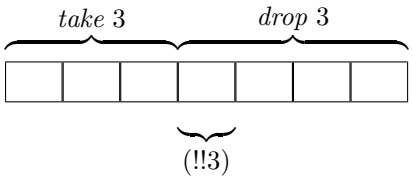


De functie `init` selecteert alles *behalve* het laatste element. Ook daarvoor is weer recursie nodig:

```
init        ::  [a] -> [a]
init (x:[])  =  []
init (x:xs)  =  x: init xs
```

Het patroon `x:[]` kan worden (en wordt meestal) geschreven als `[x]`.

In paragraaf 2.4.1 is een functie `take` ter sprake gekomen. Behalve een lijst heeft `take` een integer als parameter, die aangeeft hoeveel elementen van de lijst in het resultaat zitten. De tegenhanger van `take` is `drop`, die juist een bepaald aantal elementen van het begin van de lijst verwijdert. Tenslotte is er een operator `!!`, die één gespecificeerd element uit de lijst selecteert. Schematisch: blz. 30



Deze functies zijn als volgt gedefinieerd:

```
take, drop :: Int -> [a] -> [a]
take 0     xs      =  []
take n     []      =  []
take (n+1) (x:xs)  =  x : take n xs
drop 0     xs      =  xs
drop n     []      =  []
drop (n+1) (x:xs)  =  drop n xs
```

Als de lijst te kort is, dan worden zo veel mogelijk elementen genomen, respectievelijk weggelaten. Dat komt door de tweede regel in de definities: die zegt dat als je een lege lijst in de functies stopt, het resultaat altijd de lege lijst is, wat het gespecificeerde aantal ook is. Als deze regel niet in de definitie had gestaan, dan waren `take` en `drop` ongedefinieerd voor te korte lijsten.

De operator `!!` selecteert één element uit een lijst. De kop van de lijst telt daarbij als 'nulde' element, dus `xs!!3` geeft het *vierde* element van de lijst `xs`. Deze operator mag niet op te korte lijsten worden toegepast; er valt in dat geval immers geen zinvolle waarde op te leveren. De definitie luidt:

```
infixl 9 !!
(!!)            ::  [a] -> Int -> a
(x:xs) !! 0      =  x
(x:xs) !! (n+1)  =  xs !! n
```

Deze operator kost, vooral voor grote getallen, de nodige tijd: de hele lijst wordt er voor vanaf het begin doorlopen. Hij moet dus enigszins spaarzaam worden toegepast. De operator is geschikt om één element uit een lijst te selecteren. De functie `weekdag` uit paragraaf 2.4.1 had bijvoorbeeld zo gedefinieerd kunnen worden: blz. 30

```
weekdag d  =  [ "zondag", "maandag", "dinsdag", "woensdag",
                "donderdag", "vrijdag", "zaterdag" ] !! d
```

Moeten echter alle elementen van een lijst achtereenvolgens geselecteerd worden, dan is het beter om `map` of `foldr` te gebruiken.

Lijsten omdraaien

De functie `reverse` in de prelude zet de elementen van een lijst in omgekeerde volgorde. De functie kan eenvoudig recursief worden gedefinieerd. Een omgekeerde lege lijst blijft een lege lijst. Voor een niet-lege lijst moet het staartstuk omgekeerd worden, en het eerste element helemaal aan het eind daarvan geplaatst worden. De definitie kan dus als volgt luiden:

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

Eigenschappen van lijsten

Een belangrijke eigenschap van een lijst is zijn lengte. De lengte kan berekend worden met de functie `length`. Deze functie is in de prelude als volgt gedefinieerd:

```
length        ::  [a] -> Int
length []      =  0
length (x:xs)  =  1 + length xs
```

In de prelude zit verder een functie `elem` die test of een bepaald element in een lijst aanwezig is. De functie `elem` kan als volgt worden gedefinieerd:

```
elem      ::  a -> [a] -> Bool
elem e xs  =  or (map (==e) xs)
```

De functie vergelijkt alle elementen van `xs` met `e` (partiële parametrisering van de operator `==`). Dat levert een lijst Booleans op, waarvan `or` controleert of er minstens één `True` is. De functie kan, met gebruik van de functie-compositie-operator, ook zo geschreven worden:

```
elem e  =  or . (map (==e))
```

De functie `notElem` controleert of een element juist níet in een lijst zit:

```
notElem e xs  =  not (elem e xs)
```

Deze functie kan ook gedefinieerd worden met

```
notElem e  =  and . (map (/=e))
```

3.1.3 Hogere-orde functies op lijsten

Functies kunnen flexibeler gemaakt worden door ze een functie als parameter mee te geven. Veel standaardfuncties op lijsten hebben een functie als parameter. Het zijn daardoor hogere-orde functies.

map, filter en foldr

Eerder werden al de functies `map`, `filter` en `foldr` besproken. Deze functies doen, afhankelijk van hun functie-parameter, iets met alle elementen van een lijst. De functie `map` past zijn functie-parameter toe op alle elementen van de lijst:

```
              xs  =  [  1  ,  2  ,  3  ,  4  ,  5  ]
                     ↓     ↓     ↓     ↓     ↓
map kwadraat xs  =  [  1  ,  4  ,  9  ,  16  ,  25  ]
```

De functie `filter` gooit de elementen uit een lijst die niet aan een bepaalde Boolean functie voldoen:

```
              xs  =  [  1  ,  2  ,  3  ,  4  ,  5  ]
                     ×     ↓     ×     ↓     ×
filter even xs  =  [        2  ,        4        ]
```

De functie `foldr` zet een operator tussen alle elementen van een lijst, te beginnen aan de rechterkant met een gespecificeerde waarde:

```
              xs  =  [  1  ,  2  ,  3  ,  4  ,  5  ]
                     ↓     ↓     ↓     ↓     ↓
foldr (+) 0 xs  =     (1  +  (2  +  (3  +  (4  +  (5  + 0 )))))
```

Deze drie standaardfuncties worden in de prelude recursief gedefinieerd. Ze werden eerder bespro-
blz. 26
ken in paragraaf 2.3.1.

```
map               ::  (a->b) -> [a] -> [b]
map f []           =  []
map f (x:xs)       =  f x : map f xs
filter            ::  (a->Bool) -> [a] -> [a]
filter p []        =  []
filter p (x:xs)
       | p x       =  x : filter p xs
       | otherwise =  filter p xs
foldr             ::  (a->b->b) -> b -> [a] -> b
foldr op e []      =  e
foldr op e (x:xs)  =  x `op` foldr op e xs
```

Door veel van deze standaardfuncties gebruik te maken, kan de recursie in andere functies verborgen worden. Het 'vuile werk' wordt dan door de standaardfuncties opgeknapt, en de andere functies zien er overzichtelijker uit. De functie `or`, die kijkt of in een lijst Booleans minstens één waarde `True` is, is bijvoorbeeld zo gedefinieerd:

```
or  =  foldr (||) False
```

Maar het is ook mogelijk om deze functie direct met recursie te definiëren, zonder gebruik te maken van `foldr`:

```
or []      =  False
or (x:xs)  =  x || or xs
```

Veel functies kunnen geschreven worden als combinatie van een aanroep van `foldr` en een aanroep van `map`. De functie `elem` uit de vorige paragraaf is daar een voorbeeld van:

```
elem e   =  foldr (||) False  .  map (==e)
```

Maar ook deze functie kan natuurlijk direct, zonder gebruik te maken van standaardfuncties, gedefinieerd worden. Recursie is dan weer noodzakelijk:

```
elem e []      =  False
elem e (x:xs)  =  x==e || elem e xs
```

takewhile en dropwhile

Een variant op de functie `filter` is de functie `takeWhile`. Deze functie heeft, net als `filter`, een eigenschap (functie met Boolean resultaat) en een lijst als parameter. Het verschil is dat `filter` altijd alle elementen van de lijst bekijkt. De functie `takeWhile` begint aan het begin van de lijst, en stopt met zoeken zodra er één element niet meer aan de eigenschap voldoet. Bijvoorbeeld: `takeWhile even [2,4,6,7,8,9]` geeft `[2,4,6]`. Anders dan bij `filter` komt de 8 niet in het resultaat, want de 7 doet `takeWhile` stoppen met zoeken. De definitie in de prelude luidt:

```
takeWhile          ::  (a->Bool) -> [a] -> [a]
takeWhile p []      =  []
takeWhile p (x:xs)
       | p x        =  x : takeWhile p xs
       | otherwise  =  []
```

Vergelijk deze definitie met die van `filter`.

Zoals er bij `take` een functie `drop` hoort, zo hoort er bij `takeWhile` een functie `dropWhile`. Deze laat het beginstuk van een lijst vervallen dat aan een eigenschap voldoet. Bijvoorbeeld: `dropWhile even [2,4,6,7,8,9]` is `[7,8,9]`. De definitie luidt:

```
dropWhile          ::  (a->Bool) -> [a] -> [a]
dropWhile p []      =  []
dropWhile p (x:xs)
       | p x        =  dropWhile p xs
       | otherwise  =  x:xs
```

foldl

De functie `foldr` zet een operator tussen alle elementen van een lijst, en begint daarbij aan de rechterkant van de lijst. De functie `foldl` doet hetzelfde, maar begint aan de linkerkant. Net als `foldr` heeft `foldl` een extra parameter die aangeeft wat het resultaat is voor de lege lijst.

Een voorbeeld van de werking van `foldl` op een lijst met vijf elementen is het volgende:

```
              xs  =          [  1  ,  2  ,  3  ,  4  ,  5  ]
                             ↓     ↓     ↓     ↓     ↓
foldl (+) 0 xs  =  (((((0  +  1) +  2) +  3) +  4) +  5)
```

Om een definitie van deze functie te kunnen geven, is het handig om eerst twee voorbeelden onder elkaar te zetten:

```
foldl (⊕) a [x,y,z]  =  ((a ⊕ x) ⊕ y) ⊕ z
foldl (⊕) b [  y,z]  =  (  b  ⊕ y) ⊕ z
```

Hieruit blijkt dat aanroep van `foldl` op de lijst `x:xs` (met `xs`=$[y, z]$ in het voorbeeld) gelijk is aan `foldl xs`, mits in de recursieve aanroep als startwaarde in plaats van a de waarde $a \oplus$ `x` genomen wordt. Met deze observatie kan de definitie geschreven worden:

```
foldl op e []      =  e
foldl op e (x:xs)  =  foldl op (e`op`x) xs
```

Voor associatieve operatoren zoals `+` maakt het niet zo veel uit of je `foldr` of `foldl` gebruikt. Voor niet-associatieve operatoren zoals `-` is het resultaat van `foldl` natuurlijk anders dan dat van `foldr`.

3.1.4 Lijsten sorteren

Alle tot nu toe genoemde functies op lijsten zijn vrij eenvoudig: door middel van recursie wordt de lijst éénmaal doorlopen om het resultaat te bepalen.

Een functie die niet op deze manier geschreven kan worden, is het sorteren (in opklimmende volgorde zetten van de elementen) van een lijst. Daarvoor moeten de elementen immers helemaal door elkaar gegooid worden.

Toch is het, zeker met hulp van de standaardfuncties, niet moeilijk om een sorteer-functie te schrijven. Er zijn verschillende mogelijkheden om het sorteer-probleem aan te pakken. Deftiger gezegd: er zijn verschillende *algoritmen* mogelijk. Twee algoritmen zullen hier worden besproken. In beide algoritmen is het noodzakelijk dat de elementen van de lijst geordend kunnen worden. Het is dus mogelijk om een lijst integers of een lijst van lijsten van integers te sorteren, maar niet een lijst van functies. Dit wordt uitgedrukt door het type van de sorteerfunctie:

```
sorteer  ::  Ord a  =>  [a] -> [a]
```

dat wil zeggen: `sorteer` werkt tussen lijsten van willekeurig type `a`, mits het type `a` in de klasse van ordenbare types `Ord` zit.

Sorteren door invoegen

Stel dat een gesorteerde lijst gegeven is. Dan kan een nieuw element op de juiste plaats in deze lijst worden ingevoegd met de volgende functie:

```
insert             ::  Ord a  =>  a -> [a] -> [a]
insert e []         =  [e]
insert e (x:xs)
      |  e<=x       =  e : x : xs
      |  otherwise  =  x : insert e xs
```

Als de lijst leeg is, dan wordt het nieuwe element `e` het enige element. Als de lijst niet leeg is, en element `x` op kop heeft staan, dan hangt het ervan af of `e` kleiner is dan `x`. Zo ja, dan komt `e` helemaal op kop te staan; zo nee, dan komt `x` op kop te staan, en moet `e` elders in de lijst worden ingevoegd. Een voorbeeld van het gebruik van `insert`:

```
? insert 5 [2,4,6,8,10]
[2, 4, 5, 6, 8, 10]
```

Voor de werking van `insert` is het essentieel dat de parameter-lijst gesorteerd is; het resultaat is dan ook gesorteerd.

De functie `insert` kan gebruikt worden om een nog niet gesorteerde lijst te sorteren. Stel dat `[a,b,c,d]` gesorteerd moet worden. Je kunt dan een lege lijst nemen (die is gesorteerd) en daar het laatste element `d` in invoegen. Het resultaat is een gesorteerde lijst, waarin `c` ingevoegd kan worden. Het resultaat blijft gesorteerd, ook nadat `b` erin is ingevoegd. Tenslotte kan `a` op de juiste plaats worden ingevoegd, en het eindresultaat is een gesorteerde versie van `[a,b,c,d]`. De expressie die berekend wordt is:

```
a `insert` (b `insert` (c `insert` (d `insert` [])))
```

De structuur van deze berekening is precies die van `foldr`, met `insert` als operator en `[]` als startwaarde. Een mogelijk sorteer-algoritme luidt dus:

```
isort  =  foldr insert []
```

met de functie `insert` zoals hierboven gedefinieerd. Dit algoritme wordt *insertion sort* genoemd.

Sorteren door samenvoegen

Een ander sorteer-algoritme maakt gebruik van de mogelijkheid om twee gesorteerde lijsten samen te voegen tot één. Daartoe dient de functie `merge`:

```
merge               ::  Ord a => [a] -> [a] -> [a]
merge []     ys      =  ys
merge xs     []      =  xs
merge (x:xs) (y:ys)
       |  x <= y     =  x : merge xs (y:ys)
       |  otherwise  =  y : merge (x:xs) ys
```

Als één van beide lijsten leeg is, dan is de andere lijst het resultaat. Als beide lijsten niet-leeg zijn, dan komt de kleinste van de twee kop-elementen op kop van het resultaat, en worden de

overblijvende elementen samengevoegd door een recursieve aanroep van `merge`.

Net als `insert` gaat `merge` ervan uit dat de parameters gesorteerd zijn. In dat geval zorgt hij er voor dat ook het resultaat een gesorteerde lijst is.

Ook op de functie `merge` kan een sorteer-algoritme worden gebaseerd. Dit algoritme maakt er gebruik van dat de lege lijst en singleton-lijsten (lijsten met één element) altijd gesorteerd zijn. Langere lijsten kunnen (ongeveer) in tweeën worden gesplitst. De helften kunnen worden gesorteerd door een recursieve aanroep van het sorteer-algoritme. De twee gesorteerde resultaten kunnen tenslotte worden samengevoegd door `merge`.

```
msort           ::  Ord a => [a] -> [a]
msort xs
   |  lengte<=1  =  xs
   |  otherwise  =  merge (msort ys) (msort zs)
   where ys     =  take half xs
         zs     =  drop half xs
         half   =  lengte / 2
         lengte =  length xs
```

Dit algoritme wordt *merge sort* genoemd. In de prelude worden de functies `insert` en `merge` gedefinieerd, en een functie `sort` die werkt zoals `isort`.

3.2 Speciale lijsten

3.2.1 Strings

In een voorbeeld in paragraaf 2.4.1 werd gebruik gemaakt van teksten als waarde, bijvoorbeeld blz. 30
`"maandag"`. Een tekst die als waarde in een programma wordt gebruikt heet een *string*. Een string is een lijst, waarvan de elementen lettertekens zijn.

Alle functies die op lijsten werken, kunnen dus ook op strings gebruikt worden. Bijvoorbeeld de expressie `"zon"++"dag"` geeft de string `"zondag"`, en het resultaat van de expressie `tail (take 3 "haskell")` is de string `"as"`.

Strings worden genoteerd tussen aanhalingstekens. De aanhalingstekens geven aan dat een tekst letterlijk genomen moet worden als waarde van een string, en niet als naam van een functie. dus `"until"` is een string die uit vijf tekens bestaat, maar `until` is de naam van een functie. Om een string moeten daarom altijd aanhalingstekens geschreven worden. Ze worden alleen weggelaten door de interpreter in het eindresultaat van een opdracht:

```
? "zon"++"dag"
zondag
```

De elementen van een string zijn van het type `Char`. Dat is een afkorting van het woord *character*. Mogelijke characters zijn niet alleen de lettertekens, maar ook de cijfer-symbolen en de leestekens. Het type `Char` is één van de vier basis-types van Haskell (de andere drie zijn `Int`, `Float` en `Bool`).

Waarden van het type `Char` kunnen worden aangegeven door een letterteken tussen *enkele aanhalingstekens* oftewel *apostrofs* te zetten, bijvoorbeeld `'B'` of `'*'`. Let op het verschil met omgekeerde aanhalingstekens (back quotes), die worden gebruikt om van een functie een operator te maken. Onderstaande drie expressies hebben zeer verschillende betekenissen:

`"f"`	een lijst characters (string) die uit één element bestaat;
`'f'`	een character;
`` `f` ``	de functie `f` als operator beschouwd.

De notatie met dubbele aanhalingstekens voor strings is niets anders dan een afkorting voor een opsomming van een lijst characters. De string `"hallo"` betekent hetzelfde als de lijst `['h','a','l','l','o']` of anders gezegd `'h':'a':'l':'l':'o':[]`.

Voorbeelden waaruit blijkt dat een string inderdaad een lijst characters is, zijn de expressie `hd "aap"` die het character `'a'` oplevert, en de expressie `takeWhile (=='e') "eender"` die de string `"ee"` oplevert.

3.2.2 Characters

De waarde van een `Char` kunnen lettertekens, cijfertekens en leestekens zijn. Het is belangrijk om de aanhalingstekens rond een character neer te zetten, omdat deze tekens in Haskell normaal iets anders betekenen:

expressie	type	betekenis
`'x'`	`Char`	het letterteken `'x'`
`x`	`...`	de naam van bijv. een parameter
`'3'`	`Char`	het cijferteken `'3'`
`3`	`Int`	het getal 3
`'.'`	`Char`	het leesteken punt
`.`	`(b->c)->(a->b)->a->c`	de functie-samenstellings operator

Er zijn 128 (op sommige computers 256) mogelijke waardes voor het type `Char`:

- 52 lettertekens
- 10 cijfertekens
- 32 leestekens en de spatie
- 33 speciale tekens
- (128 extra tekens: letters met accenten, meer leestekens enz.)

Er is één leesteken dat in een string problemen geeft: het aanhalingsteken. Bij een aanhalingsteken in een string zou de string immers afgelopen zijn. Als er toch een aanhalingsteken in een string nodig is, moet daar het symbool `\` (een omgekeerde deelstreep of *backslash*) vóór gezet worden. Bijvoorbeeld:

```
"Hij zei \"hallo\" en liep door"
```

Deze oplossing geeft een nieuw probleem, want nu kan het leesteken `\` zèlf weer niet in een string staan. Als dit teken in een string moet komen, moet het daarom verdubbeld worden:

```
"het teken \\ heet backslash"
```

Zo'n dubbel symbool telt als één character. Dus de lengte van de string `"\"\"\\\""` is 4. Ook mogen deze symbolen tussen enkele aanhalingstekens staan, zoals in onderstaande definities:

```
dubbelepunt     = ':'
aanhalingsteken = '\"'
backslash       = '\\'
apostrof        = '\''
```

De 33 speciale characters worden gebruikt om de lay-out van een tekst te beïnvloeden. De belangrijkste speciale characters zijn de 'newline' en de 'tabulatie'. Ook deze characters kunnen worden weergegeven met behulp van een backslash: `'\n'` is het newline-character, en `'\t'` is het tabulatie-character. Het newline-character kan gebruikt worden om een resultaat van meer dan één regel te maken:

```
? "EEN\nTWEE\nDRIE"
EEN
TWEE
DRIE
```

Alle characters zijn genummerd volgens een door de Internationale Standaarden Organisatie (ISO) bepaalde codering[1]. Er zijn twee (ingebouwde) standaardfuncties, die de code van een character bepalen, respectievelijk het character met een bepaalde code opleveren:

```
ord  ::  Char -> Int
chr  ::  Int  -> Char
```

Bijvoorbeeld:

```
? ord 'A'
65
? chr 51
'3'
```

Een overzicht van alle characters met hun ISO/ASCII-codenummers staat in appendix C. De characters zijn geordend volgens deze codering. Het type `Char` maakt daarmee deel uit van de klasse

[1] Deze codering wordt meestal de ASCII-codering genoemd (American Standard Code for Information Interchange). Tegenwoordig is de codering internationaal erkend, en moet dus eigenlijk ISO-codering worden genoemd.

`Ord`. De ordening komt, wat de letters betreft, overeen met de alfabetische ordening, met dien verstande dat alle hoofdletters vóór de kleine letters komen. Deze ordening werkt ook door in strings; strings zijn immers lijsten, en die zijn lexicografisch geordend gebaseerd op de ordening van hun elementen:

```
? sort ["aap", "noot", "Mies", "Wim"]
["Mies", "Wim", "aap", "noot"]
```

3.2.3 Functies op characters en strings

In de prelude worden een aantal functies gedefinieerd op characters, waarmee bepaald kan worden wat voor soort teken een gegeven character is:

```
isSpace, isUpper, isLower, isAlpha, isDigit, isAlphanum :: Char->Bool
isSpace c     =  c == ' '   ||  c == '\t'  || c == '\n'
isUpper c     =  c >= 'A'   &&  c <= 'Z'
isLower c     =  c >= 'a'   &&  c <= 'z'
isAlpha c     =  isUpper c  ||  isLower c
isDigit c     =  c >= '0'   &&  c <= '9'
isAlphanum c  =  isAlpha c  ||  isDigit c
```

Deze functies kunnen goed gebruikt worden om in de definitie van een functie op characters de verschillende gevallen te onderscheiden.

In de ISO-codering is de code van het cijferteken `'3'` niet 3, maar 51. De cijfers liggen in de codering gelukkig wel opeenvolgend. Om de numerieke waarde van een cijferteken te bepalen moet dus niet alleen de functie `ord` worden toegepast, maar ook 48 van het resultaat worden afgetrokken. Dat doet de functie `digitValue`:

```
digitValue   ::  Char -> Int
digitValue c  =  ord c - ord '0'
```

Deze functie kan eventueel voor 'onbevoegd' gebruik worden beveiligd door te eisen dat de parameter inderdaad een digit is:

```
digitValue c  |  isDigit c  =  ord c - ord '0'
```

De omgekeerde operatie wordt uitgevoerd door de functie `digitChar`: deze functie maakt van een integer (die tussen 0 en 9 moet liggen) het bijbehorende cijferteken:

```
digitChar   ::  Int -> Char
digitChar n  =  chr ( n + ord '0')
```

Deze twee functies worden in de prelude helaas niet gedefinieerd (maar als ze nodig zijn kun je ze natuurlijk altijd zelf even definiëren).

In de prelude zitten wel twee functies om kleine letters naar hoofdletters om te rekenen en andersom:

```
toUpper, toLower :: Char->Char
toUpper c | isLower c  = chr (ord c - ord 'a' + ord 'A')
          | otherwise  = c
toLower c | isUpper c  = chr (ord c - ord 'A' + ord 'a')
          | otherwise  = c
```

Met behulp van `map` kunnen deze functies op alle elementen van een string worden toegepast:

```
? map toUpper "Hallo!"
HALLO!
? map toLower "Hallo!"
hallo!
```

Alle polymorfe functies die op lijsten zijn gedefinieerd zijn ook te gebruiken op strings. Daarnaast zijn er in de prelude een paar functies gedefinieerd die specifiek op strings werken:

```
words,   lines   ::   [Char]  -> [[Char]]
unwords, unlines ::  [[Char]] ->  [Char]
```

De functie `words` splitst een string op in een aantal kleine strings, die ieder één woord van de invoerstring bevatten. De woorden worden gescheiden door spaties (zoals in `isSpace`). De functie `lines` doet hetzelfde, maar dan met de afzonderlijke regels, die in de invoerstring gescheiden zijn door newline-characters (`'\n'`). Voorbeelden:

```
? words "dit is een string"
```

```
["dit", "is", "een", "string"]
? lines "eerste regel\ntweede regel"
["eerste regel", "tweede regel"]
```

De functies `unwords` en `unlines` doen het omgekeerde: ze smeden een lijst woorden, respectievelijk regels, aaneen tot één lange string:

```
? unwords ["dit", "zijn", "de", "woorden"]
dit zijn de woorden
? unlines ["eerste regel", "tweede regel"]
eerste regel
tweede regel
```

Merk hierbij op dat in het resultaat geen aanhalingstekens staan: deze worden altijd weggelaten als het resultaat van een expressie een string is.

Een variant op de functie `unlines` is de functie `layn`. Deze nummert de regels in het resultaat:

```
? layn ["eerste regel", "tweede regel"]
   1) eerste regel
   2) tweede regel
```

De precieze definitie van deze functies is op te zoeken in de prelude; het belangrijkste voor dit moment is ze te kunnen gebruiken in expressies, om een overzichtelijk resultaat te krijgen.

3.2.4 Oneindige lijsten

Het aantal elementen in een lijst kan oneindig groot zijn. De hier volgende functie `vanaf` levert een oneindig lange lijst op:

```
vanaf n  =  n : vanaf (n+1)
```

Natuurlijk kan een computer niet echt een oneindig aantal elementen bevatten. Gelukkig krijg je het beginstuk van de lijst al te zien terwijl de rest van de lijst nog wordt opgebouwd:

```
? vanaf 5
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, control-C
{Interrupted!}
?
```

Op het moment dat je genoeg elementen hebt gezien, kun je de berekening stoppen door op control-C te drukken.

Een oneindige lijst kan ook gebruikt worden als tussenresultaat, terwijl het eindresultaat toch eindig is. Dit is bijvoorbeeld het geval bij het probleem: ‘bepaal alle machten van drie die kleiner zijn dan 1000’. De eerste tien machten van drie zijn te bepalen met de volgende aanroep:

```
? map (3^) [0..9]
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683]
```

De elementen die kleiner zijn dan 1000 kunnen met de functie `takeWhile` hieruit genomen worden:

```
? takeWhile (<1000) (map (3^) [0..9])
[1, 3, 9, 27, 81, 243, 729]
```

Maar hoe weet je van tevoren dat 10 elementen genoeg is? De oplossing is om in plaats van `[0..9]` de oneindige lijst `vanaf 0` te gebruiken, om daarmee *alle* machten van drie te berekenen. Dat is zeker genoeg...

```
? takeWhile (<1000) (map (3^) (vanaf 0))
[1, 3, 9, 27, 81, 243, 729]
```

Deze methode kan worden toegepast dankzij het feit dat de interpreter nogal lui van aard is: werk wordt altijd zo lang mogelijk uitgesteld. Daarom wordt het resultaat van `map (3^) (vanaf 0)` niet in zijn geheel uitgerekend (dat zou oneindig lang duren). In plaats daarvan wordt eerst het eerste element berekend. Dat wordt doorgespeeld aan de buitenwereld, in dit geval de functie `takeWhile`. Pas als dit element verwerkt is, en `takeWhile` om een volgend element vraagt, wordt het tweede element uitgerekend. Vroeg of laat zal `takeWhile` echter niet meer om nieuwe elementen vragen (nadat het eerste getal ≥ 1000 is gepasseerd). Verdere elementen worden door `map` dan ook niet meer uitgerekend.

3.2.5 Lazy evaluatie

De evaluatiemethode (manier waarop expressies worden uitgerekend) van Haskell wordt *lazy evaluatie* ('luie berekening') genoemd. Bij lazy evaluatie wordt een (deel-)expressie pas uitgerekend als zeker is dat de waarde ècht nodig is voor het resultaat. Het tegenovergestelde van lazy evaluatie is *eager evaluatie* ('gretige berekening'). Bij eager evaluatie wordt, zodra de actuele parameter bekend is, het functieresultaat meteen berekend.

Het kunnen gebruiken van oneindige lijsten is te danken aan de lazy evaluatie. In talen waarin eager evaluatie gebruikt wordt (zoals alle imperatieve talen, en een aantal oudere functionele talen) zijn oneindige lijsten niet mogelijk.

Lazy evaluatie heeft nog meer voordelen. Bekijk bijvoorbeeld de functie `priem` uit paragraaf 2.4.1, blz. 29 die kijkt of een getal een priemgetal is:

```
priem   ::   Int -> Bool
priem x  =   delers x == [1,x]
```

Zou deze functie àlle delers van `x` bepalen, en die lijst vervolgens vergelijken met `[1,x]`? Welnee, dat is veel te veel werk! Bij de aanroep van `priem 30` gebeurt het volgende. Eerst wordt de eerste deler van 30 bepaald: 1. Deze waarde wordt vergeleken met het eerste element van de lijst `[1,30]`. Wat het eerste element betreft zijn de lijsten dus gelijk. Dan wordt de tweede deler van 30 bepaald: 2. Die wordt vergeleken met de tweede waarde van `[1,30]`: de tweede elementen van de lijsten zijn niet gelijk. De operator `==` 'weet' dat twee lijsten nooit meer gelijk kunnen worden als er een verschillend element in zit. Daarom kan er direct `False` opgeleverd worden. De overige delers van 30 worden dus niet berekend!

Het lazy gedrag van de operator `==` wordt veroorzaakt door zijn definitie. De recursieve regel uit de definitie in paragraaf 3.1.2 luidt: blz. 39

```
(x:xs) == (y:ys)  =  x==y && xs==ys
```

Als `x==y` de waarde `False` oplevert, hoeft `xs==ys` niet meer uitgerekend te worden: het totale resultaat is toch altijd `False`. Dit lazy gedrag dankt de operator `&&` op zijn beurt aan zijn definitie:

```
False && x  =  False
True  && x  =  x
```

Als de linker parameter de waarde `False` heeft, is de waarde van de rechter parameter niet nodig om het resultaat te berekenen. (Dit is de echte definitie van `&&`. De definitie in paragraaf 1.4.3 is blz. 11 ook goed, maar vertoont niet het gewenste lazy gedrag).

Functies die alle elementen van een lijst nodig hebben, mogen niet op oneindige lijsten worden toegepast. Voorbeelden van zulk soort functies zijn `sum` en `length`. Bij de aanroep `sum (vanaf 1)` of `length (vanaf 1)` helpt zelfs lazy evaluatie niet meer om in eindige tijd het eindresultaat te berekenen. De computer zal in zo'n geval in trance gaan, en nooit met een eindantwoord komen (tenzij het resultaat van de berekening nergens gebruikt wordt, want dan wordt de berekening natuurlijk niet uitgevoerd...).

3.2.6 Functies op oneindige lijsten

In de prelude worden een aantal functies gedefinieerd die oneindige lijsten opleveren.

De functie `vanaf` uit paragraaf 3.2.4 heet in werkelijkheid `enumFrom`. De functie wordt meestal blz. 48 niet als zodanig gebruikt, omdat in plaats van `enumFrom n` ook `[n..]` geschreven mag worden. (Vergelijk de notatie `[n..m]` voor `enumFromTo n m`, die in paragraaf 3.1.1 werd besproken). blz. 38

Een oneindige lijst waarin steeds één element herhaald wordt, kan worden gemaakt met de functie `repeat`:

```
repeat   ::  a -> [a]
repeat x  =  x : repeat x
```

De aanroep `repeat 't'` levert de oneindige lijst `"ttttttt...` op.

Een oneindige lijst die door `repeat` wordt gegenereerd, kan weer goed gebruikt worden als tussenresultaat door een functie die wel een eindig resultaat heeft. De functie `replicte` bijvoorbeeld maakt een eindig aantal kopieën van een element:

```
replicte     ::  Int -> a -> [a]
replicte n x  =  take n (repeat x)
```

Dankzij lazy evaluatie kan `replicte` gebruik maken van het oneindige resultaat van `repeat`. De functies `repeat` en `replicte` worden in de prelude gedefinieerd.

De meest flexibele functie is ook nu weer een hogere-orde functie, dat wil zeggen een functie met een functie als parameter. De functie `iterate` krijgt een functie en een startelement als parameter. Het resultaat is een oneindige lijst, waarin elk volgend element verkregen wordt door de functie op het vorige element toe te passen. Bijvoorbeeld:

```
iterate (+1) 3      is  [3, 4, 5, 6, 7, 8, ...
iterate (*2) 1      is  [1, 2, 4, 8, 16, 32, ...
iterate (/10) 5678  is  [5678, 567, 56, 5, 0, 0, ...
```

De definitie van `iterate`, die in de prelude staat, is als volgt:

```
iterate     ::  (a->a) -> a -> [a]
iterate f x  =  x : iterate f (f x)
```

blz. 27

Deze functie lijkt een beetje op de functie `until`, die in paragraaf 2.3.2 werd gedefinieerd. Ook `until` krijgt immers een functie en een startelement als parameter, en past de functie herhaald toe op het startelement. Het verschil is, dat `until` stopt als de waarde aan een bepaalde voorwaarde (die ook als parameter wordt meegegeven) voldoet. Bovendien levert `until` alleen de eindwaarde op (die dus aan het meegegeven stopcriterium voldoet), terwijl `iterate` alle tussenresultaten in een lijst stopt. Hij moet wel, want bij oneindige lijsten is er geen laatste element...

Hier volgen twee voorbeelden waarin `iterate` gebruikt wordt om een praktisch probleem op te lossen: de weergave van een getal als string, en het genereren van de lijst van alle priemgetallen.

Weergave van een getal als string

De functie `intString` maakt van een getal een string waarin de cijfers van dat getal zitten. Bijvoorbeeld: `intString 5678` is de string `"5678"`. Dankzij deze functie is het mogelijk om het resultaat van een berekening te combineren met een string, bijvoorbeeld zoals in `intString (3*17)++" gulden"`.

De functie `intString` kan worden samengesteld door na elkaar een aantal functies uit te voeren. Eerst moet het getal met behulp van `iterate` herhaald door 10 gedeeld worden (zoals in het derde voorbeeld van `iterate` hierboven). De oneindige staart nullen is oninteressant, en kan worden afgekapt met `takeWhile`. De gewenste cijfers zijn dan steeds het laatste cijfer van de getallen in de lijst; het laatste cijfer van een getal is de rest bij deling door 10. De cijfers staan nu nog in de verkeerde volgorde, maar dat kan worden opgelost met de functie `reverse`. Tenslotte moeten de cijfers (van type `Int`) nog worden omgezet in het overeenkomstige cijferteken (van type `Char`).

Een schema aan de hand van een voorbeeld maakt dit wat duidelijker:

```
              5678
               ↓ iterate (/10)
[5678, 567, 56, 5, 0, 0, ...
               ↓ takeWhile (/=0)
    [5678, 567, 56, 5]
               ↓ map (`rem`10)
       [8, 7, 6, 5]
               ↓ reverse
       [5, 6, 7, 8]
               ↓ map digitChar
   ['5', '6', '7', '8']
```

De functie `intString` kan simpelweg geschreven worden als samenstelling van deze vijf functies. Let er op dat de functies in omgekeerde volgorde opgeschreven moeten worden, omdat de functiesamenstellings operator (`.`) de betekenis 'na' heeft:

```
intString ::  Int -> [Char]
intString  =  map digitChar
              . reverse
              . map (`rem`10)
              . takeWhile (/=0)
              . iterate (/10)
```

Functioneel programmeren is programmeren met functies!

De lijst van alle priemgetallen

In paragraaf 2.4.1 werd een functie `priem` gedefinieerd, die bepaalt of een getal een priemgetal is. De (oneindige) lijst van alle priemgetallen kan daarmee worden berekend door blz. 29

```
filter priem [2..]
```

De functie `priem` gaat op zoek naar delers van een getal. Als zo'n deler groot is, duurt het dus vrij lang voordat de functie tot de conclusie komt dat een getal geen priemgetal is.

Door handig gebruik te maken van `iterate` is echter een veel snellere methode mogelijk. Deze methode begint ook met de oneindige lijst `[2..]`:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
```

Het eerste getal, 2, kan in de lijst van priemgetallen worden gestopt. Nu worden 2 en alle veelvouden daarvan uit de lijst weggestreept. Er blijft dan over:

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...
```

Het eerste getal, 3, is een priemgetal. Dit getal en zijn veelvouden worden uit de lijst weggestreept:

```
[5, 7, 11, 13, 17, 19, 23, 25, 29, 31, ...
```

Hetzelfde proces wordt weer uitgevoerd, maar nu met 5:

```
[7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...
```

En zo kun je doorgaan. De functie 'streep veelvouden van het eerste element weg' wordt steeds uitgevoerd op het vorige resultaat. Dit is dus een toepassing van `iterate`, met `[2..]` als startwaarde:

```
iterate streepweg [2..]
where streepweg (x:xs) = filter (not.veelvoud x) xs
      veelvoud x y  =  deelbaar y x
```

(Het getal `y` is een veelvoud van `x` als `y` deelbaar is door `x`). Doordat de beginwaarde een oneindige lijst is, is het resultaat hiervan een *oneindige lijst van oneindige lijsten*. Die super-lijst is als volgt opgebouwd:

```
[ [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
, [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, ...
, [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, ...
, [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, ...
, [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 51, 53, ...
, ...
```

Dit ding kun je nooit in zijn geheel te zien krijgen; als je hem probeert te evalueren krijg je alleen het beginstuk van de eerste rij te zien. Maar de complete lijst hoeft niet zichtbaar gemaakt te worden: de gewenste priemgetallen zijn steeds het eerste element van de lijst. De priemgetallen worden dus bepaald door van elke lijst de `head` te nemen:

```
priemgetallen ::  [Int]
priemgetallen  =  map head (iterate streepweg [2..])
           where  streepweg (x:xs) = filter (not.veelvoud x) xs
```

Door de lazy evaluatie wordt van elke lijst in de super-lijst precies het gedeelte uitgerekend dat nodig is voor het gewenste deel van het antwoord. Wil je het volgende priemgetal weten, dan wordt elke lijst het noodzakelijke stukje verder uitgerekend.

Het is vaak (zo ook in dit voorbeeld) moeilijk om je precies voor te stellen wat er op welk moment wordt uitgerekend. Maar dat hoeft ook niet: tijdens het programmeren kun je net doen alsof oneindige lijsten echt bestaan; de uitrekenvolgorde wordt door de lazy evaluatie automatisch geoptimaliseerd.

3.2.7 Lijst-comprehensies

In de verzamelingenleer is een handige notatie in gebruik om verzamelingen te definiëren:

$$V = \{\ x^2\ \mid\ x \in N, x \text{ even}\ \}$$

Naar analogie van deze notatie, de zogenaamde verzameling-comprehensie, is in Haskell een vergelijkbare notatie beschikbaar om lijsten te construeren. Deze notatie heet dan ook een *lijst-comprehensie*. Een eenvoudig voorbeeld van deze notatie is de volgende expressie:

```
[ x*x | x <- [1..10] ]
```

Deze expressie kan worden uitgesproken als '`x` kwadraat voor `x` uit 1 tot 10'. In een lijst-comprehensie staat voor de verticale streep een expressie, waarin een variabele mag voorkomen. Deze variabele (`x` in het voorbeeld) wordt gebonden in het gedeelte achter de verticale streep. De notatie '`x <- xs`' heeft de betekenis: '`x` doorloopt alle waarden van de lijst `xs`'. Voor elk van deze waarden wordt de waarde van de expressie voor de verticale streep uitgerekend.

Bovengenoemd voorbeeld heeft dus dezelfde waarde als de expressie

```
map kwadraat [1..10]
```

waarbij de functie `kwadraat` is gedefinieerd als

```
kwadraat x  =  x*x
```

Het voordeel van de comprehensie-notatie is dat de functie die steeds wordt uitgerekend (`kwadraat` in het voorbeeld) niet eerst een naam hoeft te krijgen.

De lijst-comprehensie notatie heeft nog meer mogelijkheden. Achter de verticale streep mag namelijk meer dan één lopende variabele worden gebruikt. De expressie voor de verticale streep wordt dan voor alle mogelijke combinaties uitgerekend. Bijvoorbeeld:

```
? [ (x,y) | x<-[1..2], y<-[4..6] ]
[ (1,4), (1,5), (1,6), (2,4), (2,5), (2,6) ]
```

De laatstgenoemde variabele loopt het snelst: voor elke waarde van `x` doorloopt `y` de lijst `[4..6]`.

Behalve definities van lopende variabelen mogen achter de verticale streep uitdrukkingen met de waarde `True` of `False` worden opgenomen. De betekenis daarvan wordt gedemonstreerd door het volgende voorbeeld:

```
? [ (x,y) | x<-[1..5], even x, y<-[1..x] ]
[ (2,1), (2,2), (4,1), (4,2), (4,3), (4,4) ]
```

In de resultaat-lijst worden dus alleen díe `x` verwerkt, waarvoor `even x` de waarde `True` heeft.

Door elk pijltje (`<-`) wordt een variabele gedefiniëerd, die in de verdere expressies en in de expressie links van de verticale streep gebruikt mag worden. Zo mag de variabele `x` behalve in `(x,y)` gebruikt worden in `even x` en in `[1..x]`. De variabele `y` mag echter alleen maar gebruikt worden in `(x,y)`. Het pijltje is een speciaal voor dit doel gereserveerd symbool, en is dus geen operator!

Strikt genomen is de lijst-comprehensie notatie overbodig. Hetzelfde effect kan bereikt worden door combinaties van `map`, `filter` en `concat`. De comprehensie-notatie is, zeker in ingewikkelde gevallen, echter veel gemakkelijker te begijpen. Bovenstaand voorbeeld zou anders geschreven moeten worden als

```
concat (map f (filter even [1..5]))
where  f x  =  map g [1..x]
               where g y  =  (x,y)
```

hetgeen veel minder inzichtelijk is.

Een lijst-comprehensie wordt door de interpreter direct vertaald naar een overeenkomstige expressie met `map`, `filter` en `concat`. Net als de notatie voor intervallen is de comprehensie-notatie dus puur bedoeld voor het gemak van de programmeur.

3.3 Tupels

3.3.1 Gebruik van tupels

In een lijst moet elk element hetzelfde type hebben. Het is niet mogelijk om in één lijst zowel een integer als een string te stoppen. Toch is het soms nodig om gegevens van verschillende types te groeperen. De gegevens in een bevolkingsregister bestaan bijvoorbeeld uit een string (naam), een boolean (geslacht) en drie integers (geboortedatum). Deze gegevens horen bij elkaar, maar kunnen niet in één lijst gestopt worden.

Voor dit soort gevallen is er, naast lijstvorming, nog een andere manier om samengestelde types te maken: *tupelvorming*. Een *tupel* bestaat uit een vast aantal waarden, die tot één geheel zijn gegroepeerd. De waarden mogen van verschillend type zijn (hoewel dat niet verplicht is).

Tupels worden genoteerd met ronde haakjes rond de elementen (waar bij lijsten vierkante haakjes worden gebruikt). Voorbeelden van tupels zijn:

`(1, 'a')`	een tupel met als elementen de integer 1 en het character `'a'`;
`("aap", True, 2)`	een tupel met drie elementen: de string `"aap"`, de boolean `True` en het getal 2;
`( [1,2], sqrt)`	een tupel met twee elementen: de lijst integers `[1,2]`, en de float-naar-float functie `sqrt`;
`(1, (2,3))`	een tupel met twee elementen: het getal 1, en het tupel van de getallen 2 en 3.

Voor elke combinatie van types vormt het tupel ervan een apart type. Daarbij is ook de volgorde van belang. Het type van tupels wordt geschreven door de types van de elementen op te sommen tussen ronde haakjes. De vier hierboven genoemde expressies kunnen dus als volgt getypeerd worden:

```
(1, 'a')          ::  (Int, Char)
("aap", True, 2)  ::  ([Char], Bool, Int)
([1,2], sqrt)     ::  ([Int], Float->Float)
(1, (2,3))        ::  (Int, (Int,Int))
```

Een tupel met twee elementen wordt een 2-tupel, of ook wel een *paar* genoemd. Tupels met drie elementen heten 3-tupels, enzovoort. Er bestaan geen 1-tupels: de expressie `(7)` is gewoon een integer; om elke expressie mogen immers haakjes gezet worden. Wel bestaat er een 0-tupel: de waarde `()`, die `()` als type heeft.

In de prelude zijn een paar functies gedefinieerd die op 2-tupels of 3-tupels werken. Deze zijn er meteen een voorbeeld van hoe functies op tupels gedefinieerd kunnen worden: door patroonanalyse.

```
fst          ::  (a,b) -> a
fst (x,y)     =  x
snd          ::  (a,b) -> b
snd (x,y)     =  y
fst3         ::  (a,b,c) -> a
fst3 (x,y,z)  =  x
snd3         ::  (a,b,c) -> b
snd3 (x,y,z)  =  y
thd3         ::  (a,b,c) -> c
thd3 (x,y,z)  =  z
```

Deze functies zijn polymorf, maar het is natuurlijk ook mogelijk om functies te schrijven die maar op één specifiek tupel-type werken:

```
f       ::  (Int,Char) -> [Char]
f (n,c)  =  intString n ++ [c]
```

Als twee waarden van hetzelfde type gegroepeerd moeten worden kan daarvoor een lijst gebruikt worden. In sommige gevallen is een tupel geschikter. Een punt in het platte vlak wordt bijvoorbeeld beschreven door twee `Float` getallen. Zo'n punt kan worden gerepresenteerd door een lijst, of door een 2-tupel. In beide gevallen is het mogelijk om functies te definiëren die op punten werken, bijvoorbeeld 'afstand tot de oorsprong'. De functie `afstandL` is de lijst-versie, `afstandT` de tupel-

versie hiervan:

```
afstandL    ::  [Float] -> Float
afstandL [x,y]  =  sqrt (x*x+y*y)
afstandT    ::  (Float,Float) -> Float
afstandT (x,y)  =  sqrt (x*x+y*y)
```

Zolang de functie correct wordt aangeroepen is er geen verschil. Maar het zou kunnen gebeuren dat de functie elders in het programma, door een tikfout of een denkfout, met drie coördinaten wordt aangeroepen. Bij gebruik van `afstandT` wordt daarvoor tijdens de analyse van het programma voor gewaarschuwd: een tupel met drie getallen is een ander type dan een tupel met twee getallen. In het geval van `afstandL` is het programma echter goed getypeerd. Pas als de functie inderdaad gebruikt wordt blijkt dat `afstandL` voor lijsten met drie elementen ongedefinieerd is. Het gebruik van tupels in plaats van lijsten helpt dus in dit geval om fouten zo vroeg mogelijk op te sporen.

Nog een plaats waar tupels van pas komen zijn functies die meer dan één resultaat hebben. Functies met meerdere parameters zijn mogelijk dankzij het Currying-mechanisme; functies die meerdere resultaten hebben, zijn echter alleen mogelijk door die resultaten te 'verpakken' in een tupel. Het tupel in z'n geheel is dan immers één resultaat.

Een voorbeeld van een functie die eigenlijk twee resultaten heeft, is de functie `splitAt` die in de prelude wordt gedefinieerd. Deze functie levert de resultaten van `take` en `drop` in één keer op. De functie zou dus zo gedefinieerd kunnen worden:

```
splitAt      ::  Int -> [a] -> ([a],[a])
splitAt n xs  =  (take n xs, drop n xs)
```

Het werk van beide functies kan echter in één keer worden gedaan, vandaar dat `splitAt` uit efficiëntie-overwegingen als volgt is gedefinieerd:

```
splitAt             ::  Int -> [a] -> ([a], [a])
splitAt 0     xs     =  ([]  , xs)
splitAt n     []     =  ([]  , [])
splitAt (n+1) (x:xs) =  (x:ys, zs)
                  where  (ys,zs) = splitAt n xs
```

De aanroep `splitAt 3 "haskell"` geeft bijvoorbeeld het 2-tupel `("has","kell")` als resultaat. In de definitie is (bij de recursieve aanroep) te zien hoe zo'n resultaat-tupel gebruikt kan worden: door het te onderwerpen aan een patroon-analyse (`(ys,zs)` in het voorbeeld).

3.3.2 Type-definities

Bij veelvuldig gebruik van lijsten en tupels worden type-declaraties vaak nogal ingewikkeld. Bijvoorbeeld bij het schrijven van functies op punten, zoals de functie `afstand` hierboven. De eenvoudigste functies zijn nog wel te overzien:

```
afstand  :: (Float,Float) -> Float
verschil :: (Float,Float) -> (Float,Float) -> Float
```

Maar lastiger wordt het bij lijsten van punten, en vooral bij hogere-orde functies:

```
opp_veelhoek    :: [(Float,Float)] -> Float
transf_veelhoek :: ((Float,Float)->(Float,Float))
                   -> [(Float,Float)] -> [(Float,Float)]
```

In zo'n geval komt een *type-definitie* van pas. Met een type-definitie is het mogelijk om een (duidelijkere) naam te geven aan een type, bijvoorbeeld:

```
type Punt = (Float,Float)
```

Na deze type-definitie zijn de type-declaraties eenvoudiger te schrijven:

```
afstand  :: Punt -> Float
verschil :: Punt -> Punt -> Float
opp_veelhoek    :: [Punt] -> Float
transf_veelhoek :: (Punt->Punt) -> [Punt] -> [Punt]
```

Nog beter is het om ook voor 'veelhoek' een type-definitie te maken:

```
type Veelhoek    = [Punt]
opp_veelhoek    :: Veelhoek -> Float
transf_veelhoek :: (Punt->Punt) -> Veelhoek -> Veelhoek
```

Een paar dingen om in de gaten te houden bij type-definities:

- het woord `type` is een, speciaal voor dit doel, gereserveerd woord;
- de naam van het nieuw gedefinieerde type moet met een hoofdletter beginnen (het is een constante, niet een variabele);
- een type-*declaratie* specificeert het type van een functie; een type-*definitie* definieert een nieuwe naam voor een type.

De nieuw gedefinieerde naam wordt door de interpreter puur beschouwd als afkorting. Bij het typeren van een expressie krijg je gewoon weer `(Float,Float)` te zien in plaats van `Punt`. Als er twee verschillende namen aan één type gegeven worden, bijvoorbeeld:

```
type Punt    =  (Float,Float)
type Complex =  (Float,Float)
```

dan mogen die namen door elkaar gebruikt worden. Een `Punt` is hetzelfde als een `Complex` is hetzelfde als een `(Float,Float)`. In paragraaf 3.4.3 wordt een methode beschreven hoe `Punt` als blz. 64 een echt *nieuw* type gedefinieerd kan worden.

3.3.3 Rationale getallen

Een toepassing waarbij tupels goed gebruikt kunnen worden is een implementatie van de *rationale getallen*. De rationale getallen vormen de wiskundige verzameling **Q**, getallen die als *breuk* te schrijven zijn. Voor het rekenen met rationale getallen kunnen geen `Float` getallen gebruikt worden: het is de bedoeling dat er *exact* gerekend wordt, en dat de uitkomst van $\frac{1}{2} + \frac{1}{3}$ de breuk $\frac{5}{6}$ oplevert, en niet de `Float` 0.833333.

Rationale getallen, oftewel breuken, kunnen worden gerepresenteerd door een teller en een noemer, die allebei gehele getallen zijn. De volgende type-definitie ligt daarom voor de hand:

```
type Ratio  =  (Int,Int)
```

Een aantal veelgebruikte breuken kunnen een aparte naam krijgen:

```
qNul   =  (0, 1)
qEen   =  (1, 1)
qTwee  =  (2, 1)
qHalf  =  (1, 2)
qDerde =  (1, 3)
qKwart =  (1, 4)
```

Het is de bedoeling om functies te schrijven die de belangrijkste rekenkundige operaties op rationale getallen uitvoeren:

```
qMaal  ::  Ratio -> Ratio -> Ratio
qDeel  ::  Ratio -> Ratio -> Ratio
qPlus  ::  Ratio -> Ratio -> Ratio
qMin   ::  Ratio -> Ratio -> Ratio
```

Een probleem is, dat één waarde door verschillende breuken weergegeven kan worden. Een 'half' bijvoorbeeld, wordt gerepresenteerd door het tupel `(1,2)`, maar ook door `(2,4)` en `(17,34)`. Het resultaat van twee maal een kwart (twee-vierde) zou daardoor wel eens kunnen 'verschillen' van een half (een-tweede). Om dit probleem op te lossen, is er een functie `eenvoud` nodig, die een breuk kan vereenvoudigen. Door na elke operatie op breuken deze functie toe te passen, wordt een breuk altijd op dezelfde manier gerepresenteerd. Het resultaat van twee maal een kwart kan dan veilig vergeleken worden met een half: het resultaat is `True`.

De functie `eenvoud` deelt de teller en de noemer van een breuk door hun *grootste gemene deler*. De grootste gemene deler (`ggd`) van twee getallen is het grootste getal waardoor beide deelbaar zijn. Daarnaast zorgt `eenvoud` ervoor, dat een eventueel min-teken altijd in de teller van de breuk staat. De definitie is als volgt:

```
eenvoud (t,n)  =  ( (signum n * t)/d, abs n/d )
            where  d  = ggd t n
```

Een eenvoudige definitie van `ggd x y` (die alleen werkt als `x` en `y` positief zijn) bepaalt de grootste deler van `x` waardoor `y` deelbaar is, gebruik makend van de functies `delers` en `deelbaar` uit paragraaf 2.4.1: blz. 29

```
ggd x y  =  last (filter (deelbaar y') (delers x'))
     where  x' = abs x
```

```
                y' = abs y
```

(In de prelude wordt een functie `gcd` (*greatest common divisor*) gedefinieerd, die sneller werkt:

```
gcd x y  =  gcd' (abs x) (abs y)
     where  gcd' x 0 = x
            gcd' x y = gcd' y (x `rem` y)
```

Deze methode is erop gebaseerd dat als `x` en `y` deelbaar zijn door `d`, dat dan ook `x`rem`y` (=`x-(x/y)*y`) deelbaar is door `d`).

Met behulp van de functie `eenvoud` kunnen nu de rekenkundige functies gedefinieerd worden. Om twee breuken te vermenigvuldigen, moeten de teller en de noemer vermenigvuldigd worden ($\frac{2}{3} * \frac{5}{4} = \frac{10}{12}$). Daarna kan het resultaat vereenvoudigd worden (tot $\frac{5}{6}$):

```
qMaal (x,y) (p,q)  =  eenvoud (x*p, y*q)
```

Delen door een getal is vermenigvuldigen met het omgekeerde, dus:

```
qDeel (x,y) (p,q)  =  eenvoud (x*q, y*p)
```

Voor het optellen van twee breuken moeten ze eerst gelijknamig worden gemaakt ($\frac{1}{4}+\frac{3}{10} = \frac{10}{40}+\frac{12}{40} = \frac{22}{40}$). Als gelijke noemer kan het product van de noemers dienen. De tellers moeten dan met de noemer van de andere breuk worden vermenigvuldigd, waarna ze kunnen worden opgeteld. Het resultaat moet tenslotte vereenvoudigd worden (tot $\frac{11}{20}$).

```
qPlus (x,y) (p,q)  =  eenvoud (x*q+y*p, y*q)
qMin  (x,y) (p,q)  =  eenvoud (x*q-y*p, y*q)
```

Het resultaat van berekeningen met rationale getallen wordt als tupel op het scherm gezet. Als dat niet mooi genoeg is, kan er eventueel een functie `ratioString` worden gedefinieerd:

```
ratioString :: Ratio -> String
ratioString (x,y)
         |  y'==1     =  intString x'
         |  otherwise =  intString x' ++ "/" ++ intString y'
                where   (x',y') = eenvoud (x,y)
```

3.3.4 Tupels en lijsten

Tupels komen vaak voor als elementen van een lijst. Veel gebruikt wordt bijvoorbeeld een lijst van twee-tupels, die als opzoeklijst (woordenboek, telefoonboek enz.) kan dienen. De opzoek-functie is heel eenvoudig te definiëren met behulp van patronen; voor de lijst wordt een patroon gebruikt voor 'niet-lege lijst waarvan het eerste element een 2-tupel is (en de andere elementen dus ook)'.

```
zoekOp  ::  Eq a  =>  [(a,b)] -> a -> b
zoekOp ((x,y):ts) z
           | x == z     =  y
           | otherwise  =  zoekOp ts z
```

De functie is polymorf, dus werkt op lijsten 2-tupels van willekeurig type. Wel moeten de op te zoeken elementen vergeleken kunnen worden, dus het type `a` moet in de klasse `Eq` zitten.

Het op te zoeken element (van type `a`) is opzettelijk als tweede parameter gedefinieerd, zodat de functie `zoekOp` eenvoudig partieel geparametriseerd kan worden met een specifieke opzoeklijst, bijvoorbeeld:

```
telefoonNr  =  zoekOp telefoonboek
vertaling   =  zoekOp woordenboek
```

waarbij `telefoonboek` en `woordenboek` apart als constante gedefinieerd kunnen worden.

Een andere functie waarin lijsten 2-tupels een rol spelen is de functie `zip`. Deze functie wordt in de prelude gedefinieerd. De functie `zip` heeft twee lijsten als parameter, die in het resultaat per element aan elkaar gekoppeld worden. Bijvoorbeeld: `zip [1,2,3] "abc"` geeft de lijst `[(1,'a'),(2,'b'),(3,'c')]`. Als de parameter-lijsten niet even lang zijn, is de lengte van de kortste van de twee bepalend. De definitie is zeer rechtstreeks:

```
zip               ::  [a] -> [b] -> [(a,b)]
zip []     ys      =  []
zip xs     []      =  []
zip (x:xs) (y:ys)  =  (x,y) : zip xs ys
```

De functie is polymorf, en kan dus op lijsten met elementen van willekeurige types worden toegepast. De naam *zip* betekent letterlijk 'rits': de twee lijsten worden als het ware aan elkaar geritst.

Een hogere-orde variant van `zip` is de functie `zipWith`. Deze functie krijgt behalve twee lijsten ook een functie als parameter, die aangeeft hoe de overeenkomstige elementen aan elkaar gekoppeld moeten worden:

```
zipWith                 ::  (a->b->c) -> [a] -> [b] -> [c]
zipWith f []     ys      =  []
zipWith f xs     []      =  []
zipWith f (x:xs) (y:ys)  =  f x y : zipWith f xs ys
```

Deze functie past een functie (met twee parameters) toe op alle elementen van twee lijsten. Behalve op `zip` lijkt `zipWith` ook sterk op `map`, die immers een functie (met één parameter) toepast op alle elementen van één lijst.

Gegeven de functie `zipWith` kan `zip` gedefinieerd worden als partiële parametrisatie daarvan:

```
zip  =  zipWith maak2tupel
 where  maak2tupel x y = (x,y)
```

3.3.5 Tupels en Currying

Met behulp van tupels is het mogelijk om functies met meer dan één parameter te schrijven, zonder het Curry-mechanisme te gebruiken. Een functie kan namelijk een tupel als (enige) parameter krijgen, waarmee toch twee waarden naar binnen gesmokkeld worden:

```
plus (x,y)  =  x+y
```

Deze functiedefinitie ziet er heel klassiek uit. De meeste mensen zouden zeggen dat `plus` een functie is met twee parameters, en dat parameters 'natuurlijk' tussen haakjes staan. Maar wij weten inmiddels beter: deze functie heeft één parameter, en wel een tupel; de definitie vindt plaats met behulp van een patroon voor een tupel.

De Curry-methode is overigens vaak te prefereren boven de tupel-methode. Gecurryde functies zijn immers partieel te parametriseren, en functies met een tupel-parameter niet. Alle standaardfuncties met meer dan één parameter werken dan ook volgens de Curry-methode.

In de prelude wordt een functie-transformatie (functie met functie als parameter en andere functie als resultaat) gedefinieerd, die van een gecurryde functie een functie met tupel-parameter maakt. Deze functie heet `uncurry`:

```
uncurry  ::  (a->b->c) -> ((a,b)->c)
uncurry  f (a,b)  =  f a b
```

Andersom is er een functie `curry` die van een functie met tupel-parameter een gecurryde functie maakt. Dus `curry plus`, met `plus` zoals hierboven, kan wèl partieel geparametriseerd worden.

3.4 Bomen

3.4.1 Data-definities

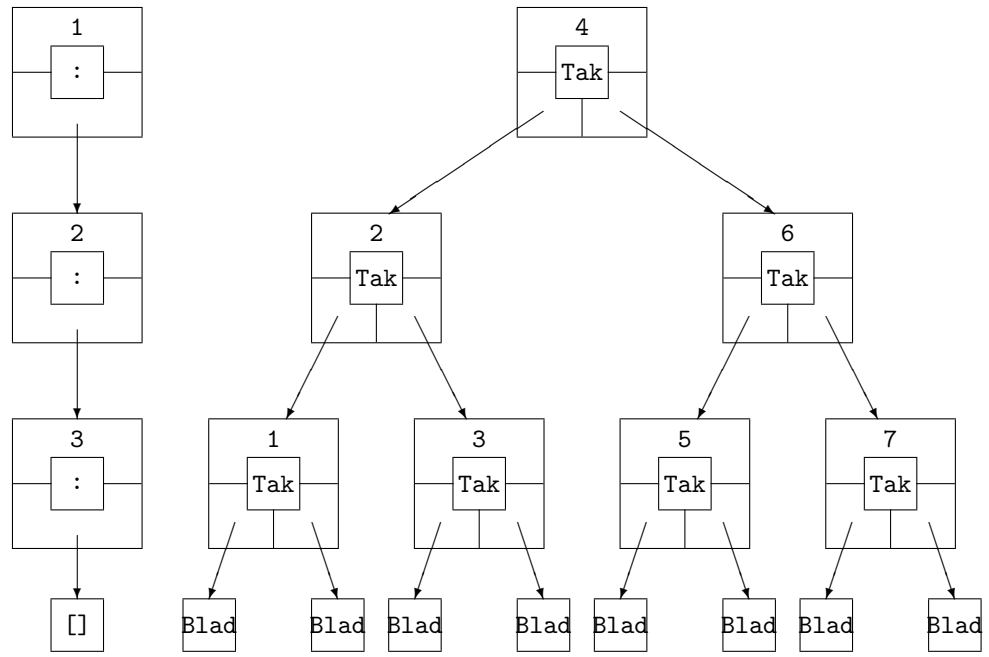
Lijsten en tupels zijn twee 'ingebouwde' manieren om gegevens te structureren. Mochten deze twee manieren niet geschikt zijn om bepaalde gegevens te representeren, dan is het mogelijk om zelf een nieuw *datatype* te definiëren.

Een datatype is een type dat gekenmerkt wordt door de manier waarop waarden van dat type kunnen worden opgebouwd. Het begrip 'lijst' is een datatype. Lijst-waarden kunnen op twee manieren worden opgebouwd:

- als de lege lijst;
- uit een element en een (kleinere) lijst door de operator `:` toe te passen.

Deze twee manieren komen terug in de patronen van definities van functies op lijsten, bijvoorbeeld:

```
length []      =  0
length (x:xs)  =  1 + length xs
```



lijststructuur *boomstructuur*

Door de functie te definiëren voor de patronen 'lege lijst' en 'operator `:` toegepast op een element en een lijst' is de functie geheel gedefinieerd.

Een lijst is een lineaire structuur; doordat steeds een element wordt bijgeschakeld, ontstaat een steeds langere keten. Soms is zo'n lineaire structuur niet gewenst, maar voldoet een *boomstructuur* beter. Er zijn verschillende boomstructuren mogelijk. In de volgende figuur wordt een lijst vergeleken met een boom die zich steeds in tweeën splitst. In kleine vierkantjes is aangegeven hoe de structuur is opgebouwd. In het geval van de lijst is dat door middel van de operator `:` met twee parameters (die eromheen getekend zijn), of door middel van `[]` zonder parameters. De boom is niet opgebouwd met operatoren maar met functies: `Tak` (met drie parameters) of `Blad` (zonder parameters).

Functies waarmee een datastructuur wordt opgebouwd heten *constructor-functies*. De constructor-functies van de boom zijn `Tak` en `Blad`. Namen van constructor-functies beginnen met een hoofdletter, om ze te onderscheiden van 'gewone' functies. Als constructor-functies als operator geschreven worden moeten ze met een dubbele punt beginnen. De constructor-operator (`:`) van lijsten is daar een voorbeeld van, en de constructor `[]` is de enige uitzondering.

Welke constructor-functies voor een nieuw type gebruikt kunnen worden, wordt gespecificeerd met een *data-definitie*. Daarin staan ook de types van de parameters van de constructor-functies, en of het nieuwe type polymorf is. De data-definitie voor bomen zoals hierboven besproken luidt bijvoorbeeld als volgt:

```
data Boom a  =  Tak a (Boom a) (Boom a)
             |  Blad
```

Je kunt deze definitie als volgt uitspreken. 'Een boom met elementen van type `a` (kortweg boom-over-`a`) kan op twee manieren worden opgebouwd: (1) door de functie `Tak` toe te passen op drie parameters (één van type `a` en twee van type boom-over-`a`), of (2) door de constante `Blad` te gebruiken.'

Bomen kunnen worden opgebouwd door de constructor-functies in een expressie te gebruiken. De boom die in de figuur getekend is, wordt bijvoorbeeld weergegeven door de volgende expressie:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad)
             (Tak 3 Blad Blad)
      )
      (Tak 6 (Tak 5 Blad Blad)
             (Tak 7 Blad Blad)
```

```
        )
```

Het hoeft niet zo mooi over de regels gespreid te worden; ook toegestaan is:

```
Tak 4(Tak 2(Tak 1 Blad Blad)(Tak 3 Blad Blad))
                    (Tak 6(Tak 5 Blad Blad)(Tak 7 Blad Blad))
```

De eerstgenoemde constructie is natuurlijk wel duidelijker. Houd ook de layout-regel uit paragraaf 1.4.5 in de gaten. blz. 14

Functies op een boom kunnen gedefinieerd worden door voor elke constructor-functie een patroon te maken. De volgende functie bepaalt bijvoorbeeld het aantal `Tak`-constructies in een boom:

```
omvang             ::  Boom a -> Int
omvang Blad         =  0
omvang (Tak x p q)  =  1 + omvang p + omvang q
```

Vergelijk deze functie met de functie `length` op lijsten.

Er zijn nog vele andere mogelijke bomen denkbaar. Een paar voorbeelden:

- Bomen waarbij de informatie in de eindpunten wordt opgeslagen (in plaats van op de splitspunten zoals bij `Boom`):

  ```
  data Boom2 a  =  Tak2 (Boom2 a) (Boom2 a)
                |  Blad2 a
  ```

- Bomen waarbij de informatie van type `a` in de splitspunten is opgeslagen, en informatie van type `b` in de eindpunten:

  ```
  data Boom3 a b  =  Tak3 a (Boom3 a b) (Boom3 a b)
                  |  Blad3 b
  ```

- Bomen die zich op elk splitspunt in drieën splitsen in plaats van in tweeën:

  ```
  data Boom4 a  =  Tak4 a (Boom4 a) (Boom4 a) (Boom4 a)
                |  Blad4
  ```

- Bomen waarin het aantal uitgaande takken in een splitspunt variabel is:

  ```
  data Boom5 a  =  Tak5 a [Boom5 a]
  ```

  In deze boom is geen aparte constructor voor 'eindpunt' nodig, omdat daarvoor een splitspunt met nul uitgaande takken gebruikt kan worden.

- Bomen waarin elk splitspunt slechts één uitgaande tak heeft:

  ```
  data Boom6 a  =  Tak6 a (Boom6 a)
                |  Blad6
  ```

  Een 'boom' volgens dit type is in feite een lijst: hij heeft een lineaire structuur.

- Bomen met verschillende soorten splitsingen:

  ```
  data Boom7 a b  =  Tak7a Int a (Boom7 a b) (Boom7 a b)
                  |  Tak7b Char  (Boom7 a b)
                  |  Blad7a b
                  |  Blad7b Int
  ```

3.4.2 Zoekbomen

Een goed voorbeeld van een situatie waarin beter bomen gebruikt kunnen worden dan lijsten, is het zoeken naar (de aanwezigheid van) een waarde in een grote collectie. Daarvoor kunnen *zoekbomen* gebruikt worden.

In paragraaf 3.1.2 werd de functie `elem` gedefinieerd, die `True` oplevert als een element in een lijst aanwezig is. Of deze functie nu met behulp van de standaardfuncties `map` en `or` wordt gedefinieerd blz. 41

```
elem      ::  Eq a  =>  a -> [a] -> Bool
elem e xs  =  or (map (==e) xs)
```

of direct met recursie

```
elem e []      =  False
elem e (x:xs)  =  x==e || elem e xs
```

maakt voor de efficiëntie ervan niet zo veel uit. In beide gevallen worden de elementen van de lijst één voor één geïnspecteerd. Op het moment dat het element gevonden is, geeft de functie direct een resultaat (dankzij lazy evaluatie), maar als het element niet aanwezig is moet de functie alle elementen van de lijst bekijken om tot die conclusie te komen.

Iets handiger werkt het als de functie mag aannemen dat de te doorzoeken lijst gesorteerd is, dat wil zeggen dat de elementen op stijgende volgorde staan. Het zoekproces kan dan namelijk ook gestopt worden als het gevorderd is tot 'voorbij' de gezochte waarde. De prijs is wel dat de elementen nu niet alleen vergelijkbaar moeten zijn (klasse `Eq`), maar ook ordenbaar (klasse `Ord`):

```
elem'         ::  Ord a  =>  a -> [a] -> Bool
elem' e []               =  False
elem' e (x:xs)  |  e<x   =  False
                |  e==x  =  True
                |  e>x   =  elem' e xs
```

Een veel grotere verbetering is het echter als de elementen niet in een lijst zijn opgeslagen, maar in een *zoekboom*. Een zoekboom is een soort 'gesorteerde boom'. Het is een boom die is opgebouwd volgens de definitie van `Boom` uit de vorige paragraaf:

```
data Boom a  =  Tak a (Boom a) (Boom a)
             |  Blad
```

Op elk splitspunt is een element opgeslagen, en twee (kleinere) bomen: een 'linker' deelboom en een 'rechter' deelboom (zie de figuur op blz. 58). In een zoekboom wordt nu bovendien geëist dat alle waarden in de linker deelboom *kleiner* zijn dan de waarde in het splitspunt, en alle waarden in de rechter deelboom *groter*. De waarden in de voorbeeldboom in de genoemde figuur zijn zo gekozen, dat de afgebeelde boom inderdaad een zoekboom is.

In een zoekboom is het zoeken naar een waarde heel eenvoudig. Als de gezochte waarde gelijk is aan de opgeslagen waarde in een splitspunt: mooi zo. Als de gezochte waarde kleiner is dan de opgeslagen waarde, dan moet doorgezocht worden in de linker deelboom (in de rechter deelboom zitten immers grotere waarden). Andersom, als de gezochte waarde groter is dan de opgeslagen waarde, moet juist in de rechter deelboom worden doorgezocht. De functie `elemBoom` is dus als volgt:

```
elemBoom   :: Ord a  =>  a -> Boom a -> Bool
elemBoom e Blad                    =  False
elemBoom e (Tak x li re)  |  e==x  =  True
                          |  e<x   =  elemBoom e li
                          |  e>x   =  elemBoom e re
```

Als de boom evenwichtig is opgebouwd, zal het te doorzoeken aantal elementen bij elke stap ongeveer halveren. Het gezochte element of een `Blad`-eindpunt is dan snel gevonden: een verzameling van duizend elementen hoeft maar 10 keer gehalveerd te worden, en een verzameling van een miljoen elementen 20 keer. Vergelijk dat met de gemiddeld half miljoen stappen die de functie `elem` kost op een verzameling met een miljoen elementen.

In het algemeen kun je zeggen dat het geheel doorzoeken van een verzameling met n elementen met `elem` n stappen kost, maar met `elemBoom` slechts $^2\log n$ stappen.

Zoekbomen zijn goed te gebruiken als een grote hoeveelheid gegevens vaak moet worden door-
blz. 56 zocht. Ook in bijvoorbeeld de functie `zoekOp` uit paragraaf 3.3.4 is met behulp van zoekbomen een dramatische snelheidswinst te boeken.

Opbouw van een zoekboom

De vorm van een zoekboom voor een bepaalde collectie gegevens kan 'met de hand' bepaald worden. De zoekboom kan vervolgens worden ingetikt als grote expressie met veel constructor-functies. Dat is echter een vervelend werk, dat eenvoudig kan worden geautomatiseerd.

Zoals de functie `insert` een element op de juiste plaats toevoegt aan een gesorteerde lijst (zie
blz. 44 paragraaf 3.1.4), voegt de functie `insertBoom` een element toe aan een zoekboom. Het resultaat blijft een zoekboom, dat wil zeggen het element wordt op de juiste plaats ingevoegd:

```
insertBoom  ::  Ord a  =>  a -> Boom a -> Boom a
insertBoom e Blad                   =  Tak e Blad Blad
```

```
insertBoom e (Tak x li re)  |  e<=x  =  Tak x (insertBoom e li) re
                            |  e>x   =  Tak x li (insertBoom e re)
```

In het geval dat het element wordt toegevoegd aan `Blad` (een 'lege' boom), wordt een klein boompje gebouwd uit `e` en twee lege boompjes. Anders is de boom niet leeg, en bevat dus een opgeslagen waarde `x`. Deze waarde wordt gebruikt om te beslissen of `e` in de linker- of rechter deelboom ingevoegd moet worden.

Door de functie `insertBoom` herhaald te gebruiken, kunnen alle elementen van een lijst in een zoekboom worden gezet:

```
lijstNaarBoom ::  Ord a  =>  [a] -> Boom a
lijstNaarBoom  =  foldr insertBoom Blad
```

Vergelijk deze functie met de functie `isort` in paragraaf 3.1.4. blz. 44

Het gebruik van `lijstNaarBoom` heeft het nadeel dat de zoekboom die het resultaat is niet altijd evenwichtig is. Bij gegevens die in een willekeurige volgorde worden ingevoegd valt dat meestal wel mee. Als de lijst die tot boom wordt gemaakt echter al gesorteerd is, is het resultaat een 'scheefgegroeide' boom:

```
? lijstNaarBoom [1..7]
Tak 7 (Tak 6 (Tak 5 (Tak 4 (Tak 3 (Tak 2 (Tak 1 Blad Blad)
Blad) Blad) Blad) Blad) Blad) Blad
```

Dit is weliswaar een zoekboom (elke waarde ligt tussen de waardes in de linker- en de rechter zoekboom), maar is helemaal scheefgetrokken zodat een bijna lineaire structuur is ontstaan. De gewenste logaritmische zoektijden zijn in deze boom dan ook niet mogelijk. Een betere (niet-scheve) boom met dezelfde waarden zou zijn:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad)
             (Tak 3 Blad Blad))
      (Tak 6 (Tak 5 Blad Blad)
             (Tak 7 Blad Blad))
```

Sorteren met zoekbomen

De hierboven ontwikkelde functies kunnen worden gebruikt in een nieuw sorteer-algoritme. Daarbij is nog één extra functie nodig: een functie die de elementen van een zoekboom op volgorde in een lijst zet. Deze functie is als volgt:

```
labels               ::  Boom a -> [a]
labels Blad           =  []
labels (Tak x li re)  =  labels li ++ [x] ++ labels re
```

In tegenstelling tot `insertBoom` doet deze functie een recursieve aanroep op de linker deelboom èn de rechter deelboom. Op deze manier wordt elk element in de complete boom bekeken. Doordat de waarde `x` er op de juiste plaats tussen wordt geplakt, is het resultaat een gesorteerde lijst (mits de parameter een zoekboom is).

Een willekeurige lijst kan nu gesorteerd worden door er een zoekboom van te maken met `lijstNaarBoom`, en de elementen vervolgens op volgorde op te sommen met `labels`:

```
sorteer  ::  Ord a  =>  [a] -> [a]
sorteer   =  labels . lijstNaarBoom
```

Weglaten uit zoekbomen

Een zoekboom kan als database gebruikt worden. Naast de operaties opsommen, invoegen en opbouwen, waarvoor al functies geschreven zijn, zou daarbij een functie voor het weglaten van een te specificeren element goed van pas komen. Deze functie lijkt een beetje op de functie `insertBoom`; de functie wordt al naar gelang de aangetroffen waarde recursief aangeroepen op de linker- of de rechter-deelboom.

```
deleteBoom  ::  Ord a   =>  a -> Boom a -> Boom a
deleteBoom e Blad    =  Blad
deleteBoom e (Tak x li re)
             | e<x   =  Tak x (deleteBoom e li) re
             | e==x  =  samenvoegen li re
             | e>x   =  Tak x li (deleteBoom e re)
```

Als de waarde echter in de boom aangetroffen wordt (het geval `e==x`), kan hij niet zomaar worden

weggelaten zonder een 'gat' achter te laten. Daarom is er een functie `samenvoegen` nodig, die twee zoekbomen samenvoegt. Deze functie werkt door het grootste element uit de linker deelboom te gebruiken als nieuw splitspunt. Als de linker deelboom leeg is, is samenvoegen natuurlijk ook geen probleem:

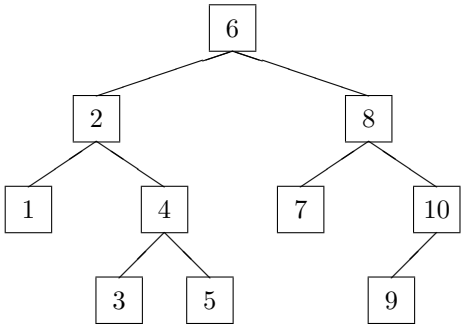
```
samenvoegen  ::  Boom a -> Boom a -> Boom a
samenvoegen Blad b2  =  b2
samenvoegen b1   b2  =  Tak x b1' b2
                 where  (x,b1') = grootsteUit b1
```

De functie `grootsteUit` levert behalve het grootste element van een boom ook de boom op die ontstaat door dit grootste element te verwijderen. Deze twee resultaten worden in een tupel samengevoegd. Het grootste element kun je vinden door steeds in de rechter deelboom af te dalen:

```
grootsteUit  ::  Boom a -> (a, Boom a)
grootsteUit (Tak x b1 Blad) =  (x, b1)
grootsteUit (Tak x b1 b2)   =  (y, Tak x b1 b2')
                        where  (y,b2') = grootsteUit b2
```

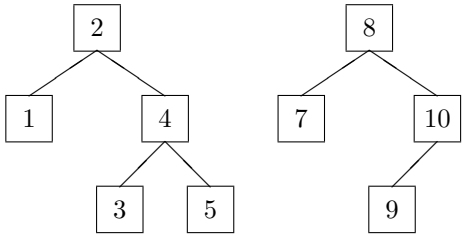
Om de werking van `deleteBoom` te demonstreren bekijken we een voorbeeld, waarbij we voor de duidelijkheid de bomen grafisch voorstellen. Bij de aanroep van

`deleteBoom 6`

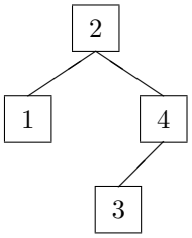


wordt de functie `samenvoegen` aangeroepen met de linker- en de rechter deelboom als parameter:

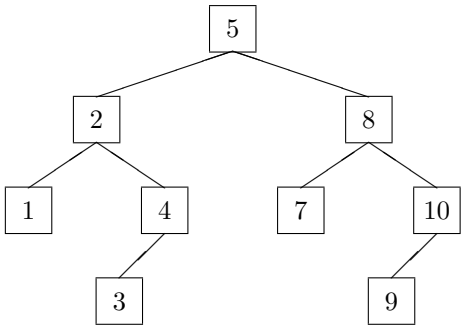
`samenvoegen` `b1` `b2`



Door `samenvoegen` wordt de functie `grootsteUit` aangeroepen met `b1` als parameter. Dat levert een tweetupel `(x,b1')` op:

`( 5 ,`  `)`

De bomen `b1'` en `b2` worden als linker- en rechter deelboom gebruikt in een nieuwe zoekboom:



Omdat de functie `grootsteUit` alleen maar wordt aangeroepen vanuit `samenvoegen`, hoeft hij niet gedefinieerd te worden op een `Blad`-boom. Hij wordt immers alleen maar met niet-lege bomen aangeroepen, omdat de lege boom in de functie `samenvoegen` al apart wordt afgehandeld.

3.4.3 Speciaal gebruik van data-definities

Behalve voor de constructie van bomen van allerlei vorm, kan een data-definitie nog op een paar opmerkelijke manieren worden gebruikt. Het datatype-mechanisme is zo universeel bruikbaar, dat er ook dingen mee gemaakt kunnen worden waarvoor in andere talen vaak aparte constructies nodig zijn.

Drie voorbeelden hiervan zijn: eindige types, vereniging van types, en beschermde types.

Eindige types

De constructor-functies in een datatype-definitie mogen ook nul parameters hebben. Dat bleek al eerder: de constructor-functie `Blad` van het type `Boom` had bijvoorbeeld geen parameters.

Het is ook toegestaan dat *geen enkele* constructor-functie parameters heeft. Het resultaat is een type dat precies zoveel elementen bevat als er constructor-functies zijn: een eindig type. De constructor-functies dienen als constanten om deze elementen aan te duiden. Een voorbeeld:

```
data Richting  =  Noord | Oost | Zuid | West
```

Functies op dit soort types kunnen gewoon met behulp van patronen worden geschreven, bijvoorbeeld:

```
move             ::  Richting -> (Int,Int) -> (Int,Int)
move Noord (x,y)  =  (x,y+1)
move Oost  (x,y)  =  (x+1,y)
move Zuid  (x,y)  =  (x,y-1)
move West  (x,y)  =  (x-1,y)
```

De voordelen van zo'n eindig type boven een codering met integers of characters zijn:

- functie-definities zijn duidelijker doordat de namen van de elementen gebruikt kunnen worden, in plaats van obscure coderingen;
- het type-systeem klaagt als je richtingen per ongeluk zou optellen (als de richtingen door integers gecodeerd werden, dan zou dit geen foutmelding geven, met alle vervelende gevolgen van dien).

Eindige types zijn niets nieuws: in feite kan het type `Bool` op deze manier gedefinieerd worden:

```
data Bool  =  False | True
```

Dit is ook de reden dat `False` en `True` met een hoofdletter geschreven moeten worden: het zijn de constructor-functies van `Bool`. (Deze definitie staat overigens niet echt in de prelude. Booleans zijn niet 'voorgedefinieerd' maar 'ingebouwd'. De reden daarvoor is, dat andere ingebouwde taalconstructies de Booleans al moeten 'kennen', zoals gevalsonderscheid met `|` in een functiedefinitie.)

Vereniging van types

Alle elementen van een lijst moeten hetzelfde type hebben. In een tupel mogen waarden van verschillend type worden opgeslagen, maar bij tupels is het aantal elementen weer niet variabel. Soms wil je echter een lijst maken, waarvan bijvoorbeeld sommige elementen integers zijn, en andere elementen characters.

Met een data-definitie is het mogelijk een type `IntOfChar` te maken, die als elementen zowel de integers als de characters heeft:

```
data IntOfChar  =  EenInt  Int
                |  EenChar Char
```

Hiermee kun je een 'gemengde' lijst maken:

```
xs ::  [IntOfChar]
xs  =  [ EenInt 1, EenChar 'a', EenInt 2, EenInt 3 ]
```

De enige prijs die je moet betalen, is dat elk element gemarkeerd moet worden met de constructor-functie `EenInt` of `EenChar`. Deze functies zijn te beschouwen als conversiefuncties:

```
EenInt  ::  Int  -> IntOfChar
EenChar ::  Char -> IntOfChar
```

waarvan het gebruik vergelijkbaar is met dat van ingebouwde conversiefuncties zoals

```
truncate ::  Float -> Int
chr      ::  Int   -> Char
```

Beschermde types

blz. 54

In paragraaf 3.3.2 werd een nadeel genoemd van type-definities: als twee types op dezelfde manier worden gedefinieerd, bijvoorbeeld

```
type Datum  =  (Int,Int)
type Ratio  =  (Int,Int)
```

dan kunnen ze door elkaar worden gebruikt. 'Datums' kunnen daardoor ineens worden verwerkt alsof het 'rationale getallen' zijn, zonder dat dat foutmeldingen van de type-checker oplevert.

Met data-definities is het mogelijk om echte nieuwe types te maken, zodat bijvoorbeeld een `Ratio` niet meer zonder meer uitwisselbaar is met elke andere `(Int,Int)`. In plaats van de type-definitie wordt daartoe de volgende data-definitie gegeven:

```
data Ratio  =  Rat (Int,Int)
```

Er is dus slechts één constructor-functie. Om een breuk te maken met een teller 3 en een noemer 5, is het nu niet meer voldoende om `(3,5)` te schrijven, maar moet je schrijven `Rat (3,5)`. Net als bij verenigings-types kan `Rat` worden beschouwd als conversiefunctie van `(Int,Int)` naar `Ratio`. Het is eigenlijk wel zo handig om ook constructor-functies te curryen. In dat geval krijgen ze niet een tupel als parameter, maar twee losse waarden. De bijbehorende datatype-definitie is:

```
data Ratio  =  Rat Int Int
```

Deze methode wordt veel gebruikt om *beschermde types* te maken. Een beschermd type bestaat uit een data-definitie en een aantal functies die op het gedefinieerde type werken (in het geval van `Ratio` bijvoorbeeld `qPlus`, `qMin`, `qMaal` en `qDeel`).

De rest van het programma (dat mogelijkerwijs door een andere programmeur geschreven kan zijn) mag van het type gebruik maken via de daarvoor bedoelde functies. Het mag echter geen gebruik maken van de manier waarop het type is opgebouwd. Dat is te bereiken door de naam van de constructor-functie 'geheim te houden'. Als later de representatie van rationale getallen om een of andere reden gewijzigd zou moeten worden, hoeven alleen de vier basisfuncties opnieuw geschreven te worden; de rest van het programma blijft gegarandeerd werken.

Als naam voor de constructor-functie wordt vaak dezelfde naam als de naam van het type gekozen, dus bijvoorbeeld

```
data Ratio  =  Ratio Int Int
```

Daar is niets op tegen; voor de interpreter is er geen verwarring mogelijk (het woord `Ratio` in een type, bijvoorbeeld achter `::`, stelt het type voor; in een expressie is het de constructor-functie).

Opgaven

3.1 Ga na dat `[1,2]++[]` volgens de definitie van `++` inderdaad `[1,2]` oplevert. Hint: schrijf `[1,2]` als `1:(2:[])`.

3.2 Schrijf de functie `concat` als aanroep van `foldr`.

3.3 Welke van de volgende expressies levert `True` op voor alle lijsten `xs`, en welke `False`:

```
[[]] ++ xs    ==  xs
[[]] ++ xs    ==  [xs]
[[]] ++ xs    ==  [ [], xs ]
[[]] ++ [xs]  ==  [ [], xs ]
[xs] ++ []    ==  [xs]
[xs] ++ [xs]  ==  [xs,xs]
```

3.4 De functie `filter` kan gedefiniëerd worden in termen van `concat` en `map`:

```
filter p  =  concat . map box
      where  box x =
```

Completeer de definitie van de functie `box` die hierin is gebruikt.

3.5 Schrijf met behulp van de functie `iterate` een niet-recursieve definitie van `repeat`.

3.6 Schrijf een functie met twee lijsten als parameter, die van elk element uit de tweede lijst het eerste voorkomen in de eerste lijst verwijdert. (Deze functie is in de prelude als operator gedefinieerd en heet \\).

3.7 Gebruik de functies `map` en `concat` in plaats van de lijstcomprehensie-notatie om de volgende lijst te definiëren:

```
[ (x,y+z) | x<-[1..10], y<-[1..x], z<-[1..y] ]
```

3.8 Het vereenvoudigen van breuken is niet nodig als breuken nooit direct met elkaar vergeleken worden. Schrijf een functie `qEq` die gebruikt kan worden in plaats van `==`. Deze functie levert `True` op als twee breuken dezelfde waarde hebben, ook als de breuken niet vereenvoudigd zijn.

3.9 Schrijf de vier rekenkundige functies voor het rekenen met *complexe getallen*. Complexe getallen zijn getallen van de vorm $a + bi$, waarbij a en b reële getallen zijn, en i een 'getal' is met de eigenschap $i^2 = -1$. Hint: leid voor de deel-functie eerst een formule af voor $\frac{1}{a+bi}$ door x en y op te lossen uit $(a+bi) * (x+yi) = (1+0i)$.

3.10 Schrijf een functie `stringInt`, die van een string de overeenkomstige integer maakt. Bijvoorbeeld: `stringInt "123"` levert de waarde 123. Beschouw daarvoor de string als lijst characters, en bepaal welke operator tussen de characters moet staan. Moet je daarbij van rechts of van links beginnen?

3.11 Definieer de functie `curry`, als tegenhanger van `uncurry` uit paragraaf 3.3.5. blz. 57

3.12 Schrijf een zoekboom-versie van de functie `zoekOp`, zoals `elemBoom` een zoekboom-versie is van `elem`. Geef ook het type van de functie.

3.13 De functie `map` kan op functies worden toegepast. Het resultaat is ook weer een functie (met een ander type). Er is geen enkele voorwaarde verbonden aan het soort functies waarop `map` toegepast kan worden. Je kunt hem dus ook op de functie `map` zelf toepassen! Wat is het type van de expressie `map map` ?

3.14 Geef een definitie van `until` die gebruik maakt van `iterate` en `dropWhile`.

3.15 Geef een directe definitie van de operator `<` op lijsten. Deze definitie mag dus geen gebruik maken van operatoren zoals `<=` op lijsten. (Als je deze definitie daadwerkelijk met de Haskell-interpretator wilt uitproberen, gebruik dan een andere naam dan `<`, omdat de operator `<` al in de prelude wordt gedefinieerd.)

3.16 Schrijf de functie `length` als aanroep van `foldr`. (Hint: begin aan de rechterkant met het getal 0, en zorg ervoor dat de operator die achtereenvolgens op alle elementen van de lijst wordt toegepast steeds 1 bij het tussenresultaat optelt, ongeacht de waarde van het lijstelement.) Wat is het type van de functie die daarbij aan `foldr` wordt meegegeven?

3.17 In paragraaf 3.1.4 werden twee sorteermethodes genoemd: de op `insert` gebaseerde functie `isort`, en de op `merge` gebaseerde functie `msort`. Een andere sorteermethode werkt volgens het volgende principe. Bekijk het eerste element van de te sorteren lijst. Neem nu alle elementen van de lijst die kleiner zijn dan deze waarde. In het eindresultaat moeten al deze waarden vóór het eerste element komen. Ze moeten wel eerst (met een recursieve aanroep) gesorteerd worden. De waarden uit de lijst die juist groter zijn dan het eerste element moeten (gesorteerd) erachter komen. (Dit algoritme staat bekend onder de naam *quicksort*).
Schrijf een functie die volgens dit principe werkt. Bedenk zelf wat het basisgeval is. Wat is het essentiële verschil tussen deze functie en `msort`? blz. 44

3.18 Beschouw de functie `groepeer` met het volgende type:

```
groepeer :: Int -> [a] -> [[a]]
```

Deze functie deelt de een gegeven lijst in deel-lijsten (die in een lijst van lijsten worden opgeleverd), waarbij de deel-lijsten een gegeven lengte hebben. Alleen de laatste deel-lijst mag zonodig wat korter zijn. De functie kan als bijvoorbeeld als volgt gebruikt worden:

```
? groepeer 3 [1..11]
[ [1,2,3], [4,5,6], [7,8,9], [10,11] ]
```

Schrijf deze functie volgens hetzelfde principe als de functie `intString` in paragraaf 3.2.6, dat wil zeggen door samenstelling van een aantal partiële parametrisaties van `iterate`, `takeWhile` en `map`. blz. 49

3.19 Bekijk bomen van het volgende type:

```
data Boom2 a  =  Blad2 a
              |  Tak2 (Boom2 a) (Boom2 a)
```

Schrijf een functie `mapBoom` en een functie `foldBoom` die op een `Boom2` werken, naar analogie van de functies `map` en `foldr` op lijsten. Geef ook het type van deze functies.

3.20 Schrijf een functie `diepte`, die oplevert uit hoeveel nivo's een `Boom2` bestaat. Geef een definitie met inductie, en een alternatieve definitie waarin je `mapBoom` en `foldBoom` gebruikt.

3.21 Schrijf een functie `toonBoom`, die een aantrekkelijk representatie als string van een boom zoals gedefinieerd op blz. 58 geeft. In de string moet elk blad op een aparte regel komen te staan (gescheiden door `"\n"`); bladeren op een dieper nivo moeten meer zijn ingesprongen dan bladeren op een minder diep nivo.

3.22 Stel dat een boom b diepte n heeft. Wat is het minimale en het maximale aantal bladeren dat b kan bevatten?

3.23 Schrijf een functie die gegeven een gesorteerde lijst een zoekboom oplevert (dus een boom die als je er `labels` op toepast een gesorteerde lijst oplevert). Zorg ervoor dat de boom niet 'scheefgroeit', zoals het geval is als je de functie `lijstNaarBoom` zou gebruiken.

Hoofdstuk 4

Algoritmen op lijsten

4.1 Combinatorische functies

4.1.1 Segmenten en deelrijen

Combinatorische functies werken op een lijst. Ze leveren een lijst van lijsten op, waarbij geen gebruik gemaakt wordt van specifieke eigenschappen van de elementen van de lijst. Het enige wat combinatorische functies kunnen doen, is elementen weglaten, elementen verwisselen, of elementen tellen.

In deze en de volgende paragraaf worden een aantal combinatorische functies gedefinieerd. Omdat ze geen gebruik maken van eigenschappen van de elementen van hun parameter-lijst, zijn het polymorfe functies:

```
inits, tails, segs ::        [a] -> [[a]]
subs, perms        ::        [a] -> [[a]]
combs              :: Int -> [a] -> [[a]]
```

Om de werking van deze functies te illustreren volgen hieronder de uitkomsten van deze functies toegepast op de lijst `[1,2,3,4]`

`inits`	`tails`	`segs`	`subs`	`perms`	`combs 2`	`combs 3`
[]	[1,2,3,4]	[]	[]	[1,2,3,4]	[1,2]	[1,2,3]
[1]	[2,3,4]	[4]	[4]	[2,1,3,4]	[1,3]	[1,2,4]
[1,2]	[3,4]	[3]	[3]	[2,3,1,4]	[1,4]	[1,3,4]
[1,2,3]	[4]	[3,4]	[3,4]	[2,3,4,1]	[2,3]	[2,3,4]
[1,2,3,4]	[]	[2]	[2]	[1,3,2,4]	[2,4]	
		[2,3]	[2,4]	[3,1,2,4]	[3,4]	
		[2,3,4]	[2,3]	[3,2,1,4]		
		[1]	[2,3,4]	[3,2,4,1]		
		[1,2]	[1]	[1,3,4,2]		
		[1,2,3]	[1,4]	[3,1,4,2]		
		[1,2,3,4]	[1,3]	[3,4,1,2]		
			[1,3,4]	[3,4,2,1]		
			[1,2]	[1,2,4,3]		
			[1,2,4]	[2,1,4,3]		
			[1,2,3]	[2,4,1,3]		
			[1,2,3,4]	[2,4,3,1]		
				[1,4,2,3]		
				(7 andere)		

Zoals uit de voorbeelden waarschijnlijk al duidelijk is, is de betekenis van deze zes functies als volgt:

- `inits` levert alle *beginsegmenten* van een lijst, dat wil zeggen aaneengesloten stukken van de lijst die aan het begin beginnen. De lege lijst telt ook als beginsegment.
- `tails` levert alle *eindsegmenten* van een lijst: aaneengesloten stukken die tot het eind doorlopen. Ook de lege lijst is een eindsegment.
- `segs` levert *alle segmenten* van een lijst: beginsegmenten en eindsegmenten, maar ook aaneengesloten stukken uit het midden.
- `subs` levert alle *subsequences* (deelrijen) van een lijst. In tegenstelling tot segmenten hoeven de elementen van een deelrij in de origninele lijst niet aaneengesloten te zijn. Er zijn dus meer deelrijen dan segmenten.
- `perms` levert alle *permutaties* van een lijst. Een permutatie van een lijst bevat dezelfde elementen, maar mogelijk in een andere volgorde.
- `combs n` levert alle *combinaties van* `n` *elementen*, dus alle manieren om `n` elementen te kiezen uit een lijst. De volgorde is daarbij hetzelfde als in de originele lijst.

Deze combinatorische functies kunnen recursief worden gedefinieerd. In de definitie worden dus steeds de gevallen `[]` en `(x:xs)` apart behandeld. In het geval `(x:xs)` wordt de functie recursief aangeroepen op de lijst `xs`.

Er is een handige manier om op een idee te komen van de definitie van een functie `f`. Kijk bij een voorbeeldlijst `(x:xs)` wat het resultaat is van de recursieve aanroep `f xs`, en probeer het resultaat aan te vullen tot de uitkomst van `f (x:xs)`.

inits

Bij de beschrijving van beginsegmenten hierboven is ervoor gekozen om de lege lijst ook als beginsegment te laten tellen. De lege lijst heeft dus één beginsegment: de lege lijst zelf. De definitie van `inits` voor het geval 'lege lijst' is daarom als volgt:

```
inits []  =  [ [] ]
```

Voor het geval `(x:xs)` kijken we naar de gewenste uitkomsten voor de lijst `[1,2,3,4]`.

```
inits [1,2,3,4]   =  [ [] , [1] , [1,2] , [1,2,3] , [1,2,3,4] ]
inits [2,3,4]     =  [ [] , [2] , [2,3] , [2,3,4] ]
```

Hieruit blijkt dat de tweede t/m vijfde elementen van `inits [1,2,3,4]` overeenkomen met de elementen van `inits [2,3,4]`, alleen steeds met een extra `1` op kop. Deze vier lijsten moeten dan nog worden aangevuld met een lege lijst.

Dit mechanisme wordt algemeen beschreven in de tweede regel van de definitie van `inits`:

```
inits []      =  [ [] ]
inits (x:xs)  =  [] : map (x:) (inits xs)
```

tails

Net als bij `inits` heeft de lege lijst één eindsegment: de lege lijst. Het resultaat van `tails []` is dus een lijst met als enige element de lege lijst.

Om op een idee te komen voor de definitie van `tails (x:xs)` kijken we eerst weer naar het voorbeeld `[1,2,3,4]`:

```
tails [1,2,3,4]   =  [ [1,2,3,4] , [2,3,4] , [3,4] , [4] , [] ]
tails [2,3,4]     =  [ [2,3,4] , [3,4] , [4] , [] ]
```

Bij deze functie zijn het tweede t/m vijfde element dus precies gelijk aan de elementen van de recursieve aanroep. Het enige wat moet gebeuren is uitbreiding met een eerste element ([1,2,3,4] in het voorbeeld).

Gebruik makend van dit idee kan de complete definitie geschreven worden:

```
tails []      =  [ [] ]
tails (x:xs)  =  (x:xs) : tails xs
```

De haakjes in de tweede regel zijn essentieel: zonder haakjes zou de typering incorrect zijn, omdat de operator `:` dan naar rechts associeert.

segs

Het enige segment van de lege lijst is weer de lege lijst. De uitkomst van `segs []` is dus, net als bij `inits` en `tails`, een singleton-lege-lijst.

Om op het spoor van de definitie van `segs (x:xs)` te komen, passen we de beproefde methode weer toe:

```
segs [1,2,3,4]   =  [ [] , [4] , [3] , [3,4] , [2] , [2,3] , [2,3,4] , [1] , [1,2] , [1,2,3] , [1,2,3,4] ]
segs [2,3,4]     =  [ [] , [4] , [3] , [3,4] , [2] , [2,3] , [2,3,4] ]
```

Als je ze maar op de goede volgorde zet, blijkt dat de eerste zeven elementen van het gewenste resultaat precies overeenkomen met de recursieve aanroep. In het tweede deel van het resultaat (de lijsten die met een 1 beginnen) zijn de beginsegmenten van [1,2,3,4] te herkennen (alleen de lege lijst is daarbij weggelaten, want die zit al in het resultaat).

Als definitie van `segs` kan dus genomen worden:

```
segs []      =  [ [] ]
segs (x:xs)  =  segs xs  ++  tail (inits (x:xs))
```

Een andere manier om de lege lijst uit de `inits` te verwijderen is:

```
segs []      =  [ [] ]
segs (x:xs)  =  segs xs  ++  map (x:) (inits xs)
```

subs

De lege lijst is de enige deelrij van de lege lijst. Voor de definitie van `subs (x:xs)` kijken we weer naar het voorbeeld:

```
subs [1,2,3,4]  =  [ [1,2,3,4] , [1,2,3] , [1,2,4] , [1,2] , [1,3,4] , [1,3] , [1,4] , [1]
                   , [2,3,4] , [2,3] , [2,4] , [2] , [3,4] , [3] , [4] , [] ]
subs [2,3,4]    =  [ [2,3,4] , [2,3] , [2,4] , [2] , [3,4] , [3] , [4] , [] ]
```

Het aantal elementen van `subs (x:xs)` (16 in het voorbeeld) is precies twee keer zo groot als het aantal elementen van de recursieve aanroep `subs xs`. De tweede helft van het totaal resultaat is precies gelijk aan het resultaat van de recursieve aanroep. Ook in de eerste helft zijn deze 8 lijsten weer te herkennen, alleen staat er daar steeds een `1` op kop.

De definitie kan dus luiden:

```
subs []      =  [ [] ]
subs (x:xs)  =  map (x:) (subs xs)  ++  subs xs
```

De functie wordt tweemaal recursief aangeroepen met dezelfde parameter. Dat is zonde van het werk: beter kan de aanroep maar éénmaal gedaan worden, waarna het resultaat tweemaal gebruikt wordt. Dit levert veel tijdwinst op, want ook voor het bepalen van `subs xs` wordt de functie weer tweemaal recursief aangeroepen, en in die recursieve aanroepen weer... Een veel efficiëntere definitie is dus:

```
subs []      =  [ [] ]
subs (x:xs)  =  map (x:) subsxs  ++  subsxs
          where  subsxs = subs xs
```

4.1.2 Permutaties en combinaties

perms

Een *permutatie* van een lijst is een lijst met dezelfde elementen, maar mogelijk in een andere volgorde. De lijst van alle permutaties van een lijst kan goed met een recursieve functie worden gedefinieerd.

De lege lijst heeft één permutatie: de lege lijst. Alle 0 elementen zitten daar namelijk in, en in dezelfde volgorde...

Het interessante geval is natuurlijk de niet-lege lijst `(x:xs)`. We kijken eerst weer naar een voorbeeld:

```
perms [1,2,3,4]  =  [ [1,2,3,4] , [2,1,3,4] , [2,3,1,4] , [2,3,4,1]
                    , [1,3,2,4] , [3,1,2,4] , [3,2,1,4] , [3,2,4,1]
                    , [1,3,4,2] , [3,1,4,2] , [3,4,1,2] , [3,4,2,1]
                    , [1,2,4,3] , [2,1,4,3] , [2,4,1,3] , [2,4,3,1]
                    , [1,4,2,3] , [4,1,2,3] , [4,2,1,3] , [4,2,3,1]
                    , [1,4,3,2] , [4,1,3,2] , [4,3,1,2] , [4,3,2,1] ]
perms [2,3,4]    =  [ [2,3,4] , [3,2,4] , [3,4,2] , [2,4,3] , [4,2,3] , [4,3,2] ]
```

Het aantal permutaties loopt flink op: van een lijst met vier elementen zijn er viermaal zoveel permutaties als van een lijst met drie elementen. In dit voorbeeld is het wat moeilijker om het resultaat van de recursieve aanroep te herkennen. Dit lukt pas door de 24 elementen in 6 groepjes van 4 te verdelen. In elke groepje zitten lijsten met dezelfde waarden als die van één lijst van de recursieve aanroep. Het nieuwe element wordt daar op alle mogelijke manieren tussen gezet.

Bijvoorbeeld, het derde groepje [[1,3,4,2], [3,1,4,2], [3,4,1,2], [3,4,2,1]] bevat steeds de elementen [3,4,2], waarbij het element 1 is toegevoegd respectievelijk aan het begin, op de tweede plaats, op de derde plaats, en aan het eind.

Voor het op alle manieren tussenvoegen van één element in een lijst kan een hulpfunctie worden geschreven, die ook weer recursief gedefinieerd is:

```
tussen          ::  a -> [a] -> [[a]]
tussen e []      =  [ [e] ]
tussen e (y:ys)  =  (e:y:ys)  :  map (y:) (tussen e ys)
```

In de definitie van `perms (x:xs)` wordt deze functie, partiëel geparametriseerd met `x`, toegepast op alle elementen van het resultaat van de recursieve aanroep. In het voorbeeld levert dat een lijst met zes lijsten van vier lijstjes. De bedoeling is echter dat er één lange lijst van 24 lijstjes uitkomt. Op de lijst van lijsten van lijstjes moet dus nog de functie `concat` worden toegepast, die immers dat effect heeft.

Al met al wordt de functie `perms` als volgt gedefinieerd:

```
perms []      =  [ [] ]
perms (x:xs)  =  concat (map (tussen x) (perms xs))
           where  tussen e []     = [ [e] ]
                  tussen e (y:ys) = (e:y:ys) : map (y:) (tussen e ys)
```

combs

Een laatste voorbeeld van een combinatorische functie is de functie `combs`. Deze functie heeft, behalve de lijst, ook een getal als parameter:

```
combs :: Int -> [a] -> [[a]]
```

De bedoeling is dat in het resultaat van `combs n xs` alle deelrijen van `xs` met lengte `n` zitten. De functie kan dus eenvoudigweg gedefinieerd worden door

```
combs n xs  =  filter goed (subs xs)
        where  goed xs  =  length xs == n
```

Deze definitie is echter niet zo erg efficiënt. Het aantal deelrijen is namelijk meestal erg groot, dus `subs` kost veel tijd, terwijl de meeste deelrijen door `filter` weer worden weggegooid. Een betere definitie is te verkrijgen door `combs` direct te definiëren, zonder `subs` te gebruiken.

In de definitie van `combs` worden voor de integer-parameter de gevallen `0` en `n+1` onderscheiden. In het geval `n+1` worden ook voor de lijst-parameter twee gevallen onderscheiden. De definitie krijgt dus de volgende vorm:

```
combs 0     xs     =  ...
combs (n+1) []     =  ...
combs (n+1) (x:xs) =  ...
```

Deze drie gevallen worden hieronder apart bekeken.

- Voor het kiezen van nul elementen uit een lijst is er één mogelijkheid: de lege lijst. Het resultaat van `combs 0 xs` is daarom een singleton-lege-lijst. Het maakt daarbij niet uit of `xs` leeg is of niet.
- Het patroon `n+1` betekent '1 of meer'. Het kiezen van minstens één element uit de lege lijst is onmogelijk. Het resultaat van `combs (n+1) []` is dan ook de lege lijst: er is geen oplossing mogelijk. Let wel: hier is dus sprake van een *lege lijst oplossingen* en niet van *de lege lijst als enige oplossing* zoals in het vorige geval. Een belangrijk verschil!
- Het kiezen van `n+1` elementen uit de lijst `x:xs` is wel mogelijk, mits het kiezen van `n` elementen uit `xs` mogelijk is. De oplossingen zijn te verdelen in twee groepen: lijstjes waar `x` in zit, en lijstjes waar `x` niet in zit.
  - Voor de lijstjes waar `x` wel in zit, moeten uit de overige elementen `xs` nog `n` elementen gekozen worden. Daarin moet dan steeds `x` op kop gezet worden.
  - Voor de lijstjes waar `x` niet in zit, moeten alle `n+1` elementen uit `xs` gekozen worden.

  Voor beide gevallen kan `combs` recursief aangeroepen worden. De resultaten kunnen gecombineerd worden met `++`.

De definitie van `combs` komt er dus als volgt uit te zien:

```
combs 0     xs     =  [ [] ]
combs (n+1) []     =  [ ]
combs (n+1) (x:xs) =  map (x:) (combs n xs)  ++  combs (n+1) xs
```

Bij deze functie kunnen niet, zoals bij `subs`, de twee recursieve aanroepen gecombineerd worden. De twee aanroepen hebben hier namelijk verschillende parameters.

4.1.3 De @-notatie

De definitie van `tails` heeft iets omslachtigs:

```
tails []      =  [ [] ]
tails (x:xs)  =  (x:xs) : tails xs
```

In de tweede regel wordt de parameter-lijst door het patroon gesplitst in een kop `x` en een staart `xs`. De staart wordt gebruikt bij de recursieve aanroep, maar de kop en de staart worden ook weer samengevoegd tot de lijst `(x:xs)`. Dat is zonde van het werk, want deze lijst is in feite ook al beschikbaar als parameter.

Een andere definitie van `tails` zou kunnen luiden:

```
tails []  =  [ [] ]
tails xs  =  xs : tails (tail xs)
```

Nu is het opnieuw opbouwen van de parameter-lijst niet nodig, omdat hij helemaal niet gesplitst wordt. Maar nu moet, om de staart bij de recursieve aanroep te kunnen meegeven, expliciet de functie `tail` worden gebruikt. Het leuke van patronen was nu juist, dat dat niet nodig is.

Ideaal zou het zijn om het goede van deze twee definities te combineren. De parameter moet dus zowel als geheel beschikbaar zijn, als gesplitst in een kop en een staart. Voor deze situatie is een speciale notatie beschikbaar. Vóór een patroon mag een naam worden geschreven die het geheel aanduidt. De naam wordt van het patroon gescheiden door het symbool `@`.

Met gebruik van deze constructie wordt de definitie van `tails` als volgt:

```
tails []           =  [ [] ]
tails lyst@(x:xs)  =  lyst : tails xs
```

Hoewel het symbool `@` in operator-symbolen gebruikt mag worden, is een losse `@` speciaal gereserveerd voor deze constructie.

Bij functies die al eerder in dit diktaat gedefinieerd werden, komt een @-patroon ook goed van pas. Bijvoorbeeld in de functie `dropWhile`:

```
dropWhile p []       = []
dropWhile p ys@(x:xs)
          | p x      = dropWhile p xs
          | otherwise = ys
```

Dit is ook de manier waarop `dropWhile` in werkelijkheid in de prelude is gedefinieerd.

4.2 Matrixrekening

4.2.1 Vectoren en matrices

Matrixrekening is een tak van wiskunde die zich bezighoudt met lineaire afbeeldingen in meerdimensionale ruimtes. In deze paragraaf worden de belangrijkste begrippen uit de matrixrekening ingevoerd. Verder wordt aangegeven hoe deze begrippen in Haskell als type of functie gemodelleerd kunnen worden. De Haskell-definitie van de functies volgt in de volgende twee paragrafen.

De generalisatie van de een-dimensionale lijn, het twee-dimensionale platte vlak en de driedimensionale ruimte is de n-dimensionale ruimte. In een n-dimensionale ruimte kan elk 'punt' aangeduid worden door n getallen. Zo'n aanduiding wordt ook wel een *vector* genoemd. In Haskell zou een vector gerepresenteerd kunnen worden als element van het volgende type:

```
type Vector  =  [Float]
```

Om achter getalconstanten niet steeds `.0` te hoeven schrijven om de type-checker tevreden te stellen, zullen we in deze sectie veronderstellen dat de `+i` optie van de interpreter aan staat, zodat `Int`-getallen zonodig automatisch naar `Float` geconverteerd worden.

Het aantal elementen in de lijst die een `Vector` voorstelt bepaalt de dimensie van de ruimte. Om geen verwarring te krijgen met andere lijsten-van-floats, die geen vectoren voorstellen, is het beter om een 'beschermd type' te definiëren zoals beschreven in paragraaf 3.4.3: blz. 64

```
data Vector  =  Vec [Float]
```

Op een lijst getallen moet dus de constructorfunctie `Vec` worden toegepast om er een `Vector` van te maken.

In plaats van als *punt* in de (n-dimensionale) ruimte kan een vector ook worden beschouwd als *(gericht) lijnstuk* van de oorsprong naar dat punt. Een nuttige functie bij het werken met vectoren is het bepalen van de afstand van een punt tot de oorsprong, of, in de lijnstuk-interpretatie, de *lengte* van het lijnstuk:

```
vecLengte    :: Vector -> Float
```

Bij twee vectoren (in dezelfde ruimte) kan bepaald worden of ze *loodrecht* op elkaar staan, en meer algemeen welke *hoek* ze met elkaar maken:

```
vecLoodrecht :: Vector -> Vector -> Bool
vecHoek      :: Vector -> Vector -> Float
```

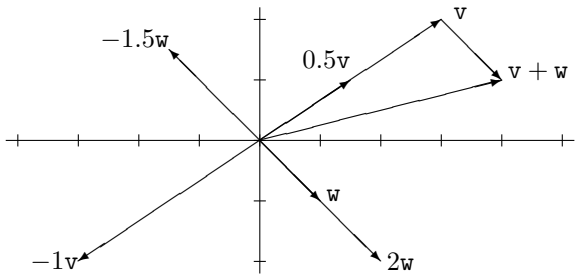
Verder kan een vector met een getal worden 'vermenigvuldigd' door alle getallen in de lijst (alle coördinaten) met dat getal te vermenigvuldigen. Twee vectoren worden 'opgeteld' door alle coördinaten op te tellen:

```
vecScale     :: Float  -> Vector -> Vector
vecPlus      :: Vector -> Vector -> Vector
```

In de volgende paragraaf zullen deze functies geschreven worden. In de gericht-lijnstuk-interpretatie van vectoren hebben deze functies de volgende meetkundige interpretatie:

- `vecScale`: de vector blijft in dezelfde richting wijzen, maar wordt 'verlengd' met een factor volgens het aangegeven getal. Als de absolute waarde van dit getal < 1 is wordt de vector verkort; als het getal < 0 is gaat de vector de andere kant op wijzen.
- `vecPlus`: de twee vectoren worden 'kop aan staart' gelegd, en wijzen zo een nieuw punt aan (de volgorde maakt daarbij niet uit).

Als voorbeeld bekijken we een paar vectoren in de 2-dimensionale ruimte:



`v`	=	$[3, 2]$
`w`	=	$[1, -1]$
`vecPlus v w`	=	$[4, 1]$
`vecScale 0 v`	=	$[0, 0]$
`vecScale 0.5 v`	=	$[1.5, 1]$
`vecScale -1 v`	=	$[-3, -2]$
`vecScale 2 w`	=	$[2, -2]$
`vecScale -1.5 w`	=	$[-1.5, 1.5]$

Functies van vectoren naar vectoren heten *afbeeldingen*. Van bijzonder belang zijn *lineaire* afbeeldingen. Dat zijn afbeeldingen waarbij elke coördinaat van de beeld-vector een lineaire combinatie is van coördinaten van het origineel.

In de 2-dimensionale ruimte kan elke lineaire afbeelding geschreven worden als

$$f\ (x, y)\ =\ (\ a * x + b * y\ ,\ c * x + d * y\ )$$

waarbij a, b, c en d vrij gekozen kunnen worden. De n^2 getallen die een lineaire afbeelding in de n-dimensionale ruimte beschrijven, worden in de wiskunde als rechthoekig blok getallen met haken eromheen geschreven. Bijvoorbeeld:

$$\begin{pmatrix} \frac{1}{2}\sqrt{3} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2}\sqrt{3} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

beschrijft een lineaire afbeelding in de 3-dimensionale ruimte (rotatie over 30° om de z-as). Zo'n blok getallen heet een *matrix* (meervoud: *matrices*).

In Haskell kan een matrix gerepresenteerd worden door een lijst van lijsten. We maken er maar meteen een beschermd (data)type van:

```
data Matrix  =  Mat [[Float]]
```

Daarbij moet gekozen worden of de lijsten de rijen of de kolommen van de matrix voorstellen. In dit diktaat is voor de rijen gekozen (dat is nu de meest logische keuze, omdat elke rij met één vergelijking in de lineaire afbeelding overeenkomt).

Mocht de kolom-representatie nodig zijn, dan is er een functie die de rijen van een matrix tot kolommen maakt en andersom. Dit heet *transponeren* van een matrix. De functie die dat doet heeft als type:

```
matTransp  ::  Matrix -> Matrix
```

Er geldt dus bijvoorbeeld

$$\texttt{matTransp}\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

De belangrijkste functie die op matrices werkt, is de functie die de lineaire afbeelding uitvoert. Dit wordt wel het *toepassen* van een matrix op een vector genoemd. Het type van deze functie is:

```
matApply  ::  Matrix -> Vector -> Vector
```

De samenstelling van twee lineaire afbeeldingen is weer een lineaire afbeelding. Bij twee matrices horen twee lineaire afbeeldingen; de samenstelling van deze afbeeldingen wordt weer beschreven door een matrix. Het bepalen van deze matrix wordt het *vermenigvuldigen* van matrices genoemd. Er is dus een functie:

```
matProd  ::  Matrix -> Matrix -> Matrix
```

Net als functiesamenstelling (de operator `(.)`) is de functie `matProd` associatief (dus $A \times (B \times C) = (A \times B) \times C$). Matrixvermenigvuldiging is echter niet commutatief ($A \times B$ is niet altijd gelijk aan $B \times A$). Het matrixproduct $A \times B$ is de afbeelding die eerst matrix B toepast, en dan matrix A.

De identieke afbeelding is ook een lineaire afbeelding. Hij wordt beschreven door de *identiteitsmatrix*. Dit is een matrix waarin het getal 1 staat op de diagonaal, en het getal 0 op de andere plaatsen. Voor elke dimensie is er zo'n identiteitsmatrix, die wordt bepaald door de volgende functie:

```
matId  ::  Int -> Matrix
```

Sommige lineaire afbeeldingen zijn *inverteerbaar*. De inverse afbeelding (als die bestaat) is ook lineair, en wordt dus beschreven door een matrix:

```
matInv  ::  Matrix -> Matrix
```

De dimensie van het beeld van een lineaire afbeelding hoeft niet hetzelfde te zijn als die van het origineel. Een afbeelding van de 3-dimensionale ruimte naar het 2-dimensionale vlak wordt bijvoorbeeld beschreven door een matrix met de vorm

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$$

Een afbeelding van een p-dimensionale naar een q-dimensionale ruimte heeft dus p kolommen en q rijen. Bij het samenstellen van afbeeldingen (vermenigvuldigen van matrices) moeten deze afmetingen kloppen. In het matrixproduct $A \times B$ moet het aantal kolommen van A gelijk zijn aan het aantal rijen van B. Het aantal kolommen van A is immers de dimensie van het origineel van de afbeelding A; deze moet gelijk zijn aan de dimensie van het beeld van de afbeelding B. De samenstelling $A \times B$ heeft hetzelfde origineel als B, en dus evenveel kolommen als B; het heeft hetzelfde beeld als A, en dus evenveel rijen als A. Bijvoorbeeld:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 2 & 2 \\ 2 & 3 & 4 & 1 \\ 1 & 3 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 16 & 22 & 10 \\ 2 & 5 & 8 & 5 \end{pmatrix}$$

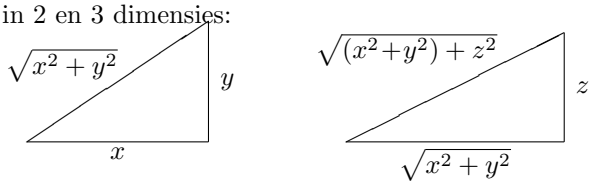
Het is duidelijk dat de begrippen 'identiteitsmatrix' en 'inverse' alleen zin hebben voor vierkante matrices, dus matrices met dezelfde origineel- en beeldruimte. En zelfs voor vierkante matrices is de inverse niet altijd gedefinieerd. De matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$ bijvoorbeeld heeft geen inverse.

4.2.2 Elementaire operaties

In deze en de volgende paragraaf worden de definities gegeven van een aantal functies op vectoren en matrices.

Lengte van een vector

De lengte van een vector wordt bepaald volgens de stelling van Pythagoras. De volgende figuren illustreren de situatie in 2 en 3 dimensies:

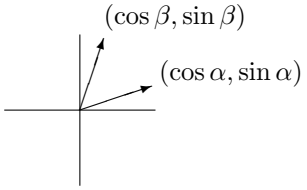


In het algemeen (willekeurige dimensie) kan de lengte van een vector uitgerekend worden door de wortel van de som van de kwadraten van de coördinaten te berekenen. De functie luidt dus:

```
vecLengte (Vec xs) = sqrt (sum (map kwadraat xs))
```

Hoek van twee vectoren

Bekijk twee vectoren met lengte 1. De eindpunten van deze vectoren liggen dus op de eenheidscirkel. Als de hoek die deze vectoren met de x-as maken respectievelijk α en β is, dan zijn de coördinaten van het eindpunt respectievelijk $(\cos\alpha, \sin\alpha)$ en $(\cos\beta, \sin\beta)$.



De hoek die de twee vectoren met elkaar maken is $\beta - \alpha$. Voor het verschil van twee hoeken geldt de volgende rekenregel:

$$\cos(\beta - \alpha) \;=\; \cos\alpha\ \cos\beta \;+\; \sin\alpha\ \sin\beta$$

In het geval van de twee vectoren is de cosinus van de ingesloten hoek dus gelijk aan de som van de producten van overeenkomstige coördinaten (dit is ook zo in hogere dimensies dan 2). Deze formule is zo belangrijk, dat hij een aparte naam heeft gekregen: het *inwendig product* van twee vectoren (of kortweg *inproduct*). De waarde wordt berekend door de volgende functie:

```
vecInprod (Vec xs) (Vec ys)  =  sum (zipWith (*) xs ys)
```

Voor vectoren met een andere lengte dan 1 moet het inproduct door de lengte gedeeld worden om de cosinus van de hoek te bepalen. De hoek kan dus als volgt berekend worden:

```
vecHoek v w  =  acos (vecInprod v w / (vecLengte v * vecLengte w))
```

De functie `acos` is de inverse van de cosinus. Als hij niet ingebouwd zou zijn, kon hij berekend
blz. 34 worden met de functie `inverse` uit paragraaf 2.4.5.

De cosinus van zowel $90°$ als $-90°$ is 0. Om te bepalen of twee vectoren loodrecht op elkaar staan, hoeft de `arccos` helemaal niet berekend te worden: het inproduct volstaat. Dit hoeft zelfs niet door de lengtes van de vectoren gedeeld te worden, omdat alleen het nul-zijn van belang is:

```
vecLoodrecht v w  =  vecInprod v w == 0
```

Vectoren optellen en verlengen

De functies `vecScale` en `vecPlus` zijn eenvoudige toepassingen van de standaardfuncties `map` en `zipWith`:

```
vecScale  ::  Float -> Vector -> Vector
vecScale k (Vec xs)  =  Vec (map (k*) xs)
vecPlus   ::  Vector -> Vector -> Vector
vecPlus (Vec xs) (Vec ys)  =  Vec (zipWith (+) xs ys)
```

Soortgelijke functies zijn ook op matrices van nut. Het is handig om eerst twee functies te maken die werken zoals `map` en `zipWith`, maar dan op de elementen van *lijsten van* lijsten. Deze functies zullen we `mapp` en `zippWith` noemen (dit zijn geen standaardfuncties).

Om een functie toe te passen op alle elementen van een lijst van lijstjes, moet 'het toepassen op alle elementen van een lijstje' worden toegepast op alle elementen van de grote lijst. Dus `map f` moet worden toegepast op alle elementen van de grote lijst:

```
mapp  :: (a->b) -> [[a]] -> [[b]]
mapp f  =  map (map f)
```

Anders gezegd:

```
mapp  =  map . map
```

Voor `zippWith` geldt iets dergelijks:

```
zippWith :: (a->b->c) -> [[a]] -> [[b]] -> [[c]]
zippWith  =  zipWith . zipWith
```

Deze functies kunnen gebruikt worden in `matScale` en `matPlus`:

```
matScale  ::   Float -> Matrix -> Matrix
matScale k (Mat xss) =  Mat (mapp (k*) xss)
matPlus   ::  Matrix -> Matrix -> Matrix
matPlus (Mat xss) (Mat yss)  =  Mat (zippWith (+) xss yss)
```

Matrices transponeren

Een getransponeerde matrix is een matrix waarvan de rijen en de kolommen zijn omgewisseld. Deze operatie is ook op gewone lijsten van lijsten (zonder de constructor `Mat`) van belang. Daarom schrijven we eerst een functie

```
transpose  ::  [[a]] -> [[a]]
```

Daarna is `matTransp` eenvoudig:

```
matTransp  (Mat xss)  =  Mat (transpose xss)
```

De functie `transpose` is een generalisatie van `zip`. Waar `zip` *twee* lijsten aan elkaar ritst tot lijst van *tweetallen*, ritst `transpose` een *lijst van* lijsten aan elkaar tot een lijst van *lijsten*.

De functie kan recursief worden gedefinieerd, maar eerst bekijken we een voorbeeld. Er moet gelden:

`transpose` [[1,2,3] , [4,5,6] , [7,8,9] , [10,11,12]] = [[1,4,7,10] , [2,5,8,11] , [3,6,9,12]]

Als de lijst van lijsten maar uit één rij bestaat, is de functie eenvoudig: de rij van n elementen worden n kolommetjes van ieder één element. Dus:

```
transpose [rij]  =  map singleton rij
             where  singleton x = [x]
```

Voor het recursieve geval gaan we ervan uit dat de getransponeerde van alles behalve de eerste rij al bepaald is. Dus in het voorbeeld van zoëven:

`transpose` [[4,5,6] , [7,8,9] , [10,11,12]] = [[4,7,10] , [5,8,11] , [6,9,12]]

Hoe moet de eerste rij [1,2,3] nu gecombineerd worden met deze deel-oplossing tot een totale oplossing? De elementen ervan moeten steeds op kop gezet worden van de recursieve oplossing. Dus 1 komt op kop van [4,7,10], 2 komt op kop van [5,8,11], en 3 komt op kop van [6,9,12]. Dit kan eenvoudig met `zipWith`, als volgt:

```
transpose (xs:xss)  =  zipWith (:) xs (transpose xss)
```

Hiermee is de functie `transpose` gedefinieerd. Hij kan alleen niet toegepast worden op een matrix met nul rijen, maar dat is ook een beetje onzinnig geval.

Niet-recursieve matrix transponering

Er is ook een niet-recursieve definitie van `transpose` mogelijk, die het 'vuile werk' laat opknappen door de standaardfunctie `foldr`. De definitie heeft namelijk de vorm

```
transpose (y:ys)  =  f y (transpose ys)
```

(met voor `f` de partiëel geparametriseerde functie `zipWith (:)`). Functies die deze vorm hebben zijn een speciaal geval van `foldr` (zie paragraaf 2.3.1). Als functie-parameter van `foldr` kan `f`, dat wil zeggen `zipWith (:)`, genomen worden. Blijft de vraag wat het 'neutrale element' is, dat wil zeggen het resultaat van `transpose []`.

blz. 26

Een 'lege matrix' kan beschouwd worden al matrix met 0 rijen van ieder n elementen. De getransponeerde daarvan is een matrix met n rijen van ieder 0 elementen, dus een lijst met daarin n lege

lijstjes. Maar hoe groot is n? Er zijn slechts nul rijen beschikbaar, dus we kunnen niet even kijken hoe lang de eerste rij is. Om geen risico te lopen dat we n te klein kiezen, nemen we n oneindig groot: de getransponeerde van een matrix met 0 rijen van ∞ elementen is een matrix met ∞ rijen van 0 elementen. De functie `zipWith` zorgt er later wel voor dat deze oneindige lijst wordt ingekort op de gewenste lengte (het resultaat van `zipWith` op twee lijsten heeft de lengte van de kortste).

Deze wat ingewikkelde redenering levert een zeer elegante definitie voor `transpose` op:

```
transpose  =  foldr f e
       where  f = zipWith (:)
              e = repeat []
```

of zelfs eenvoudigweg

```
transpose  =  foldr (zipWith (:)) (repeat [])
```

Functioneel programmeren is programmeren met functies...

Matrix op een vector toepassen

Een matrix is een representatie van een lineaire afbeelding tussen vectoren. De functie `matApply` voert deze lineaire afbeelding uit. Bijvoorbeeld:

`matApply` (`Mat` [[1,2,3] , [4,5,6]]) (`Vec` $[x, y, z]$) = `Vec` [$1x+2y+3z$, $4x+5y+6z$]

Het aantal *kolommen* van de matrix is gelijk aan het aantal coördinaten van de *originele* vector; het aantal *rijen* van de matrix is gelijk aan de dimensie van de *beeld*vector.

Voor elke coördinaat van de beeldvector zijn alleen maar de getallen uit de overeenkomstige rij van de matrix nodig. Het ligt dus voor de hand om `matApply` te schrijven als de `map` van een of andere functie op de (rijen van de) matix:

```
matApply (Mat m) v  =  Vec (map f m)
```

De functie `f` werkt daarbij op één rij van de matrix, en levert één element van de beeldvector op. Bijvoorbeeld op de tweede rij:

`f` $[4, 5, 6] \;=\; 4x + 5y + 6z$

De functie `f` berekent dus het inproduct van zijn parameter met de vector `v` ($[x, y, z]$ in het voorbeeld). De complete functie `matApply` is dus:

```
matApply :: Matrix -> Vector -> Vector
matApply (Mat m) v  =  Vec (map f m)
                 where  f rij = vecInprod (Vec rij) v
```

Twee dingen om op te letten in deze definitie:

- De constructorfunctie `Vec` wordt op de juiste plaatsen toegepast om het type correct te krijgen. De functie `vecInprod` verwacht twee vectoren. De parameter `v` is al een vector, maar `rij` is een ordinaire lijst, waarvan met `Vec` eerst een vector gemaakt moet worden.
- De functie `f` mag, behalve van zijn parameter `rij`, ook gebruik maken van `v`. Lokale functies mogen altijd gebruik maken van de parameters van de functie waarbinnen ze gedefinieerd worden.

De identiteitsmatrix

In elke dimensie is er een identieke afbeelding. Deze wordt beschreven door een vierkante matrix met een 1 op de diagonaal, en een 0 op alle andere plaatsen. Om een functie te schrijven die voor elke dimensie de juiste identiteitsmatrix oplevert, is het handig om eerst een oneindig grote identiteitsmatrix te definiëren, dus de matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ \vdots & & & & \ddots \end{pmatrix}$$

De eerste rij daarvan is een oneindige lijst nullen met een 1 op kop, dus `1:repeat 0`. De overige rijen worden bepaald door steeds een extra 0 op kop te zetten. Dit kan met de functie `iterate`:

```
matIdent ::  Matrix
matIdent  =  Mat (iterate (0:) (1:repeat 0))
```

Een identiteitsmatrix van dimensie n is nu te verkrijgen door n rijen van deze oneindige matrix te nemen, en elke rij af te breken op n elementen:

```
matId :: Int -> Matrix
matId n  =  Mat( map (take n) (take n xss))
     where  (Mat xss) = matIdent
```

Matrixvermenigvuldiging

Het product van twee matrices beschrijft de afbeelding die de samenstelling is van de afbeeldingen die bij de twee matrices horen. Om te bekijken hoe het product berekend kan worden, passen we de matrices $\binom{a\ b}{c\ d}$ en $\binom{e\ f}{g\ h}$ na elkaar toe op de vector $\binom{x}{y}$. (We noteren $\otimes$ voor matrixproduct en $\odot$ voor toepassing van een matrix op een vector. Let op het verschil tussen matrices en vectoren.)

$$
\begin{aligned}
& \left(\begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} e & f \\ g & h \end{pmatrix}\right) \odot \begin{pmatrix} x \\ y \end{pmatrix} \\
= \ & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \left(\begin{pmatrix} e & f \\ g & h \end{pmatrix} \odot \begin{pmatrix} x \\ y \end{pmatrix}\right) \\
= \ & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \odot \begin{pmatrix} ex+fy \\ gx+hy \end{pmatrix} \\
= \ & \begin{pmatrix} a(ex+fy)+b(gx+hy) \\ c(ex+fy)+d(gx+hy) \end{pmatrix} \\
= \ & \begin{pmatrix} (ae+bg)x+(af+bh)y \\ (ce+dg)x+(cf+dh)y \end{pmatrix} \\
= \ & \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix} \odot \begin{pmatrix} x \\ y \end{pmatrix}
\end{aligned}
$$

Het product van twee matrices wordt dus als volgt berekend: elk element is het *inproduct* tussen een *rij* van de linker matrix en een *kolom* van de rechter matrix. De lengte van een rij (het aantal kolommen) van de linker matrix moet gelijk zijn aan de lengte van een kolom (het aantal rijen) van de rechter matrix. (Dat was aan het eind van de vorige paragraaf ook al opgemerkt).

Blijft de vraag hoe matrixvermenigvuldiging als functie geschreven kan worden. Daartoe kijken we nog eens naar het voorbeeld uit de vorige paragraaf:

$$
\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 2 & 2 \\ 2 & 3 & 4 & 1 \\ 1 & 3 & 4 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 16 & 22 & 10 \\ 2 & 5 & 8 & 5 \end{pmatrix}
$$

Het aantal rijen van het resultaat is hetzelfde als het aantal rijen van de linker matrix. We proberen `matProd` daarom te schrijven als `map` op de linker matrix:

```
matProd (Mat a) (Mat b)  =  Mat (map f a)
```

Daarbij werkt de functie `f` op één rij van de linker matrix, en levert één rij van het resultaat. Bijvoorbeeld de tweede rij:

```
f [2,1,0] = [2,5,8,5]
```

Hoe worden de getallen `[2,5,8,5]` berekend? Door het inproduct van `[2,1,0]` met alle *kolommen* van de rechter matrix te bepalen.

Matrices worden echter opgeslagen als *rijen*. Om de kolommen te krijgen, moet de `transpose`-functie worden toegepast. Het resultaat is dus:

```
matProd (Mat a) (Mat b)  =  Mat (map f a)
                       where  f aRij  = map (vecInprod (Vec aRij)) bKols
                              bKols   = map Vec (transpose b)
```

De functie `transpose` werkt op een 'kale' lijst van lijsten (dus niet op matrices). Hij levert weer een lijst van lijsten op. Met `map Vec` wordt daar een lijst van vectoren van gemaakt. Van al deze vectoren kan het inproduct met de rijen van `a` (beschouwd als vectoren) berekend worden.

4.2.3 Determinant en inverse

Nut van de determinant

Alleen bijectieve (één-op-één en 'op') afbeeldingen zijn inverteerbaar. Als er beeldpunten zijn met meer dan één origineel, is het namelijk onduidelijk waarheen de inverse afbeelding hem moet terugsturen. Ook als er punten in de beeldruimte zijn zonder origineel, kan de inverse afbeelding niet worden gedefinieerd.

Als een matrix niet vierkant is, is hij dus niet inverteerbaar (de beeldruimte heeft dan immers een lagere dimensie (waardoor er beeldpunten zijn met meer originelen) of een hogere dimensie (waardoor er beeldpunten zijn zonder origineel)). Maar zelfs vierkante matrices stellen niet altijd bijectieve afbeeldingen voor. De matrix $\binom{0\ 0}{0\ 0}$ beeldt bijvoorbeeld elk punt af op de oorsprong, en heeft dus geen inverse. De matrix $\binom{1\ 1}{2\ 2}$ beeldt elk punt af op een punt van de lijn $y = 2x$, en heeft dus ook geen inverse. Ook de matrix $\binom{1\ 2}{3\ 6}$ beeldt alle punten af op een lijn.

Alleen als de tweede rij van een 2-dimensionale matrix geen veelvoud is van de eerste, is de matrix inverteerbaar. Een matrix $\binom{a\ b}{c\ d}$ is dus alleen inverteerbaar als $\frac{a}{c} \neq \frac{b}{d}$, oftewel $ad - bc \neq 0$. Deze waarde wordt de *determinant* van de matrix genoemd. Als de determinant nul is, is de matrix niet inverteerbaar; als hij ongelijk aan nul is, is de matrix wel inverteerbaar.

Ook voor (vierkante) matrices van hogere dimensie kan een determinant berekend worden. Voor een 3×3-matrix gaat dat als volgt:

$$\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \times \det \begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \times \det \begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \times \det \begin{pmatrix} d & e \\ g & h \end{pmatrix}$$

Je begint dus met de matrix te splitsen in een eerste rij (a, b, c) en overige rijen $\binom{d\ e\ f}{g\ h\ i}$. Dan bereken je de determinanten van de 2×2-matrices die je krijgt door uit de 'overige rijen' steeds één kolom weg te laten ($\binom{e\ f}{h\ i}$, $\binom{d\ f}{g\ i}$ en $\binom{d\ e}{g\ h}$). Die vermenigvuldig je met de overeenkomstige elementen uit de eerste rij. Tenslotte tel je ze op, waarbij om de andere term een min-teken krijgt. Dit werkt ook in hogere dimensies dan 3.

Definitie van de determinant

Van deze informele beschrijving van de determinant gaan we een functie-definitie maken. Er is sprake van het afsplitsen van de eerste rij van de matrix, dus de definitie heeft de vorm

```
det (Mat (ry:rys))  =  ...
```

Uit de rest-rijen `rys` moeten kolommen worden weggelaten. Om van rijen kolommen te maken moeten ze getransponeerd worden. De definitie wordt dus zoiets als

```
det (Mat (ry:rys))  =  ...
                       where  kols = transpose rys
```

Uit de lijst van kolommen moet op alle mogelijke manieren één kolom worden weggelaten. Dat
blz. 84 kan met de combinatorische functie `gaps` uit opgave 4.5. Het resultaat is een lijst van n lijsten van lijsten. Die moeten weer terug-getransponeerd worden, en er moeten met `Mat` kleine matrixjes van gemaakt worden:

```
det (Mat (ry:rys))  =  ...
                       where  kols = transpose rys
                              mats = map (Mat.transpose) (gaps kols)
```

Van al deze matrixjes moet de determinant berekend worden. Dat gebeurt natuurlijk door de functie recursief aan te roepen. De determinanten die het resultaat zijn, moeten vermenigvuldigd worden met de overeenkomstige elementen van de eerste rij:

```
det (Mat (ry:rys))  =  ...
                       where  kols  = transpose rys
                              mats  = map (Mat.transpose) (gaps kols)
                              prods = zipWith (*) ry (map det mats)
```

De producten `prods` die hierdoor worden opgeleverd, moet worden opgeteld, waarbij de termen afwisselend een plus- en een minteken krijgen. Dat kan bijvoorbeeld met behulp van de functie `altsum` (voor 'alternerende som') die gedefinieerd kan worden met behulp van een oneindige lijst:

```
altsum xs  =  sum (zipWith (*) xs plusMinEen)
```

```
            where  plusMinEen  =  1 : -1 : plusMinEen
```

Het functie-definitie wordt dus:

```
det (Mat (ry:rys))  =  altsum prods
                where  kols  = transpose rys
                       mats  = map (Mat.transpose) (gaps kols)
                       prods = zipWith (*) ry (map det mats)
```

Dit kan nog wat vereenvoudigd worden. Het terug-transponeren van de matrixjes is namelijk niet nodig, omdat de determinant van een getransponeerde matrix hetzelfde is als van de matrix zelf. Bovendien is het niet nodig om de tussenresultaten (`kols`, `mats` en `prods`) een naam te geven. De definitie kan dus luiden:

```
det (Mat (ry:rys)) =
  altsum (zipWith (*) ry (map det (map Mat (gaps (transpose rys)))))
```

Omdat `det` een recursieve functie is, moeten we niet vergeten een niet-recursief basisgeval toe te voegen. Daarvoor kan het 2×2-geval gebruikt worden, maar nog makkelijker is het om het 1×1-geval te definiëren:

```
det (Mat [[x]])  =  x
```

Het eindresultaat, waarin nog wat haakjes zijn weggewerkt door functie-samenstelling te gebruiken, is als volgt:

```
det  ::  Matrix -> Float
det (Mat [[x]])    =  x
det (Mat (ry:rys)) =
  (altsum . zipWith (*) ry . map det . map Mat . gaps . transpose) rys
```

Inverse van een matrix

De determinant is niet alleen van nut om te bepalen of de inverse bestaat, maar ook om de inverse daadwerkelijk uit te rekenen. De determinant moet dan natuurlijk wel ongelijk aan nul zijn.

De inverse van een 3×3-matrix kan als volgt worden uitgerekend: bepaal een matrix van negen 2×2-matrixjes, die onstaan door een rij en een kolom weg te laten uit de matrix. Bereken overal de determinant van, zet afwisselend plus- en min-tekens, en deel alles door de determinant van de gehele matrix (die $\neq 0$ moet zijn!).

Een voorbeeld is waarschijnlijk duidelijker. De inverse van $\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$ wordt als volgt uitgerekend:

$$\frac{\begin{pmatrix} +\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ \cancel{b} & e & h \\ \cancel{c} & f & i \end{pmatrix} & -\det\begin{pmatrix} \cancel{a} & d & g \\ \cancel{b} & \cancel{e} & \cancel{h} \\ \cancel{c} & f & i \end{pmatrix} & +\det\begin{pmatrix} \cancel{a} & d & g \\ \cancel{b} & e & h \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \\ -\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ b & \cancel{e} & h \\ c & \cancel{f} & i \end{pmatrix} & +\det\begin{pmatrix} \cancel{a} & d & \cancel{g} \\ \cancel{b} & \cancel{e} & \cancel{h} \\ c & \cancel{f} & i \end{pmatrix} & -\det\begin{pmatrix} a & \cancel{d} & g \\ b & \cancel{e} & h \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \\ +\det\begin{pmatrix} \cancel{a} & \cancel{d} & \cancel{g} \\ b & e & h \\ c & f & i \end{pmatrix} & -\det\begin{pmatrix} a & \cancel{d} & g \\ \cancel{b} & \cancel{e} & \cancel{h} \\ c & \cancel{f} & i \end{pmatrix} & +\det\begin{pmatrix} a & d & \cancel{g} \\ b & e & \cancel{h} \\ \cancel{c} & \cancel{f} & \cancel{i} \end{pmatrix} \end{pmatrix}}{\det\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}}$$

De doorgestreepte elementen staan in dit voorbeeld alleen maar om aan te geven welke elementen weggelaten moeten worden; er staan dus negen 2×2-matrixjes.

Let goed op welke elementen worden weggelaten: in de r-de rij van de grote matrix wordt steeds de r-de *kolom* van de matrixjes weggelaten, terwijl in de k-de kolom van de grote matrix juist de k-de *rij* van de matrixjes wordt weggelaten.

De matrix-inverse functie `matInv` kan nu op vergelijkbare wijze als de functie `det` geschreven worden, dat wil zeggen door op de juiste momenten gebruik te maken van `gaps`, `transpose`, `Mat`, enzovoort. Het wordt aan de lezer overgelaten om de details uit te werken (zie opgave 4.8). blz. 84

4.3 Polynomen

4.3.1 Representatie

Een *polynoom* is een som van *termen*, waarbij elke term bestaat uit het product van een reëel getal en een natuurlijke macht van een variabele.

$x^2 + 2x + 1$
$4.3x^3 + 2.5x^2 + 0.5$
$6x^5$
x
3

De hoogste macht die voorkomt heet de *graad* van het polynoom. In bovenstaande voorbeelden is de graad dus achtereenvolgens 2, 3, 5, 1 en 0. Het lijkt misschien raar om 3 een polynoom te noemen; het getal 3 is echter gelijk aan $3x^0$, en is dus inderdaad een product van een getal en een natuurlijke macht van x.

Met polynomen kun je rekenen: je kunt polynomen bij elkaar optellen, van elkaar aftrekken en met elkaar vermenigvuldigen. Het product van de polynomen $x + 1$ en $x^2 + 3x$ is bijvoorbeeld $x^3 + 4x^2 + 3x$. Als je twee polynomen echter door elkaar deelt is het resultaat niet altijd een polynoom. Het komt nu goed uit dat getallen ook polynomen zijn: zo is het resultaat van het optellen van $x + 1$ en $-x$ het polynoom 1.

In deze paragraaf wordt een datatype `Poly` ontworpen, waarmee polynomen kunnen worden gerepresenteerd. In de volgende paragraaf worden een aantal functies gedefinieerd, die op dat soort polynomen werken:

```
pPlus  :: Poly -> Poly -> Poly
pMin   :: Poly -> Poly -> Poly
pMaal  :: Poly -> Poly -> Poly
pEq    :: Poly -> Poly -> Bool
pGraad :: Poly -> Int
pEval  :: Float -> Poly -> Float
polyString :: Poly -> String
```

Een mogelijke representatie voor polynomen is 'functie van float naar float'. Het nadeel daarvan is echter dat je het resultaat van het vermenigvuldigen van twee polynomen niet meer als polynoom kunt inspecteren; je hebt dan een functie die je alleen nog maar op waarden kunt loslaten. Ook is het dan niet mogelijk om een gelijkheids-operator te schrijven; het is dus niet mogelijk om te testen of het product van de polynomen x en $x + 1$ gelijk is aan het polynoom $x^2 + x$.

Het is dus beter om een polynoom te representeren als een datastructuur met getallen. Daarbij ligt het voor de hand om een polynoom voor te stellen als lijst termen, waarbij elke term gekenmerkt wordt door een `Float` (de coëfficiënt) en een `Int` (de exponent). Een polynoom kan dus worden gerepresenteerd als een lijst twee-tupels. We maken er echter meteen maar een *data*type van met de volgende definitie:

```
data Poly  =  Poly [Term]
data Term  =  Term (Float,Int)
```

Let op: de namen `Poly` en `Term` worden dus zowel als naam van het type gebruikt, als als naam van de (enige) constructorfunctie. Dit is toegestaan, want het is uit de context altijd duidelijk welke van de twee bedoeld wordt. Het woord `Poly` is een type in type-declaraties zoals

```
pEq  ::  Poly -> Poly -> Bool
```

maar het is een constructorfunctie in functiedefinities zoals

```
pGraad (Poly [])  =  ...
```

Een aantal voorbeelden van representaties van polynomen is:

$3x^5 + 2x^4$	`Poly [Term(3.0,5), Term(2.0,4)]`
$4x^2$	`Poly [Term(4.0,2)]`
$2x + 1$	`Poly [Term(2.0,1), Term(1.0,0) ]`
3	`Poly [Term(3.0,0)]`
0	`Poly []`

Net als bij de rationale getallen uit paragraaf 3.3.3 hebben we hier weer het probleem dat er meerdere representaties zijn voor één polynoom. Het polynoom $x^2 + 7$ kan bijvoorbeeld worden gerepresenteerd door de volgende expressies: blz. 55

```
Poly [ Term(1.0,2), Term(7.0,0) ]
Poly [ Term(7.0,0), Term(1.0,2) ]
Poly [ Term(1.0,2), Term(3.0,0), Term(4.0,0) ]
```

Net als bij rationale getallen is het dus nodig om een polynoom te 'vereenvoudigen' nadat er operaties op zijn uitgevoerd. Vereenvoudigen bestaat in dit geval uit:

- sorteren van de termen, zodat de termen met de hoogste exponent voorop staan;
- samenvoegen van termen met gelijke exponent;
- verwijderen van termen met coëfficiënt nul.

Een alternatieve methode is om de polynomen niet te vereenvoudigen, maar dan moet er extra werk gedaan worden in de functie `pEq` waarmee polynomen vergeleken worden.

4.3.2 Vereenvoudiging

Voor het vereenvoudigen van polynomen schrijven we een functie

```
pEenvoud  ::  Poly -> Poly
```

Deze functie voert de drie genoemde aspecten van het vereenvoudigen uit, en kan dus geschreven worden als functie-samenstelling:

```
pEenvoud (Poly xs) = Poly (eenvoud xs)
                where  eenvoud = verwijderNul . samenvExpo . sortTerms
```

Blijft de taak over om de drie samenstellende functies te schrijven. Alledrie werken ze op lijsten van termen.

In paragraaf 3.1.4 werd een functie gedefinieerd die een lijst sorteert. Daarbij moesten de waarden echter ordenbaar zijn, en werd de lijst gesorteerd van klein naar groot. Er is een algemenere sorteer-functie denkbaar, waarbij als extra parameter een criterium wordt meegegeven dat beslist in welke volgorde de elementen komen te staan. Deze functie zou hier goed van pas komen, want hij kan dan gebruikt worden met 'heeft een grotere exponent' als sorteer-criterium. Daarom schrijven we eerst een functie `sortVolgens`, die dan gebruikt kan worden bij het schrijven van `sortTerms`. blz. 44

De algemene sorteer-functie `sortVolgens`, heeft als type:

```
sortVolgens  ::  (a->a->Bool) -> [a] -> [a]
```

Behalve de te sorteren lijst heeft de functie een functie als parameter. Die parameter-functie levert `True` op als zijn eerste parameter vóór zijn tweede parameter moet komen. De definitie van `sortVolgens` lijkt sterk op die van `isort` in paragraaf 3.1.4. Het verschil is dat nu de als extra parameter meegegeven vergelijk-functie wordt gebruikt in plaats van `<`. De definitie wordt dan: blz. 44

```
sortVolgens komtVoor xs  =  foldr (insertVolgens komtVoor) [] xs
insertVolgens komtVoor e []  =  [e]
insertVolgens komtVoor e (x:xs)
           | e `komtVoor` x  =  e : x : xs
           | otherwise       =  x : insertVolgens komtVoor e xs
```

Voorbeelden van het gebruik van `sortVolgens` zijn:

```
? sortVolgens (<) [1,3,2,4]
[1, 2, 3, 4]
? sortVolgens (>) [1,3,2,4]
[4, 3, 2, 1]
```

Bij het sorteren van termen op grond van hun exponent kan `sortVolgens` nu worden gebruikt. Deze functie krijgt als `komtVoor`-functie een functie mee die kijkt of de exponent groter is:

```
sortTerms ::  [Term] -> [Term]
sortTerms  =  sortVolgens expoGroter
       where  Term(c1,e1) `expoGroter` Term(c2,e2)  =  e1>e2
```

De tweede functie die nodig is, is de functie die termen met gelijke exponenten samenvoegt. Deze functie mag er van uitgaan dat de termen al zijn gesorteerd op exponent. Termen met gelijke exponent staan dus naast elkaar. De functie laat lijsten met nul of één element ongemoeid. Bij lijsten met twee of meer elementen zijn er twee mogelijkheden:

- de exponenten van de eerste twee elementen zijn gelijk; de elementen worden samengevoegd, het nieuwe element wordt op kop van de rest gezet, en de functie wordt opnieuw aangeroepen, zodat het nieuwe element eventueel met nog meer elementen samengevoegd kan worden.
- de exponenten van de eerste twee elementen zijn *niet* gelijk; het eerste element komt dan onveranderd in het resultaat, de rest wordt aan een nadere inspectie onderworden (misschien is het tweede element wel gelijk aan het derde).

Dit alles komt terug in de definitie:

```
samenvExpo   :: [Term] -> [Term]
samenvExpo []         =  []
samenvExpo [t]        =  [t]
samenvExpo (Term(c1,e1):Term(c2,e2):ts)
         | e1==e2     =  samenvExpo (Term(c1+c2,e1):ts)
         | otherwise  =  Term(c1,e1) : samenvExpo (Term(c2,e2):ts)
```

De derde benodigde functie is eenvoudig te maken:

```
verwijderNul ::  [Term] -> [Term]
verwijderNul  =  filter coefNietNul
          where  coefNietNul (Term(c,e))  =  c/=0.0
```

Desgewenst kunnen de drie functies lokaal gedefinieerd worden in `pEenvoud`:

```
pEenvoud (Poly xs) =  Poly (eenvoud xs)
      where  eenvoud  = vN . sE . sT
             sT  =  sortVolgens expoGroter
             sE []         =  []
             sE [t]        =  [t]
             sE (Term(c1,e1):Term(c2,e2):ts)
              | e1==e2     =  sE (Term(c1+c2,e1):ts)
              | otherwise  =  Term(c1,e1) : sE (Term(c2,e2):ts)
             vN  =  filter coefNietNul
             coefNietNul (Term(c,e))  =  c/=0.0
             Term(c1,e1) `expoGroter` Term(c2,e2)  =  e1>e2
```

De functie `pEenvoud` verwijdert *alle* termen waarvan de coëfficiënt nul is. Het nul-polynoom wordt dus gerepresenteerd door `Poly []`, een lege lijst termen. Daarmee verschilt het nul-polynoom van andere polynomen waarin de variabele niet voorkomt. Het polynoom '3' wordt bijvoorbeeld gerepresenteerd door `Poly [Term(3.0,0)]`.

4.3.3 Rekenkundige operaties

Het optellen van twee polynomen is eenvoudig. De lijsten van termen kunnen gewoon geconcateneerd worden. Daarna zorgt `pEenvoud` ervoor dat de termen gesorteerd worden, gelijke exponenten samengenomen worden, en nul-termen verwijderd worden:

```
pPlus  ::  Poly -> Poly -> Poly
pPlus (Poly xs) (Poly ys)  =  pEenvoud (Poly (xs++ys))
```

Voor het aftrekken van twee polynomen tellen we het eerste polynoom op bij het tegengestelde van het tweede:

```
pMin  ::  Poly -> Poly -> Poly
pMin p1 p2  =  pPlus p1 (pNeg p2)
```

Blijft natuurlijk de vraag hoe het tegengestelde van een polynoom berekend wordt: daartoe moet het tegengestelde van elke term berekend worden.

```
pNeg  ::  Poly -> Poly
pNeg (Poly xs)   =  Poly (map tNeg xs)
tNeg  ::  Term -> Term
tNeg (Term(c,e)) = Term(-c,e)
```

Vermenigvuldigen van polynomen is wat moeilijker. Daarvoor moet elke term van het eerste polynoom vermenigvuldigd worden met elke term van het andere polynoom. Dat vraagt om een hogere-orde functie: 'doe iets met elk element van een lijst in combinatie met elk element van een andere lijst'. Deze functie noemen we `cpWith`. De `cp` staat voor *cross product*, de `With` is naar analogie van de functie `zipWith`. Als de eerste lijst leeg is, valt er niets samen te stellen. Als de eerste lijst de vorm `x:xs` heeft, moet het eerste element `x` met alle elementen van de tweede lijst

samengesteld worden, en moet bovendien het cross-product van `xs` en de tweede lijst nog bepaald worden. Dit geeft de definitie:

```
cpWith  ::  (a->b->c) -> [a] -> [b] -> [c]
cpWith f [] ys     = []
cpWith f (x:xs) ys = map (f x) ys ++ cpWith f xs ys
```

Deze definitie kan meteen gebruikt worden bij het vermenigvuldigen van polynomen:

```
pMaal  ::  Poly -> Poly -> Poly
pMaal (Poly xs) (Poly ys)  =  pEenvoud (Poly (cpWith tMaal xs ys))
```

Hierin wordt de functie `tMaal` gebruikt, die twee termen vermenigvuldigd. Zoals uit het voorbeeld $3x^2$ maal $5x^4$ is $15x^6$ blijkt, moeten daartoe de coëfficiënten worden vermenigvuldigd, en de exponenten opgeteld:

```
tMaal  :: Term -> Term -> Term
tMaal (Term(c1,e1)) (Term(c2,e2))  =  Term (c1*c2,e1+e2)
```

Doordat we polynomen steeds vereenvoudigen, en dus steeds de term met de hoogste exponent voorop staat, is de graad van een polynoom gelijk aan de exponent van de eerste term. Alleen voor het nul-polynoom hebben we een aparte definitie nodig.

```
pGraad  ::  Poly -> Int
pGraad (Poly [])             =  0
pGraad (Poly (Term(c,e):ts)) =  e
```

Twee vereenvoudigde polynomen zijn gelijk als alle termen gelijk zijn. Twee termen zijn gelijk als de coëfficiënt en de exponent overeenstemmen. Dit alles laat zich gemakkelijk naar functies vertalen:

```
pEq  :: Poly -> Poly -> Bool
pEq (Poly xs) (Poly ys)  =  length xs==length ys
                            && and (zipWith tEq xs ys)
tEq  :: Term -> Term -> Bool
tEq (Term(c1,e1)) (Term(c2,e2))  =  c1==c2 && e1==e2
```

De functie `pEval` moet een polynoom uitrekenen met een specifieke waarde voor x ingevuld. Daartoe moeten alle termen geëvalueerd worden, en de resultaten opgeteld:

```
pEval :: Float -> Poly -> Float
pEval w (Poly xs)   =  sum (map (tEval w) xs)
tEval :: Float -> Term -> Float
tEval w (Term(c,e)) =  c * w^e
```

Tenslotte schrijven we een functie voor de weergave van een polynoom als string. Hiervoor moeten de termen worden weergegeven als string, en ertussen moet een +-teken komen:

```
polyString  :: Poly -> String
polyString (Poly [])     =  "0"
polyString (Poly [t])    =  termString t
polyString (Poly (t:ts)) =  termString t ++ " + "
                            ++ polyString (Poly ts)
```

Bij de weergave van een term laten we de coëfficiënt en de exponent weg als die 1 is. Als de exponent 0 is, wordt de variabele weggelaten, maar de coëfficiënt nooit. De exponent duiden we aan met een ^-teken, net zoals dat in Haskell-expressies gebruikelijk is. De functie wordt daarmee:

```
termString  :: Term -> String
termString (Term(c,0))    =  floatString c
termString (Term(1.0,1))  =  "x"
termString (Term(1.0,e))  =  "x^" ++ intString e
termString (Term(c,e))    =  floatString c ++ "x^" ++ intString e
```

De functie `floatString` is nog niet gedefinieerd. Met een boel moeite is dat wel mogelijk, maar dat is niet nodig: in paragraaf 6.2.4 wordt hiervoor de functie `show` geïntroduceerd. blz. 120

Opgaven

4.1 Hoe wordt de volgorde van de elementen van `segs [1,2,3,4]` als de parameters van `++` in de definitie van `segs` worden omgewisseld?

4.2 Schrijf `segs` als combinatie van `inits`, `tails` en standaardfuncties. De volgorde van de elementen in het resultaat hoeft niet hetzelfde te zijn als op blz. 67.

4.3 Gegeven is een lijst `xs` met `n` elementen. Bepaal het aantal elementen van `inits xs`, `segs xs`, `subs xs`, `perms xs`, en `combs k xs`.

4.4 Schrijf een functie `bins :: Int -> [[Char]]` die alle getallen in het tweetallig stelsel (als strings nullen en enen) bepaalt met het gegeven aantal cijfers. Vergelijk de functie met de functie `subs`.

4.5 Schrijf een functie `gaps` die alle mogelijkheden geeft om één element uit een lijst weg te laten. Bijvoorbeeld:

`gaps` [1,2,3,4,5] = [[2,3,4,5] , [1,3,4,5] , [1,2,4,5] , [1,2,3,5] , [1,2,3,4]]

4.6 Vergelijk de voorwaarden wat betreft de afmetingen van de matrices bij matrixvermenigvuldiging met het type van de functie-samenstellings-operator `(.)`.

4.7 Ga na dat de recursieve definitie van matrix-determinant `det` overeenkomt met de expliciete definitie voor determinanten van 2×2-matrices uit paragraaf 4.2.3.
blz. 78

4.8 Schrijf de functie `matInv`.

4.9 In paragraaf 4.2.2 werd de functie `transpose` beschreven als generalisatie van `zip`. In paragraaf 3.3.4 werd `zip` geschreven als `zipWith maak2tupel`. De functie `cp` wordt nu gedefinieerd als `cpWith maak2tupel`. Schrijf een functie `crossprod` die een generalisatie is van `cp` zoals `transpose` een generalisatie is van `zip`. Hoe kan `length (crossprod xs)` berekend worden zonder `crossprod` te gebruiken?
blz. 75
blz. 56

4.10 Een andere mogelijke representatie van polynomen is een lijst van coëfficiënten. De exponenten worden dus niet opgeslagen. De prijs daarvan is dat 'ontbrekende' termen als `0.0` opgeslagen moeten worden. Het is het handigst om de termen met de kleinste exponent aan het begin van de lijst op te slaan. Dus bijvoorbeeld:

$x^2 + 2x$	`[0.0, 2.0, 1.0]`
$4x^3$	`[0.0, 0.0, 0.0, 4.0]`
5	`[5.0]`

Schrijf functies voor de graad van een polynoom en voor de som en het product van twee polynomen in deze representatie.

Hoofdstuk 5

Programmatransformatie

5.1 Efficiëntie

5.1.1 Tijdgebruik

Elke aanroep van een functie die bij het uitrekenen van een expressie wordt gedaan, kost tijd. De gedane functie-aanroepen worden door de interpreter geteld: het zijn de *reductions* die na het antwoord worden vermeld.

Het aantal benodigde reducties kan de tweede keer dat een expressie wordt uitgerekend kleiner zijn dan de eerste keer. Bekijk bijvoorbeeld het volgende geval. De functie `f` is een ingewikkelde functie, bijvoorbeeld de faculteit-functie. De constante `k` is gedefinieerd met behulp van een aanroep van `f`. De functie `g` gebruikt `k`:

```
f x = product [1..x]
k   = f 5
g x = k + 1
```

De constante `k` wordt dankzij lazy evaluatie niet tijdens het analyseren van de file uitgerekend. Hij wordt pas uitgerekend als hij voor het eerst gebruikt wordt, bijvoorbeeld door een aanroep van `g`:

```
? g 1
121
(57 reductions, 78 cells)
```

Als daarna nog eens `g` aangeroepen wordt, wordt de waarde van `k` niet opnieuw uitgerekend. De tweede aanroep van `g` is dus veel sneller:

```
? g 1
121
(3 reductions, 8 cells)
```

Je kunt in een file dus gerust allerlei constante-definities zetten. Pas als deze voor het eerst gebruikt worden kosten ze tijd. Functies die zijn gedefinieerd door een hogere-orde functie partieel te parametriseren, bijvoorbeeld

```
som  =  foldr (+) 0
```

kosten de tweede keer dat ze gebruikt worden één reductie minder:

```
? som [1,2,3]
6
(9 reductions, 19 cells)
? som [1,2,3]
6
(8 reductions, 17 cells)
```

Na twee keer aanroepen blijft het aantal reducties verder hetzelfde.

5.1.2 Complexiteitsanalyse

Bij de analyse van het tijdgebruik van een functie is vaak niet het precieze aantal reducties van belang. Veel interessanter is het om te zien hoe dit aantal reducties verandert als de grootte van de invoer verandert.

Bekijk bijvoorbeeld drie lijsten met 10, 100 en 1000 elementen (de getallen vanaf 1 in omgekeerde volgorde). Het aantal reducties van enkele functies op deze lijsten is als volgt:

elementen	`head`	`last`	`length`	`sum`	`(++x)`	`isort`
10	2	11	42	32	52	133
100	2	101	402	302	502	10303
1000	2	1001	4002	3002	5002	1003003
n	2	$n+1$	$4n+2$	$3n+2$	$5n+2$	n^2+3n+3

Het bepalen van de `head` is altijd in twee reducties te doen. De head van een lijst hoeft immers alleen maar van het begin geplukt te worden. De functie `last` kost echter meer tijd naarmate de lijst langer is. Deze lijst moet immers helemaal doorlopen worden, en dat duurt langer naarmate de lijst langer is. Ook bij `length`, `sum` en `++` is het aantal benodigde reducties een lineaire functie van de lengte van de lijst. Bij `isort` is het aantal reducties echter een kwadratische functie van de lengte van de lijst.

Het is niet altijd zo gemakkelijk om te bepalen wat het aantal reducties is als functie van de lengte van de parameter. In de praktijk wordt er daarom mee volstaan om de *grootte-orde* van deze functie aan te geven: de benodigde tijd voor `head` is *constant*, voor `last`, `length`, `sum` en `++` is hij *lineair*, en voor `isort` is hij *kwadratisch* in de lengte van de parameter. De grootte-orde van een functie wordt genoteerd met de letter $\mathcal{O}$. Een kwadratische functie wordt aangegeven met $\mathcal{O}(n^2)$, een lineaire functie met $\mathcal{O}(n)$, en een constante functie met $\mathcal{O}(1)$.

Het aantal benodigde reducties voor zowel `length` als `sum` is $\mathcal{O}(n)$. Als de benodigde tijdsduur voor twee functies dezelfde grootte-orde heeft, kan die dus nog een constante factor schelen.

De volgende tabel geeft enig gevoel voor de snelheid waarmee de verschillende functies stijgen:

orde	$n = 10$	$n = 100$	$n = 1000$	naam
$\mathcal{O}(1)$	1	1	1	constant
$\mathcal{O}(\log n)$	3	7	10	logaritmisch
$\mathcal{O}(\sqrt{n})$	3	10	32	
$\mathcal{O}(n)$	10	100	1000	lineair
$\mathcal{O}(n \log n)$	30	700	10000	
$\mathcal{O}(n\sqrt{n})$	30	1000	31623	
$\mathcal{O}(n^2)$	100	10000	10^6	kwadratisch
$\mathcal{O}(n^3)$	1000	10^6	10^9	kubisch
$\mathcal{O}(2^n)$	1024	10^{30}	10^{300}	exponentieel

In deze tabel is voor de constante factor 1 genomen, en voor het grondtal van de logaritme 2. Voor de grootte-orde is het grondtal niet van belang, omdat wegens ${}^g\log n = {}^2\log n / {}^2\log g$ verandering van grondtal slechts een constante factor scheelt.

Let op dat, bijvoorbeeld, een $\mathcal{O}(n\sqrt{n})$ algoritme niet altijd 'beter' is dan een $\mathcal{O}(n^2)$ algoritme voor hetzelfde probleem. Als de constante factor in het $\mathcal{O}(n\sqrt{n})$ algoritme bijvoorbeeld tien keer zo groot is als die in het $\mathcal{O}(n^2)$ algoritme, dan is het eerste algoritme alleen sneller voor $n > 100$. Soms doen algoritmen met een lage complexiteit veel meer operaties per stap dan algoritmen met een hogere complexiteit, en hebben dus ook een veel grotere constante factor. Echter, als n maar groot genoeg is, zijn algoritmen met een lagere complexiteit sneller.

Aan de definitie van een functie is vaak eenvoudig de grootte-orde van het aantal benodigde reducties af te lezen. Die grootte-orde kan gebruikt worden als maat voor de *complexiteit* van de functie. Hier volgt een aantal voorbeelden. Het eerste voorbeeld gaat over een niet-recursieve functie. Dan volgen drie voorbeelden van recursieve functies waarbij de parameter van de recursieve aanroep iets kleiner is (1 korter of 1 minder), dus functies met een definitie van de vorm

```
f (x:xs)  =  ... f xs ...
g (n+1)   =  ... g n  ...
```

Tenslotte volgen drie voorbeelden van recursieve functies waarbij de parameter van de recursieve aanroep ongeveer halveert, zoals in

```
f (Tak x li re)  =  ... f li ...
g (2*n)          =  ... g n  ...
```

- De functie is niet recursief, en bevat alleen aanroepen van functies met complexiteit $\mathcal{O}(1)$. Voorbeeld: `head`, `tail`, `abs`. (Let op: de tijd die nodig is om het resultaat van `tail` af te drukken is lineair, maar het bepalen van de `tail` op zich duurt constante tijd). Een aantal malen een constante blijft een constante, dus deze functies hebben zelf ook complexiteit $\mathcal{O}(1)$.

- De functie doet één recursieve aanroep van zichzelf met een iets kleinere parameter. Het eindantwoord wordt vervolgens in constante tijd bepaald. Voorbeeld: `fac`, `last`, `insert`, `length`, `sum`. Bovendien: `map f` en `foldr f`, waarbij `f` constante complexiteit heeft. Deze functies hebben complexiteit $\mathcal{O}(n)$.
- De functie doet één recursieve aanroep van zichzelf met een iets kleinere parameter. Het eindantwoord wordt vervolgens in lineaire tijd bepaald. Voorbeeld: `map f` of `foldr f`, waabij `f` lineaire complexiteit heeft. Een speciaal geval hiervan is `isort`; deze functie is immers gedefinieerd als `foldr insert []`, waarin `insert` lineaire complexiteit heeft. Deze functies hebben complexiteit $\mathcal{O}(n^2)$.
- De functie doet *twee* recursieve aanroepen van zichzelf met iets kleinere parameter. Voorbeeld: de voorlaatste versie van `subs` in paragraaf 4.1.1. Dit soort functies zijn heel vreselijk: bij elke stap in de recursie verdubbelt het aantal reducties. Deze functies hebben complexiteit $\mathcal{O}(2^n)$. blz. 69
- De functie doet één recursieve aanroep van zichzelf met een parameter die ongeveer twee keer zo klein geworden is. Het eindantwoord wordt vervolgens in constante tijd berekend. Voorbeeld: `elemBoom` in paragraaf 3.4.2. Deze functies hebben complexiteit $\mathcal{O}(\log n)$. blz. 60
- De functie deelt zijn parameter ongeveer in tweeën en roept zichzelf recursief aan op *beide* delen. Het eindantwoord wordt vervolgens in constante tijd berekend. Voorbeeld: `omvang` in paragraaf 3.4.1. Deze functies hebben complexiteit $\mathcal{O}(n)$. blz. 59
- De functie deelt zijn parameter ongeveer in tweeën en roept zichzelf recursief aan op beide delen. Het eindantwoord wordt vervolgens in lineaire tijd berekend. Voorbeeld: `labels` in paragraaf 3.4.2, `msort` in paragraaf 3.1.4. Deze functies hebben complexiteit $\mathcal{O}(n \log n)$. blz. 61 blz. 45

Samengevat in een tabel:

parameter	aant.rec.aanr.	vervolgens	voorbeeld	complexiteit
geen	0	constant	`head`	$\mathcal{O}(1)$
kleiner	1	constant	`sum`	$\mathcal{O}(n)$
kleiner	1	lineair	`isort`	$\mathcal{O}(n^2)$
kleiner	2	will.	`subs`	$\mathcal{O}(2^n)$
helft	1	constant	`elemBoom`	$\mathcal{O}(\log n)$
helft	1	lineair		$\mathcal{O}(n)$
helft	2	constant	`omvang`	$\mathcal{O}(n)$
helft	2	lineair	`msort`	$\mathcal{O}(n \log n)$

5.1.3 Verbeteren van efficiëntie

Hoe kleiner de parameter van een recursieve functie is, des te korter duurt het om de functie uit te rekenen. Uit het voorgaande volgt dat er daarnaast nog drie manieren zijn om de efficiëntie van een functie te verbeteren:

- doe de recursieve aanroep liever op gehalveerde parameters dan op een met één verminderde parameter;
- doe liever één dan twee recursieve aanroepen;
- houd het vervolgwerk na de recursieve aanroep(en) liever constant dan lineair.

Bij sommige functies kunnen deze methoden inderdaad gebruikt worden om een snellere functie te krijgen. We bekijken van elke methode een voorbeeld.

a. Kleine parameters

De operator `++` is gedefinieerd met inductie naar de linker parameter:

```
[]     ++ ys  = ys
(x:xs) ++ ys  =  x : (xs++ys)
```

De benodigde tijd voor het uitrekenen van `++` is daarom lineair in de lengte van de linker parameter. De lengte van de rechter parameter beïnvloedt de tijdsduur niet. Als twee lijsten samengevoegd moeten worden, maar de volgorde van de elementen doet er niet toe, dan is het dus efficiënter om de kortste van de twee voorop te zetten.

De operator is associatief, dat wil zeggen dat de waarde van `(xs++ys)++zs` gelijk is aan die van `xs++(ys++zs)`. Voor de complexiteit maakt het echter wèl uit welke van de twee expressies berekend wordt.

Stel dat de lengte van `xs` gelijk is aan x, de lengte van `ys` y en die van `zs` z. Dan geldt:

tijd voor...	`(xs++ys)++zs`	`xs++(ys++zs)`
eerste `++`	x	x
tweede `++`	$x + y$	y
totaal	$2x + y$	$x + y$

Als `++` naar rechts associeert, wordt hij dus sneller berekend dan als hij naar links associeert. blz. 22 Daarom wordt `++` in de prelude als rechts-associërende operator gedefinieerd (zie paragraaf 2.1.3).

b. Halveren i.p.v. verminderen (lineair vervolgwerk)

Bekijk functies waarbij het vervolg-werk na de recursieve aanroep lineair is. Een recursieve functie die zichzelf twee keer aanroept met een parameter die half zo groot is (bijvoorbeeld een linker- en een rechter deelboom van een gebalanceerde boom) heeft complexiteit $\mathcal{O}(n \log n)$. Dat is dus beter dan een functie die zichzelf één keer aanroept met een parameter die slechts iets kleiner is geworden (bijvoorbeeld de staart van een lijst), die complexiteit $\mathcal{O}(n^2)$ heeft.

De sorteerfunctie `msort`

```
msort xs |  lengte<=1  =  xs
         |  otherwise  =  merge (msort ys) (msort zs)
   where ys     =  take half xs
         zs     =  drop half xs
         half   =  lengte / 2
         lengte =  length xs
```

is dus efficiënter dan de functie `isort`

```
isort  []     =  []
isort (x:xs)  =  insert x (isort xs)
```

Bij veel functies op lijsten ligt een algoritme op de `isort`-manier het meest voor de hand. Het loont de moeite om altijd even na te gaan of er ook een `msort`-achtig algoritme mogelijk is. Denk daarbij aan de slagzin

Verdeel en Heers

die in dit geval betekent: 'verdeel de parameter in twee ongeveer even grote lijsten; pas de functie recursief toe op de helften, en voeg de deel-oplossingen vervolgens samen'. In de functie `msort` gebeurt het verdelen door de functies `take` en `drop`; het heersen gebeurt door de functie `merge`.

c. Halveren i.p.v. verminderen (constant vervolgwerk)

Bekijk functies die zichzelf één keer aanroepen met constant vervolgwerk. Een voorbeeld daarvan is de machtsverhef-functie:

```
x ^ 0      =  1
x ^ (n+1)  =  x  *  x ^ n
```

Deze functie heeft complexiteit $\mathcal{O}(n)$. Zou de parameter gehalveerd worden bij de recursieve aanroep, dan bedroeg de complexiteit slechts $\mathcal{O}(\log n)$. In het geval van machtsverheffen is dat mogelijk:

```
x ^ 0       =  1
x ^ (2*n)   =  kwadraat (x^n)
x ^ (2*n+1) =  x * kwadraat (x^n)
kwadraat x  =  x * x
```

In deze definitie wordt de bekende rekenregel $x^{2n} = (x^n)^2$ uitgebuit. In Haskell kunnen de gevallen 'even parameter' en 'oneven parameter' onderscheiden worden met behulp van patronen. In talen waarin dat niet mogelijk is kan het altijd nog met behulp van de functie `even`.

d. Weinig recursieve aanroepen (halverende parameter)

Als je het voor het kiezen hebt, is het beter om één recursieve aanroep te doen dan twee. Daarom is het in een zoekboom mogelijk om in $\mathcal{O}(\log n)$ tijd het gezochte element te vinden:

```
zoek e Blad                    =  False
zoek e (Tak x li re)  |  e==x  =  True
                      |  e<x   =  zoek e li
                      |  e>x   =  zoek e re
```

de functie wordt immers maar recursief aangeroepen op één helft: de linker deelboom als `e<x`, of de rechter deelboom als `e>x`. In een boom waarin de elementen niet geordend zijn, kost zoeken in het algemeen $\mathcal{O}(n)$ tijd, omdat een recursieve aanroep op de linker èn de rechter deelboom nodig is:

```
zoek e Blad                       =  False
zoek e (Tak x li re)  | e==x       =  True
                      | otherwise  =  zoek e li || zoek e re
```

als het element in de linkerboom wordt aangetroffen heb je geluk, maar anders zal toch `zoek` nog eens worden aangeroepen op de rechterboom.

e. Weinig recursieve aanroepen (verminderende parameter)

Het uitvoeren van één recursieve aanroep in plaats van twee is helemaal spectaculair als de parameter niet halveert maar slechts iets kleiner wordt: het maakt van een exponentieel algoritme een lineair algoritme! Het is dus uit den boze om twee recursieve aanroepen te doen met dezelfde parameter, zoals gebeurt in `subs`:

```
subs []      =  [[]]
subs (x:xs)  =  map (x:) (subs xs)  ++  subs xs
```

Zoals in paragraaf 4.1.1 is aangegeven, kan de recursieve aanroep in een `where`-constructie worden gezet, waardoor hij nog maar één keer wordt uitgevoerd: blz. 69

```
subs []      =  [[]]
subs (x:xs)  =  map (x:) (subsxs)  ++  subsxs
         where  subsxs = subs xs
```

Weliswaar wordt `subs` hiermee niet lineair, omdat de `++` meer dan constante tijd kost, maar toch wordt er tijd bespaard.

Als de twee recursieve aanroepen niet dezelfde parameter hebben, gaat deze methode niet op. Er is soms echter toch een verbetering mogelijk. Bekijk bijvoorbeeld de functie van Fibonacci:

```
fib 0      =  0
fib 1      =  1
fib (n+2)  =  fib n + fib (n+1)
```

De functie wordt tweemaal recursief aangeroepen met parameter die niet veel kleiner is, in ieder geval niet halveert. De complexiteit van `fib` is dus $\mathcal{O}(2^n)$.

Een verbetering hiervan is mogelijk door een extra functie te schrijven, die meer doet dan het gevraagde. In dit geval maken we een functie `fib' n`, die behalve de waarde van `fib n` ook `fib (n-1)` berekent. Deze functie levert dus een 2-tupel op. De functie `fib` kan dan geschreven worden door gebruik te maken van `fib'`:

```
fib 0  =  0
fib n  =  snd (fib' n)
   where  fib' 1     =  (0,1)
          fib' (n+1) =  (b,a+b) where (a,b)=fib' n
```

Deze functie bevat nog maar één recursieve aanroep, en heeft dus complexiteit $\mathcal{O}(n)$.

f. Vervolgwerk beperkt houden

Bij 'verdeel en heers'-functies, die het resultaat van twee recursieve aanroepen combineren tot een eindantwoord is het van belang hoeveel werk het combineren kost. Kost het combineren constante tijd, dan is de complexiteit $\mathcal{O}(n)$; kost het combineren lineaire tijd, dan is de complexiteit $\mathcal{O}(n \log n)$.

De functie `labels`, die de elementen van een boom in een lijst zet, heeft als de boom gebalanceerd is complexiteit $\mathcal{O}(n \log n)$:

```
labels Blad           =  []
labels (Tak x li re)  =  labels li ++ [x] ++ labels re
```

Voor het 'heersen' wordt immers de operator `++` gebruikt, die lineaire tijd kost.

Ook nu is verbetering mogelijk, door een extra functie te maken, die eigenlijk te veel doet:

```
labelsVoor  ::  Boom a -> [a] -> [a]
labelsVoor boom xs  =  labels boom ++ xs
```

Gegeven deze functie kan `labels` geschreven worden als

```
labels boom  =  labelsVoor boom []
```

Op deze manier levert dat natuurlijk nog geen tijdwinst op. Maar er is een andere definitie van `labelsVoor` mogelijk:

```
labelsVoor Blad          xs  =  xs
labelsVoor (Tak x li re) xs  =  labelsVoor li (x:labelsVoor re xs)
```

Deze functie doet twee recursieve aanroepen op de deelbomen (net als `labels`), maar het 'heersen' kost nu nog maar constante tijd (voor de operator `:`). De complexiteit van het algoritme is daarom $\mathcal{O}(n)$, een verbetering ten opzichte van de $\mathcal{O}(n \log n)$ van het oorspronkelijke algoritme.

5.1.4 Geheugengebruik

Behalve op de benodigde tijd kan een algoritme ook beoordeeld worden op de benodigde geheugenruimte. Het geheugen wordt bij berekeningen voor drie doeleinden gebruikt:

- het opslaan van het programma;
- het opbouwen van datastructuren met behulp van constructorfuncties;
- het bijhouden van nog uit te rekenen deel-expressies.

Het geheugen voor de datastructuren wordt de *heap* ('hoop') genoemd; het geheugen voor de administratie van de berekening de *stack* ('stapel').

De heap

Op de heap worden datastructuren (lijsten, tupels, bomen enz.) opgebouwd door constructorfuncties te gebruiken. De heap bestaat uit *cellen*, waarin de informatie wordt opgeslagen.[1] Bij elke aanroep van de constructor-operator `:` worden twee nieuwe cellen gebruikt. Ook bij gebruik van zelf-gedefinieerde constructorfuncties worden cellen gebruikt (voor elke parameter van een constructorfunctie één).

Vroeg of laat is de gehele heap verbruikt. Op dat moment treedt de *garbage collector* ('vuilnisophaler') in werking. Alle cellen die niet meer nodig zijn worden daarbij voor hergebruik geschikt gemaakt. Bij het uitrekenen van de expressie `length (subs [1..8])` worden de deelrijen van `[1..8]` bepaald en geteld. Zodra een deelrij geteld is, is de deelrij zelf niet meer belangrijk. Bij garbage collection zal de betreffende heap-ruimte voor hergebruik worden vrijgegeven.

Als er tijdens het uitrekenen van een expressie garbage collection(s) hebben plaatsgevonden, wordt dit na afloop door de interpreter vermeld:

```
? length (subs [1..8])
256
(4141 reductions, 9286 cells, 1 garbage collection)
? length (subs [1..10])
1024
(18487 reductions, 43093 cells, 6 garbage collections)
?
```

Het is ook mogelijk om elke afzonderlijke garbage-collection vermeld te krijgen. Daartoe moet de optie `+g` gezet worden:

```
? :set +g
```

Bij het begin van elke garbage collection wordt dan `{{Gc:` op het scherm gezet.Na afloop van elke garbage collection wordt daarachter het aantal opnieuw te gebruiken cellen vermeld:

```
? length (subs [1..10])
{{Gc:7987}}{{Gc:7998}}{{Gc:7994}}{{Gc:7991}}{{Gc:7997}}{{Gc:8004}}1024
(18487 reductions, 43093 cells, 6 garbage collections)
```

Als na garbage collection nog steeds niet genoeg geheugen beschikbaar is om een nieuwe cell aan te maken, wordt de berekening afgebroken met de melding

[1] Afhankelijk van de gebruikte computer kost elke cell 4 of 8 bytes in het geheugen.

```
ERROR: Garbage collection fails to reclaim sufficient space
```

Het enige wat er dan nog opzit is om Haskell te starten met een grotere heap. Haskell starten met een heap van 200000 cellen gaat, mits er voldoende geheugenruimte beschikbaar is voor zo'n grote heap, als volgt:

```
% hugs -h200000
```

Op het gebruik van de heap valt niet veel te bezuinigen. De opgebouwde datastructuren zijn meestal gewoon nodig. De enige manier om het gebruik van constructorfuncties, en daarmee het gebruik van de heap, te vermijden is het zo veel mogelijk gebruiken van `@`-patronen, zoals beschreven in paragraaf 4.1.3. Daar werd op drie manieren de functie `tails` gedefinieerd: blz. 71

- zonder patronen

```
tails []  =  [ [] ]
tails xs  =  xs : tails (tail xs)
```

- met gewone patronen

```
tails []      =  [ [] ]
tails (x:xs)  =  (x:xs) : tails xs
```

- met `@`-patronen

```
tails []           =  [ [] ]
tails lyst@(x:xs)  =  lyst : tails xs
```

Bij toepassing van deze functies op de lijst `[1..10]` is het tijd- en geheugengebruik als volgt:

zonder patronen	210 reductions	569 cells
met gewone patronen	200 reductions	579 cells
met `@`-patronen	200 reductions	559 cells

Hieruit blijkt nogmaals dat de definitie met `@`-patronen de voordelen van de andere twee definities combineert.

De stack

De stack wordt door de interpreter gebruikt om de expressie op te slaan die uitgerekend moet worden. Ook worden de tussenresultaten op de stack opgeslagen. Elke 'reduction' levert een nieuw tussenresultaat op.

De stackruimte die nodig is om de tussenresultaten op te slaan kan groter zijn dan de uit te rekenen expressie en het resultaat. Bekijk bijvoorbeeld de tussenresultaten in de berekening `foldr (+) 0 [1..5]`, waarin de som van de getallen 1 t/m 5 wordt uitgerekend:

```
foldr (+) 0 [1..5]
1+foldr (+) 0 [2..5]
1+(2+foldr (+) 0 [3..5])
1+(2+(3+foldr (+) 0 [4..5]))
1+(2+(3+(4+foldr (+) 0 [5])))
1+(2+(3+(4+(5+foldr (+) 0 []))))
1+(2+(3+(4+(5+0))))
1+(2+(3+(4+5)))
1+(2+(3+9))
1+(2+12)
1+14
15
```

Het feitelijke uitrekenen van de `+`-operatoren begint pas nadat op de stack de complete expressie `1+(2+(3+(4+(5+0))))` is opgebouwd. In de meeste gevallen heb je daar geen last van, maar het wordt vervelend als je erg lange lijsten gaat optellen:

```
? foldr (+) 0 [1..5000]
12502500
? foldr (+) 0 [1..6000]
ERROR: Control stack overflow
```

Als je in plaats van `foldr` de functie `foldl` gebruikt worden de getallen op een andere manier opgeteld. Ook hier is echter bij het optellen van lange lijsten veel stackruimte nodig:

```
foldl (+) 0 [1..5]
```

```
foldl (+) (0+1) [2..5]
foldl (+) ((0+1)+2) [3..5]
foldl (+) (((0+1)+2)+3) [4..5]
foldl (+) ((((0+1)+2)+3)+4) [5]
foldl (+) (((((0+1)+2)+3)+4)+5) []
((((0+1)+2)+3)+4)+5
(((1+2)+3)+4)+5
((3+3)+4)+5
(6+4)+5
10+5
15
```

Vanwege het lazy-evaluatie principe wordt het uitrekenen van parameters zo lang mogelijk uitgesteld. Dat geldt ook voor de tweede parameter van `foldl`. In dit geval is dat een beetje vervelend, want uiteindelijk wordt de expressie tòch uitgerekend. Het uitstellen is dus nergens goed voor geweest, en kost alleen maar een boel stackruimte.

Zonder lazy evaluatie was de berekening als volgt verlopen; zodra dat mogelijk is wordt de tweede parameter van `foldl` uitgerekend:

```
foldl (+) 0 [1..5]
foldl (+) (0+1) [2..5]
foldl (+) 1 [2..5]
foldl (+) (1+2) [3..5]
foldl (+) 3 [3..5]
foldl (+) (3+3) [4..5]
foldl (+) 6 [4..5]
foldl (+) (6+4) [5]
foldl (+) 10 [5]
foldl (+) (10+5) []
foldl (+) 15 []
15
```

De benodigde stackruimte is bij deze evaluatievolgorde gedurende de berekening constant.

Hoe krijg je de interpreter nu zo ver dat hij een functie (in dit geval de partieel geparametriseerde functie `foldl (+)`) niet-lazy uitrekent? Speciaal voor dit doel is er een ingebouwde functie, `strict` genaamd. Het effect van `strict` is als volgt:

```
strict f x  =  f x
```

De functie krijgt dus een functie en een waarde als parameter, en past de functie toe op de waarde. Op zich heb je daar niet zo veel aan, maar `strict` doet nog meer: hij rekent zijn tweede parameter niet-lazy uit. Zo'n functie kun je zelf nooit schrijven; `strict` is dan ook niet slechts voorgedefinieerd, maar echt ingebouwd.

Met behulp van de functie `strict` kun je elke gewenste functie niet-lazy doen uitrekenen. In het voorbeeld hierboven zou het handig zijn als `foldl` zijn *tweede* parameter niet-lazy uitrekent. Dat wil zeggen dat de partieel geparametriseerde functie `foldl (+)` niet-lazy moet zijn in zijn eerste parameter. Dat kan, door de functie `strict` erop toe te passen; we schrijven dus `strict (foldl (+))`. Let op het extra paar haakjes: `foldl (+)` moet als geheel aan `strict` worden meegegeven.

In de prelude wordt een functie `foldl'` gedefinieerd, die hetzelfde doet als `foldl`, maar dan met niet-lazy tweede parameter. Ter vergelijking geven we eerst nog eens de definitie van `foldl`:

```
foldl f e []      =  e
foldl f e (x:xs)  =  foldl f (f e x) xs
```

In de functie `foldl'` wordt bij de recursieve aanroep de functie `strict` gebruikt zoals hiervoor beschreven:

```
foldl' f e []      =  e
foldl' f e (x:xs)  =  strict (foldl' f) (f e x) xs
```

Gebruikmakend van `foldl'` is de stackruimte constant, zodat rustig de som van hele lange lijsten berekend kan worden:

```
? foldl (+) 0 [1..6000]
ERROR: Control stack overflow
? foldl' (+) 0 [1..6000]
```

```
18003000
```

Voor operatoren zoals `+` en `*`, die de waarde van beide parameters nodig hebben om hun resultaat te berekenen, is het aan te bevelen om `foldl'` te gebruiken in plaats van `foldr` of `foldl`. Er zal daardoor nooit tekort aan stackruimte optreden. In de prelude is dan ook gedefinieerd:

```
sum     =  foldl' (+) 0
product =  foldl' (*) 1
```

Voor de operator `&&` wordt echter wel `foldr` gebruikt:

```
and     =  foldr (&&) True
```

Bij deze operator zorgt lazy evaluatie er immers voor dat de tweede parameter niet wordt uitgerekend als dat niet nodig is. Dat betekent dat `and` stopt met rekenen zodra er een waarde `False` in de lijst staat. Bij gebruik van `foldl'` zou hij altijd de hele lijst verwerken.

5.2 Wetten

5.2.1 Wiskundige wetten

Wiskundige functies hebben de prettige eigenschap dat hun resultaat niet afhangt van de context van de berekening. De waarde van $2 + 3$ is altijd 5, of deze expressie nu deel uitmaakt van de expressie $4 \times (2 + 3)$ of bijvoorbeeld $(2 + 3)^2$.

Veel operatoren voldoen aan bepaalde rekenregels. Zo geldt bijvoorbeeld voor alle getallen x en y dat $x + y = y + x$. Zo'n rekenregel wordt een *wet* genoemd. Enkele rekenkundige wetten zijn:

commutatieve wet voor $+$	$x + y = y + x$
commutatieve wet voor $\times$	$x \times y = y \times x$
associatieve wet voor $+$	$x + (y + z) = (x + y) + z$
associatieve wet voor $\times$	$x \times (y \times z) = (x \times y) \times z$
distributieve wet	$x \times (y + z) = (x \times y) + (x \times z)$
wet voor herhaald machtsverheffen	$(x^y)^z = x^{(y \times z)}$

Dit soort rekenregels kun je goed gebruiken om expressies te tranformeren tot expressies die dezelfde waarde hebben. Daardoor kun je uitgaande van bestaande wetten nieuwe wetten afleiden. Het bekende merkwaardige product $(a+b)^2 = a^2+2ab+b^2$ volgt bijvoorbeeld uit bovenstaande wetten:

$(a+b)^2$
$=$ (definitie kwadraat)
$(a+b) \times (a+b)$
$=$ (distributieve wet)
$((a+b) \times a) + (a+b) \times b$
$=$ (commutatieve wet voor $\times$ (twee keer))
$(a \times (a+b)) + (b \times (a+b))$
$=$ (distributieve wet (twee keer))
$(a \times a + a \times b) + (b \times a + b \times b)$
$=$ (associatieve wet voor $+$)
$a \times a + (a \times b + b \times a) + b \times b$
$=$ (definitie kwadraat (twee keer))
$a^2 + (a \times b + b \times a) + b^2$
$=$ (commutatieve wet voor $\times$)
$a^2 + (a \times b + a \times b) + b^2$
$=$ (definitie '$(2\times)$')
$a^2 + 2 \times a \times b + b^2$

In elke tak van wiskunde worden nieuwe functies en operatoren gedefinieerd, waarvoor ook weer rekenregels gelden. In de propositielogica bijvoorbeeld gelden de volgende regels om te 'rekenen' met boolse waarden:

commutatieve wet voor $\wedge$	$x \wedge y = y \wedge x$
associatieve wet voor $\wedge$	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
distributieve wet	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
wet van de Morgan	$\neg(x \wedge y) = \neg x \vee \neg y$
wet van Howard	$(x \wedge y) \rightarrow z = x \rightarrow (y \rightarrow z)$

Het handige van wetten is dat je er een aantal achter elkaar kunt toepassen, zonder dat je je druk hoeft te maken om de betekenis van de tussenliggende stappen. Zo kun je bijvoorbeeld de wet $\neg((a \vee b) \vee c) \rightarrow \neg d = \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d))$ afleiden met gebruik van bovenstaande wetten:

$\neg((a \vee b) \vee c) \rightarrow \neg d$
$=$ (wet van de Morgan)
$(\neg(a \vee b) \wedge \neg c) \rightarrow \neg d$
$=$ (wet van de Morgan)
$((\neg a \wedge \neg b) \wedge \neg c) \rightarrow \neg d$
$=$ (wet van Howard)
$(\neg a \wedge \neg b) \rightarrow (\neg c \rightarrow \neg d)$
$=$ (wet van Howard)
$\neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d))$

Zelfs als je niet zou weten wat $\wedge$, $\vee$ en $\neg$ betekenen kun je de geldigheid van de nieuwe wet inzien, mits je de geldigheid van de gebruikte wetten accepteert. Bij elke gelijkheid staat namelijk een aanwijzing ('hint') die aangeeftvolgens welke wet deze gelijkheid waar is. De hints zijn erg belangrijk. Als ze ontbreken, moet een lezer die twijfelt aan een stap zelf uitvinden welke wet gebruikt is. De aanwezigheid van hints draagt in belangrijke mate bij aan de 'leesbaarheid' van een afleiding.

5.2.2 Haskell-wetten

Haskell-functies hebben dezelfde eigenschap als wiskundige functies: een aanroep van een functie met dezelfde parameter levert altijd dezelfde waarde. Als je ergens in een expressie een deel-expressie vervangt door een andere deel-expressie die dezelfde waarde heeft, dan maakt dat voor het eindantwoord niet uit.

Voor allerlei functies is het mogelijk om wetten te formuleren waar ze aan voldoen. Zo geldt bijvoorbeeld voor alle functies `f` en alle lijsten `xs`:

```
(map f . map g) xs  =  map (f.g) xs
```

Dit is geen *definitie* van `map` of `.` (die zijn al eerder gedefinieerd); het is een *wet* waar deze functies aan blijken te voldoen.

Wetten voor Haskell-functies kun je gebruiken bij het schrijven van een programma. Je kunt er programma's overzichtelijker of sneller mee maken. In de definitie van de determinant-functie op
blz. 78 matrices (zie paragraaf 4.2.3) komt bijvoorbeeld de volgende expressie voor:

```
(altsum . zipWith (*) ry . map det . map Mat . gaps . transpose) rys
```

Door bovenstaande wet hierop toe te passen blijkt dat dit ook geschreven kan worden als

```
(altsum . zipWith (*) ry . map (det.Mat) . gaps . transpose) rys
```

Door gebruik te maken van wetten kan met complete programma's 'gerekend' worden. Programma's worden op die manier getransformeerd tot andere programma's. Net als bij het rekenen met getallen of proposities is het weer niet nodig om alle tussenliggende programma's (of zelfs maar het uiteindelijke programma) te begrijpen. Als het begin-programma goed is, de wetten zijn geldig, en je maakt geen fouten bij het toepassen van de wetten, dan doet het uiteindelijke programma gegarandeerd hetzelfde als het begin-programma.

Een aantal belangrijke wetten die gelden voor de Haskell-standaardfuncties zijn de volgende:

- functiecompositie is associatief, dus

  ```
  f . (g . h)  =  (f . g) . h
  ```

- `map f` distribueert over `++`, dus

  ```
  map f (xs++ys)  =  map f xs ++ map f ys
  ```

- de generalisatie hiervan naar een *lijst van* lijsten in plaats van *twee* lijsten:

```
map f . concat  =  concat . map (map f)
```

- `map` distribueert over samenstelling:

```
map (f.g)  =  map f . map g
```

- Als `f` associatief is (dus `x‘f‘(y‘f‘z)` = `(x‘f‘y)‘f‘z`), en `e` is het neutrale element van `f` (dus `x‘f‘e` = `e‘f‘x` = `x` voor alle `x`), dan geldt voor eindige lijsten `xs`:

```
foldr f e xs  =  foldl f e xs
```

- Als bij `foldr` de beginwaarde het neutrale element van de operator is, dan is `foldr` over een singleton-lijst de identiteit:

```
foldr f e [x]  =  x
```

- Een element op kop zetten van een lijst kan verwisseld worden met `map`, mits je dan de functiewaarde van het element op kop zet:

```
map f . (x:)  =  (f x :) . map f
```

- Elk gebruik van `map` kan ook geschreven worden als aanroep van `foldr`:

```
map f xs  =  foldr g [] xs  where g x ys = f x : ys
```

Veel van dit soort wetten komen overeen met wat men in imperatieve talen ‘programmeertrucs’ zou noemen. De vierde wet in bovenstaand rijtje zou men in een imperatieve taal bijvoorbeeld beschrijven als ‘samenvoegen van twee loops’. In imperatieve talen zijn de met de wetten overeenkomende programmatransformaties echter niet blindelings toe te passen: je moet er daar altijd op verdacht zijn dat functies onverwachte neveneffecten hebben. In functionele talen zoals Haskell mogen de wetten *altijd* toegepast worden; functies hebben immers altijd dezelfde waarde ongeacht de context.

5.2.3 Bewijzen van wetten

Van een aantal wetten is het intuïtief duidelijk dat ze gelden. Van andere wetten is de geldigheid niet op het eerste gezicht duidelijk. Vooral in het laatste geval, maar ook in het eerste, is het nuttig om de wet te *bewijzen*. Daarbij kan gebruik gemaakt worden van de definities van functies, en van eerder bewezen wetten. In paragraaf 5.2.1 werden op die manier al de wetten $(a+b)^2 = a^2+2ab+b^2$ blz. 93 en $\neg((a \vee b) \vee c) \rightarrow \neg d = \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d))$ bewezen.

Ook het bewijs van wetten voor Haskell-functies ziet er zo uit. Het is handig wetten een naam te geven, zodat je er later (bij het bewijs van andere wetten) eenvoudig aan kunt refereren. De formulering van een wet, compleet met bewijs, ziet er dan bijvoorbeeld als volgt uit:

Wet *foldr over een singleton-lijst*
Als `e` het neutrale element is van de operator `f`, dan geldt

```
foldr f e [x]  =  x
```

Bewijs:

```
foldr f e [x]
=   (notatie lijst-opsomming)
foldr f e (x:[])
=   (def. foldr)
f x (foldr f e [])
=   (def. foldr)
f x e
=   (voorwaarde van de wet)
x
```

Als een wet de gelijkheid van twee *functies* beschrijft, dan kan deze bewezen worden door te bewijzen dat het resultaat van de functie gelijk is voor alle mogelijke parameters. Beide functies worden dus op een variabele `x` toegepast, waarna gelijkheid wordt aangetoond. Dit is bijvoorbeeld het geval in de volgende wet:

Wet *functiecompositie is associatief*
Voor alle functies `f`, `g` en `h` van het juiste type geldt:

```
f . ( g . h )  =  ( f . g ) . h
```

Bewijs:

```
(f . (g.h)) x
=   (def. (.))
f ((g.h) x)
=   (def. (.))
f (g (h x))
=   (def. (.))
(f.g) (h x)
=   (def. (.))
((f.g) . h) x
```

Het nalezen van zo'n bewijs is niet altijd even spannend. Dat het schrijven van een bewijs lastig is, merk je pas als je het zelf moet bedenken. De eerste paar stappen zijn meestal niet zo moeilijk. Maar vaak kom je halverwege vast te zitten. Het is dan handig om ook een stukje van de andere kant te werken. Om aan te geven dat een bewijs op die manier opgebouwd is, zullen we het in het vervolg in twee kolommen weergeven:

	`f . ( g . h )`	`( f . g ) . h`

`x`	`(f . (g.h)) x`
	= (def. `(.)`)
	`f ((g.h) x)`
	= (def. `(.)`)
	`f (g (h x))`
		`((f.g) . h) x`
		= (def. `(.)`)
		`(f.g) (h x)`
		= (def. `(.)`)
		`f (g (h x))`

In de twee kolommen van deze opzet worden de twee kanten van de wet bewezen gelijk te zijn aan dezelfde expressie. Twee dingen die aan dezelfde expressie gelijk zijn, zijn natuurlijk ook aan elkaar gelijk, en dat moest bewezen worden. In de eerste kolom van het schema is nog aangegeven op welke parameter (`x`) de linker- en de rechterfunctie worden toegepast.

Deze bewijs-methode kan ook gebruikt worden om een andere wet te bewijzen:

Wet *map na op-kop*
Voor alle waarden `x` en functies `f` van het juiste type geldt:

```
map f . (x:)  =  ((f x):) . map f
```

Bewijs:

	`map f . (x:)`	`((f x):) . map f`

`xs`	`(map f . (x:)) xs`
	= (def. `(.)`)
	`map f ((x:) xs)`
	= (section-notatie)
	`map f (x:xs)`
	= (def. `map`)
	`f x : map f xs`
		`( ((f x):) . map f) xs`
		= (def. `(.)`)
		`((f x):) (map f xs)`
		= (section-notatie)
		`f x : map f xs`

Net als het vorige bewijs had dit bewijs natuurlijk ook als één lange afleiding geschreven kunnen worden. In deze tweekoloms-notatie is het echter duidelijker hoe het bewijs onstaan is: de twee kanten van de wet zijn gelijk bewezen aan een derde expressie, en daarmee ook aan elkaar.

5.2.4 Inductieve bewijzen

Functies op lijsten hebben vaak een inductieve definitie. De functie wordt daarbij apart gedefinieerd voor `[]`. Dan wordt de functie gedefinieerd voor het patroon `(x:xs)`, waarbij de functie recursief aangeroepen mag worden op `xs`.

Bij het bewijs van wetten waarin eindige lijsten een rol spelen, kan ook inductie worden gebruikt. De wet wordt daarbij apart bewezen voor het geval dat de lijst `[]` is. Vervolgens wordt de wet bewezen voor een lijst van de vorm `(x:xs)`. Bij dat bewijs mag al aangenomen worden dat de wet geldig is voor de lijst `xs`. Een voorbeeld van een wet die met inductie bewezen kan worden is de distributie van `map` over `++`.

Wet *map na* `++`
Voor alle functies `f` en alle lijsten `xs` en `ys` van het juiste type geldt:

```
map f (xs++ys)  =  map f xs ++ map f ys
```

Bewijs met inductie naar `xs`:

`xs`	`map f (xs++ys)`	`map f xs ++ map f ys`

`[]`	`map f ([]++ys)`
	= (def. `++`)
	`map f ys`
		`map f [] ++ map f ys`
		= (def. `map`)
		`[] ++ map f ys`
		= (def. `++`)
		`map f ys`

`x:xs`	`map f ((x:xs)++ys)`
	= (def. `++`)
	`map f (x:(xs++ys))`
	= (def. `map`)
	`f x : map f (xs++ys)`
		`map f (x:xs) ++ map f ys`
		= (def. `map`)
		`(f x : map f xs) ++ map f ys`
		= (def. `++`)
		`f x : (map f xs ++ map f ys)`

In dit bewijs wordt in het eerste gedeelte de wet geformuleerd voor de lijst `xs`. Dit heet de *inductiehypothese*. In het tweede gedeelte wordt de wet bewezen voor de lege lijst: hier is dus `[]` ingevuld waar in de originele wet `xs` stond. In het derde gedeelte wordt de wet bewezen met `(x:xs)` ingevuld voor `xs`. De laatste regel van de twee kolommen is weliswaar niet dezelfde expressie, maar omdat in dit gedeelte van het bewijs de inductiehypothese aangenomen mag worden, is de gelijkheid toch geldig.

In de wet 'map na functiecompositie' worden twee functies gelijkgesteld. Om deze gelijkheid te bewijzen, worden linker- en rechterkant op een parameter `xs` toegepast. Daarna verloopt het bewijs met inductie naar `xs`:

Wet *map na functiecompositie*
Voor alle samenstelbare functies `f` en `g` geldt:

```
map (f.g)  =  map f . map g
```

Bewijs met inductie naar `xs`:

	`map (f.g)`	`map f . map g`

`xs`	`map (f.g) xs`	`(map f . map g) xs`
		= (def. `(.)`)
		`map f (map g xs)`

`[]`	`map (f.g) []`
	= (def. `map`)
	`[]`
		`map f (map g [])`
		= (def. `map`)
		`map f []`
		= (def. `map`)
		`[]`

`x:xs`	`map (f.g) (x:xs)`
	= (def. `map`)
	`(f.g) x : map (f.g) xs`
	= (def. `(.)`)
	`f(g x) : map (f.g) xs`
		`map f (map g (x:xs))`
		= (def. `map`)
		`map f (g x : map g xs)`
		= (def. `map`)
		`f(g x) : map f (map g xs)`

In dit bewijs is de rechterkant van de stelling eerst nog vereenvoudigd, voordat de eigenlijke inductie begint; deze stap zou anders in beide gedeeltes van het inductieve bewijs gedaan moeten worden. Dat gebeurt ook in het bewijs van de volgende wet:

Wet *map na concat*
Voor alle functies `f` geldt:

```
map f . concat  =  concat . map (map f)
```

Dit is een generalisatie van de distributiewet van `map` over `++`.

Bewijs met inductie naar `xss`:

	`map f . concat`	`concat . map (map f)`

`xss`	`(map f . concat) xss`
	= (def. `(.)`)
	`map f (concat xss)`
		`(concat . map (map f)) xss`
		= (def. `(.)`)
		`concat (map (map f) xss)`

`[]`	`map f (concat [])`
	= (def. `concat`)
	`map f []`
	= (def. `map`)
	`[]`
		`concat (map (map f) [])`
		= (def. `map`)
		`concat []`
		= (def. `concat`)
		`[]`

`xs:xss`	`map f (concat (xs:xss))`
	= (def. `concat`)
	`map f (xs++concat xss)`
	= (distributie-wet)
	`map f xs`
	`++ map f (concat xss)`
		`concat (map (map f) (xs:xss))`
		= (def. `map`)
		`concat (map f xs : map (map f) xss)`
		= (def. `concat`)
		`map f xs`
		`++ concat (map (map f) xss)`

Ook nu weer zijn de expressies in de onderste regels gelijk vanwege de aanname van de inductiehypothese. In dit bewijs wordt behalve de definitie van functies en notaties ook een andere wet gebruikt, namelijk de eerder bewezen distributiewet van `map` over `++`.

Niet altijd is het gevalsonderscheid `[]`/`(x:xs)` voldoende om een wet te bewijzen. Datzelfde geldt trouwens voor de definitie van functies. In het bewijs van de volgende wet worden *drie* gevallen onderscheiden: de lege lijst, een singleton-lijst, en een lijst met minstens twee elementen `(x1:x2:xs)`.

Wet *dualiteitswet*
Als `f` een associatieve operator is (dus `x‘f‘(y‘f‘z) = (x‘f‘y)‘f‘z`), en `e` is het neutrale element van `f` (dus `f x e = f e x = x` voor alle `x`), dan geldt:

```
foldr f e  =  foldl f e
```

Bewijs met inductie naar `xs`:

	`foldr f e`	`foldl f e`
`xs`	`foldr f e xs`	`foldl f e xs`

`[]`	`foldr f e []`
	= (def. `foldr`)
	`e`
		`foldl f e []`
		= (def. `foldl`)
		`e`

`[x]`	`foldr f e [x]`
	= (def. `foldr`)
	`f x (foldr f e [])`
	= (def. `foldr`)
	`f x e`
	= (`e` neutraal element)
	`x`
		`foldl f e [x]`
		= (def. `foldl`)
		`foldl f (e‘f‘x) []`
		= (def. `foldl`)
		`e‘f‘x`
		= (`e` neutraal element)
		`x`

`x1:x2:xs`	`foldr f e (x1:x2:xs)`
	= (def. `foldr`)
	`x1 ‘f‘ foldr f e (x2:xs)`
	= (def. `foldr`)
	`x1 ‘f‘ (x2 ‘f‘ foldr f e xs)`
	= (`f` associatief)
	`(x1‘f‘x2) ‘f‘ foldr f e xs`
	= (def. `foldr`)
	`foldr f e ((x1‘f‘x2):xs)`
		`foldl f e (x1:x2:xs)`
		= (def. `foldl`)
		`foldl f (e‘f‘x1) (x2:xs)`
		= (def. `foldl`)
		`foldl f ((e‘f‘x1)‘f‘x2) xs`
		= (`f` associatief)
		`foldl f (e‘f‘(x1‘f‘x2)) xs`
		= (def. `foldl`)
		`foldl f e ((x1‘f‘x2):xs)`

De laatste twee regels zijn gelijk omdat de lijst `(x1‘f‘x2):xs` korter is dan `x1:x2:xs`. De wet mag voor deze lijst dus al aangenomen worden.

5.2.5 Verbetering van efficiëntie

Wetten kunnen gebruikt worden om functies te transformeren in efficiëntere functies. Twee expressies waarvan de gelijkheid bewezen is, hoeven immers niet even snel berekend te worden; in dat geval kan de langzamere definitie vervangen worden door de snellere. In deze paragraaf worden twee voorbeelden van deze techniek bekeken:

- de `reverse`-functie wordt verbeterd van $\mathcal{O}(n^2)$ tot $\mathcal{O}(n)$;
- de Fibonacci-functie, die al eerder was verbeterd van $\mathcal{O}(2^n)$ tot $\mathcal{O}(n)$ wordt verder verbeterd tot $\mathcal{O}(\log n)$.

reverse

Voor het verbeteren van de `reverse`-functie bewijzen we drie wetten:

- Naast het in de vorige paragraaf bewezen verband tussen `foldr` en `foldl` is er nog een verband: de *tweede dualiteitswet*:

```
foldr f e (reverse xs)  =  foldl (flip f) e xs
```

- Het bewijs van bovenstaande wet lukt niet in één keer. Er is een andere wet bij nodig, die apart met inductie bewezen kan worden:

```
foldr f e (xs++[y])  =  foldr f (f y e) xs
```

- Zo ongeveer de eenvoudigste wet die met inductie bewezen kan worden is:

```
foldr (:) []  =  id
```

We beginnen met de derde wet. Daarna volgt een bewijs van de hulp-wet, en vervolgens de tweede dualiteitswet zelf. Met deze dualiteitswet verbeteren we tenslotte de `reverse`-functie.

Wet *foldr met constructorfuncties*
Als aan `foldr` de constructorfuncties van lijsten worden meegegeven, te weten `(:)` en `[]`, is het resultaat de identiteit:

```
foldr (:) []  =  id
```

Bewijs met inductie naar `xs`:

	`foldr (:) []`	`id`

`xs`	`foldr (:) [] xs`	`id xs`
		= (def. `id`)
		`xs`

`[]`	`foldr (:) [] []`
	= (def. `foldr`)
	`[]`
		`[]`

`x:xs`	`foldr (:) [] (x:xs)`
	= (def. `foldr`)
	`x : foldr (:) [] xs`
		`x : xs`

Wet *hulpwet voor de volgende wet*

```
foldr f e (as++[b])  =  foldr f (f b e) as
```

Bewijs met inductie naar `as`:

`as`	`foldr f e (as++[b])`	`foldr f (f b e) as`

`[]`	`foldr f e ([]++[b])`
	= (def. `++`)
	`foldr f e [b]`
	= (def. `foldr`)
	`f b (foldr f e [])`
	= (def. `foldr`)
	`f b e`
		`foldr f (f b e) []`
		= (def. `foldr`)
		`f b e`

`a:as`	`foldr f e ((a:as)++[b])`
	= (def. `++`)
	`foldr f e (a:(as++[b]))`
	= (def. `foldr`)
	`f a (foldr f e (as++[b]))`
		`foldr f (f b e) (a:as)`
		= (def. `foldr`)
		`f a (foldr f (f b e) as)`

Wet *tweede dualiteitswet*
Voor alle functies `f`, waardes `e` en lijsten `xs` van het juiste type geldt:

```
foldr f e (reverse xs)  =  foldl (flip f) e xs
```

Bewijs met inductie naar `xs`:

`xs`	`foldr f e (reverse xs)`	`foldl (flip f) e xs`

`[]`	`foldr f e (reverse [])`
	= (def. `reverse`)
	`foldr f e []`
	= (def. `foldr`)
	`e`
		`foldl (flip f) e []`
		= (def. `foldl`)
		`e`

`x:xs`	`foldr f e (reverse (x:xs))`
	= (def. `reverse`)
	`foldr f e (reverse xs++[x])`
	= (hulpwet hierboven)
	`foldr f (f x e) (reverse xs)`
		`foldl (flip f) e (x:xs)`
		= (def. `foldl`)
		`foldl (flip f) (flip f e x) xs`
		= (def. `flip`)
		`foldl (flip f) (f x e) xs`

De laatste expressies in dit bewijs zijn gelijk vanwege de inductiehypothese. De inductiehypothese mag worden aangenomen voor *alle* `e`, dus ook met `f x e` ingevuld voor `e`. (De inductiehypothese mag daarentegen voor de variabele waarnaar de inductie verloopt, `xs`, alleen voor vaste `xs` aangenomen worden; anders valt de hele inductie in duigen.)

blz. 41 De functie `reverse` is in paragraaf 3.1.2 gedefinieerd als

```
reverse []     =  []
reverse (x:xs) =  reverse xs ++ [x]
```

of het daaraan equivalente

```
reverse  =  foldr post []
     where  post x xs = xs++[x]
```

Op deze manier gedefinieerd kost de functie $\mathcal{O}(n^2)$ tijd, waarbij n de lengte van de om te keren lijst is. De operator `++` in `post` kost immers $\mathcal{O}(n)$ tijd, en dit moet vermenigvuldigd worden met de $\mathcal{O}(n)$ van de recursie (al of niet verborgen in `foldr`).

Maar uit de zojuist bewezen wetten kunnen we afleiden:

```
reverse xs
=   (def. id)
id (reverse xs)
=   (foldr met constructorfuncties)
foldr (:) [] (reverse xs)
=   (tweede dualiteitswet)
foldl (flip (:)) [] xs
```

De nieuwe definitie

```
reverse  =  foldl (flip (:)) []
```

kost slechts $\mathcal{O}(n)$ tijd. De operator die voor `foldl` gebruikt wordt, `flip (:)`, kost immers slechts constante tijd.

Fibonacci

Ook wetten waarin natuurlijke getallen een rol spelen, kunnen soms met inductie worden bewezen. Daarbij wordt de wet apart bewezen voor het geval 0, en daarna voor het patroon $n{+}1$, waarbij de wet voor het geval n al gebruikt mag worden. In sommige bewijzen wordt een ander inductieschema aangehouden, bijvoorbeeld $0/1/n{+}2$; of $1/n{+}2$ als de wet niet hoeft te gelden voor het geval 0.

Inductie over natuurlijke getallen kunnen we goed gebruiken om een verbetering in de efficiëntie van de Fibonacci-functie te bereiken. De oorspronkelijke definitie daarvan was:

```
fib 0      =  0
fib 1      =  1
fib (n+2)  =  fib n + fib (n+1)
```

blz. 89 De benodigde tijd voor het berekenen van `fib n` is $\mathcal{O}(2^n)$. In paragraaf 5.1.3 werd al een verbetering tot $\mathcal{O}(n)$ bereikt, maar dankzij de volgende wet is een nog grotere verbetering mogelijk.

Wet *Fibonacci door machtsverheffen*
Stel $p = \frac{1}{2} + \frac{1}{2}\sqrt{5}$, $q = \frac{1}{2} - \frac{1}{2}\sqrt{5}$, en $c = 1/\sqrt{5}$. Dan geldt:

```
fib n  =  c * (p^n - q^n)
```

De waarden p en q zijn de oplossingen van de vierkantsvergelijking $x^2 - x - 1 = 0$. Daarom geldt $p^2 = p + 1$ en $q^2 = q + 1$. Bovendien geldt $p - q = \sqrt{5} = 1/c$. Gebruik makend van deze eigenschappen bewijzen we de stelling.

Bewijs van 'Fibonacci door machtsverheffen' met inductie naar n:

n	`fib` n	$c \times (p^n - q^n)$
0	`fib 0` = (def. `fib`) 0 = (def. $\times$) $c \times 0$	$c \times (p^0 - q^0)$ = (def. machtsverheffen) $c \times (1 - 1)$ = (eigenschap $-$) $c \times 0$
1	`fib 1` = (def. `fib`) 1 = (def. /) $c \times (1/c)$	$c \times (p^1 - q^1)$ = (def. machtsverheffen) $c \times (p - q)$ = (eigenschap c) $c \times (1/c)$
n+2	`fib` (n+2) = (def. `fib`) `fib` n `+ fib` (n+1)	$c \times (p^{n+2} - q^{n+2})$ = (eigenschap machtsverheffen) $c \times (p^n p^2 - q^n q^2)$ = (eigenschap p en q) $c \times (p^n(1+p) - q^n(1+q))$ = (distributie $\times$) $c \times ((p^n + p^{n+1}) - (q^n + q^{n+1}))$ = (commutativiteit en associativiteit $+$) $c \times ((p^n - q^n) + (p^{n+1} - q^{n+1}))$ = (distributie $\times$) $c \times (p^n - q^n) + c \times (p^{n+1} - q^{n+1})$

Het opmerkelijke aan deze wet is dat ondanks al die wortels het eindantwoord toch weer geheeltallig is. Deze wet kan gebruikt worden om een $\mathcal{O}(\log n)$ versie van `fib` te maken. Machtsverheffen kan immers in $\mathcal{O}(\log n)$ tijd, met de halverings-methode. Door `fib` te definiëren door

```
fib n  =  c * (p^n - q^n)
   where  c = 1/wortel5
          p = 0.5 * (1+wortel5)
          q = 0.5 * (1-wortel5)
          wortel5 = sqrt 5
```

kan ook `fib` in logaritmische tijd berekend worden. Een laatste optimalisatie is mogelijk door op te merken dat $|q| < 1$, en dat dus q^n, zeker voor grote n, verwaarloosd kan worden. Het is voldoende om $c \times p^n$ af te ronden op de dichtstbijzijnde integer.

5.2.6 Eigenschappen van functies

Behalve voor het verbeteren van de efficiëntie van bepaalde functies zijn wetten ook gewoon handig om meer inzicht te krijgen in de werking van bepaalde functies. Bij functies die een lijst opleveren is het bijvoorbeeld interessant om te weten hoe de lengte afhangt van de parameter van die functie. Hieronder volgen vier wetten over de lengte van het resultaat van een functie. Daarna volgen drie wetten over de som van de resultaat-lijst van een functie. De wetten worden daarna gebruikt om iets te kunnen zeggen over de lengte van het resultaat van combinatorische functies.

Wetten over lengte

In deze paragraaf wordt de `length`-functie geschreven als `len` (om schrijfwerk te besparen).

Wet *lengte na op-kop*
Door een element op kop te zetten van een lijst wordt de lengte één groter:

```
len . (x:)  =  (1+) . len
```

Deze wet volgt vrijwel direct uit de definitie. Er is geen inductie nodig:

	`len . (x:)`	`(1+) . len`

`xs`	`(len . (x:)) xs`
	= (def. `(.)`)
	`len (x:xs)`
	= (def. `len`)
	`1 + len xs`
		`((1+) . len) xs`
		= (def. `(.)`)
		`1 + len xs`

Wet *lengte na map*
Door het `map`-pen van een functie op een lijst blijft de lengte van de lijst onveranderd:

```
len . map f  =  len
```

Het bewijs verloopt met inductie naar `xs`:

	`len . map f`	`len`

`xs`	`(len . map f) xs`
	= (def. `(.)`)
	`len (map f xs)`
		`len xs`

`[]`	`len (map f [])`
	= (def. `map`)
	`len []`
		`len []`

`x:xs`	`len (map f (x:xs))`
	= (def. `map`)
	`len (f x : map f xs)`
	= (def. `len`)
	`1 + len (map f xs)`
		`len (x:xs)`
		= (def. `len`)
		`1 + len xs`

Wet *lengte na ++*
De lengte van de concatenatie van twee lijsten is de som van de lengtes van die lijsten:

```
len (xs++ys)  =  len xs + len ys
```

Het bewijs verloopt met inductie naar `xs`:

`xs`	`len (xs++ys)`	`len xs + len ys`

`[]`	`len ([]++ys)`
	= (def. `++`)
	`len ys`
		`len [] + len ys`
		= (def. `len`)
		`0 + len ys`
		= (def. `+`)
		`len ys`

`x:xs`	`len ((x:xs)++ys)`
	= (def. `++`)
	`len (x:(xs++ys))`
	= (def. `len`)
	`1 + len (xs++ys)`
		`len (x:xs) + len ys`
		= (def. `len`)
		`(1+len xs) + len ys`
		= (associativiteit `+`)
		`1 + (len xs + len ys)`

De volgende wet is een generalisatie hiervan: in deze wet komt een lijst van lijsten voor, in plaats van twee lijsten, en de operator `+` is dan ook vervangen door `sum`.

Wet *lengte na concatenatie*
De lengte van een concatenatie van een lijst van lijsten is de som van de lengtes van al die lijsten:

```
len . concat  =  sum . map len
```

Het bewijs verloopt met inductie naar `xss`:

	`len . concat`	`sum . map len`
`xss`	`len (concat xss)`	`sum (map len xss)`

`[]`	`len (concat [])`
	= (def. `concat`)
	`len []`
	= (def. `len`)
	`0`
		`sum (map len [])`
		= (def. `map`)
		`sum []`
		= (def. `sum`)
		`0`

`xs:xss`	`len (concat (xs:xss))`
	= (def. `concat`)
	`len (xs++concat xss)`
	= (lengte na `++`)
	`len xs + len (concat xss)`
		`sum (map len (xs:xss))`
		= (def. `map`)
		`sum (len xs : map len xss)`
		= (def. `sum`)
		`len xs + sum (map len xss)`

Wetten over sum

Net als voor `len` zijn er voor `sum` twee wetten om hem over concatenatie te distribueren (van twee lijsten of van een lijst van lijsten).

Wet *sum na ++*
De som van de concatenatie van twee lijsten is gelijk aan de sommen van die twee lijsten opgeteld:

```
sum (xs++ys)  =  sum xs + sum ys
```

Bewijs met inductie naar `xs`:

`xs`	`sum (xs++ys)`	`sum xs + sum ys`

`[]`	`sum ([]++ys)`
	= (def. `++`)
	`sum ys`
		`sum [] + sum ys`
		= (def. `sum`)
		`0 + sum ys`
		= (def. `+`)
		`sum ys`

`x:xs`	`sum ((x:xs)++ys)`
	= (def. `++`)
	`sum (x:(xs++ys))`
	= (def. `sum`)
	`x + sum(xs++ys)`
		`sum (x:xs) + sum ys`
		= (def. `sum`)
		`(x+sum xs) + sum ys`
		= (associativiteit `+`)
		`x + (sum xs + sum ys)`

Net als bij `len` wordt deze wet gebruikt in het bewijs van de generalisatie naar lijsten van lijsten.

Wet *sum na concatenatie*
De som van de concatenatie van een lijst van lijsten is de som van de sommen van die lijsten:

```
sum . concat  =  sum . map sum
```

Het bewijs verloopt met inductie naar `xss`:

	`sum . concat`	`sum . map sum`
`xss`	`sum (concat xss)`	`sum (map sum xss)`

`[]`	`sum (concat [])`
	= (def. `concat`)
	`sum []`
		`sum (map sum [])`
		= (def. `map`)
		`sum []`

`xs:xss`	`sum (concat (xs:xss))`
	= (def. `concat`)
	`sum (xs ++ concat xss)`
	= (sum na `++`)
	`sum xs + sum (concat xss)`
		`sum (map sum (xs:xss))`
		= (def. `map`)
		`sum (sum xs : map sum xss)`
		= (def. `sum`)
		`sum xs + sum (map sum xss)`

Er is geen wet voor de `sum` van een `map` op een lijst, zoals die voor `len` gold. De som van de kwadraten van een getal is immers niet gelijk aan het kwadraat van de som of iets dergelijks. Wel is er een wet te formuleren voor het geval de ge`map`te functie de functie `(1+)` is.

Wet *sum na map-plus-1*
De som van een lijst opgehoogde getallen is de som van de oorspronkelijke lijst plus de lengte ervan:

```
sum (map (1+) xs)  =  len xs + sum xs
```

Bewijs met inductie naar `xs`:

`xs`	`sum (map (1+) xs)`	`len xs + sum xs`

`[]`	`sum (map (1+) [])`
	= (def. `map`)
	`sum []`
		`len [] + sum []`
		= (def. `len`)
		`0 + sum []`
		= (def. `+`)
		`sum []`

`x:xs`	`sum (map (1+) (x:xs))`
	= (def. `map`)
	`sum (1+x : map (1+) xs)`
	= (def. `sum`)
	`(1+x) + sum (map (1+) xs)`
		`len (x:xs) + sum (x:xs)`
		= (def. `len` en `sum`)
		`(1+len xs) + (x+sum xs)`
		= (`+` associatief en commutatief)
		`(1+x) + (len xs + sum xs)`

In opgave 5.4 wordt deze wet gegeneraliseerd naar een willekeurige lineaire functie. blz. 111

Wetten over combinatorische functies

Met behulp van een aantal hierboven genoemde wetten zijn wetten te bewijzen over combinatorische functies uit sectie 4.1. We bewijzen voor `inits`, `segs` en `combs` een wet die aangeeft hoeveel elementen het resultaat heeft (zie ook opgave 4.3): blz. 67 blz. 84

```
len . inits    =  (1+) . len
len . segs     =  f    . len     where f n = 1 + (n*n+n)/2
len . combs k  =  (`boven` k) . len
```

Wet *aantal beginsegmenten*
Het aantal beginsegmenten van een lijst is één meer dan het aantal elementen van de lijst:

```
len . inits  =  (1+) . len
```

Het bewijs verloopt met inductie naar `xs`:

	`len . inits`	`(1+) . len`
`xs`	`len (inits xs)`	`1 + len xs`

`[]`	`len (inits [])`
	= (def. `inits`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
		`1 + len []`

`x:xs`	`len (inits (x:xs))`
	= (def. `inits`)
	`len ([] : map (x:) (inits xs))`
	= (def. `len`)
	`1 + len (map (x:) (inits xs))`
	= (lengte na map)
	`1 + len (inits xs)`
		`1 + len (x:xs)`
		= (def. `len`)
		`1 + (1+len xs)`

Wet *aantal segmenten*
Het aantal segmenten van een lijst is een kwadratische functie van het aantal elementen van de lijst:

```
len . segs  =  f . len  where f n = 1 + (n*n+n)/2
```

Het bewijs verloopt met inductie naar `xs`. We schrijven n voor `len xs`.

	`len . segs`	`f . len where f` n `=` $1+(n^2+n)/2$
`xs`	`len (segs xs)`	`f (len xs)`

`[]`	`len (segs [])`
	= (def. `segs`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
	= (def. `len`)
	`1 + 0`
		`f (len [])`
		= (def. `len`)
		`f 0`
		= (def. `f`)
		$1+(0^2+0)/2$
		= (uitrekenen)
		`1 + 0`

`x:xs`	`len (segs (x:xs))`
	= (def. `segs`)
	`len ( segs xs`
	`++ map (x:)(inits xs))`
	= (lengte na `++`)
	`len (segs xs) +`
	`len (map (x:)(inits xs))`
	= (lengte na map)
	`len (segs xs) + len (inits xs)`
	= (aantal beginsegmenten)
	`len (segs xs) + 1 + len xs`
		`f (len (x:xs))`
		= (def. `len`)
		`f` $(1+n)$
		= (def. `f`)
		$1+((1+n)^2+(1+n))/2$
		= (merkwaardig product)
		$1+((1+2n+n^2)+(1+n))/2$
		= (+ associatief en commutatief)
		$1+((n^2+n)+(2+2n))/2$
		= (distributie /)
		$1+(n^2+n)/2+1+n$

blz. 4 De definitie van `boven` in paragraaf 1.2.2 was niet volledig. De complete definitie luidt:

```
boven n k  |  n>=k  =  fac n / (fac k * fac (n-k))
           |  n<k   =  0
```

Deze functie speelt een rol in de volgende wet.

Wet *aantal combinaties*
Het aantal combinaties van k elementen uit een lijst is de binomiaalcoëfficiënt 'lengte van de lijst boven k':

```
len . combs k  =  (`boven` k) . len
```

In het bewijs gebruiken we de klassieke wiskundige notatie $n!$ voor `fac n`, en $\binom{n}{k}$ voor `n `boven` k`. We schrijven n voor `len xs`. Het bewijs verloopt met inductie naar `k` (met de gevallen `0` en `k+1`). Het bewijs van de inductiestap (geval `k+1`) verloopt opnieuw met inductie, ditmaal naar `xs` (met

de gevallen `[]` en `x:xs`). Deze inductiestructuur komt overeen met die van de definitie van `combs`.

	`len . combs k`	`(‘boven‘ k) . len`
`k` `xs`	`len (combs k xs)`	$\binom{\texttt{len xs}}{k}$

`0`
`xs`
	`len (combs 0 xs)`
	= (def. `combs`)
	`len [[]]`
	= (def. `len`)
	`1 + len []`
	= (def. `len`)
	`1 + 0`
	= (eigenschap `+`)
	`1`
		$\binom{\texttt{len xs}}{0}$
		= (def. `boven`)
		$\frac{n!}{(n-0)!*0!}$
		= (def. `fac` en $-$)
		$\frac{n!}{n!*1}$
		= (def. / en $*$)
		1

`k+1`
`[]`
	`len (combs (k+1) [])`
	= (def. `combs`)
	`len []`
	= (def. `len`)
	`0`
		$\binom{\texttt{len []}}{k+1}$
		= (def. `len`)
		$\binom{0}{k+1}$
		= (def. `boven`)
		0

`k+1`
`x:xs`
	`len (combs (k+1) (x:xs))`
	= (def. `combs`)
	`len (map (x:)(combs k xs)`
	`++ combs (k+1) xs)`
	= (lengte na `++`)
	`len (map (x:)(combs k xs))`
	`+ len (combs (k+1) xs)`
	= (lengte na map)
	`len (combs k xs)`
	`+ len (combs (k+1) xs)`
		$\binom{\texttt{len (x:xs)}}{k+1}$
		= (def. `len`)
		$\binom{n+1}{k+1}$
		= (def. `boven` $(n \geq k)$)
		$\frac{(n+1)!}{((n+1)-(k+1))!*(k+1)!}$
		= (teller: def. `fac`; noemer: rekenen)
		$\frac{(n+1)*n!}{(n-k)!*(k+1)!}$
		= (teller: rekenen; noemer: def. `fac` $(n>k)$)
		$\frac{(k+1)*n!}{(n-k)!*(k+1)*k!} + \frac{(n-k)*n!}{(n-k)*(n-k-1)!*(k+1)!}$
		= (delen $(n>k)$)
		$\frac{n!}{(n-k)!*k!} + \frac{n!}{(n-(k+1))!*(k+1)!}$
		= (def. `boven`)
		$\binom{n}{k} + \binom{n}{k+1}$

De rechterkolom van het inductiestap-bewijs is alleen geldig voor $n > k$. Voor $n = k$ verloopt het bewijs als volgt:

$$\binom{n+1}{k+1} \;=\; 1 \;=\; 1+0 \;=\; \binom{n}{k} + \binom{n}{k+1}$$

Voor $n < k$ luidt het bewijs:

$$\binom{n+1}{k+1} \;=\; 0 \;=\; 0+0 \;=\; \binom{n}{k} + \binom{n}{k+1}$$

5.2.7 Polymorfie

De volgende wetten zijn geldig voor alle functies `f`:

```
inits . map f  =  map (map f) . inits
segs  . map f  =  map (map f) . segs
subs  . map f  =  map (map f) . subs
perms . map f  =  map (map f) . perms
```

Intuïtief is het wel duidelijk dat deze wetten gelden. Stel bijvoorbeeld dat je met `inits` de beginsegmenten van een lijst berekent, nadat je van alle elementen de `f`-waarde hebt berekend (door

`map f`). Je had dan ook eerst de `inits` kunnen bepalen, en daarna in *elk* resulterend beginsegment op alle elementen `f` toepassen. In dat laatste geval (voorgesteld door de rechterkant van de wet) moet je `f` toepassen op de elementen van een lijst van lijsten, vandaar de dubbele `map`.

We bewijzen de eerste van de genoemde wetten; de andere zijn niet veel moeilijker.

Wet *beginsegmenten na map*
Voor all functies `f` op lijsten geldt:

```
inits . map f  =  map (map f) . inits
```

Het bewijs verloopt met inductie naar `xs`:

	`inits . map f`	`map (map f) . inits`
`xs`	`inits (map f xs)`	`map (map f) (inits xs)`

`[]`	`inits (map f [])`
	= (def. `map`)
	`inits []`
	= (def. `inits`)
	`[[]]`
		`map (map f) (inits [])`
		= (def. `inits`)
		`map (map f) [[]]`
		= (def. `map`)
		`[ map f [] ]`
		= (def. `map`)
		`[[]]`

`x:xs`	`inits (map f (x:xs))`
	= (def. `map`)
	`inits (f x:map f xs)`
	= (def. `inits`)
	`[] : map (f x:)`
	`(inits (map f xs))`
		`map (map f)(inits (x:xs))`
		= (def. `inits`)
		`map (map f)([]:map (x:)(inits xs))`
		= (def. `map`)
		`map f [] :`
		`map (map f)(map (x:)(inits xs))`
		= (def. `map`)
		`[] : map (map f)(map (x:)(inits xs))`
		= (map na functiecompositie)
		`[] : map (map f.(x:)) (inits xs)`
		= (map na op-kop)
		`[] : map ((f x:).map f) (inits xs)`
		= (map na functiecompositie)
		`[] : map (f x:)`
		`(map (map f)(inits xs))`

Een soortgelijke wet als de zojuist bewezene geldt voor *elke* combinatorische functie. Dat wil zeggen: als `combinat` een combinatorische functie is, dan geldt voor alle functies `f` dat

```
combinat . map f  =  map (map f) . combinat
```

Dat komt door de definitie van wat voor soort functies 'combinatorische functie' genoemd worden: functies van lijsten naar lijsten van lijsten, die geen gebruik mogen maken van specifieke eigenschappen van elementen. Anders gezegd: combinatorische functies zijn polymorfe functies met als type

```
combinat  ::  [a] -> [[a]]
```

Het is zelfs zo, dat bovengenoemde wet als *definitie* van combinatorische functies gebruikt kan worden. Dus: een functie `combinat` heet 'combinatorisch' als voor alle functies `f` geldt:

```
combinat . map f  =  map (map f) . combinat
```

Met zo'n definitie, die een duidelijke omschrijving geeft met behulp van een wet, kun je meestal wat beter uit de voeten dan de enigszins vage omschrijving 'mag geen gebruik maken van specifieke
blz. 67 eigenschappen van elementen', die in sectie 4.1 werd gebruikt.

blz. 96 Er zijn wetten die lijken op deze 'wet van de combinatorische functies'. In paragraaf 5.2.4 werd bijvoorbeeld de wet 'map na concat' bewezen. Die wet stelt dat voor alle functies `f` geldt:

```
map f . concat  =  concat . map (map f)
```

De wet kun je natuurlijk ook andersom lezen. Dan staat er:

```
concat . map (map f)  =  map f . concat
```

In deze vorm lijkt de wet op de wet van de combinatorische functies. Het enige verschil is, dat de 'dubbele map' nu aan de andere kant staat. Dat is ook niet zo gek, want het type van `concat` is:

```
concat  ::  [[a]] -> [a]
```

Neemt bij gebruik van combinatorische functies het aantal lijst-nivo's toe, bij `concat` vermindert

dat aantal juist. De dubbele map moet dan ook gebruikt worden *voordat* `concat` wordt toegepast; de enkele map er*na*.

De functie `concat` is geen combinatorische functie, om de eenvoudige reden dat hij niet aan de daarvoor geldende wet voldoet. Wel is de functie een *polymorfe* functie. In paragraaf 1.5.3 werd een polymorfe functie gedefinieerd als 'een functie met een type waar type-variabelen in voorkomen'. Net als het begrip 'combinatorische functie' is het begrip 'polymorfe functie' met een wet minder vaag te definiëren. Dat gaat als volgt: blz. 16

> Een functie `poly` tussen lijsten heet een *polymorfe* functie als voor alle functies `f` geldt:
>
> $$\texttt{poly . } \underbrace{\texttt{map (...(map f))}}_{n\ \texttt{map's}} = \underbrace{\texttt{map (...(map f))}}_{k\ \texttt{map's}} \texttt{ . poly}$$
>
> Deze functie heeft dan het type:
>
> $$\texttt{poly :: } \underbrace{\texttt{[...[a]...]}}_{n\text{ dimensionale lijst}} \texttt{ -> } \underbrace{\texttt{[...[a]...]}}_{k\text{ dimensionale lijst}}$$

Alle combinatorische functies zijn polymorf. De wet die voor combinatorische functies moet gelden is immers een speciaal geval van de wet voor polymorfe functies, met $n = 1$ en $k = 2$. Ook `concat` is polymorf: de wet 'map na concat' heeft de geëiste vorm, met $n = 2$ en $k = 1$.

Ook voor andere datastructuren dan lijsten (bijvoorbeeld tupel, of bomen) kan het begrip 'polymorfe functie' met behulp van een wet gedefinieerd worden. Er is dan een equivalent van `map` op de betreffende datastructuur nodig, die in de wet gebruikt kan worden in plaats van `map`.[2]

5.2.8 Bewijzen van rekenkundige wetten

In paragraaf 5.2.1 is een aantal wiskundige wetten genoemd, zoals 'vermenigvuldigen is associatief'. Deze wetten kunnen ook bewezen worden. Bij een bewijs van een wet waarin een bepaalde functie een rol speelt, is echter de definitie van die functie nodig. Tot nu toe hebben we nog geen definitie gegeven van optellen en vermenigvuldigen; deze functies werden als 'ingebouwd' beschouwd. blz. 93

In theorie is het niet nodig dat de getallen, althans de natuurlijke getallen, in Haskell zijn ingebouwd. Het is namelijk mogelijk om ze te definiëren door middel van een data-declaratie. In deze paragraaf zullen we de definitie van het type `Nat` (de natuurlijke getallen) geven, om twee redenen:

- om aan te tonen hoe krachtig het data-declaratie mechanisme is (je kunt er zelfs de natuurlijke getallen mee definiëren!);
- om met inductie de rekenkundige operatoren te definiëren, waarna de rekenkundige wetten met inductie bewezen kunnen worden.

In de praktijk kun je de zo gedefinieerde natuurlijke getallen beter niet gebruiken, omdat de rekenkundige operaties niet erg efficiënt verlopen (vergeleken met de ingebouwde operatoren). Maar voor het gebruik bij het bewijzen van wetten voldoet de definitie uitstekend.

De definitie van natuurlijke getallen met een data-declaratie verloopt volgens een procédé dat al in de vorige eeuw werd bedacht door Giuseppe Peano (al bediende hij zich natuurlijk niet van onze notaties). Het datatype `Nat` (voor 'natuurlijk getal') luidt:

```
data Nat  =  Nul
          |  Volg Nat
```

Een natuurlijk getal is dus òf het getal `Nul`, of het wordt opgebouwd door de constructor-functie `Volg` toe te passen op een ander natuurlijk getal. Elk natuurlijk getal kan worden opgebouwd door maar vaak genoeg `Volg` toe te passen op `Nul`. Zo kan bijvoorbeeld gedefinieerd worden:

```
een   =  Volg Nul
twee  =  Volg (Volg Nul)
drie  =  Volg (Volg (Volg Nul))
vier  =  Volg (Volg (Volg (Volg Nul)))
```

[2] De tak van wiskunde waarin deze constructie wordt uitgevoerd heet 'categorietheorie'. In de categorietheorie wordt een functie die aan deze wet voldoet een 'natuurlijke transformatie' genoemd.

Dit is misschien een wat omslachtige notatie vergeleken bij 1, 2, 3, en 4, maar bij het definiëren van functies en het bewijzen van wetten heb je daar geen last van.

De functie 'plus' kan nu met inductie naar één van de twee parameters gedefinieerd worden, bijvoorbeeld de linker:

```
Nul    + y  =  y
Volg x + y  =  Volg (x+y)
```

In de tweede regel wordt de te definiëren functie recursief aangeroepen. Dit is toegestaan, omdat `x` een kleinere datastructuur is dan `Volg x`. Met behulp van de plus-functie kan, ook weer met inductie, een vermenigvuldig-functie gedefinieerd worden:

```
Nul    * y  =  Nul
Volg x * y  =  y + (x*y)
```

Ook hier wordt de te definiëren functie recursief aangeroepen met een kleinere parameter. Met behulp van deze functie kan de machtsverhef-functie worden gedefinieerd, ditmaal met inductie naar de tweede parameter:

```
x ^ Nul     =  Volg Nul
x ^ Volg y  =  x * (x^y)
```

Nu de operatoren gedefinieerd zijn, is het mogelijk om de rekenkundige wetten te bewijzen. Dat moet in de goede volgorde gebeuren, omdat voor het bewijs van sommige wetten andere wetten nodig zijn. Alle bewijzen verlopen met inductie naar één van de variabelen. Sommige worden zo vaak gebruikt dat ze een naam hebben; sommige worden hier alleen maar bewezen omdat ze in het bewijs van andere wetten nodig zijn. De bewijzen zijn niet moeilijk. Het enige lastige is, om tijdens de bewijzen niet per ongeluk een nog niet bewezen wet te gebruiken omdat het 'natuurlijk zo is' – dan zou je wel meteen kunnen stoppen. Dit alles is misschien scherpslijperij, maar het is toch wel eens leuk om te zien dat de bekende wetten ook inderdaad bewezen kunnen worden.

Dit zijn de wetten die we zullen bewijzen:

1.	`x + Nul`	=	`x`	
2.	`x + Volg y`	=	`Volg (x+y)`	
3.	`x + y`	=	`y + x`	`+` is commutatief
4.	`(x+y) + z`	=	`x + (y+z)`	`+` is associatief
5.	`x * Nul`	=	`Nul`	
6.	`x * Volg y`	=	`x + (x*y)`	
7.	`x * y`	=	`y * x`	`*` is commutatief
8.	`x * (y+z)`	=	`x*y + x*z`	`*` distribueert links over `+`
9.	`(y+z) * x`	=	`y*x + z*x`	`*` distribueert rechts over `+`
10.	`(x*y) * z`	=	`x * (y*z)`	`*` is associatief
11.	`x ^ (y+z)`	=	`x^y * x^z`	
12.	`(x*y)^z`	=	`x^z * y^z`	
13.	`(x^y)^z`	=	`x ^ (y*z)`	herhaald machtsverheffen

Bewijs van wet 1, met inductie naar `x`:

`x`	`x + Nul`	`x`

`Nul`	`Nul + Nul`
	= (def. `+`)
	`Nul`
		`Nul`

`Volg x`	`Volg x + Nul`
	= (def. `+`)
	`Volg (x+Nul)`
		`Volg x`

Bewijs van wet 2, met inductie naar `x`:

`x`	`x + Volg y`	`Volg (x+y)`

`Nul`	`Nul + Volg y`
	= (def. `+`)
	`Volg y`
		`Volg (Nul+y)`
		= (def. `+`)
		`Volg y`

`Volg x`	`Volg x + Volg y`
	= (def. `+`)
	`Volg (x + Volg y)`
		`Volg (Volg x + y)`
		= (def. `+`)
		`Volg (Volg (x+y))`

Bewijs van wet 3 (plus is commutatief), met inductie naar `x`:

x	x + y	y + x
Nul	Nul + y = (def. +) y	y + Nul = (wet 1) y
Volg x	Volg x + y = (def. +) Volg (x+y)	y + Volg x = (wet 2) Volg (y+x)

Bewijs van wet 4 (plus is associatief), met inductie naar x:

x	(x+y) + z	x + (y+z)
Nul	(Nul+y) + z = (def. +) y + z	Nul + (y+z) = (def. +) y + z
Volg x	(Volg x + y) + z = (def. +) Volg (x+y) + z = (def. +) Volg ((x+y) + z)	Volg x + (y+z) = (def. +) Volg (x + (y+z))

Bewijs van wet 5, met inductie naar x:

x	x * Nul	Nul
Nul	Nul * Nul = (def. *) Nul	Nul
Volg x	Volg x * Nul = (def. *) Nul + (x*Nul)	Nul = (def. +) Nul+Nul

Bewijs van wet 6, met inductie naar x:

x	x * Volg y	x + (x*y)
Nul	Nul * Volg y = (def. *) Nul	Nul + Nul*y = (def. *) Nul+Nul = (def. +) Nul
Volg x	Volg x * Volg y = (def. *) Volg y + (x*Volg y) = (def. +) Volg (y + (x*Volg y))	Volg x + (Volg x * y) = (def. *) Volg x + (y + (x*y)) = (def. +) Volg (x + (y + (x*y))) = (+ associatief) Volg ((x + y) + (x*y)) = (+ commutatief) Volg ((y + x) + (x*y)) = (+ associatief) Volg (y + (x + (x*y)))

Bewijs van wet 7 (* is commutatief), met inductie naar x:

x	x * y	y * x
Nul	Nul * y = (def. *) Nul	y * Nul = (wet 5) Nul
Volg x	Volg x * y = (def. *) y + (x*y)	y * Volg x = (wet 6) y + (y*x)

Bewijs van wet 8 (* distribueert links over +), met inductie naar x:

x	x * (y+z)	x*y + x*z
Nul	Nul * (y+z) = (def. *) Nul	Nul*y + Nul*z = (def. *) Nul + Nul = (def. +) Nul
Volg x	Volg x * (y+z) = (def. *) (y+z) + (x*(y+z))	(Volg x*y) + (Volg x*z) = (def. *) (y+x*y) + (z + x*z) = (+ associatief) ((y+x*y)+z) + x*z = (+ associatief) (y+(x*y+z)) + x*z = (+ commutatief) (y+(z+x*y)) + x*z = (+ associatief) ((y+z)+x*y) + x*z = (+ associatief) (y+z) + (x*y + x*z)

Bewijs van wet 9 (* distribueert rechts over +):

	(y+z) * x = (* commutatief) x * (y+z) = (wet 8) x*y + x*z	y*x + z*x = (* commutatief) x*y + x*z

Bewijs van wet 10 (* is associatief), met inductie naar x:

x	(x*y) * z	x * (y*z)
Nul	(Nul*y) * z = (def. *) Nul * z = (def. *) Nul	Nul * (y*z) = (def. *) Nul
Volg x	(Volg x*y) * z = (def. *) (y+(x*y)) * z = (wet 9) (y*z) + ((x*y)*z)	Volg x * (y*z) = (def. *) (y*z) + (x*(y*z))

Bewijs van wet 11, met inductie naar y:

y	x^(y+z)	x^y * x^z
Nul	x^(Nul+z) = (def. +) x^z	x^Nul * x^z = (def. ^) Volg Nul * x^z = (def. *) x^z + Nul*x^z = (def. *) x^z + Nul = (wet 1) x^z
Volg y	x ^ (Volg y+z) = (def. +) x ^ (Volg (y+z)) = (def. ^) x * x^(y+z)	x^Volg y * x^z = (def. ^) (x*x^y) * x^z = (* associatief) x * (x^y * x^z)

Het bewijs van wet 12 en wet 13 wordt als opgave aan de lezer overgelaten.

Opgaven

blz. 68 **5.1** Welke versie van segs is efficiënter: die uit paragraaf 4.1.1 of die uit opgave 4.1?

blz. 83

blz. 87 **5.2** Bekijk het (derde) schema in paragraaf 5.1.2. Geef bij de optimalisatie-methodes **b** t/m **f**

in paragraaf 5.1.3 aan in welke regel van het schema de algoritmen voor en na optimalisatie vallen. blz. 87

5.3 De functie `concat` kan gedefinieerd worden door

```
concat xss  =  fold (++) [] xss
```

Daarbij kan voor `fold` één van de drie functies `foldr`, `foldl` of `foldl'` gebruikt worden. Bespreek het effect van deze keuze voor de efficiëntie, zowel voor het geval dat `xss` uitsluitend eindige lijsten bevat, als voor het geval dat er ook oneindige lijsten in zitten.

5.4 In paragraaf 5.2.6 wordt de wet blz. 103

```
sum (map (1+) xs)  =  len xs + sum xs
```

bewezen. Formuleer een dergelijke wet voor een willekeurige lineaire functie in plaats van `(1+)`, dus

```
sum (map ((k+).(n*)) xs)  =  ...
```

Bewijs de geformuleerde wet.

5.5 Bewijs de volgende

Wet *fold na concatenatie*
Als `op` een associatieve operator is, dan geldt:

```
foldr op e . concat  =  foldr op e . map (foldr op e)
```

5.6 Bepaal een functie `g` en een waarde `e` waarvoor geldt:

```
map f  =  foldr g e
```

Bewijs de gelijkheid voor de gevonden `g` en `e`.

5.7 Bewijs de volgende wet:

```
len . subs  =  (2^) . len
```

5.8 Bewijs dat `subs` een combinatorische functie is.

5.9 Bewijs wet 12 en wet 13 uit paragraaf 5.2.8. blz. 107

Hoofdstuk 6

Klassen en hun instances

6.1 Numerieke types

6.1.1 Overloading

Veel functies en operatoren kunnen op waarden van verschillend type worden toegepast. Er zijn twee mechanismen waardoor dat mogelijk is:

- *Polymorfie.* Een polymorfe functie werkt op een bepaalde datastructuur (bijvoorbeeld lijsten), zonder gebruik te maken van eigenschappen van de elementen. De functie kan dus op datastructuren met een willekeurig element-type worden toegepast. Voorbeelden van polymorfe functies: `length`, `concat` en `map`.
- *Overloading.* Een overloaded functie kan op een aantal verschillende types werken die niets met elkaar te maken hebben. De operator `+` werkt bijvoorbeeld zowel op type `Int` als op type `Float`. De operator `<=` kan op `Int` en `Float` werken, en daarnaast ook op `Char`, twee-tupels en lijsten.

In paragraaf 1.5.5 is aangegeven dat overloading mogelijk is dankzij het bestaan van klassen van types (*classes*). Een klasse is een groep types waarop een bepaalde operator kan worden toegepast. De types `Int` en `Float` vormen samen bijvoorbeeld `Num`, de klasse van numerieke types. De operator `+` is op alle types in de klasse `Num` gedefinieerd. Dit komt tot uiting in het type van de operator `+`: blz. 17

```
(+)  ::  Num a  =>  a -> a -> a
```

De tekst `Num a` kan gelezen worden als 'type `a` zit in klasse `Num`'. Het geheel heeft de betekenis: '`+` heeft het type `a->a->a` mits `a` in klasse `Num` zit'.

Andere klassen die veel gebruikt worden zijn `Eq` en `Ord`. De klasse `Eq` is de klasse van types waarvan de elementen vergeleken kunnen worden; `Ord` is de klasse van ordenbare types. Operatoren die op types uit deze klassen gedefinieerd zijn, zijn bijvoorbeeld:

```
(==)  ::  Eq a  => a -> a -> Bool
(<=)  ::  Ord a => a -> a -> Bool
```

Let op het verschil tussen de pijltjes: het pijltje met enkele stok kan meer dan eens voorkomen in een type, en wordt gebruikt in de betekenis 'functie van... naar...'. Het pijltje met dubbele stok kan maar één keer voorkomen in een typedeclaratie. Links ervan staat vermeld dat een bepaalde type-variabele in een bepaalde klasse zit, rechts ervan staat een type waar deze typevariabele in gebruikt wordt.

6.1.2 Classes en instances

Het is mogelijk om zelf nieuwe klassen te definiëren, naast de reeds bestaande klassen `Num`, `Eq` en `Ord`. Ook is het mogelijk om nieuwe types toe te voegen aan een klasse (zowel aan de drie bestaande klassen als aan zelf-gedefinieerde).

Het definiëren van een klasse heet een *klasse-declaratie*, het toevoegen van een type aan een klasse een *instance-declaratie*.[1] De drie standaard-klassen zijn niet ingebouwd in Haskell. Ze worden in de prelude gedefinieerd door middel van gewone klasse-declaraties. Ook het feit dat `Int` en `Float` deel

[1] Het woord *instance* betekent letterlijk 'voorbeeld' (vergelijk de uitdrukking *for instance*). Het type `Int` is als het ware een 'voorbeeld' van een type in de klasse `Num`. Zo wordt dat overigens in het Nederlands meestal niet gezegd. Het woord *instance* wordt doorgaans onvertaald gelaten, of vervangen door het anglicisme 'instantie'.

uitmaken van de klasse `Num` wordt in de prelude gedefinieerd door middel van instance-declaraties. Er is dus niets speciaals aan de drie standaard-klassen.

De definitie van de klasse `Num` is een goed voorbeeld van een klasse-declaratie. Deze ziet er, iets vereenvoudigd, als volgt uit:

```
class  Num a  where
  (+), (-), (*), (/) ::  a -> a -> a
  negate             ::  a -> a
```

Een klasse-declaratie bestaat dus uit de volgende onderdelen:

- het (speciaal voor dit doel) gereserveerde woord `class`;
- de naam van de klasse (`Num` in het voorbeeld);
- een type-variabele (`a` in het voorbeeld);
- het gereserveerde woord `where`;
- type-declaraties voor operatoren en functies, waarbij de genoemde type-variabele gebruikt mag worden.

In een instance-declaratie wordt voor de aldus gedefinieerde operatoren een definitie gegeven. De instance-declaratie waarmee wordt aangegeven dat `Int` in de klasse `Num` zit ziet er als volgt uit:

```
instance  Num Int  where
   (+)    =  primPlusInt
   (-)    =  primMinusInt
   (*)    =  primMulInt
   (/)    =  primDivInt
   negate =  primNegInt
```

Een instance-declaratie bestaat dus uit de volgende onderdelen:

- het gereserveerde woord `instance`;
- de naam van een klasse (`Num` in het voorbeeld);
- een type (`Int` in het voorbeeld);
- het gereserveerde woord `where`;
- definities voor de operatoren en functies die in de klasse-declaratie werden gedeclareerd.

Bij de instance-declaratie `Num Int` zijn de functie-definities een beetje flauw, omdat simpelweg wordt aangegeven dat voor elke functie de betreffende ingebouwde functie op integers genomen moet worden. De instance-declaratie `Num Float` (uit te spreken als '`Float` is een instance van `Num`') is al even flauw:

```
instance  Num Float  where
   (+)    =  primPlusFloat
   (-)    =  primMinusFloat
   (*)    =  primMulFloat
   (/)    =  primDivFloat
   negate =  primNegFloat
```

Het leuke van deze declaraties is wel, dat alleen de functies `prim...` ingebouwd zijn; de operatoren `+`, `*` enz. zijn, compleet met hun overloading, in de prelude gewoon in Haskell gedefinieerd.

6.1.3 Nieuwe numerieke types

Als je zelf een type hebt gedefinieerd, waarop de numerieke operatoren zouden moeten werken, dan kan het nieuwe type met een instance-declaratie lid gemaakt worden van de klasse `Num`. In paragraaf 3.3.3 werd bijvoorbeeld het type `Ratio` der rationale getallen gedefinieerd (de verzameling **Q**). Rationale getallen werden opgeteld met de functie `qPlus`, vermenigvuldigd met `qMaal`, enzovoort. Maar het is natuurlijk veel handiger om optelling tussen `Ratio`'s gewoon als `+` te kunnen schrijven. Daartoe dient de volgende instance-declaratie:

blz. 55

```
instance  Num Ratio  where
   (+)  =  qPlus
   (-)  =  qMin
   (*)  =  qMaal
   (/)  =  qDeel
   negate =  qMin (0,1)
```

In plaats van de functies eerst `qPlus`, `qMaal` enz. te noemen, kan de functie-definitie ook direct in de instance-declaratie geschreven worden. Vaak worden de instance-declaraties direct na de

type-declaratie geschreven, zoals hieronder:

```
type  Ratio  =  (Int,Int)
instance  Num Ratio  where
   (x,y) + (p,q)  =  eenvoud (x*q+y*p, y*q)
   (x,y) - (p,q)  =  eenvoud (x*q-y*p, y*q)
   (x,y) * (p,q)  =  eenvoud (x*p, y*q)
   (x,y) / (p,q)  =  eenvoud (x*q, y*p)
   negate (x,y)   =  (negate x, y)
```

Het is overigens verstandig om geen 'kale' tupels tot instance van een klasse te maken, maar ze te beschermen met een beschermd datatype (zie paragraaf 3.4.3). Je gebruikt daartoe `data` in plaats van `type`, en patronen in de definitie van de functies: blz. 64

```
data  Ratio  =  Rat (Int,Int)
instance  Num Ratio  where
   Rat (x,y) + Rat (p,q)  =  eenvoud (Rat (x*q+y*p, y*q))
   Rat (x,y) - Rat (p,q)  =  eenvoud (Rat (x*q-y*p, y*q))
   Rat (x,y) * Rat (p,q)  =  eenvoud (Rat (x*p, y*q))
   Rat (x,y) / Rat (p,q)  =  eenvoud (Rat (x*q, y*p))
   negate (Rat (x,y))     =  Rat (negate x, y)
```

Aan de hand van de typering bepaalt de interpreter welke versie van de operatoren gebruikt moet worden. Bij de operator `*` op het type `Ratio` redeneert de interpreter ongeveer als volgt:

> Dit is een definitie van de operator `*`. Volgens de klasse-declaratie van `Num` heeft die operator het type `a->a->a`, waarbij `a` het type is van een instance van `Num`. In deze instance-declaratie is dat `Ratio`. Dus de parameters van `*` zijn in deze definitie van type `Ratio`. Een `Ratio` is een (beschermd) tupel van twee integers. Dus `x`, `y`, `p` en `q` hebben het type `Int`. Volgens de definitie moeten `x*p` en `y*q` uitgerekend worden. Eens kijken, kan `*` toegepast worden op integers? Ja, want `Int` behoort volgens de definitie in de prelude ook tot de klasse `Num`. Dan weet ik dus ook hoe `x` en `p` vermenigvuldigd moeten worden...

Dankzij de typering ziet de interpreter dus dat `x*p` niet een recursieve aanroep is van het vermenigvuldigen van `Ratio`'s, maar dat hier sprake is van het gebruik van de operator `*` uit één van de andere instances van `Num`.

6.1.4 Numerieke constanten

Door het klasse-mechanisme kan voor optelling de operator `+` gebruikt worden, ongeacht of de op te tellen waardes integers zijn, `Float`'s, of zelfgedefinieerde numerieke types, zoals `Ratio` of `Complex`. Lastig is echter, dat het voor de notatie van constanten wèl belangrijk is wat het gewenste type is. Zo moet voor de waarde 'drie' geschreven worden:

met het type `Int`:	`3`
met het type `Float`:	`3.0`
met het type `Ratio`:	`Rat (3,1)`
met het type `Complex`:	`Comp (3.0, 0.0)`

Als bijvoorbeeld is gedefinieerd: `half = Rat(1,2)`, dan kun je niet schrijven:

```
3 * half
```

De waarde `half` heeft immers het type `Ratio`, terwijl `3` het type `Int` heeft.

Dit is vooral vervelend bij het definiëren van functies die op alle types in een klasse moeten kunnen werken. Het is bijvoorbeeld wel mogelijk om een overloaded functie `verdubbel` te schrijven, maar een functie `halveer` lukt niet. Het enige wat er op zou zitten is om hier verschillende versies van te maken:

```
verdubbel  ::  Num a  => a -> a
verdubbel x  =  x + x
halveerInt   ::  Int -> Int
halveerInt n   =  n / 2
halveerFloat ::  Float -> Float
halveerFloat x =  x / 2.0
```

enzovoort. Een oplossing zou kunnen zijn om de functie `halveer` in de klasse `Num` te specificeren, en in elke instance te definiëren. Maar dan kan je wel aan de gang blijven, want waarom wel een functie `halveer` maar geen functie `deelInVieren`?

Om het probleem beter op te lossen, is er in de prelude voor gekozen om nog één functie aan de klasse `Num` toe te voegen: de functie `fromInteger`. De volledige klasse-declaratie luidt dus:

```
class  Num a  where
  (+), (-), (*), (/) ::  a -> a -> a
  negate             ::  a -> a
  fromInteger        ::  Int -> a
```

Hiermee wordt gespecificeerd dat er voor elk instance-type van `Num` een conversie-functie moet zijn van `Int` naar dat type. Die conversiefunctie moet gedefinieerd worden in de instance-declaratie. Voor het type `Int` is dat gemakkelijk:

```
instance Num Int  where
    ....
    fromInteger n  =  n
```

Voor het type `Float` zit er niets anders op dan een ingebouwde functie te gebruiken

```
instance  Num Float  where
    ....
    fromInteger  =  primIntToFloat
```

Voor zelf-gedefinieerde types is het echter wel mogelijk om `fromInteger` zonder `prim`-magie te definiëren, bijvoorbeeld:

```
instance  Num Ratio  where
   ....
   fromInteger n  =  Rat (n,1)
```

Functies zoals `halveer` kunnen nu gedefinieerd worden door:

```
halveer  ::  Num a  =>  a -> a
halveer x  =  x / fromInteger 2
```

Omdat het in de praktijk tamelijk vervelend is om op iedere getal-constante de functie `fromInteger` toe te passen, is het in Haskell mogelijk om dit automatisch te laten doen. Nadat aan de interpreter
blz. 4 de opdracht `:set +i` is gegeven (zie paragraaf 1.2.3), wordt voortaan op elke constante van type `Int` direct de functie `fromInteger` toegepast. Daarmee wordt wel een raar mengsel van 'ingebouwde' en 'voorgedefinieerde' faciliteiten gebruikt: de functie `fromInteger` is in de prelude netjes in Haskell gedefinieerd, maar het automatisch toepassen ervan op elke `Int`-constante is een niet in Haskell definieerbaar, en daarom ingebouwd mechanisme.

Handig is het wel. Functies die op `Float`'s werken, zoals `sqrt`, lijken nu ook op integer-constanten te kunnen werken:

```
? sqrt 2
1.41421
```

'Lijken te werken', want wat er eigenlijk uitgerekend wordt is `sqrt (fromInteger 2)`.

6.2 Ordening en gelijkheid

6.2.1 Default-definities

In de prelude wordt een klasse `Eq` gedefinieerd. De instances van deze klasse zijn de types waarvan de elementen met elkaar vergeleken kunnen worden. De klasse-declaratie is als volgt:

```
class  Eq a  where
   (==), (/=)  ::  a -> a -> Bool
```

Bij elke instance-declaratie voor deze klasse moeten dus de operatoren `==` (gelijkheid) en `/=` (ongelijkheid) worden gedefinieerd. De vier standaard-types `Int`, `Float`, `Char` en `Bool` worden in de prelude alle als instance van `Eq` gedefinieerd:

```
instance  Eq Int  where
     x == y  =  primEqInt x y
     x /= y  =  not (x==y)
```

```
instance  Eq Float  where
     x == y  =  primEqFloat x y
     x /= y  =  not (x==y)
instance  Eq Char  where
     x == y  =  ord x == ord y
     x /= y  =  not (x==y)
instance  Eq Bool  where
     True  == True   =  True
     False == False  =  True
     True  == False  =  False
     False == True   =  False
     x     /= y      =  not (x==y)
```

Waarden van type `Int` en `Float` worden vergeleken door aanroep van een ingebouwde functie. Characters worden vergeleken door hun iso/ascii-codes te vergelijken met de zojuist gedefineerde operator `==` op integers. Gelijkheid op `Bool`'s tenslotte wordt direct door middel van patronen gedefinieerd.

In alle vier de gevallen wordt ongelijkheid (`/=`) gedefinieerd door het resultaat van `==` om te keren met `not`. De definitie is in alle gevallen precies hetzelfde (behalve natuurlijk dat telkens de `==` uit een andere instance wordt gebruikt). Dit soort definities mag ook reeds in de klasse-declaratie worden gezet. Het is dan niet nodig om ze in iedere instance te herhalen. Zo'n definitie van een operator heet een *default*-definitie: een definitie die bij ontbreken[2] van een definitie in de instance-declaraties wordt gebruikt. Wordt een functie waarvoor een default-definitie bestaat tòch in de instance-declaratie gedefinieerd, dan gaat die definitie voor.

De klasse-declaratie `Eq` zoals die werkelijk in de prelude staat is dus als volgt:

```
class  Eq a  where
   (==), (/=)  ::  a -> a -> Bool
   x /= y  =  not (x==y)
```

De definitie van `/=` in de instance-declaraties is weggelaten, omdat de default-definitie voldoet.

Ook zelfgedefinieerde types kunnen tot instance van `Eq` gemaakt worden. Voor het type `Ratio` kan de gelijkheids-definitie uit opgave 3.8 gebruikt worden: blz. 65

```
instance  Eq Ratio  where
    Rat (x,y) == Rat (p,q)  =  x*q == y*p
```

De gelijkheid die in de rechterkant van de definitie gebruikt wordt is de gelijkheid tussen integers. De Haskell-interpreter kan dat afleiden aan de hand van de typering. Een definitie van `/=` kan, net als in de instance-declaraties in de prelude, achterwege blijven. De default-definitie is ook in dit geval immers bruikbaar.

6.2.2 Klassen met voorwaarden

De types waarvan de elementen ordenbaar zijn (met operatoren zoals `<=`) zijn instances van de klasse `Ord`. In de klasse-declaratie voor `Ord` wordt aangegeven dat een type ook een instance van `Eq` moet zijn, wil het ordenbaar zijn:

```
class  Eq a => Ord a  where
   (<=), (<), (>=), (>)  ::  a -> a -> Bool
   max, min              ::  a -> a -> a
```

Alle operatoren en functies in deze klasse met uitzondering van `<=` hebben een default-definitie. De enige operator die in instance-declaraties gedefinieerd hoeft te worden is dus `<=`. Het is vanwege deze default-definities dat geëist wordt dat instances van `Ord` ook instances van `Eq` zijn. In de default-definitie van `<` wordt namelijk de operator `/=` gebruikt. De default-definities luiden:

```
x <  y   =   x <= y && x /= y
x >= y   =   y <= x
x >  y   =   y <  x
max x y | x>=y  =  x
        | y>=x  =  y
min x y | x<=y  =  x
        | y<=x  =  y
```

[2] *default* betekent letterlijk 'ontbreken'

De instance-declaratie `Ord Int` en `Ord Float` doen voor de definitie van `<=` een beroep op een ingebouwde functie. Voor characters wordt de ordening bepaald door de integer-ordening van hun iso/ascii codes:

```
instance  Ord Char  where
    x <= y  =  ord x <= ord y
```

Net als `Ord` eist de klasse-declaratie voor `Num` in de prelude dat de instances ook een instance van de klasse `Eq` zijn. De, nu helemaal complete klasse-declaratie voor `Num` luidt derhalve:

```
class  Eq a => Num a  where
   (+), (-), (*), (/)  ::  a -> a -> a
   negate              ::  a -> a
   fromInteger         ::  Int -> a
```

De vergelijkbaarheid van de elementen van instances van `Num` wordt echter niet gebruikt in default-definities, zoals dat bij `Ord` het geval was. De enige reden dat `Eq a` wordt geëist als voorwaarde voor `Num a` is dat 'numerieke types', waarvan de elementen niet eens vergelijkbaar zijn, onzinnig beschouwd worden.

6.2.3 Instances met voorwaarden

Ook bij instance-declaraties kan als voorwaarde worden opgegeven, dat een bepaald type deel uitmaakt van een bepaalde klasse. Deze constructie wordt o.a. gebruikt om in één keer alle denkbare lijsten tot instance van `Eq` te maken:

```
instance  Eq a => Eq [a]  where
   []    == []     =  True
   []    == (y:ys) =  False
   (x:xs)== []     =  False
   (x:xs)== (y:ys) =  x==y && xs==ys
```

De eerste regel van deze instance-declaratie kan zo gelezen worden: 'als `a` een instance is van `Eq`, dan is ook `[a]` een instance van `Eq`'. Het vierde geval in de definitie van `==` is opmerkelijk: aan de rechterkant komt twee maal een aanroep van `==` voor. De eerste stelt gelijkheid op (de eerste) elementen van de lijst voor (dat kan, want dat was immers de voorwaarde van de instance-declaratie). De tweede aanroep van `==` is een recursieve aanroep van gelijkheid op lijsten.

Dankzij deze declaratie kunnen lijsten van integers vergeleken worden, lijsten van floats, lijsten van characters en lijsten van booleans. Maar ook lijsten van lijsten van integers (want lijsten van integers zijn ondertussen ook vergelijkbaar). En daarom dus ook lijsten van lijsten van lijsten van integers, enzovoort...

blz. 39 In paragraaf 3.1.2 werd al opgemerkt dat lijsten een ordening kennen: de lexicografische ordening. Deze ordening wordt gedefinieerd door een instance-declaratie in de prelude:

```
instance  Ord a => Ord [a]  where
    []     <= ys     =  True
    (x:xs) <= []     =  False
    (x:xs) <= (y:ys) =  x<y || (x==y && xs<=ys)
```

Met deze definitie wordt aangegeven dat de lege lijst de kleinste lijst is. Voor niet-lege lijsten is het eerste element bepalend; als het eerste element van de twee lijsten gelijk is, wordt de rest van de lijsten recursief vergeleken. De andere ordenings-operatoren hoeven niet gedefinieerd te worden: daarvoor worden de default-definities gebruikt.

Een instance-declaratie kan meer dan één voorwaarde hebben. Die moeten dan tussen haakjes genoteerd worden, met komma's ertussen. Hiermee kan bijvoorbeeld de gelijkheid van twee-tupels gedefinieerd worden, zoals in de prelude gebeurt:

```
instance (Eq a, Eq b) => Eq (a,b)  where
   (x,y) == (u,v)  =  x==u && y==v
```

De elementen van een tweetupel `(a,b)` zijn dus vergelijkbaar mits zowel `a` als `b` een instance is van `Eq`. Twee tweetupels zijn volgens deze definitie alleen maar gelijk, als beide elementen gelijk zijn (volgens de gelijkheidsdefinitie van hun respectievelijke types). In de prelude wordt alleen een definitie gegeven van de gelijkheid van tweetupels. Drie- en meertupels zijn niet vergelijkbaar. In voorkomende gevallen kan zo'n gelijkheid natuurlijk wel zelf gedefinieerd worden.

Het pijltje met dubbele stok (`=>`) kan gebruikt worden in type-declaraties, in instance-declaraties

en in klasse-declaraties. Let op het verschil in betekenis van deze drie vormen:
- `f :: Num a => a->a` is een type-declaratie: `f` is een functie met type `a->a` mits `a` een type is in de klasse `Num`.
- `instance Eq a => Eq [a]` is een instance-declaratie: `[a]` is een instance van `Eq` mits `a` dat ook is.
- `class Eq a => Ord a` is een klasse-declaratie: alle instances van de nieuwe klasse `Ord` moeten ook instances zijn van `Eq`.

6.2.4 Standaard-klassen

In de prelude worden de volgende klassen gedefinieerd:
- `Eq`, de klasse van vergelijkbare types;
- `Ord`, de klasse van ordenbare types;
- `Num`, de klasse van numerieke types;
- `Enum`, de klasse van opsombare types;
- `Ix`, de klasse van index-types;
- `Text`, de klasse van afdrukbare types.

De eerste drie werden al eerder besproken. Hieronder volgt een korte beschrijving van de andere drie (die minder vaak gebruikt worden).

de klasse Enum

De klasse `Enum` is als volgt gedefinieerd:

```
class  Ord a => Enum a  where
    enumFrom       :: a -> [a]
    enumFromThen   :: a -> a -> [a]
    enumFromTo     :: a -> a -> [a]
    enumFromThenTo :: a -> a -> a -> [a]
```

De bedoeling van de functie `enumFrom` is dat de lijst van waarden 'vanaf' een bepaalde waarde wordt opgeleverd. De functie `enumFromThen` krijgt een beginwaarde en een tweede waarde; de functie `enumFromTo` krijgt een beginwaarde en een eindwaarde; de functie `enumFromThenTo` tenslotte krijgt ze alledrie.

Standaard-instances van deze klasse zijn `Int`, `Float` en `Char`. Een paar voorbeelden van het resultaat van de functies voor verschillende types:

```
enumFrom 4                  =  [4,5,6,7,8,...
enumFromTo 'c' 'f'          =  ['c','d','e','f']
enumFromThenTo 1.0 1.5 3.0  =  [1.0,1.5,2.0,2.5,3.0]
```

Deze vier functies worden door de interpreter gebruikt om de speciale notaties `[x..]`, `[x,y..]`, `[x..y]` en `[x,y..z]` uit te rekenen, die in paragraaf 3.1.1 werden besproken. Deze notaties kunnen blz. 38 dus evenals de vier functies gebruikt worden voor verschillende types, bijvoorbeeld `['c'..'f']`. Zou je zelf types definiëren als instance van `Enum`, dan kan ook voor die types de notatie `[x..y]` gebruikt worden.

De instance-declaratie `Enum Int` luidt als volgt:

```
instance  Enum Int  where
    enumFrom n       =  iterate (1+)     n
    enumFromThen n m =  iterate ((m-n)+) n
```

Voor het type `Float` is de instance-declaratie hetzelfde, maar dan met `1.0` in plaats van `1`. Voor characters luidt de declaratie:

```
instance  Enum Char  where
    enumFrom c       =  map chr (enumFrom (ord c))
    enumFromThen c d =  map chr (enumFromThen (ord c) (ord d))
```

De in de definitie gebruikte functies zijn natuurlijk geen recursieve aanroepen, maar de overeenkomstige functies van de `Int`-instance.

Voor de functies `enumFromTo` en `enumFromThenTo` is er een default-definitie in de klasse-declaratie:

```
class  Ord a => Enum a  where
     ....
```

```
enumFromTo n m         =  takeWhile (m>=) (enumFrom n)
enumFromThenTo n n' m
           | n' > n    =  takeWhile (m>=) (enumFromThen n n')
           | otherwise =  takeWhile (m<=) (enumFromThen n n')
```

Omdat in deze definities de ordenings-operatoren gebruikt worden, is het noodzakelijk dat elke instance van `Enum` ook een instance is van `Ord`. Instances van `Enum` hoeven echter niet noodzakelijk een instance van `Num` te zijn. Dat de instance-declaraties voor `Int` en `Float` de operator `+` gebruiken is hun zaak; er zijn instances van `Enum` denkbaar die geen numerieke operatoren nodig hebben (`Char` is daar een voorbeeld van).

de klasse Ix

De klasse `Ix` lijkt op `Enum`. De klasse-declaratie is als volgt:

```
class  Ord a => Ix a  where
   range   :: (a,a) -> [a]
   index   :: (a,a) -> a -> Int
   inRange :: (a,a) -> a -> Bool
```

De functie `range` is vergelijkbaar met `enumFromTo`. Er wordt nu echter ook een functie `index` gevraagd, die het rangnummer van een waarde in een bepaald interval geeft. Daarom kan het type `Float` geen instance zijn van `Ix`. De 'discrete' types `Int` en `Char` zijn wel een instance van `Ix`, bijvoorbeeld:

```
instance  Ix Int  where
   range   (m,n)    =  [m..n]
   index   (m,n) i  =  i - m
   inRange (m,n) i  =  m<=i && i<=n
```

de klasse Text

De klasse `Text` declareert twee functies, waarmee respectievelijk een element en een lijst van het instance-type omgezet kunnen worden in een `String`. Deze functies worden zelden direct gebruikt. Meestal wordt in plaats daarvan de functie `show` gebruikt, die in de prelude wordt gedefinieerd:

```
show  ::  Text a  =>  a -> String
```

Met de functie `show` kan elke waarde van een type dat een instance is van `Text` worden omgezet naar een `String`. Standaard-instances van `Text` zijn `Int`, `Float`, `Char` en lijsten en tweetupels waarvan de element-types ook instances zijn van `Text`.

6.2.5 Problemen met klassen

Bij het analyseren van een expressie of een file met definities kan de interpreter een aantal fouten melden die te maken hebben met het gebruik van klassen. Hieronder worden drie soorten fouten besproken.

'Cannot derive instance'

Deze foutmelding is het gevolg als je een operator uit een klasse gebruikt met parameters waarvoor er geen instance is gedeclareerd. Bijvoorbeeld:

```
? (1,2,3) == (4,5,6)
ERROR: Cannot derive instance in expression
*** Expression        : (1,2,3) == (4,5,6)
*** Required instance : Eq (Int,Int,Int)
```

In de prelude staat geen declaratie waarmee drietupels tot instance van `Eq` gemaakt worden (wel voor tweetupels en lijsten). Daarom kan de operator `==` niet zonder meer op drietupels toegepast worden. Een oplossing van dit probleem is de ontbrekende instance-declaratie aan het programma toe te voegen.

Deze foutmelding wordt ook gegeven als je functies probeert te vergelijken. Functie-types zijn immers geen instance van `Eq`:

```
? tail == drop 1
ERROR: Cannot derive instance in expression
*** Expression        : tail == drop 1
*** Required instance : Eq ([a]->[a])
```

Dat wij met de technieken uit sectie 5.2zelf in staat zijn om de gelijkheid van twee functies te bewijzen, wil nog niet zeggen dat deze functies ook in de taal Haskell vergeleken mogen worden. De interpreter zou in zo'n geval immers de twee functies op alle mogelijke parameters moeten toepassen (wat oneindig lang duurt) of een inductief bewijs moeten leveren (waar hij niet creatief genoeg voor is). blz. 93

'Overlapping instances'

Het is niet mogelijk om twee declaraties te geven waarmee een type instance wordt van dezelfde klasse. Bij gebruik van een operator op een waarde van dat type zou de interpreter dan namelijk niet kunnen kiezen uit de twee definities.

Hetzelfde probleem treedt op als het type in een instance-declaratie een speciaal geval is van een type waarvoor al een andere instance-declaratie bestaat. Bijvoorbeeld: in de prelude worden tweetupels gedeclareerd als instance van `Eq`:

```
instance  (Eq a,Eq b) => Eq (a,b)  where
   (x,y)==(u,v)  =  x==u && y==v
```

Als rationale getallen als tweetupel van twee integers worden gedefinieerd, zou je daarop wel een andere gelijkheid willen definiëren:

```
type Ratio = (Int,Int)
instance  Eq Ratio  where
   (x,y)==(u,v)  =  x*v == u*y
```

Bij een aanroep van `(1,2)==(2,4)` kan de interpreter nu niet kiezen: wordt de standaard tupel-gelijkheid bedoeld of de `Ratio`-gelijkheid? Bij het analyseren van de instance-declaratie wordt daarom een foutmelding gegeven:

```
ERROR "file" (line 12): Overlapping instances for class "Eq"
*** This instance   : Eq (Int,Int)
*** Overlaps with   : Eq (a,a)
*** Common instance : Eq (Int,Int)
```

Dit probleem kan worden opgelost door types waar een 'rare' gelijkheid op gedefinieerd moet worden als beschermd type te definiëren, dus met gebruikmaking van `data` in plaats van `type`:

```
data  Ratio = Rat (Int,Int)
```

Dan kan `Ratio` rustig tot instance van `Eq` gemaakt worden, omdat `Ratio` een ander type is dan `(Int,Int)`.

'Unresolved overloading'

Bij het gebruik van een overloaded operator besluit de interpreter op grond van de types van de parameters welke definitie gekozen moet worden. Zo wordt in `1+2` de integer-versie van `+` gebruikt, en in `1.0+2.0` de float-versie. Maar als de parameters zelf het resultaat zijn van een overloaded functie, kan het zijn dat er meerdere mogelijkheden zijn. Dat is bijvoorbeeld het geval in de volgende expressie:

```
? fromInteger 1 + fromInteger 2
ERROR: Unresolved overloading
*** type        : Num a => a
```

Voor `fromInteger` kan de integer-versie of de float-versie gekozen worden. In het eerste geval moet ook de integer-versie van `+` gebruikt worden, in het tweede geval de float-versie. Als er verder geen context is waardoor de keuze gemaakt kan worden (bijvoorbeeld de hele expressie is parameter van de functie `sqrt`), dan volgt er een *unresolved overloading* foutmelding.

Deze foutmelding treedt vooral op als de optie 'pas `fromInteger` toe op elk getal' aan staat (zoals beschreven in paragraaf 6.1.4). Een onschuldig ogende expressie als `1+2` heeft dan immers al een unresolved overloading tot gevolg. blz. 115

Meestal is deze fout te herstellen door de interpreter een extra hint te geven over het gewenste type. Dat kan bijvoorbeeld door een type-declaratie te geven voor de kritieke functies, of door de dubieuze expressie direct te typeren:

```
? fromInteger 1 + fromInteger 2 :: Int
3
```

6.3 Klassen en wetten

6.3.1 Wetten voor standaardklassen

De namen van de diverse klassen en de operatoren daarin suggereren dat die operatoren aan allerlei eigenschappen voldoen. Er is echter niemand die er op toeziet dat dit inderdaad het geval is. Er volgt bijvoorbeeld geen foutmelding als je zou definiëren:

```
type  Bliep  =  (Int,[Char])
instance  Ord Bliep  where
    (n,xs) <= (k,ys)  =  n+k==length ys
```

(om maar eens iets onzinnigs te noemen). Toch is het niet gangbaar om dit soort definities een 'ordening' te noemen. Maar waarom is deze definitie niet 'zinvol', en de toch ook niet voor de hand liggende definitie van `<=` op rationale getallen (zie hieronder) wel?

```
instance  Ord Ratio  where
   Rat (x,y) <= Rat (u,v)  =  x*v <= u*y
```

Het antwoord is: normaliter wordt er van uitgegaan dat operatoren zoals `<=` aan bepaalde eigenschappen voldoen. Van die eigenschappen wordt gebruik gemaakt in andere functies. Sorteerfuncties maken bijvoorbeeld gebruik van het feit dat als $x \leq y$ en $y \leq z$, dat dan ook $x \leq z$.

Eigenschappen waar operatoren in een klasse aan dienen te voldoen, kunnen vastgelegd worden in *wetten*. Deze wetten zouden als commentaar bij de klasse-declaratie toegevoegd kunnen worden. Helemaal mooi zou het zijn als Haskell voor elke instance zou controleren of hij aan de gegeven wetten voldoet. Helaas... dat is een beetje te veel gevraagd. Wel zou je als programmeur kunnen *bewijzen* dat de definities die je in een bepaalde instance geeft, aan de vereiste wetten voldoet. Zo'n bewijs kan dienen om de *correctheid* van een implementatie aan te tonen.[3]

Wil de operator `==` zijn naam 'gelijkheids-operator' waardig zijn, dan moet hij aan de volgende wetten voldoen (voor alle f, x, y en z):

reflexiviteit	er geldt $x = x$;
symmetrie	als $x = y$, dan $y = x$;
transitiviteit	als $x = y$ en $y = z$, dan $x = z$;
congruentie	als $x = y$, dan $f\ x = f\ y$.

De laatste wet geeft problemen als die inderdaad wordt geëist voor *alle* functies f. Gelijkheid op rationale getallen voldoet bijvoorbeeld niet aan de congruentiewet voor de functie `gemeen`:

```
gemeen (Rat (t,n))  =  t + n
```

Als het om beschermde datatypes gaat, wordt de congruentie-eis daarom meestal afgezwakt; de wet hoeft alleen maar te gelden voor een bepaalde verzameling functies en combinaties daarvan. Bij de rationale getallen zijn dat bijvoorbeeld `qPlus`, `qMin`, `qMaal` en `qDeel`. Voor functies zoals `gemeen`, die met patroon-herkenning direct gebruik maken van de representatie van het datatype, hoeft de congruentiewet niet te gelden.

De wetten waar de ordenings-operator `<=` aan pleegt te voldoen, zijn de volgende (voor alle x, y en z):

reflexiviteit	er geldt $x \leq x$;
antisymmetrie	als $x \leq y$ en $y \leq x$, dan $x = y$;
transitiviteit	als $x \leq y$ en $y \leq z$, dan $x \leq z$.

De in de vorige paragrafen gedefinieerde instances van `Ord` voldoen inderdaad aan deze drie wetten. Voor de types `Int` en `Float` is dat moeilijk na te gaan, omdat die ingebouwde operatoren gebruiken (je zou kunnen zeggen dat die per definitie aan deze wetten voldoen). Voor de zelf-gedefinieerde instances, zoals de rationale getallen en de lexicografische ordening op lijsten, zijn de wetten inderdaad te bewijzen. Een ordening die aan deze wetten voldoet heet een *partiële ordening*. Het is namelijk niet nodig dat elk element van het type met elk ander element vergelijkbaar is. Daarom
blz. 117 staat er in de default-definitie van `min` en `max` in paragraaf 6.2.2 niet `otherwise` in de tweede regel. Als twee elementen onderling niet geordend zijn, is de waarde van `min` en `max` ongedefinieerd.

[3] Er zijn enkele pogingen gedaan om een programmeertaal te ontwerpen waarbij wetten automatisch gecontroleerd worden. Hoewel er aardige resultaten zijn geboekt, heeft deze benadering nog niet tot grote doorbraken geleid.

Numerieke types moeten, willen ze met recht zo genoemd worden, voldoen aan de wetten uit paragraaf 5.2.8. De bewijzen in die paragraaf waren in feite het bewijs dat het type `Nat` een waarlijk numeriek type is. Naast deze wetten zijn er nog meer wetten waaraan numerieke types moeten voldoen. Bijvoorbeeld een wet die het gedrag van `-` definiëert:

blz. 107

$-$ is de inverse van $+$ $\qquad y + (x - y) = x$

Een overeenkomstige wet voor de delings-operator `/` geeft echter problemen:

$/$ is de inverse van $*$ $\qquad$ als $y \neq 0$ dan $y * (x/y) = x$

Deze wet is bijvoorbeeld niet geldig voor de ingebouwde deling op gehele getallen. Bovendien is er sprake van een getal 0, en wat moeten we daarvoor nemen in een kandidaat-numeriek type?

In Haskell zijn alle numerieke types een instance van de klasse `Num`. De klasse-declaratie en de instance-declaraties die daarvoor nodig zijn staan in de prelude, en werden in paragraaf 6.1.1 besproken. Voor een precieze beschrijving van het onderscheid tussen verschillende numerieke types is een indeling in meerdere klassen eigenlijk geschikter.

blz. 113

6.3.2 Een klasse-hiërarchie

In de wiskunde is het al heel lang gebruikelijk om verschillende soorten verzamelingen te onderscheiden, al naar gelang de operatoren die er op gedefinieerd zijn en de wetten die ervoor gelden. Het indelen van verzamelingen in soorten is in feite niets anders dan het indelen van types in klassen. In deze en de volgende paragrafen wordt de gebruikelijke wiskundige indeling dan ook beschreven aan de hand van klasse- en instance-declaraties in Haskell. Wat eerst een programmeertaal was, is daarmee een gereedschap geworden om wiskunde te bedrijven.

We zullen, overeenkomstig het wiskundige gebruik, de numerieke types indelen in vijf klassen: *monoïden*, *groepen*, *ringen*, *euclidische ringen* en *lichamen*. Deze vijf klassen vormen een hiërarchie: elke klasse voegt een aantal operatoren (en wetten) toe aan de vorige klasse. In de klasse-declaraties komt dat tot uiting in het feit dat de 'hogere' klassen de 'lagere' klassen als voorwaarde hebben:

```
class  Eq a      => Monoid a   where ...
class  Monoid a  => Groep a    where ...
class  Groep a   => Ring a     where ...
class  Ring a    => Euclid a   where ...
class  Euclid a  => Lichaam a  where ...
```

Een type kan dus alleen een instance van `Ring` zijn als het ook een instance van `Groep` is; daarvoor moet het weer een instance van `Monoid` zijn, en dat kan alleen als het een instance van `Eq` is. Anders gezegd: er zijn veel types die een instance zijn van `Monoid`, maar naarmate er meer operatoren en wetten nodig zijn, vallen er steeds meer types af; uiteindelijk zijn er maar een paar types instance van `Lichaam`.

In de vijf klassen worden operatoren, onder andere 'plus' en 'maal', gedefinieerd. Om ze niet te verwarren met de operatoren `+` en `*` die in `Num` worden gedefinieerd, zullen we ze hieronder noteren als `<+>` en `<*>`. Als je onderstaande definities zou beschouwen als *vervanging* in plaats van als *aanvulling* van de prelude, dan zou je ze ook gewoon `+` en `*` kunnen noemen.

Hieronder volgen de klasse-declaraties van de vijf genoemde klassen. Bij elke klasse staan de wetten genoemd waaraan de operatoren moeten voldoen. Deze definities komen misschien wat abstract over, maar in de paragrafen paragraaf 6.3.4 tot paragraaf 6.3.7 staan allerlei instance-declaraties, waarmee concrete types ingedeeld worden in de abstracte klassen. Een wiskundige zou zeggen: in die paragrafen worden *voorbeelden* gegeven van de abstracte soorten (wat betekent 'instance' ook alweer?).

blz. 126

blz. 130

Monoïden

Er is maar heel weinig nodig om een type een monoïde te kunnen noemen. Er moet een *associatieve operator* zijn, die een *neutraal element* heeft. De klasse-declaratie is als volgt:

```
class  Eq a => Monoid a  where
    nul  ::  a
    <+>  ::  a -> a -> a
```

De wetten waar deze operatoren aan moeten voldoen zijn:

M1 `x <+> (y <+> z) = (x <+> y) <+> z`
M2 `nul <+> x = x`
M3 `x <+> nul = x`

Enigszins suggestief hebben we de operator `<+>` genoemd, en het neutrale element `nul`. Dat je hierbij niet per se aan optellen en het getal 0 hoeft te denken blijkt in paragraaf 6.3.4.
blz. 126

In de klasse-declaratie wordt als voorwaarde gesteld dat de elementen van het type vergelijkbaar zijn (een instance van `Monoid` moet ook instance zijn van `Eq`). De gelijkheid tussen elementen is nodig, omdat je anders bezwaarlijk wetten kunt opstellen: daarin wordt immers gelijkheid van bepaalde elementen gesproken.

Groepen

Wil een monoïde ook als groep beschouwd kunnen worden, dan is er naast `<+>` nog een functie nodig. Deze functie geeft bij elk element een *tegenovergestelde*. De klasse-declaratie is als volgt:

```
class  Monoid a => Groep a  where
    neg  ::  a -> a
```

In de wetten voor een `Groep` wordt gesteld dat een element opgeteld bij zijn tegenovergestelde de waarde `nul` oplevert. Bovendien moet de operator `<+>` nu behalve associatief ook *commutatief* zijn[4]. De wetten voor een `Groep` luiden:

G1 `x <+> neg x = nul`
G2 `x <+> y = y <+> x`

Ringen

Bij een ring doet de operator `<*>` de intrede. Deze operator moet associatief zijn, en een neutraal element ('`een`') hebben. De instance-declaratie luidt:

```
class  Groep a => Ring a  where
    een  ::  a
    <*>  ::  a -> a -> a
```

In de wetten voor een `Ring` wordt naast associativiteit van `<*>` en neutraliteit van `een`, gesteld dat de operator `<*>` *distribueert* over `<+>`. Er zijn twee distributieve wetten nodig, omdat `<*>` niet commutatief hoeft te zijn. De wetten voor `Ring` luiden:

R1 `x <*> (y <*> z) = (x <*> y) <*> z`
R2 `een <*> x = x`
R3 `x <*> een = x`
R4 `x <*> (y<+>z) = (x<*>y) <+> (x<*>z)`
R5 `(y<+>z) <*> x = (y<*>x) <+> (z<*>x)`

Hoewel de namen '`<*>`' en '`een`' enigszins suggestief gekozen zijn, moet je je wel realiseren dat `<*>` niet in elke instance 'vermenigvuldigen' hoeft te betekenen.

Domeinen

Een *domein* is een ring waarin nog enkele extra wetten gelden. Er worden echter geen nieuwe operatoren toegevoegd, dus we maken er geen aparte klasse van. De wetten die in een domein gelden zijn de volgende:

D1 `x <*> y = y <*> x`
D2 als `x`$\neq$`nul` en `y`$\neq$`nul`, dan is `x<*>y` $\neq$ `nul`
D3 `een` $\neq$ `nul`

[4] Je ziet ook vaak definities van 'groep' waarin commutativiteit van de operator niet wordt verondersteld. Als de operator wel commutatief is, spreekt men van een 'commutatieve groep'. Omdat we het hier verder alleen over commutatieve groepen zullen hebben, noemen we dat eenvoudigweg 'groep'.

Euclidische ringen

Een Euclidische ring is een domein waarin *deling met rest* mogelijk is. De klasse-declaratie is:

```
class  Ring a => Euclid a  where
    orde  ::  a -> Int
    quot  ::  a -> a -> a
    rest  ::  a -> a -> a
```

Bij elk twee elementen kan dus een 'quotiënt' en een 'rest' berekend worden. Deze enigszins suggestieve namen worden gerechtvaardigd door de wetten. De 'rest' moet kleiner zijn dan de 'noemer' van de deling, en als je het 'quotiënt' met de noemer vermenigvuldigt, houd je precies de rest over. In deze wet wordt het begrip 'kleiner' gebruikt. In plaats van de voorwaarde te stellen dat een instance van `Euclid` ook een instance van `Ord` is, is er een functie `orde` in de klasse opgenomen. Elementen kunnen dan 'geordend' worden door hun respectievelijke ordes te ordenen.

E1 `(n <*> quot t n) <+> rest t n` = `t`
E2 `orde (rest t n)` < `orde n`

De definitie van `rest` kan als default-definitie in de klasse-declaratie worden opgenomen. Op grond van wet E1 geldt namelijk:

```
rest t n  =  t <-> (n <*> quot t n)
```

Lichamen

Een `Lichaam` heeft de rijkste structuur van de vijf genoemde klassen. Naast alle eerder genoemde functies is er in een lichaam bij elk element (behalve `nul`) een 'omgekeerde':

```
class  Euclid a => Lichaam a  where
    omg :: a -> a
```

In de wetten wordt gespecificeerd dat het product van een element en zijn omgekeerde de waarde `een` oplevert:

L1 `x <*> omg x` = `een`

6.3.3 Afgeleide operatoren

Gebruikmakend van de operatoren en functies in de verschillende klassen, is het mogelijk om nieuwe operatoren te definiëren. Deze operatoren worden daardoor vanzelf ook overloaded.

Op types uit de klasse `Groep` kan de operator `<->` gedefinieerd worden als het optellen van het tegengestelde:

```
(<->)   ::  Groep a  =>  a -> a -> a
x <-> y  =  x <+> neg y
```

Voor deze operator gelden allerlei wetten, die volgen uit de gegeven wetten en de definitie. In een ring geldt bijvoorbeeld dat 'vermenigvuldigen' distribueert over 'aftrekken': `x <*> (y<->z)` = `(x<*>y) <-> (x<*>z)` (zie opgave 6.5). blz. 131

In een ring kun je vermenigvuldigen. Machtsverheffen kan gedefinieerd worden als herhaald vermenigvuldigen:

```
(<^>)      ::  Ring a  =>  a -> Int -> a
x <^> 0     =  een
x <^> (n+1) =  x <*> (x <^> n)
```

Het komt goed van pas dat er in de ring een element `een` beschikbaar is om te dienen als resultaat van `x<^>0`. Merk op dat de rechter parameter van `<^>` een gewone integer is, waarvoor we dus `0` schrijven, en niet `nul`. Omdat `<*>` een associatieve operator is, maakt het niet uit of we links of rechts van de recursieve aanroep vermenigvuldigen met `x`.

In een Euclidische ring kun je 'delen met rest'. Je kunt dus ook bepalen of de rest gelijk aan nul is, en daarmee het begrip 'deelbaar' definiëren:

```
deelbaar  ::  Euclid a  =>  a -> a -> Bool
deelbaar x y  =  rest x y == nul
```

Omdat de elementen dankzij de functie `orde` ordenbaar zijn, is het in een Euclidische ring mogelijk om van een 'grootste gemene deler' te spreken. Daarvoor kan het algoritme in paragraaf 3.3.3 blz. 56

gegeneraliseerd worden naar een willekeurige Euclidische ring:

```
ggd x y  |  y==nul     =  x
         |  otherwise  =  ggd y (rest x y)
```

6.3.4 Instances van de klassen

Diverse types die in dit diktaat aan de orde zijn geweest, kunnen als instance gedeclareerd worden van de klassen in de hiërarchie. Het type `Int` bijvoorbeeld is een `Monoid`, een `Groep`, een `Ring` en een `Euclid` (maar geen `Lichaam`):

```
instance Monoid Int where
      (<+>)  =  (+)
      nul    =  0
instance Groep Int where
      neg    =  negate
instance Ring Int where
      (<*>)  =  (*)
      een    =  1
instance Euclid Int where
      orde   =  abs
      quot   =  (/)
      rest   =  rem
```

Het type `Float` is behalve dat ook een instance van `Lichaam`. Dat `Float` een instance is van `Euclid` is haast te flauw om te definiëren: elke deling gaat in `Float` altijd precies op, dus de rest is altijd nul (dat geldt ook in andere lichamen).

```
instance Monoid Float where
      (<+>) = (+)
      nul   = 0.0
instance Groep Float where
      neg   = negate
instance Ring Float where
      (<*>) = (*)
      een   = 1.0
instance Euclid Float where
      orde x    = 0
      quot      = (/)
      rest x y  = 0.0
instance Lichaam Float where
      omg   =  (1.0/)
```

Ook de rationale getallen zijn een instance van alle vijf de klassen. Daarbij kan de benadering
blz. 55 gekozen worden uit paragraaf 3.3.3 (alle rationale getallen worden na een berekening direct ver-
blz. 65 eenvoudigd), of die uit opgave 3.8 (gelijkheid wordt zodanig gedefinieerd, dat hij ook werkt op niet-vereenvoudigde breuken). Hieronder is voor die laatste benadering gekozen. De rationale getallen zijn als beschermd datatype gedefinieerd.

```
data  Ratio  =  Rat (Int,Int)
instance  Eq Ratio  where
    Rat (a,b) == Rat (c,d)  =  a*d == c*b
instance  Monoid Ratio  where
    Rat (a,b) <+> Rat (c,d) =  Rat (a*d+c*b, b*d)
    nul                     =  Rat (0,1)
instance  Groep Ratio  where
    neg (Rat (a,b))         =  Rat (-a, b)
instance  Ring Ratio  where
    Rat (a,b) <*> Rat (c,d) =  Rat (a*c, b*d)
    een                     =  Rat (1,1)
instance  Lichaam Ratio  where
    omg (Rat (a,b))         =  Rat (b,a)
```

De hele indeling in vijf klassen was erom begonnen dat er typen zijn die niet tot alle klassen behoren. Lijsten zijn daar een voorbeeld van: de enige van de vijf klassen waarvan lijsten een instance zijn, is de `Monoid`. De rol van de operator wordt gespeeld door lijstconcatenatie (`++`), wat immers een associatieve operator is met een neutraal element:

```
instance  Eq a => Monoid [a]  where
    (<+>)  =  (++)
    nul    =  []
```

Lijsten vormen geen groep, omdat er geen functie `neg` op lijsten gedefinieerd kan worden met de gewenste eigenschap.

Het type `Bool` vormt een `Ring`. De rol van de operator `<+>` wordt daarbij gespeeld door de ongelijkheids-operator (een voorbeeld van het feit dat je bij `<+>` niet direct aan optelling hoeft te denken). De 'and'-operator speelt de rol van `<*>`. De instance-declaraties luiden:

```
instance  Monoid Bool  where
    (<+>)  =  (/=)
    nul    =  False
instance  Groep Bool  where
    neg    =  id
instance  Ring Bool  where
    (<*>)  =  (&&)
    een    =  True
```

Een ander voorbeeld van een ring zijn lijsten van integers. De operaties worden daarbij 'puntsgewijs' uitgevoerd door middel van `zipWith` (voor operatoren met twee parameters), `map` (voor operatoren met één parameter) en `repeat` (voor constanten). De instance-declaraties luiden:

```
instance  Monoid [Int]  where
    (<+>)  =  zipWith (+)
    nul    =  repeat 0
instance  Groep [Int]  where
    neg    =  map negate
instance  Ring [Int]  where
   (<*>)   =  zipWith (*)
   een     =  repeat 1
```

We kunnen niet op deze manier doorgaan, en lijsten van integers tot Euclidische ring maken. De gedefinieerde operatoren voldoen namelijk niet aan wet D2. Wel kunnen de declaraties nog wat algemener gemaakt worden. De elementen van de lijst hoeven namelijk niet per se integers te zijn; het is voldoende als het element-type een instance is van `Ring`:

```
instance  Monoid a  =>  Monoid [a]  where
    (<+>)  =  zipWith (<+>)
    nul    =  repeat nul
instance  Groep a  =>  Groep [a]  where
    neg    =  map neg
instance  Ring a  =>  Ring [a]  where
   (<*>)   =  zipWith (<*>)
   een     =  repeat een
```

Met deze declaratie zijn bijvoorbeeld lijsten van `Bool`'s een ring, en daarom ook lijsten van lijsten van `Bool`'s, enzovoort. Het enige probleem van deze declaratie is dat hij zich niet verdraagt met de eerder gegeven instance-declaratie voor `Monoid [a]`. Wat zou immers het resultaat moeten zijn van `[1,2]<+>[3,4]` – de concatenatie van de twee lijsten `[1,2,3,4]`, of de puntsgewijze optelling `[4,6]`? Als beide instance-declaraties in één programma staan, is het gevolg daarom een foutmelding.

6.3.5 Polynoomringen

In paragraaf 4.3.1 werd een datastructuur `Poly` gedefinieerd, waarmee polynomen beschreven kunnen worden: blz. 80

```
data Poly  =  Poly [Term]
data Term  =  Term (Float,Int)
```

In paragraaf 4.3.3 werden vervolgens een aantal functies gedefinieerd: `pPlus` om polynomen op te tellen, `pNeg` om het tegenovergestelde van een polynoom te bepalen, en `pMaal` om polynomen te vermenigvuldigen. Met deze drie functies vormen de polynomen een `Ring`: blz. 82

```
instance  Monoid Poly  where
    (<+>)  =  pPlus
    nul    =  Poly []
```

```
instance  Groep Poly  where
    neg    =  pNeg
instance  Ring Poly  where
   (<*>)   =  pMaal
   een     =  Poly [Term (1.0,0)]
```

Nagegaan kan worden, dat de genoemde operaties inderdaad aan de vereiste eigenschappen voldoen.

Een polynoom is een lijst termen, waarbij elke term bestaat uit een *coëfficiënt* en een *exponent*. Een voorbeeld van een term is $1.5x^3$. Dat de exponent een geheel getal is (zelfs een natuurlijk getal), is wezenlijk in een polynoom. Maar waarom zou de coëfficiënt een `Float` moeten zijn? Er zijn evengoed polynomen denkbaar met integers als coëfficiënten, of met complexe getallen.

Door naar de definities van de functies op polynomen te kijken, is het te achterhalen welke eigenschappen van de coëfficiënten er precies nodig zijn. In de functie `pPlus` wordt de functie `pEenvoud`
blz. 81 gebruikt. In deze functie (gedefinieerd in paragraaf 4.3.2) worden coëfficiënten opgeteld en met nul vergeleken. In de functie `pNeg` wordt de functie `tNeg` gebruikt, die van een coëfficiënt het tegenovergestelde bepaalt. In de functie `pMaal` tenslotte wordt de functie `tMaal` gebruikt, die coeëfficiënten vermenigvuldigt. Nergens worden coëfficiënten op elkaar gedeeld. Oftewel: het is voldoende als de coëfficiënten een `Ring` vormen. Een Euclidische ring, of zelfs een lichaam, is dus niet vereist.

De vrijheid in de keuze van het type van de coëfficënten komt tot uitdrukking in een nieuwe definitie van polynomen:

```
data  Poly a  =  Poly [Term a]
data  Term a  =  Term (a,Int)
```

Dit is een polymorfe definitie van `Poly a`, uit te spreken als 'polynomen met coëfficiënten in `a`'.

Een type `Poly a` is een instance van `Ring` als het type `a` dat is. Dit komt tot uiting in de voorwaardelijke instance-declaratie

```
instance  Ring a => Ring (Poly a)  where ...
```

De definities van de functies op polynomen kunnen aangepast worden, zodat ze werken in een willekeurige ring. Eerst de hulpfuncties op termen (vergelijk de overeenkomstige definities in pa-
blz. 82 ragraaf 4.3.3):

```
tNeg  ::  Groep a  =>  Term a -> Term a
tNeg (Term (c,e))  =  Term (neg c, e)
tMaal ::  Ring a   =>  Term a -> Term a -> Term a
tMaal (Term(c1,e1)) (Term(c2,e2))  =  Term (c1<*>c2, e1+e2)
```

blz. 131 Ook de functie `pEenvoud` kan zo aangepast worden (opgave 6.9).

De nieuwe aangepaste definities van de functies op polynomen kunnen direct in de instance-declaraties worden gegeven:

```
instance  Monoid a  =>  Monoid (Poly a)  where
    Poly xs <+> Poly ys  =  pEenvoud (Poly (xs++ys))
    nul                  =  Poly []
instance  Groep a   =>  Groep (Poly a)  where
    neg (Poly xs)        =   Poly (map tNeg xs)
instance  Ring a    =>  Ring (Poly a)  where
    Poly xs <*> Poly ys  =   pEenvoud (Poly (cpWith tMaal xs ys))
    een                  =   Poly [Term (een,0)]
```

blz. 125 De relevante afgeleide functies uit paragraaf 6.3.3, zoals `<->` en `<^>`, krijgen we kado. De functie `kwadraat` bijvoorbeeld werkt immers op een willekeurige instance van `Ring`, dus ook op polynomen. In een sessie kan bijvoorbeeld het polynoom dat het resultaat is van $(x+1)^4$ berekend worden:

```
? (Poly [ Term(1,1), Term(1,0) ]) <^> 4
Poly [ Term(1,4), Term(4,3), Term(6,2), Term(4,1), Term(1,0) ]
```

(Dat klopt, want $(x+1)^4$ is $x^4 + 4x^3 + 6x^2 + 4x + 1$).

Polynomen als Euclidische ring

Het is mogelijk om op polynomen een 'deling met rest' uit te voeren. De graad van het polynoom
blz. 82 (de functie `pGraad` uit paragraaf 4.3.3) vervult daarbij de rol van `orde`-functie. Een voorbeeld is

de volgende deling:

$$\frac{2x^4+5x^3+4x^2-3x+2}{x^2+x+1}$$

Het quotiënt van deze deling is $2x^2+3x-1$, en de rest is $-5x+3$. Deze uitkomsten voldoen aan de vereiste wetten E1 (reken maar na) en E2 (de graad van de rest (1) is kleiner dan de graad van de noemer (2)).

Voor het algoritme en de daarvoor benodigde voorwaarden bekijken we eerst hoe het voorbeeldresultaat berekend wordt. De polynomen kunnen gedeeld worden met een soort staartdeling:

$$\begin{array}{l} x^2{+}x{+}1 / 2x^4{+}5x^3{+}4x^2{-}3x{+}2 \backslash 2x^2{+}3x{-}1 \\ \quad \underline{2x^4{+}2x^3{+}2x^2} \\ \qquad\quad 3x^3{+}2x^2{-}3x \\ \qquad\quad \underline{3x^3{+}3x^2{+}3x} \\ \qquad\qquad\quad -x^2{-}6x{+}2 \\ \qquad\qquad\quad \underline{-x^2\ -x{-}1} \\ \qquad\qquad\qquad\quad -5x{+}3 \end{array}$$

Om te beginnen wordt gekeken naar de eerste term van de teller ($2x^4$) en de eerste term van de noemer (x^2). Deze twee worden door elkaar gedeeld, en dat levert de eerste term van het resultaat ($2x^2$). Vervolgens wordt deze $2x^2$ vermenigvuldigd met de complete noemer (x^2+x+1). Het product ($2x^4{+}2x^3{+}2x^2$) wordt afgetrokken van de teller. De term met de hoogste exponent ($2x^4$) is daarmee verdwenen. Het procédé wordt herhaald met het overgebleven deel ($3x^3{+}2x^2{-}3x{+}2$). Dat levert de tweede term van het resultaat ($3x$). De herhaling gaat door, totdat de eerste coëfficiënt van de teller een lagere graad heeft dan de noemer (in het voorbeeld $-5x+3$). Dat is de gezochte rest, en inmiddels is ook het hele quotiënt opgebouwd.

In dit algoritme wordt gebruik gemaakt van het vermenigvuldigen en aftrekken van polynomen. Dat is geen probleem, want polynomen vormen een ring. Maar daarnaast is het nodig dat termen door elkaar gedeeld kunnen worden (bijvoorbeeld $3x^3/x^2 = 3x$). Daarvoor moeten de coëfficiënten door elkaar gedeeld worden, en de exponenten afgetrokken. Deze deling moet exact geschieden (dus zonder rest). Conclusie:

> *Polynomen vormen een Euclidische ring,*
> *mits de coëfficiënten een lichaam vormen.*

Deze voorwaarde wordt in onderstaande instance-declaratie gesteld. Het gebruikte algoritme verloopt precies zoals in het behandelde voorbeeld: eerst wordt een polynoom `h` bepaald, die één term van het resultaat bevat. Daarbij wordt het resultaat van een recursieve aanroep opgeteld. In de recursieve aanroep wordt de teller (`f`) verminderd met `h` keer de noemer (`g`).

```
instance  Lichaam a  =>  Euclid (Poly a)  where
    orde (Poly [])             =  -1
    orde (Poly (Term(c,e):ts)) =  e
    quot f g | n<m   =  nul
             | n>=m  =  h <+> quot (f <-> h <*> g) g
                where  n = orde f
                       m = orde g
                       h = Poly [Term(coef1 f </> coef1 g, n-m)]
                       coef1 (Poly (Term(c,e):ts)) = c
```

Voor de functie `rest` kan de default-definitie uit de klasse-declaratie gebruikt worden.

Omdat de rationale getallen (`Ratio`) een lichaam vormen, kan van polynomen met rationale getallen als coëfficiënten nu ook het quotiënt en de rest bepaald worden. Er zijn geen nieuwe functiedefinities nodig; door de typering weet de interpreter alle gegeven definities op de juiste manier te combineren. Zo heeft bijvorbeeld de deling

$$\frac{\frac{1}{3}x^3+\frac{1}{5}x-\frac{1}{2}}{\frac{1}{2}x+1}$$

als quotiënt het polynoom $\frac{2}{3}x^2-\frac{4}{3}x+\frac{46}{15}$, en als rest het polynoom met alleen de constante term $-\frac{107}{30}$.

Doordat de polynomen nu een Euclidische ring vormen, krijgen we de functie `ggd` kado. Wel moeten de coëfficiënten van de polynoom daarvoor een lichaam vormen.

6.3.6 Quotiëntenlichamen

blz. 126

De rationale getallen vormen een lichaam, zoals gedefinieerd in paragraaf 6.3.4. Net als de definitie van polynomen kan deze definitie gegeneraliseerd worden. De teller en de noemer van een breuk hoeven immers geen integer te zijn. Als je naar de definitie van de functies op rationale getallen kijkt, zie je wat van de teller en noemer verwacht wordt: ze moeten optelbaar en vermenigvuldigbaar zijn. Nergens worden ze meegegeven aan functies die per se een integer verwachten, zoals `take` of `repeat`. Teller en noemer van een breuk mogen dus als type een willekeurige instance van `Ring` hebben.

Deze generalisatie van de rationale getallen noemen we `Quot a`: dit is het polymorfe type 'breuken met teller en noemer van type `a`'. Dit type kan instance gemaakt worden van alle vijf de klassen uit deze sectie (`Monoid`, `Groep`, `Ring`, `Euclid` en `Lichaam`), en bovendien van de klasse `Eq`. Voorwaarde is steeds dat het type van teller en noemer een ring is. Zelfs voor het `Eq`-zijn van het quotiënttype geldt deze voorwaarde al: om te bepalen of twee breuken gelijk zijn moeten ze immers kruiselings vermenigvuldigd worden.

De complete definitie van quotiëntenlichamen is als volgt (dit is een rechtstreekse generalisatie van de rationale getallen):

```
data  Quot a  =  Quot (a,a)
instance  Ring a  =>  Eq (Quot a)  where
     Quot (a,b) == Quot (c,d) =  a<*>d == c<*>b
instance  Ring a  =>  Monoid (Quot a)  where
     Quot (a,b) <+> Quot (c,d) = Quot (a<*>d <+> c<*>b, b<*>d)
     nul  = Quot (nul , een)
instance  Ring a  =>  Groep (Quot a)  where
     neg (Quot (a,b)) = Quot (neg a, b)
instance  Ring a  =>  Ring (Quot a)  where
     Quot (a,b) <*> Quot (c,d) = Quot (a<*>c, b<*>d)
     een = Quot (een, een)
instance  Ring a  =>  Lichaam (Quot a)  where
     omg (Quot (a,b))  =  Quot (b,a)
```

De definitie van `Euclid (Quot a)` is hier maar weggelaten, omdat die (net als bij `Float`) triviaal is.

Omdat quotiënten van elementen van een ring een lichaam vormen, spreken we van 'quotiëntlichamen'. Zo bestaat er bijvoorbeeld het lichaam 'quotiënten van polynomen met integers als coëfficiënten'.

6.3.7 Matrixringen

blz. 71

Ook de definitie van matrices uit sectie 4.2 kan gegeneraliseerd worden. De elementen van een matrix hoeven niet per se `Float`'s te zijn; het kunnen elementen van een willekeurige ring zijn. Alleen voor de matrix-inverteerfunctie is lichaam-zijn van de elementen vereist.

Als de elementen in een matrix een ring zijn, vormen de matrices zelf ook een ring. Daarbij speelt matrixvermenigvuldiging de rol van `<*>`, en de identiteitsmatrix de rol van `een`. Aan de voorwaarden is voldaan. Zo is wet R1 bijvoorbeeld geldig omdat matrixvermenigvuldiging werd gedefinieerd als het na elkaar toepassen van lineaire afbeeldingen. Het na elkaar toepassen van
blz. 35
afbeeldingen is associatief, zoals in opgave 2.4 werd bewezen.

De matrix-functies kunnen eenvoudig worden aangepast, zodat de volgende declaraties kunnen worden gedaan:

```
data  Matrix a  =  Mat [[a]]
instance  Eq a     => Eq     (Matrix a)  where ...
instance  Monoid a => Monoid (Matrix a)  where ...
instance  Ring a   => Ring   (Matrix a)  where ...
```

Door matrices in te passen in de klasse-hiërarchie komen er allerlei nieuwe mogelijkheden beschikbaar. Er kunnen bijvoorbeeld berekeningen gedaan worden met matrices van polynomen met

blz. 132 complexe getallen als coëfficiënten (zie opgave 6.12 voor een toepassing hiervan). En wat te denken van integer-matrices als coëfficiënten van polynomen? En daar weer het quotiëntlichaam van... de mogelijkheden zijn onbegrensd.

Opgaven

6.1 Schrijf een declaratie waardoor de operatoren `+`, `-`, `*` en `/` ook op complexe getallen (zie opgave 3.9) gebruikt kunnen worden. blz. 65

6.2 Verklaar uit de manier waarop `+` gedefinieerd is, dat het uitrekenen van `1+2` de tweede keer één reductie minder kost dan de eerste keer (zie ook paragraaf 5.1.1). blz. 85

6.3 Definieer een type `Set a` dat verzamelingen voorstelt met elementen uit `a`. Gebruik lijsten om dit type te implementeren. Definieer een functie

```
subset :: Set a -> Set a -> Bool
```

die controleert of een verzameling een deelverzameling is van een andere. Schrijf vervolgens een instance-declaratie waarmee `Set a` een instance van `Eq` wordt (met als voorwaarde dat `a` een instance is van `Eq`). Bedenk dat bij verzamelingen, anders dan bij lijsten, volgorde en verdubbelingen van elementen geen rol speelt. Waarom moet het type gedefinieerd worden als beschermd datatype, dus met behulp van `data` en niet met `type`?

6.4 Definieer een klasse `Finite` ('eindig'). Deze klasse bezit geen operatoren en functies, maar wel een constante: de lijst van alle elementen van het type. Het is de bedoeling dat die lijst eindig is, vandaar de naam. Definieer de volgende types als instances van `Finite`:

- `Bool`;
- `Char`;
- `(a,b)`, mits `a` en `b` eindig zijn;
- `Set a` (zoals gedefinieerd in de vorige opgave), mits `a` eindig is;
- `(a->b)`, mits `a` en `b` eindig zijn en `Eq a` (moeilijk, voor de liefhebber).

Maak nu `(a->b)` tot instance van `Eq` mits `a` eindig is. Zijn er nog meer voorwaarden voor deze instance-declaratie?

6.5 Bewijs dat in elke ring geldt: $a * (b - c) = a * b - a * c$, en $a * 0 = 0$. Gebruik de definitie van $-$ uit paragraaf 6.3.3. Let goed op dat je alleen maar de voor ringen geldende wetten gebruikt! blz. 125

6.6 In de laatste zin van paragraaf 6.3.4 is sprake van een 'foutmelding'. Welke foutmelding wordt bedoeld? blz. 126

6.7 Beschouw het type `Klein`, dat gedefinieerd is als:

```
data Klein = E | A | B | C
```

(genoemd naar de Duitse wiskundige Felix Klein). Dit type wordt een instance van `Groep` gemaakt, door:

```
instance  Monoid Klein  where
    (<+>)  =  f
    nul    =  E
instance  Groep Klein  where
    neg    =  id
```

Hoe kan de functie `f` worden gedefinieerd?

6.8 In welke klasse zou de functie `fromInteger` geplaatst moeten worden: `Monoid`, `Groep`, `Ring`, `Euclid` of `Lichaam`? Geef een default-definitie.

6.9 Welke wijzigingen zijn er nodig in de functie `pEenvoud` uit paragraaf 4.3.2 om te werken voor polynomen met coëfficiënten in een willekeurige ring, zoals beschreven in paragraaf 6.3.5? Wat wordt het type daardoor? blz. 81 blz. 127

6.10 Wat gebeurt er als de functie `quot` in de instance-declaratie van polynomen als `Euclid` (paragraaf 6.3.5), als tweede parameter het nul-polynoom meekrijgt? blz. 129

6.11 Geef een zinvolle definitie van `(a->a)` als instance van `Monoid` (stoor je even niet aan het feit dat functies geen instance zijn van `Eq`). Welke deelverzameling van `(Float->Float)` is een instance van `Groep`?

6.12 Bij het oplossen van differentiaalvergelijkingen (zoals die in de natuurkunde veel voorkomen) spelen *eigenwaardes* een belangrijke rol. Eigenwaardes zijn als volgt gedefinieerd. Als A een vierkante matrix is, dan heet k een eigenwaarde van A als voor alle vectoren v geldt: $A@v = k \cdot v$. (Met '@' wordt hier de functie `matApply` bedoeld, en met '$\cdot$' de functie `vecScale`). Schrijf een functie die gegeven een matrix de vergelijking bepaalt waaraan de eigenwaardes moeten voldoen.

Hoofdstuk 7

Programmeertechnieken

7.1 Expressiebomen

7.1.1 Rekenkundige expressies

Data-declaraties worden gebruikt om de vorm van datastructuren te beschrijven. Een veel voorkomende niet-lijstvormige datastructuur is de *ontleedboom*. Een ontleedboom is een symbolische beschrijving van een expressie. Een expressie wordt in een ontleedboom dus in de vorm van een datastructuur opgeslagen.

Numerieke expressies worden bijvoorbeeld beschreven door bomen die zijn opgebouwd volgens de volgende data-declaratie:

```
data Expr = Con Float
          | Var String
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
```

De data-declaratie beschrijft de opbouw van rekenkundige expressies. Voor elk soort expressie (constante, variabele, optelling, aftrekking, vermenigvuldiging en deling) is er een constructor waarmee de representatie van de expressie kan worden opgebouwd.

Twee van de zes contructoren (`Con` en `Var`) hebben één parameter. De andere vier hebben twee parameters. Voor de duidelijkheid schrijven we ze als infix-operator, dus tussen de parameters in plaats van er voor. Operatoren die een constructor voorstellen in plaats van een gewone functie moeten met een dubbele punt beginnen. Voor de symmetrie (het oog wil ook wat) eindigen de operatoren in de gegeven data-declaratie ook op een dubbele punt. De drie symbolen in `:+:` vormen één operator, en moeten dan ook zonder spatie ertussen geschreven worden.

De constructor-operatoren uit de data-declaratie kunnen van een prioriteit en een associatievolgorde worden voorzien met behulp van een infix-declaratie. Deze werd ingevoerd in paragraaf 2.1.4. Het is het handigste om de operatoren van dezelfde prioriteit te voorzien als de gewone rekenkundige operatoren: blz. 23

```
infixl  7  :*:
infix   7  :/:
infixl  6  :+:, :-:
```

Na deze declaraties kan bijvoorbeeld de expressie $3x+4y$ als datastructuur worden gerepresenteerd door de Haskell-expressie

```
Con 3.0 :*: Var "x" :+: Con 4.0 :*: Var "y"
```

Het is belangrijk om onderscheid te maken tussen expressies in Haskell, en rekenkundige expressies in het taaltje dat door de data-declaratie wordt beschreven. Zo is `Var "x"` een Haskell-expressie die als waarde de datastructuur heeft, die de rekenkundige expressie x beschrijft.

7.1.2 Symbolisch differentiëren

Het voordeel van de representatie van expressies als datastructuren is dat we functies kunnen schrijven die op expressies werken, en het resultaat kunnen bekijken. Dit wordt *symbolische manipulatie* van expressies genoemd. Een goed voorbeeld van symbolische manipulatie is het *differentiëren*

van een expressie. Als we de expressie `Var "x" :*: Var "x"` differentiëren, dan komt daar de expressie `Con 2.0 :*: Var "x"` uit, of iets wat daar equivalent aan is.

Het symbolische differentiëren van een expressie biedt veel meer mogelijkheden dan het numeriek
blz. 31 differentiëren, waarvoor in paragraaf 2.4.2 de volgende functie werd geschreven:

```
diff f  =  f'
    where  f' x  =  (f (x+h) - f x) / h
           h     =  0.0001
```

Een numeriek gedifferentiëerde functie is immers een functie; het enige wat we met die functie kunnen doen is hem op een parameter toepassen. Het is niet mogelijk om het functievoorschrift van de numeriek gedifferentieerde functie te zien te krijgen. Bij het gebruik van expressie-bomen en symbolisch differentiëren is dat wèl mogelijk.

De symbolische differentieerfunctie heeft behalve een expressie een `Char` als parameter die aangeeft naar welke variabele de expressie gedifferentieerd moet worden (dit is ook iets wat bij numeriek differentiëren niet mogelijk was). In de definitie wordt voor elk van de zes constructoren van expressies aangegeven hoe de expressie gedifferentieerd kan worden. Daarbij kunnen de bekende rekenregels voor differentiëren gevolgd worden: de afgeleide van een som is bijvoorbeeld de som van de afgeleiden, en voor de afgeleide van een product geldt de 'productregel' ($(fg)' = fg' + gf'$).

```
afg :: Expr -> String -> Expr
afg (Con c)   dx  =  Con 0.0
afg (Var x)   dx
     | x==dx      =  Con 1.0
     | otherwise  =  Con 0.0
afg (f :+: g) dx  =  afg f dx :+: afg g dx
afg (f :-: g) dx  =  afg f dx :-: afg g dx
afg (f :*: g) dx  =  f :*: afg g dx  :+:  g :*: afg f dx
afg (f :/: g) dx  =  ( g :*: afg f dx :-: f :*: afg g dx )
                     :/: ( g :*: g )
```

Uit de definitie blijkt verder dat de afgeleide van een constante de constante 0 is. De afgeleide van een variabele is de constante 1 als het de variabele betreft waarnaar gedifferentieerd wordt (in de definitie suggestief `dx` genoemd). Andere variabelen gedragen zich als constanten.

7.1.3 Andere expressiebomen

Naast de data-declaratie die rekenkundige expressies beschrijft, zijn ook andere soorten expressies te beschrijven met data-declaraties. Onderstaande data-declaratie beschrijft bijvoorbeeld Boolese expressies oftewel *proposities*:

```
data Prop  =  Cons Bool
           |  Vari Char
           |  Not Prop
           |  Prop :/\: Prop
           |  Prop :\/: Prop
           |  Prop :->: Prop
```

De propositie $b \vee \neg b$ wordt bijvoorbeeld gerepresenteerd door de datastructuur

```
Vari 'b' :\/: Not (Vari 'b')
```

Functies op het type `Prop` worden natuurlijk weer geschreven door voor alle vijf constructoren een patroon te gebruiken. De functie `verw` bijvoorbeeld, verwijdert alle voorkomens van `:\/:` en `:->:` uit een propositie. Daarvoor worden rekenregels uit de propositielogica, zoals de wet van De Morgan toegepast.

```
verw (Cons b)     =  Cons b
verw (Vari x)     =  Vari x
verw (Not p)      =  Not (verw p)
verw (p :/\: q)   =  verw p :/\: verw q
verw (p :\/: q)   =  Not (Not (verw p) :/\: Not (verw q))
verw (p :->: q)   =  verw (Not p :\/: q)
```

Met data-declaraties kunnen naast expressies ook taalconstructies uit programmeertalen worden beschreven. Statements uit een Pascal-achtige taal kunnen bijvoorbeeld worden beschreven door de volgende data-declaratie:

```
data Stat  =  Assign Char Expr
           |  If Prop Stat Stat
           |  While Prop Stat
           |  Repeat Stat Prop
           |  Compound [Stat]
```

Door functies te schrijven die op dit soort datastructuren werken, is het mogelijk om Pascal-programma's (althans de boom-representaties daarvan) te transformeren volgens 'rekenregels' die daarvoor gelden.

7.1.4 Stringrepresentatie van een boom

We keren weer terug naar de expressiebomen uit paragraaf 7.1.1. De Haskell-expressies die nodig zijn om een rekenkundige expressie als boom te representeren, lijken sterk op die expressies zelf. Zo wordt bijvoorbeeld de expressie $x*x+1$ gerepresenteerd door `Var "x":*:Var "x":+:Con 1.0`. Het is alleen jammer dat op elke constante eerst de constructor `Con` moet worden toegepast, op elke variabele `Var`, en dat elke operator van dubbele punten moet worden voorzien. Makkelijker zou het zijn als de expressie direct kan worden ingetikt. De simpelste manier om een datastructuur te maken waar Haskell mee uit de voeten kan, is om de rekenkundige expressie in een string te zetten: `"x*x+1"`.

blz. 133

In de paragraaf 7.3.3 zullen we een functie

blz. 141

```
ontleed :: String -> Expr
```

schrijven, die zo'n string omzet in de overeenkomstige boomstructuur. Maar eerst bekijken we het omgekeerde probleem: een functie `weerg` die een expressie-boom weergeeft als string. Als beide functies beschikbaar zijn, dan kunnen we bijvoorbeeld de differentieer-functie `afg` 'inpakken' zodat het een functie tussen strings wordt:

```
afgel :: String -> String -> String
afgel exprstr dx = weerg ( afg (ontleed exprstr) dx )
```

De ingepakte functie is eenvoudiger te gebruiken dan de oorspronkelijke functie `afg`. Een sessie kan er bijvoorbeeld als volgt uitzien:

```
? afgel "x*x+1" "x"
x*1+1*x+0
```

Dit gebruik van de symbolische expressie-manipulatie functies is natuurlijk eenvoudiger dan

```
? afg (Var "x" :*: Var "x" :+: Con 1.0) "x"
Var "x":*:Con 1:+:Con 1:*:Var "x":+:Con 0
```

De functie `weerg` werkt op expressie-bomen, en wordt daarom met zes patronen gedefinieerd voor alle constructoren van `Expr`.

```
weerg          ::  Expr -> String
weerg (Con n)   = show n
weerg (Var x)   = x
weerg (a :+: b) = "("++ weerg a ++ "+" ++ weerg b ++")"
weerg (a :-: b) = "("++ weerg a ++ "-" ++ weerg b ++")"
weerg (a :*: b) =       weerg a ++ "*" ++ weerg b
weerg (a :/: b) =       weerg a ++ "/" ++ weerg b
```

De functie `show` is een standaardfunctie die een stringrepresentatie geeft van onder andere `Float` waarden. De functie `weerg` wordt waar nodig recursief aangeroepen; de resulterende strings worden samengevoegd tot één lange string, waarin ook nog het symbool voor de betreffende operator wordt opgenomen. Bij het optellen en aftrekken worden er ook nog haakjes in het resultaat gezet, anders zou de boom-expressie `(Var "a":+:Var "b"):*:(Var "c":+:Var "d")` worden omgezet in de string `"a+b*c+d"`, die de verkeerde interpretatie heeft.

7.2 Invoer en uitvoer

7.2.1 Interactief gebruik van Haskell

Tot nu toe hebben we de Haskell-interpretator steeds gebruikt door een functie aan te roepen met een bepaalde parameter. Het resultaat werd dan op het scherm geschreven. Hoewel dit model

goed voldoet in eenvoudige gevallen, zijn er toch een aantal nadelen aan verbonden:

- De gebruiker moet altijd een Haskell-expressie intikken, bestaande uit een functienaam en parameters.
- Als een parameter een string is, moet die tussen aanhalingstekens geplaatst worden.
- De invoer van het programma (parameters van de functie) moet bij aanroep van de functie bekend zijn; het is bijvoorbeeld niet mogelijk om te reageren op uitvoer van de functie.
- Het is niet mogelijk om files te lezen en te schrijven.

Al deze problemen hebben te maken met de aanwezigheid van een interactieve gebruiker en met het afwisselend doen van invoer en uitvoer. Om de problemen op te kunnen lossen is er in Haskell een primitief type `Dialogue` beschikbaar. Als de interpretator een expressie evalueert waarvan het type `Dialogue` is, wordt de waarde niet op het scherm geschreven; in plaats daarvan wordt de waarde als 'dialoog' beschouwd, die met de gebruiker 'gevoerd' kan worden.

In de prelude zijn een aantal functies gedefinieerd die een `Dialogue` opleveren. Een handige functie is bijvoorbeeld `interact`:

```
interact :: (String->String) -> Dialogue
```

Deze functie heeft als parameter een functie die strings in strings omzet. Als de dialoog die het resultaat is met de gebruiker wordt gevoerd, wordt alles wat de gebruiker intikt als één lange string beschouwd. De gegeven functie wordt daarop toegepast; de string die het resultaat is wordt op het scherm afgebeeld.

Als eerste voorbeeld bekijken we het gebruik van `interact`, waarbij als functie eenvoudigweg de identiteit wordt meegegeven:

```
?  interact id
dit is tekst      ← dit tikt de gebruiker in
dit is tekst      ← dit is het antwoord
dit ook           ← dit tikt de gebruiker weer in
dit ook           ← en dit verschijnt weer op het scherm
```

De dialoog gaat net zolang door tot het programma wordt afgebroken met control-C, of totdat het einde van de invoer wordt aangegeven met control-D. De invoer en de uitvoer verschijnt door elkaar heen. Dit is een gevolg van lazy evaluatie: een gedeelte van de uitvoer is al bekend, nog voordat de gehele invoer is ingetikt.

In het volgende voorbeeld geven we een wat interessantere functie mee aan `interact`. De functie `toUpper` maakt kleine letters tot hoofletters, en laat andere karakters ongemoeid. Met de functie `map` kunnen we deze functie elementsgewijs op een string toepassen. Door `map toUpper` als parameter van `interact` te gebruiken, wordt de functie interactief op de invoer toegepast:

```
?  interact (map toUpper)   ← zo wordt de dialoog gestart
Dit is tekst                ← dit tik je in
DIT IS TEKST                ← en dit is de uitvoer
Dit ook
DIT OOK
```

Een variant op `interact` is de functie `run`, die ook in de prelude wordt gedefinieerd. Deze functie laat de ingetikte letters niet zien, maar alleen de uitvoer. Bovendien wacht `run` in tegenstelling tot `interact` niet totdat er een hele regel ingetikt is, maar gaat hij meteen aan het werk. Een dialoog die bij het gebruik van `run` gevoerd kan worden zou er zo uit kunnen zien:

```
?  run (map toUpper)
DIT IS TEKST           ← je ziet alleen de uitvoer
DIT OOK
```

Dankzij de lazy evaluatie kan `run` gebruikt worden om een interactief Haskell-programma te schrijven. We moeten daarvoor een functie schrijven die als parameter de ingetikte invoer krijgt, en als resultaat de gewenste uitvoer oplevert. Bijvoorbeeld:

```
groet xs  =  "wat is je naam? " ++ naam ++
             "\nhallo, " ++ naam
 where naam = takeWhile (/='\n') xs
```

Dit 'programma' kan worden uitgevoerd door het als parameter aan `run` mee te geven:

```
? run groet
wat is je naam? Jeroen
hallo, Jeroen
(89 reductions, 221 cells)
?
```

Dankzij de lazy evaluatie wordt precies op het goede moment gewacht op invoer. Merk ook op dat deze dialoog stopt nadat de gebruiker is begroet; in de functie `groet` wordt het gedeelte van de invoer na het newline-karakter niet gebruikt; alweer dankzij de lazy evaluatie gaat de interpreter dan ook niet zitten wachten tot die ingetikt wordt.

7.2.2 De werking van 'interact'

Tot nu toe hebben we `interact` beschouwd als een primitieve functie. Dat is hij echter niet: in de prelude wordt `interact` in Haskell gedefinieerd in termen van nog primitievere functies. Enkele van deze functies zijn de volgende:

```
done       ::                                                  Dialogue
readChan   :: Channel ->           FailCont -> StrCont  -> Dialogue
appendChan :: Channel -> String -> FailCont -> SuccCont -> Dialogue
```

De constante `done` is een dialoog die direct is afgelopen. Dat lijkt zinloos, maar net zoals 0 een zinvol getal is, en de lege lijst een zinvolle lijst, zal ook de dialoog `done` erg nuttig blijken te zijn.

De functies `readChan` en `appendChan` lezen uit, respectievelijk schrijven naar een gegeven `Channel`. Daarbij is er, net als in veel imperatieve talen, de keuze uit de volgende kanalen:

```
stdin, stdout, stderr  :: Channel
```

Het kanaal `stdin` is de plaats waar de invoer vandaan komt (normaliter het toetsenbord), `stdout` de plaats waar uitvoer geschreven wordt (meestal het beeldscherm) en `stderr` de plaats waar foutmeldingen geschreven worden (meestal ook het beeldscherm).

De `String`-parameter van `appendChan` is de tekst die naar het betreffende kanaal wordt geschreven. Maar wat zijn die andere twee parameters, die zowel bij `readChan` als bij `appendChan` worden gegeven?

We bekijken eerst de functie `readChan`. Als het lezen gelukt is, is de inhoud van het betreffende kanaal beschikbaar als string. Deze string kan echter niet als resultaat opgeleverd worden, want het resultaat van `readChan` moet een `Dialogue` zijn. De redding uit dit probleem vormen functies-als-parameter: we geven aan `readChan` een extra parameter mee, die aangeeft hoe van de gelezen string een dialoog gemaakt kan worden. Dit heet een 'string-continuatie': een functie die aangeeft hoe de dialoog verder moet gaan nadat er een string gelezen is.

Voor het geval het lezen niet gelukt is, wordt er ook een 'fail-continuatie' aan `readChan` meegegeven. In plaats van de gelezen string krijgt deze continuatie een beschrijving van de fout die is opgetreden als parameter. Ook de functie `appendChan` krijgt twee continuaties als parameter: een fail-continuatie voor het geval het schrijven onverhoopt niet gelukt is, en een 'succes-continuatie' als het schrijven succesvol verlopen is. De typen van de drie gebruikte continuaties zijn in de prelude als volgt gedefinieerd:

```
type StrCont   =  String  -> Dialogue
type FailCont  =  IOError -> Dialogue
type SuccCont  =             Dialogue
```

Een eenvoudig voorbeeld van het gebruik van deze functies is een dialoog die 'Hallo' op het scherm schrijft:

```
hallo ::  Dialogue
hallo  =  appendChan stdout "Hallo!" abort done
```

De functie `abort` die hierin gebruikt is, is een in de prelude gedefinieerde fail-continuatie die een gegeven foutmelding negeert; de simpelste manier om een fout-situatie op te vangen:

```
abort     ::  FailCont
abort err  =  done
```

Als tweede voorbeeld bekijken we een programma dat de standaard-invoer kopieert naar de standaard-uitvoer. Voor de string-continuatie die we aan `readChan` moeten meegeven is het vaak

blz. 28 handig om een lambda-expressie (zie paragraaf 2.3.4) te gebruiken, zodat we niet eerst een aparte functie hoeven te definiëren. Zo ook nu:

```
cat :: Dialogue
cat  =  readChan stdin abort
        (\invoer -> appendChan stdout invoer abort done)
```

In plaats van de invoer-string ongewijzigd naar het scherm te schrijven, kan deze ook eerst door een functie worden omgezet in een andere string. We zullen, als variant op de functie `cat`, een functie `transform` schrijven die dit doet. De omzet-functie wordt als parameter aan deze functie meegegeven.

We kunnen de definitie van de functie `cat` gebruiken, maar op de plaats waar die functie `invoer` meegeeft aan `appendChan`, schrijven we nu `f invoer`. De functie `f` is de parameter van `transform`; het is een functie `String->String`. De functiedefinitie wordt daarmee:

```
transform :: (String->String) -> Dialogue
transform f  =  readChan stdin abort
                (\invoer -> appendChan stdout (f invoer) abort done)
```

De functie `transform` doet hetzelfde als de functie `interact`, die in de prelude is gedefinieerd.

7.2.3 Invoer/uitvoer met files

Naast de eerder genoemde functies zijn er de volgende standaardfuncties die dialogen opleveren:

```
readFile   :: String ->           FailCont -> StrCont  -> Dialogue
writeFile  :: String -> String -> FailCont -> SuccCont -> Dialogue
appendFile :: String -> String -> FailCont -> SuccCont -> Dialogue
```

Deze functies werken in plaats van op een kanaal op een file met de gegeven naam.

Dialogen waarin veel invoer en uitvoer gedaan wordt, hebben de volgende vorm:

```
f1 ... (\xs -> f2 ... (\ys -> f3 ... (\zs -> f4 ... done)))
```

Hierin is `f`n steeds een functie zoals `readFile`. Als we de expressie over meerdere regels spreiden, is het duidelijker wat er gebeurt:

```
f1 ...
(\xs -> f2 ...
(\ys -> f3 ...
(\zs -> f4 ... done )))
```

Na elkaar worden de acties die beschreven worden door `f1`, `f2`, `f3` en `f4` uitgevoerd. Dit wordt uitgedrukt door deze functies steeds als continuatie-functie aan de eerder uit te voeren functies mee te geven.

Bij lange dialogen doet de behoefte zich voelen om van deel-dialogen te abstraheren in functies. Om deze deel-dialogen later in grotere dialogen te kunnen gebruiken, moeten ze, net als de standaardfuncties, een continuatie-parameter hebben. Het is dus een goede gewoonte om zelf functies te schrijven met een continuatie-parameter.

Een voorbeeld hiervan is een functie `toonFile`, die de inhoud van een file met een gegeven `naam` toont, en daarna doorgaat met een dialoog die wordt gegeven door de continuatie `cont`:

```
toonFile :: String -> SuccCont -> Dialogue
toonFile naam cont = readFile naam
                     (\fout -> schrijf ("can't open "++naam++"\n") cont)
                     (\inh  -> schrijf (naam++":\n"++inh)          cont)
```

De functie `schrijf` die hierin gebruikt wordt is een partiële parametrisatie van `appendChan`, om niet steeds de eerste en derde parameter van deze functie te hoeven opschrijven:

```
schrijf tekst succes  =  appendChan stdout tekst abort succes
```

De functie `toonfile` kunnen we gebruiken om na elkaar de inhoud van drie files te tonen:

```
toonFile "aap" (toonFile "noot" (toonFile "mies" done)))
```

7.2.4 De Haskell-compiler

Een Haskell-programma kan gecompileerd worden. Daarvoor is een programma beschikbaar dat een Haskell-programma vertaalt naar een programma in de taal C: de Haskell-compiler `gofc`.

Dankzij de compiler is het mogelijk om in Haskell toepassingsprogramma's te schrijven, zonder dat een gebruiker daarvan hoeft te merken dat het programma in Haskell geschreven was.

Het Haskell-programma moet, om het te kunnen compileren, een definitie bevatten van een dialoog die `main` heet, bijvoorbeeld:

```
main :: Dialogue
main = readChan stdin abort
       (\xs -> toonFile (lines xs) done)
```

Deze dialoog wordt bij uitvoeren van het gecompileerde programma met de gebruiker gevoerd.

Als de file waarin de definitie staat `toon.gs` heet, geef je vanuit het operating systeem het commando

```
gofc toon.gs
```

Het resultaat is een file `toon.c` waarin het Haskell-programma (compleet met de relevante gedeelten uit de prelude) is vertaald naar de taal C. Deze file moet vervolgens door de C-compiler worden vertaald, waarbij de file `runtime.o` moet worden meegelinkt. Om dit wat gemakkelijker te maken is er een programma `gofcc` gemaakt die dit doet:

```
gofcc toon.c
```

Dit programma genereert de executabele file `a.out`.

7.3 Ontleden

7.3.1 Ontleedfuncties

In functionele talen is het niet mogelijk om globale variabelen te gebruiken, die door verschillende functies geïnspecteerd en veranderd kunnen worden. Meer algemeen gezegd: het is niet mogelijk om functies te schrijven die een *neveneffect* hebben. Het voordeel daarvan is, dat functies alleen maar afhankelijk zijn van hun parameter, en dus niet verschillende antwoorden kunnen geven afhankelijk van de plaats waar ze van werden aangeroepen. Dat maakt ook het redeneren over programma's gemakkelijker.

Dit lijkt in sommige gevallen wel eens lastig. Een voorbeeld daarvan is interactieve input/output. Eerder zagen we hoe dit met behulp van hogere-orde functies opgelost kon worden: de lees-functie krijgt een functie mee (een zogenaamde *continuatie*), die op de gelezen input moet worden toegepast.

Een ander probleem lijkt *ontleden* te vormen. In Pascal of C zou je een aantal (wederzijds) recursieve functies schrijven (bijvoorbeeld *parsExpr* en *parsTerm*), die in een globale variabele *symbol* het eerste symbool op de invoer verwachten aan te treffen, en daarin ook weer het eerste symbool dat ze niet meer nodig hebben achterlaten. Het functieresultaat van de ontleedfuncties is de herkende constructie als boomstructuur.

Hoe moet dit nu functioneel, als er geen globale variabele is toegestaan? De oplossing is om de functies niet alleen de herkende constructie te laten opleveren, maar ook de onverwerkte input. Het resultaat van de functies is dus een twee-tupel. Op deze manier wordt alles waar de functie op werkt expliciet gemaakt in parameters en resultaat.

Het type van een ontleedfunctie zou dus als volgt kunnen zijn:

```
type Parser  =  String -> ( Boom , String )
```

Omdat hierin nog niet vermeld is wat `Boom` voor type is, kunnen we beter een polymorf type definiëren:

```
type Parser a  =  String -> ( a, String )
```

Sommige strings zouden echter op meerdere manieren ontleedbaar kunnen zijn. We maken dus het type van de ontleedfuncties zó, dat ze niet één boom-met-reststring opleveren, maar alle mogelijke manieren waarop dat kan:

```
type Parser a  =  String -> [( a, String )]
```

Als voorbeeld schrijven we een parsefunctie die de letter 'a' herkent:

```
lettera  ::  Parser Char
lettera []                  =  []
```

```
lettera (x:xs) | x=='a'     =  [ ('a', xs) ]
               | otherwise  =  []
```

Het komt goed uit dat we een *lijst* van ontledingen opleveren, zodat we de lege lijst kunnen opleveren als ontleding niet mogelijk is (omdat de invoer niet met een 'a' begint).

Deze functie is natuurlijk parametriseerbaar, zodat hij ook andere letters kan herkennen:

```
letter  ::  Char -> Parser Char
letter a []                 =  []
letter a (x:xs) | x==a      =  [ (a, xs) ]
                | otherwise  =  []
```

Een andere elementaire ontleedfunctie is een functie die een string herkent, bijvoorbeeld het woord 'begin'. We noemen deze functie `tok`, omdat hij een 'token' herkent:

```
tok  ::  String -> Parser String
tok k xs  |  k==take n xs  =  [ (k, drop n xs) ]
          |  otherwise     =  []
                      where  n = length k
```

Met behulp van deze functies zijn ontleedfuncties te construeren voor terminal symbolen uit een grammatica.

7.3.2 Parser-combinators

Interessanter zijn de ontleedfuncties voor de non-terminal symbolen. Die kun je natuurlijk met de hand schrijven, maar handiger is het om ze te *genereren* door hogere-orde functies partieel te parametriseren. Die hogere-orde functies noemen we *parser-combinators*.

De functie `seq` bijvoorbeeld, levert een ontleedfunctie die twee constructies naast elkaar herkent. De te herkennen constructies worden aangeleverd in de vorm van functie-parameters. Een lijst-
blz. 52 comprehensie (zie paragraaf 3.2.7) komt daarbij goed van pas:

```
seq                 :: Parser a -> Parser b -> Parser (a,b)
(p1 `seq` p2) xs     = [  ((v1,v2),xs2)
                       |  (v1,xs1) <- p1 xs
                       ,  (v2,xs2) <- p2 xs1
                       ]
```

De functie levert dus alle mogelijke tupels `(v1,v2)` met reststring `xs2`, waarbij `v1` het resultaat van aanroep van parse-functie `p1` is, waarvan de reststring `xs1` aan parse-functie `p2` wordt gevoed om waarde `v2` te bepalen.

Behalve 'opeenvolging' moeten we ook iets hebben om 'keuze' te representeren. Daarvoor dient de parser-combinator `orelse`:

```
orelse              :: Parser a -> Parser a -> Parser a
(p1 `orelse` p2) xs = p1 xs ++ p2 xs
```

Deze functie probeert eenvoudigweg beide parsers op de input uit, en concateneert alle gevonden ontledingen.

Naast de parser-combinators `seq` en `orelse` zijn er functies om parsers te transformeren in andere parsers. De eerste parser-transformeerder is `just`. Gegeven een parser `p` levert deze functie een parser die hetzelfde doet als `p`, maar bovendien garandeert dat de rest-string leeg is:

```
just :: Parser a -> Parser a
just p xs  =  [ (v,"")
              | (v,ys) <- p xs
              , ys==""
              ]
```

Een andere parser-transformeerfunctie is `do`. Deze functie maakt van een parser een andere parser, die ook nog een bepaalde functie toepast op de resultaat-structuur. Deze functie kun je bijvoorbeeld gebruiken om tijdens het ontleden een bepaalde waarde op te bouwen (bij het ontleden van een computerprogramma kan dat bijvoorbeeld de gegenereerde code zijn, of een lijst van alle variabelen met hun type, enz.) De functie `do` levert gegeven een parser `p` en een functie `f` een parser op die hetzelfde doet als `p`. maar ook nog `f` op het resultaat toepast:

```
do  :: Parser a -> (a->b) -> Parser b
```

```
do p f xs  =  [ (f v,ys)
              | (v,ys) <- p xs
              ]
```

Met de tot nu toe geschreven parsergenereer- en -transformeerfuncties kunnen we een ontleedfunctie schrijven die correct geneste haakjesparen ontleedt. Met behulp van `do` berekenen we als functiereultaat de nestingsdiepte (dit staat model voor het berekenen van een semantische functie naar keuze).

```
haakjes  =   (     letter '('
             `seq` haakjes
             `seq` letter ')'
             `seq` haakjes
             )
             `do`  (\(a,(b,(c,d)))-> (1+b)`max`d )
             `orelse` tok "" `do` \x -> 0
```

Om het aantal haakjes te beperken hebben we de functies als infix-operator gebruikt, met een handig gekozen prioriteit:

```
infixr 5 `seq`
infixr 3 `orelse`
infix  7 `do`
```

Een voorbeeld-sessie met de functie `haakjes`:

```
? just haakjes "()(())()"
[(2,[])]
(254 reductions, 615 cells)
? just haakjes "())"
[]
(63 reductions, 131 cells)
```

Inderdaad herkent `just haakjes` alleen correct geformeerde haakjesparen, en berekent daarbij de nesting-diepte.

7.3.3 Ontleden van expressies

We kunnen de parser-combinators nu ook gebruiken om rekenkundige expressies te ontleden. Daarbij gaan we uit van de volgende grammatica voor expressies:

expressie	::=	*term* `+` *expressie*
	|	*term* `-` *expressie*
	|	*term*
term	::=	*factor* `*` *term*
	|	*factor* `/` *term*
	|	*factor*
factor	::=	*constante*
	|	*variabele*
	|	`(` *expressie* `)`

Door gebruik van deze grammatica krijgen `*` en `/` automatisch een hogere prioriteit dan `+` en `-`.

Voor elk begrip uit deze grammatica (expressie, term, factor, constante en variabele) maken we een overeenkomstige ontleedfunctie. Het schrijven van deze functies is erg gemakkelijk: de vertical strepen in de grammatica worden aanroepen van `orelse`, en de onderdelen van elk alternatief worden samengevoegd met `seq`. Het enige wat we dan nog moeten doen is met behulp van de `do`-functie de gewenste boom op te bouwen uit de onderdelen.

```
expr :: Parser Expr
expr = term
       `orelse` term
                `seq` letter '+'
                `seq` expr
                `do` (\(e1,(c,e2)) ->  e1 :+: e2)
       `orelse` term
                `seq` letter '-'
```

```
                `seq` expr
                `do` (\(e1,(c,e2)) ->  e1 :-: e2)
term :: Parser Expr
term = factor
       `orelse` factor
                `seq` letter '*'
                `seq`    term
                `do`  (\(e1,(c,e2)) ->  e1 :*: e2)
       `orelse` factor
                `seq` letter '/'
                `seq`    term
                `do`  (\(e1,(c,e2)) ->  e1 :/: e2)
factor :: Parser Expr
factor   =   constant `do` Con
             `orelse` variable `do` Var
             `orelse` ( letter '(' `seq` expr `seq` letter ')' )
                      `do` (\(ho,(e,hs)) -> e)
```

Voor de ontleedfunctie `variabele` zijn geen parser-combinators nodig. Deze functie is eenvoudig met de hand te schrijven:

```
variabele :: Parser String
variabele xs  = [ (takeWhile isAlpha xs, dropWhile isAlpha xs) ]
```

Het schrijven van de ontleedfunctie `constant` wordt aan de lezer overgelaten.

Gebruikmakend van deze functies kunnen we nu expressies ontleden:

```
? just expr "2*x+5"
[ ( (Con 2 :*: Var "x") :+: Con 5, [] ) ]
```

Er is blijkbaar één oplossing, waarbij de geproduceerde boom `(Con 2 :*: Var "x") :+: Con 5` is, en de rest-string leeg. (Natuurlijk is de rest-string leeg, want dat wordt gegarandeerd door
blz. 135
`just`). Het is eenvoudig om nu de in paragraaf 7.1.4 beloofde functie `ontleed` te schrijven:

```
ontleed :: String -> Expr
ontleed  =  fst . head . (just expr)
```

Deze functie eist (met `just`) dat er geen extra symbolen zijn, pakt met `head` de eerste oplossing (die er hopelijk is, anders volgt een foutmelding) en selecteert met `fst` alleen het ontleed-resultaat (de rest-string is niet meer interessant).

7.3.4 Meer parser-combinators

In veel toepassingen komt het voor dat een constructie willekeurig vaak mag worden herhaald. Een parser-genereerfunctie die daarvoor gebruikt kan worden is `many`:

```
many    :: Parser a  -> Parser [a]
many p  =  q
           where q = ( p `seq` q) `do` makelist )
                     `orelse` (okay [])
```

Hierin zijn de volgende functies gebruikt:

```
makelist (x,xs) = x:xs
okay :: a -> Parser a
okay v xs  =  [ (v,xs) ]
```

De functie `okay` is een ontleedfunctie die altijd slaagt met de te specificeren waarde `v` als resultaat.

Voor elke veel in grammatica's voorkomende constructie kunnen we een parser-generator maken. Bijvoorbeeld voor een constructie die herhaald mag worden, maar minstens eenmaal moet voorkomen:

```
many1    :: Parser a -> Parser [a]
many1 p  = p `seq` many p `do` makeList
```

Of voor een lijst van constructies die door `p` ontleed worden, gescheiden door constructies die door `s` ontleed worden:

```
listOf      :: Parser a -> Parser b -> Parser [a]
listOf p s  = p `seq` many (s `seq` p) `do` f
```

```
    `orelse` okay []
    where f (x,xs) = x:(map snd xs)
```

als je eenmaal de juiste standaardfuncties hebt geschreven, is het heel gemakkelijk om ingewikkelde parsers te schrijven. De belangrijkste truc die we hebben gebruikt is *generalisatie*:

- een *polymorf* type `Parser`
- functies die niet één oplossing leveren, maar de *lijst van alle mogelijke oplossingen*
- *hogere-orde functies*, die gegeven elementaire parsers meer ingewikkelde parsers maken.

Er is helemaal geen aparte datastructuur nodig geweest om grammatica's in op te slaan: alles wordt geregeld met behulp van functies.

Opgaven

7.1 Breid de data-declaratie van `Expr` uit zodat er vier nieuwe expressievormen onststaan: de sinus van een expressie, de cosinus van een expressie, de exponentiële functie (e tot de macht een expressie), en de natuurlijke logaritme van een expressie. Breid vervolgens de definitie van `afg` uit voor deze vier nieuwe expressievormen. Verwerk daarin de 'kettingregel' voor het differentiëren: $(f \circ g)' = (f' \circ g) * g'$.

7.2 Schrijf een functie `norep` die gegeven een `Stat` alle `Repeat`-statements daaruit verwijdert, door ze te vervangen door een equivalente `While`-statements.

7.3 Schrijf een dialoog `fileDial` die één regel invoer uit het kanaal `stdin` leest, de gelezen string als filenaam gebruikt en de inhoud van deze file leest, en deze inhoud op het scherm toont.

7.4 Schrijf een functie

```
toonFiles :: [String] -> SuccCont -> Dialogue
```

die de inhoud van een aantal files, waarvan een lijst namen gegeven is, op het scherm toont, en daarna doorgaat met de gegeven continuatie.

7.5 a. Schrijf een Haskell-functie `fileSize` met als type:

```
fileSize :: String -> SuccCont -> Dialogue
```

die een gegeven string als filenaam gebruikt, de inhoud van de file leest, en op het scherm aangeeft hoeveel regels en hoeveel karakters de file bevat. Daarna moet de dialoog doorgaan met de gegeven succes-continuatie. Hint: om de uitvoer-string op te bouwen kun je de functie `show` gebruiken om van een integer een string te maken.

b. Verander nu de functie zodat de melding niet op het scherm geschreven wordt, maar als extra regel aan de file wordt toegevoegd.

7.6 Schrijf de functie `constant` die in paragraaf 7.3.3 wordt gebruikt. blz. 141

7.7 Schrijf een ontleedfunctie voor expressies in talen met negen in plaats van twee prioriteitsnivo's, zoals C (en Haskell zelf). Probeer de regelmaat in de benodigde functies te 'vangen' in een hogere-orde functie.

7.8 Maak een parser-generator die gegeven een lijst parsers één parser voor de compositie daarvan maakt.

Bijlage A

Lisp voor Haskell-kenners

A.1 Expressies

A.1.1 Functie-aanroep

In Lisp kun je, net als in Haskell, een functie aanroepen door de naam van de functie en de parameters naast elkaar te schrijven. In Lisp moet echter het geheel nog tussen haakjes geschreven worden.

Het resultaat van een functie-aanroep kan gebruikt worden als parameter van een andere functie. In Haskell moet de deel-aanroep dan tussen haakjes gezet worden om aan te geven dat het één parameter betreft; in Lisp staat de deel-aanroep tussen haakjes omdat elke aanroep tussen haakjes moet staan.

Al met al lijken Lisp-expressies sterk op Haskell-expressies. Het enige verschil is dat er in Lisp ook om de buitenste aanroep in een expressie haakjes moeten staan.

Haskell	Lisp
`sqrt 2.0` `sqrt (exp 1.0)`	`(sqrt 2.0)` `(sqrt (exp 1.0))`

A.1.2 Operatoren

In Haskell zijn er functies en operatoren. De naam van een functie begint met een letter, de naam van een operator bestaat uit symbolen. Functie-namen worden *voor* de parameters geschreven, operatoren er *tussen* (operatoren hebben altijd twee parameters).

In Lisp worden functies en operatoren altijd *voor* de parameter geschreven. Afgezien van de schrijfwijze (letters, resp. symbolen) is er dus geen verschil tussen functies en operatoren.

Prioriteit en associatievolgorde zijn eigenschappen van infix-operatoren. Dit is in Lisp dus geen issue; de berekeningsvolgorde wordt altijd expliciet aangegeven.

Functies kunnen in Lisp een variabel aantal parameters hebben. De functie `+` is daarvan een voorbeeld.

Haskell	Lisp
`f 1 2`	`(f 1 2)`
`1 + 2`	`(+ 1 2)`
`x <= y`	`(<= x y)`
`x+y*z`	`(+ x (* y z))`
`x*(y+z)`	`(* x (+ y z))`
`v+w+x+y+z`	`(+ v w x y z)`

A.1.3 Lijsten

In Haskell kun je lijsten opschrijven door de elementen, gescheiden door komma's, tussen vierkante haken te zetten. In Lisp wordt voor lijsten dezelfde notatie gebruikt als voor functie-aanroep: de elementen staan naast elkaar, gescheiden door spaties, en het geheel staat tussen ronde haken.

Om te voorkomen dat het eerste element van de lijst wordt opgevat als functie, die moet worden uitgerekend, wordt het geheel voorafgegaan door het symbool `'` (spreek uit: quote).

De lege lijst wordt in Lisp aangduid met de naam `nil`, of desgewenst met een leeg, rond haakjespaar.

Net als Haskell kent Lisp strings, die mogen worden gebruikt als lijst van characters.

Haskell	Lisp
`[1,2,3]`	`'(1 2 3)`
`[]`	`nil`
`[]`	`()`
`length [1,2,3]`	`(length '(1 2 3))`
`length "aap"`	`(length "aap")`

A.2 Functies op lijsten

Zowel in Haskell als in Lisp kan een lijst ook worden opgebouwd met de op-kop-van functie. In Haskell heet die `:`, in Lisp is dat `cons`. Lijsten kunnen geconcateneerd worden met `++` in Haskell, en met `append` in Lisp.

Haskell	Lisp
`1:2:3:4:[]`	`(cons 1 (cons 2 (cons 3 (cons 4 ()))))`
`[1,2,3] ++ [4,5,6]`	`(append '(1 2 3) '(4 5 6))`

A.3 Functiedefinitie

A.3.1 Simpele definitie

Ingrediënten van een functiedefinitie zijn de naam van de functie, declaratie van parameters, en een body (expressie waarin de parameters gebruikt mogen worden).

In Haskell is er een speciale syntax voor functiedefinities, te herkennen aan het `=` symbool in het midden. Links daarvan staan de naam en namen voor de parameters, rechts staat de body.

In Lisp heeft een functiedefinitie dezelfde vorm als een functie-aanroep: ronde haakjes met daartussen een aantal onderdelen, gescheiden door spaties. De onderdelen zijn achtereenvolgens: het speciale symbool `defun`, de naam van de functie, een lijst met parameternamen, en de body van de functie.

Haskell	Lisp

`kwadraat x = x*x`
`boven n k = fac n`
`/ (fac k * fac (n-k))`
	`(defun kwadraat (x) (* x x))`
	`(defun boven (n k)`
	`(/ (fac n)`
	`(* (fac k) (fac (- n k)))))`

A.3.2 Definitie met gevalsonderscheid

Haskell heeft een aparte syntax voor een functiedefinitie met gevalsonderscheid: voor het =-teken in een functiedefinitie mag een `|` staan gevolgd door een voorwaarde. In een functiedefinitie kunnen meerdere voorwaarden en bijbehorende functie-bodies staan.

In Lisp worden de gevallen *in* de body van de functie uitgesplitst, door aanroep van de speciale functie `cond`. Deze functie heeft een aantal paren voorwaarde–body als parameter.

Haskell	Lisp

`signum x | x<0 = -1`
`| x==0 = 0`
`| x>0 = 1`
	`(defun signum (x)`
	`(cond ( (< x 0) -1 )`
	`( (== x 0) 0 )`
	`( (> x 0) 1 )))`

A.3.3 Definitie met patronen

Een andere manier om gevallen te onderscheiden in Haskell is het gebruik van *patronen*. Een formele parameter is dan niet een simpele naam, maar een aanduiding van de opbouw van de parameter. Veel gebruikt worden de patronen `[]` en `(x:xs)` voor respectievelijk lege en niet-lege lijsten.

In de body van een functiedefinitie met patronen zijn de onderdelen van de parameter, in het voorbeeld `x` en `xs`, los aanspreekbaar.

In Lisp kunnen formele parameters geen patronen zijn. In plaats daarvan kun je de gevallen in de body van de functie uitsplitsen met `cond`. Je kunt bijvoorbeeld de functie `null` gebruiken om te testen of een lijst leeg is. In de body dien je expliciet de onderdelen van de parameter te selecteren. De kop en staart van een lijst kun je bijvoorbeeld pakken met `first` en `rest`.

Haskell	Lisp

`lengte [] = 0`
`lengte (x:xs) = 1 + lengte xs`
 	
`som [] = 0`
`som (x:xs) = x + som xs`
	`(defun lengte (xs)`
	`  (cond ((null xs) 0)`
	`  (t (+ 1 (lengte (rest xs))))))`
	`(defun som (xs)`
	`  (cond ((null xs) 0)`
	`  (t (+ (first xs)`
	`  (som (rest xs))))))`

In de Lisp-versie is hier de constante `t` gebruikt als conditie die altijd waar is (vergelijk de constante `otherwise` in Haskell).

In beide talen hadden deze functies ook gedefinieerd kunnen worden met gebruik van `foldr`, respectievelijk `reduce`. We hebben dat hier niet gedaan om de structuur van een definitie met patronen te demonstreren.

A.3.4 Evaluatie

In Haskell worden alle functies lazy geëvalueerd, dat wil zeggen dat de parameters pas worden uitgerekend als ze echt nodig zijn. Van lijst-parameters worden alleen díe elementen uitgerekend die voor het resultaat van belang zijn. Desgewenst kun je een niet-lazy versie van een functie maken door de functie `strict` er op toe te passen.

In Lisp worden alleen een paar standaardfuncties (zoals `cond`, `or` en `and`) lazy uitgerekend. Zelfgemaakte functies worden altijd strict uitgerekend. Daardoor is het onmogelijk om bijvoorbeeld oneindige lijsten te gebruiken, zelfs als het eindresultaat eindig is.

Haskell	Lisp

`repeat x = x : repeat x`
 	
`> repeat 3`
`[3,3,3,3,3,3,3,3,3,3,3,3,3,3,3...`
`> head (repeat 3)`
`3`
	`(defun repeat (x) (cons x (repeat x)))`
	
	`> (repeat 3)`
	`Error: Invocation history stack overflow.`
	`> (first (repeat 3))`
	`Error: Invocation history stack overflow.`

A.4 Typering

A.4.1 Statisch versus dynamisch

De types in een Haskell-programma worden gecontroleerd tijdens het compileren. Een eenmaal ingelezen programma is gegarandeerd type-correct. De types in een Lisp-programma worden gecontroleerd tijdens het uitvoeren. Operatoren controleren alvorens het resultaat te berekenen of het type correct is. Er kunnen dus typerings-fouten in het programma zitten die pas na een tijdlang gebruik aan het licht komen.

Haskell	Lisp

`> 1<='a'`
`Error: Type error in application`
 	
`> 2<=3 || 1<='a'`
`Error: Type error in application`
	`> (<= 1 'a)`
	`Error: A is not of type FLOAT`
	
	`> (or (<= 2 3) (<= 1 'a))`
	`T`

A.4.2 Types van lijsten

In Haskell moeten alle elementen van een lijst van hetzelfde type zijn. In Lisp hoeft dat niet. Een apart tupel-type voor een vast aantal elementen van verschillend type is in Lisp dus ook niet nodig.

In Haskell zijn lijsten van lijsten mogelijk, mits dan ook alle elementen een lijst zijn. In Lisp kunnen sommige elementen van een lijst atomiar zijn, en andere elementen een lijst.

Haskell	Lisp

`[1,2,3]`
`(1,"aap")`
`[ [1,2,3], [4,5,6] ]`
`[ 5, [4,5,6] ]  -- typeringsfout`
	`'(1 2 3)`
	`'(1 "aap")`
	`'( '(1 2 3) '(4 5 6) )`
	`'( 5 '(4 5 6) )`

A.4.3 Overloading

In Haskell kan een functie gedefinieerd worden op meerdere types. Zo'n functie moet dan een member zijn van een class, waarna de verschillende types instance worden gemaakt van die class. door de compiler wordt de juiste versie van de fuctie gekozen aan de hand van het type.

In Lisp kun je run-time het type van de parameter opvragen, en op die manier een functie op meerdere types toepasbaar maken. De juiste versie wordt met gevalsonderscheid expliciet gekozen.

Haskell	Lisp

`class Seq a where`
`  nummer :: a -> Int`
`instance Seq Int where`
`  nummer n = n`
`instance Seq Char where`
`  nummer c = ord c`
`instance Seq Bool where`
`  nummer False = 0`
`  nummer True  = 1`
	`(defun nummer (x)`
	`  (cond ((integerp x) x)`
	`  ((characterp x) (ord x))`
	`  (x 1)`
	`  (t 0)))`

In Haskell kun je reeds bestaande overloaded operatoren (zoals `==`) nog verder overladen, door een nieuw type ook instance te maken van de betreffende klasse (`Eq` in het geval van `==`). In Lisp is dat onmogelijk.

A.4.4 Polymorfie

Haskell is getypeerd volgens het Hindley-Milner typesysteem, waarin bijvoorbeeld onderscheid wordt gemaakt tussen lijsten-van-integer en lijsten-van-lijsten-van-boolean, en tussen functie-van-int-naar-char en functie-van-int-naar-bool. Door middel van type-variabelen is het mogelijk om aan te geven dat een functie bijvoorbeeld het type functie-van-willekeurig-type-naar-datzelfde-type heeft.

In Lisp zijn er types voor getallen (met subtypes voor integer, float, rational etc.), lijsten, en functies. Er kan geen onderscheid gemaakt worden tussen verschillende typen van functies.

A.5 Hogere-ordefuncties

A.5.1 Map / Mapcar

Zowel in Haskell als in Lisp is het mogelijk om een functie mee te geven als parameter aan een andere functie. Een bekend voorbeeld is de functie `map`, die in Lisp bekend staat onder de naam `mapcar`. In Lisp moet expliciet worden aangegeven dat de functie die als parameter wordt meegegeven moet worden opgezocht in de verzameling van alle eerder gegeven functie-definities. Daarom staat er in Lisp `#'f` in plaats van `f` als parameter van `map`.

In Lisp heeft de functie `map` een variabel aantal parameters. In plaats van op één lijst kan hij ook op twee lijsten worden toegepast. De functie-parameter moet dan een functie met twee parameters zijn; het effect is hetzelfde als dat van `zipWith` in Haskell. Ook generalisaties naar meer lijsten zijn mogelijk.

Haskell	Lisp
`map f [1,2,3]` `zipWith (+) [1,2,3] [4,5,6]`	`(mapcar #'f '(1 2 3))` `(mapcar #'+ '(1 2 3) '(4 5 6))`

A.5.2 Foldr / Reduce

De functie overeenkomstig met Haskell's `foldr` heet in Lisp `reduce`. Een neutrale waarde hoeft om mij onduidelijke redenen niet te worden meegegeven; toch kan `reduce` op lege lijsten worden toegepast.

Haskell	Lisp
`foldr (+) 0 [1,2,3]`	`(reduce #'+ '(1 2 3))`

A.5.3 Currying / Lambda-notatie

In Lisp zijn functies niet gecurried, en ze kunnen dus ook niet partieel geparametriseerd worden. Het effect kan wel worden bereikt met behulp van de lambda-notatie. Hiermee worden nieuwe, naamloze functies gemaakt, meestal met het doel om ze aan een andere functies mee te geven. De lambda-notatie bestaat ook in Haskell, maar is minder vaak nodig omdat in plaats daarvan partiële parametrisatie kan worden gebruikt.

Haskell	Lisp

`\x -> x*x`
`map (\x->x*x) [1,2,3]`
`map (+1) [1,2,3]`
`until (>9) (*2) 1`
	`(lambda (x) (* x x))`
	`(mapcar #'(lambda (x) (* x x)) '(1 2 3))`
	`(mapcar #'(lambda (x) (+ x 1)) '(1 2 3))`
	`(until #'(lambda (x) (> x 9))`
	`  #'(lambda (x) (* x 2)) 1)`

A.6 Filosofie

A.6.1 Haskell: referentieel transparant

Het resultaat van een functie-aanroep in Haskell is geheel bepaald door de waarden van de parameters. Het functieresultaat is de enige manier waarop een functie zijn omgeving kan beïnvloeden. Haskell-functies hebben dus geen *neveneffecten*; er zijn geen globale variabelen die veranderd kunnen worden.

Deze eigenschap maakt lazy evaluatie mogelijk: zonder dat het voor het eindresultaat uitmaakt, kan een berekening later, of helemaal niet worden uitgevoerd. Als deze berekeningen neveneffecten gehad zouden hebben, dan kan de uitreken-volgorde het eindresultaat wèl beïnvloeden.

De afwezigheid van neveneffecten maakt het mogelijk om over Haskell-programma's te redeneren als waren het wiskundige formules; zo zijn bijvoorbeeld in alle omstandigheden de aanroepen

`map f (xs++ys)` en `map f xs ++ map f ys` gelijk. Als de functie `f` neveneffecten op globale variabelen zou hebben, kun je niet zonder meer op de geldigheid van deze wet vertrouwen.

In Lisp is het wel mogelijk een waarde toe te kennen aan een globale variabele: door aanroep van de functie `setf` wordt een waarde aan een variabele toegekend.

A.6.2 Lisp: meta-circulair

De syntax van Lisp-programma's (functie-aanroepen zoals `(f 1 2)` lijkt sterk op die van Lisp-data (lijsten zoals `'(1 2 3)`). De Lisp-interpreter is als het ware een functie die een lijst, voorstellende een functie-aanroep, interpreteert. Dit maakt het eenvoudig om Lisp-programma's te schrijven die andere programma's manipuleren. Het is zelfs mogelijk om een programma door een programma te laten schrijven, en dat vervolgens uit te voeren.

Dat laatste effect zie je in Haskell meestal in de vorm van hogere-ordefuncties. Functies kunnen functie opleveren, die pas later op werkelijke data worden toegepast.

A.7 Haskell en Prolog

A.7.1 Lijsten

De notatie van lijsten is enigzins verwarrend als je afwisselend in Haskell en Prolog programmeert. Een opsomming gebeurt op dezelfde manier: komma's tussen de elementen en vierkante haken om het geheel. Singletons en de lege lijst worden op dezelfde manier gerepresenteerd. Maar het patroon waarbij een lijst in een kop en een staart wordt verdeeld is verschillend: in Haskell wordt de operator `:` gebruikt, in Prolog wordt het symbool `|` tussen kop en staart gezet, maar *bovendien* vierkante haken om het geheel. In Haskell moet je die extra vierkante haken vooral *niet* neerzetten: daarmee zou je een lijstnivo toevoegen!

Haskell	Prolog

`[1,2,3]`
`[5]`
`[]`
`(x:xs)`
`[x:xs]`
	`[1,2,3]`
	`[5]`
	`[]`
	`[x|xs]`
	`[[x|xs]]`

A.7.2 Functies en relaties

Het belangrijkste verschil tussen Haskell en Prolog is dat je in Haskell *functies* specificeert, en in Prolog *relaties*. Bij een functie geef je aan wat bij bepaalde invoer de uitvoer is; bij een relatie geef je aan dat invoer en uitvoer met elkaar een relationeel verband hebben. Daarom hebben Prolog-relaties altijd een parameter meer dan de overeenkomstige Haskell-functies.

Daar waar je in Haskell een aanroep doet van een andere functie, krijgt de Prolog-relatie *voorwaarden*. Je zult merken dat je een extra naam nodig hebt, om het tussenresultaat te kunnen benoemen.

Haskell	Prolog
`d x = 3` `f x = h (g x)`	`d(x,3).` `f(x,z) :- g(x,y), h(y,z) .`

A.7.3 Patronen

Gelukkig kent Prolog wel patronen, waarmee bijvoorbeeld de gevallen 'lege lijst' en 'niet-lege lijst' onderscheiden kunnen worden. Zo kan een relatie worden gedefinieerd waarmee wordt aangeduid dat twee lijsten aan elkaar geplakt een derde lijst opleveren. In Prolog heet deze relatie `append`; de overeenkomstige functie in Haskell heet `++`.

Haskell	Prolog

`[]     ++ ys  =  ys`
`(x:xs) ++ ys  =  x:(xs++ys)`
	`append([],ys,ys).`
	`append([x|xs],ys,[x|zs])`
	`  :- append(xs,ys,zs) .`

Merk op dat het resultaat van de recursieve aanroep in de Prolog-versie een aparte naam moet krijgen. Bij wat meer ingewikkelde functies/relaties kan dat tamelijk onoverzichtelijk worden:

Haskell	Prolog

`ins x [] = [x]`
`ins x (y:ys)`
`  | x<y  = x:y:ys`
`  | x>=y = y:ins x ys`
	`ins(x,[],[x]).`
	`ins(x,[y|ys],[x|y|ys])`
	`  :- x<y .`
	`ins(x,[y|ys],[y|zs])`
	`  :- x>=y, ins(x,ys,zs) .`

A.7.4 Richtingloze definities

Een voordeel van de Prolog-definities is dat ze ook omgekeerd gebruikt kunnen worden. De append-relatie kan behalve voor het samenvoegen van twee lijsten ook worden gebruikt om een lijst in alle mogelijke delen te splitsen. In Haskell zou hiervoor een andere functie nodig zijn.

Haskell	Prolog

`> [1,2]++[3,4]`
`[1,2,3,4]`
 	
`splits [] = [([],[])]`
`splits (x:xs)`
`  = ([[],(x:xs)) :`
`  map f (splits xs)`
`  where f (a,b) = (x:a,b)`
 	
`> splits [1,2,3]`
`[([],[1,2,3])`
`,([1],[2,3])`
`,([1,2],[3])`
`,([1,2,3],[])`
`]`
	`> append([1,2],[3,4],Z).`
	`Z = [1,2,3,4]`
	
	`> append(X,Y,[1,2,3]).`
	`X=[] Y=[1,2,3]`
	`X=[1] Y=[2,3]`
	`X=[1,2] Y=[3]`
	`X=[1,2,3] Y=[]`

Helaas is dit gebruik van relaties in twee richtingen alleen mogelijk als bij de definitie van relaties geen gebruik is gemaakt van ingebouwde relaties die niet in twee richtingen gebruikt mogen worden, zoals `plus`.

A.7.5 Constructorfuncties

In Prolog kunnen op de parameterplaatsen van predicaten ook weer functies aangeroepen worden. Dit stelt dan echter geen functie-aanroep voor van een elders gedefinieerde functie – dat dient immers te gebeuren met het eerder beschreven mechanisme.

Functies in Prolog komen overeen met wat in Haskell constructorfuncties genoemd zou worden.

Haskell	Prolog

`data Boom = Tip Int`
`  | Tak Boom Boom`
 	
`size (Tip x)  = 1`
`size (Tak p q) = size p + size q`
	`size(tip(x),1).`
	`size(tak(p,q),a+b)`
	`  :- size(p,a), size(q,b).`

Bijlage B

Practicumopgaven

B.0 Oefenopgaven

1. Log in op een Sun-computer met behulp van de uitgedeelde login-naam en password. Open (als dat standaard niet gebeurt) een terminal window: druk op de rechter-muisknop, en kies onder 'Programs' het item 'Terminal'.

   Start nu de Haskell-interpreter op door achter de unix-prompt `hugs` in te typen. Probeer (als dit gelukt is) het resultaat van enkele expressies op te vragen. Voorbeelden:
   - `[1..10]`
   - `sum [1..10]`
   - `reverse [1..10]`

   Probeer eventueel ook andere expressies zelf samen te stellen.

2. We gaan nu een Haskell-script dat functies bevat maken.
   - Start de Haskell-interpreter door `hugs` in te typen.
   - Start de editor met `:e nieuw.hs`
   - Type een Haskell-script in m.b.v. de editor en save het
   - Laad het script in Haskell m.b.v. `:load nieuw.gs`

   Probeer de functie `revsort` (van het werkcollege) die een lijst in omgekeerde volgorde sorteert in een file te bewaren en in te lezen in Haskell. Vraag het type op van de gedefinieerde functie.

   Vraag het type op van "*jouw naam*". Vraag het type op van `length`. Type nu in `length` "*jouw naam*". Hoe zou `String` gedefinieerd zijn?

3. Zoek de definitie van `reverse` op in de file `Prelude.hs`.

4. Nog een extra werkcollegeopgave was:
   - Definieer een functie `verhoog1` zodat `verhoog1 xs` alle elementen van lijst xs met 1 verhoogt (bijv. `verhoog1 [1,9,9,6] == [2,10,10,7]`).
   - Definieer een functie `verhoog` zodat `verhoog n xs` alle elementen van lijst xs met n verhoogt.

   Maak een Haskell-script met deze functies, en controleer hun werking.

5. Maak een functie, `deelDoorTweeLijst :: Float -> [Float]`, Die gegeven een float een lijst van Floats oplevert, die als volgt is gedefinieerd.
   - Het eerste element van de lijst is de eerste Float
   - Het volgende element van de lijst is het laatste element van de lijst gedeeld door twee

   Voorbeeld:

```
? deelDoorTweeLijst 16.0
[16.0, 8.0, 4.0, 2.0, 1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.0{Interrupted!}

(43 reductions, 185 cells)
? take 4 (deelDoorTweeLijst 16.0)
[16.0, 8.0, 4.0, 2.0]
(21 reductions, 79 cells)
```

6. Tik het programma voor numeriek differentiëren in uit paragraaf 2.4.2. Bereken de waarde van f' in de punten 0.0, 1.0, 2.0 tot en met 15.0 (laat deze waarden in een lijst zetten) waarbij f de derdemachtsfunctie is. Zijn het de waarden die je verwacht?

   Tik het programma voor het uitrekenen van de wortel uit paragraaf 2.4.3. Reken een paar wortels uit.

   Leuker is het om niet alleen het eindresultaat (de uitgerekende wortel) te zien, maar ook de tussenresultaten (zoals in de berekening van $\sqrt{3}$ in paragraaf 2.4.3). Maak daartoe een variant op de functie `wortel`. Noem deze functie `wortels`, en laat hem een *lijst* van floats opleveren. Maak daarvoor een eigen versie van `until`, die tussenresultaten oplevert.

B.1 Complexe getallen

Probleemstelling

In deze opgave moeten twee types worden gedefinieerd waarmee *complexe getallen* kunnen worden gerepresenteerd. Vervolgens moeten een aantal functies worden gedefinieerd, die op waarden van deze types werken. Alle dingen die gedefinieerd moeten worden zijn in onderstaande tekst aangegeven met een •.

De types

Een complex getal is een getal van de vorm $a+bi$, waarbij a en b reële getallen zijn, en i een waarde is met de eigenschap $i^2 = -1$. Deze representatie heet de *cartesische* representatie: een complex getal is als het ware een punt (a, b) in het platte vlak.

- Definieer een type `Cart` die een complex getal representeert als tupel van twee reële getallen. Maak er een 'beschermd type' van zoals beschreven op blz. 90.

Elk punt (a, b) in het vlak wordt éénduidig beschreven door de afstand r tot de oorsprong, en de hoek φ tussen de x-as en de lijn van de oorsprong tot het punt (φ in radialen, $0 \leq \varphi \leq 2\pi$). Een complex getal kan dus ook worden beschreven door de reële getallen r en φ. Dit wordt de *poolcoördinaten*-representatie genoemd.

- Definieer een type `Pool` waarmee complexe getallen in poolcoördinaten worden gerepresenteerd. (Deze representatie is hetzelfde als `Cart`, dus het is goed dat we beschermde types gebruiken).

Naamgeving

Het is belangrijk dat je alle functies precies dezelfde naam geeft zoals hieronder beschreven. De programma's zullen namelijk automatisch getest worden door een programma dat ervan uitgaat dat de functies inderdaad zo heten. Daarnaast mag je zonodig zelf extra functies definiëren. De namen van de functies die gedefinieerd moeten worden zijn systematisch opgebouwd.

Conversiefuncties

- Schrijf vier conversiefuncties waarmee integers en reële getallen naar beide representaties van complexe getallen kunnen worden omgezet. Schrijf ook twee functies waarmee beide representaties in elkaar worden omgezet:

```
float'cart   ::  Float -> Cart
float'pool   ::  Float -> Pool
int'cart     ::  Int   -> Cart
int'pool     ::  Int   -> Pool
cart'pool    ::  Cart  -> Pool
pool'cart    ::  Pool  -> Cart
```

Basisfuncties

In de Cartesische representatie kunnen complexe getallen worden opgeteld door x- en y-'coördinaat' paarsgewijs op te tellen. Ze kunnen worden vermenigvuldigd door in $(a + bi) * (c + di)$ de haakjes uit te werken, en gebruik te maken van $i^2 = -1$. Voor de deling van cartesische complexe getallen, zie opgave 3.7.

- Schrijf de vier rekenkundige functies voor `Cart`:

```
cart'plus   ::  Cart -> Cart -> Cart
cart'min    ::  Cart -> Cart -> Cart
cart'maal   ::  Cart -> Cart -> Cart
cart'deel   ::  Cart -> Cart -> Cart
```

In de pool-representatie kunnen complexe getallen worden vermenigvuldigd door hun afstanden tot de oorsprong te vermenigvuldigen (als reële getallen), en de hoeken met de x-as op te tellen. Voor delen van complexe getallen: delen van de afstanden en aftrekken van de hoeken. Optellen en aftrekken van `Pool`-complexe getallen is niet eenvoudig; het makkelijkste is om te converteren naar `Cart`, dan de operatie uit te voeren en daarna weer terug te transformeren.

- Schrijf de vier rekenkundige functies voor `Pool`:

```
pool'plus   ::  Pool -> Pool -> Pool
pool'min    ::  Pool -> Pool -> Pool
pool'maal   ::  Pool -> Pool -> Pool
pool'deel   ::  Pool -> Pool -> Pool
```

Machtsverheffen met een natuurlijk getal als macht kan worden uitgevoerd door herhaald te vermenigvuldigen.

- Schrijf de machtsverhef-functie voor beide representaties:

```
cart'macht  ::  Cart -> Int -> Cart
pool'macht  ::  Pool -> Int -> Pool
```

Andere functies

De n-de-machts-wortel is het gemakkelijkst uit te rekenen in de pool-representatie: van de afstand tot de oorsprong wordt de (reële) $\sqrt[n]{\ }$ getrokken, en de hoek φ wordt door n gedeeld. (Ga na dat als je de uitkomst weer tot de n-de macht verheft, je inderdaad het originele getal terugkrijgt). Maar er zijn meer oplossingen: alle getallen met hoek $\frac{\varphi}{n} + \frac{k*2\pi}{n}$ voor $k \in \{0..n-1\}$ voldoen. Er zijn dus altijd n oplossingen van de complexe n-de-machts-wortel.

Voor de functie in de Cartesische representatie kun je heen en weer transformeren naar de pool-representatie.

- Schrijf twee functies die de lijst van alle mogelijke n-de-machts wortels van een complex getal oplevert. Schrijf ook twee functies die de tweedemachts-wortel berekenen.

```
cart'root   ::  Int -> Cart -> [Cart]
pool'root   ::  Int -> Pool -> [Pool]
cart'sqrt   ::         Cart -> [Cart]
pool'sqrt   ::         Pool -> [Pool]
```

De exponentiële functie kan berekend worden met de volgende machtreeks:

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Gebruik makend daarvan kun je de sinus en cosinus definiëren voor complexe getallen, volgens de formules:

$$\begin{aligned} \cos x &= (e^{ix} + e^{-ix})/2 \\ \sin x &= (e^{ix} - e^{-ix})/2i \end{aligned}$$

- Schrijf de functies

```
cart'exp    ::  Cart -> Cart
pool'exp    ::  Pool -> Pool
```

door tien termen van de machtreeks te sommeren. Schrijf vervolgens de functies

```
cart'sin  ::  Cart -> Cart
pool'sin  ::  Pool -> Pool
cart'cos  ::  Cart -> Cart
pool'cos  ::  Pool -> Pool
```

Een toepassing

- Schrijf tenslotte twee functies die gegeven drie complexe getallen a, b en c de lijst met de twee complexe oplossingen van $ax^2 + bx + c = 0$ berekent (met de abc-formule):

```
cart'opl2  ::  Cart -> Cart -> Cart -> [Cart]
pool'opl2  ::  Pool -> Pool -> Pool -> [Pool]
```

B.2 Teksten

Een waarde van het type `String` kun je goed gebruiken om een rijtje letters in te zetten. In zo'n string komt echter het twee-dimensionale karakter van een tekst niet goed tot uiting. Bij het manipuleren van teksten zijn de twee dimensies vaak wel van belang. Bij het lay-outen van een krantepagina heb je bijvoorbeeld operaties nodig zoals 'zet deze tekst naast die tekst, en dat in zijn geheel boven deze foto'.

Deze opgave bestaat uit het schrijven van een aantal functies die op een type `Tekst` werken. Het tweede gedeelte van de opgave bestaat uit het toepassen van deze functies op een konkreet probleem: het maken van een kalender van een gegeven jaartal.

Een twee-dimensionale tekst wordt gerepresenteerd als een lijst strings: elke regel van de tekst is een element van deze lijst. De typedefinitie luidt:

```
type Tekst = [[Char]]
```

We gaan ervoor zorgen dat een tekst altijd rechthoekig is, dus dat alle regels van een tekst even lang zijn. De functies die op teksten werken mogen deze eigenschap zonder verdere controle aannemen, maar moeten er wel ook voor zorgen dat hun resultaat dan ook deze eigenschap heeft.

Functies op teksten

Hieronder volgen de types van functies die op teksten werken, met een korte beschrijving. De opgave is om deze functies te schrijven.

```
leeg :: Tekst
```

Het is altijd handig om een eenheidselement beschikbaar te hebben: `leeg` is een tekst met 0 regels.

```
toon :: Tekst -> [Char]
```

De functie `toon` kan gebruikt worden om een `Tekst` te prepareren voor weergave op het scherm. Alle regels worden hiertoe aan elkaar gekoppeld, gescheiden door newline-karakters.

```
breedte :: Tekst -> Int
hoogte  :: Tekst -> Int
```

Deze functies bepalen de breedte, respectievelijk de hoogte van een tekst.

```
maxBreedte :: [Tekst] -> Int
maxHoogte  :: [Tekst] -> Int
```

Deze functies bepalen wat de maximale breedte, respectievelijk hoogte, van een aantal teksten is.

```
maakBreed :: Int -> Tekst -> Tekst
maakHoog  :: Int -> Tekst -> Tekst
```

Deze functies leveren een nieuwe tekst op met de gegeven breedte, respectievelijk hoogte. Als de gegeven tekst te breed (hoog) is, wordt het eind van de regels (regels aan het einde) afgekapt; is de tekst niet breed (hoog) genoeg, dan worden spaties (lege regels) aan het eind toegevoegd.

```
tekst :: Int -> Int -> Tekst -> Tekst
```

Deze functie zorgt ervoor dat een tekst de gegeven breedte (eerste parameter) en hoogte (tweede parameter) krijgt.

```
naast :: Tekst -> Tekst -> Tekst
```

Deze functie zet twee teksten naast elkaar, en maakt er zo één grote tekst van. Je mag zonder controle aannemen dat de teksten even hoog zijn.

```
boven :: Tekst -> Tekst -> Tekst
```

Deze functie zet twee teksten boven elkaar, en maakt er zo één grote tekst van. Je mag zonder controle aannemen dat de teksten even breed zijn.

```
rij    :: [Tekst] -> Tekst
stapel :: [Tekst] -> Tekst
```

Deze functies zetten een lijst teksten allemaal naast, respectievelijk boven elkaar. De teksten in de lijst hebben nog niet gegarandeerd dezelfde hoogte resp. breedte!

```
groepeer :: Int -> [a] -> [[a]]
```

Deze functie groepeert de elementen van een lijst in deel-lijstjes van de aangegeven lengte. De laatste deel-lijst mag eventueel iets korter zijn. Bijvoorbeeld `groepeer 2 [1..5]` levert de lijst van lijsten `[[1,2],[3,4],[5]]`.

```
hblok :: Int -> [Tekst] -> Tekst
vblok :: Int -> [Tekst] -> Tekst
```

Deze functies maken een aantal kleine teksten tot één grote tekst. Daarbij geeft de `Int` parameter aan hoeveel kleine teksten er steeds naast elkaar (bij `hblok`), respectievelijk boven elkaar (bij `vblok`) gezet moeten worden.

Kalenderfuncties

Schrijf, gebruik makend van de tekstmanipulatie-functies, nu twee functie:

```
maand :: Int -> Int -> Tekst
jaar  :: Int -> Tekst
```

die de kalender van een gegeven maand, respectievelijk jaar opleveren. De functies hoeven niet te werken voor jaartallen voor 1753.

De functie `maand` krijgt als eerste parameter een jaartal, en als tweede parameter het nummer van een maand. De tekst die het resultaat is bevat als eerste regel de naam van de maand en het jaartal, en daaronder in vertikale kolommetjes de weken, voorafgegaan door afkortingen van de dagen, te beginnen bij maandag.

De functie moet bijvoorbeeld als volgt gebruikt kunnen worden:

```
? toon (maand 1994 10)
oktober 1994
ma     3  10 17 24 31
di     4  11 18 25
wo     5  12 19 26
do     6  13 20 27
vr     7  14 21 28
za  1  8  15 22 29
zo  2  9  16 23 30
```

De functie `jaar` geeft de kalender van een heel jaar, waarbij de kalenders voor januari, februari en maart naast elkaar staan, daaronder de kalenders voor april, mei en juni, enzovoorts:

```
? toon (jaar 1994)
januari 1994            februari 1994           maart 1994
ma     3  10 17 24 31    ma     7  14 21 28      ma     7  14 21 28
di     4  11 18 25       di  1  8  15 22         di  1  8  15 22 29
wo     5  12 19 26       wo  2  9  16 23         wo  2  9  16 23 30
do     6  13 20 27       do  3  10 17 24         do  3  10 17 24 31
vr     7  14 21 28       vr  4  11 18 25         vr  4  11 18 25
za  1  8  15 22 29       za  5  12 19 26         za  5  12 19 26
zo  2  9  16 23 30       zo  6  13 20 27         zo  6  13 20 27
april 1994              mei 1994                juni 1994
ma     4  11 18 25       ma     2  9  16 23 30    ma     6  13 20 27
di     5  12 19 26       di     3  10 17 24 31    di     7  14 21 28
wo     6  13 20 27       wo     4  11 18 25       wo  1  8  15 22 29
```

```
do     7  14 21 28       do     5  12 19 26       do  2  9  16 23 30
vr  1  8  15 22 29       vr     6  13 20 27       vr  3  10 17 24
za  2  9  16 23 30       za     7  14 21 28       za  4  11 18 25
zo  3  10 17 24          zo  1  8  15 22 29       zo  5  12 19 26
juli 1994                augustus 1994            september 1994
ma     4  11 18 25       ma  1  8  15 22 29       ma     5  12 19 26
di     5  12 19 26       di  2  9  16 23 30       di     6  13 20 27
wo     6  13 20 27       wo  3  10 17 24 31       wo     7  14 21 28
do     7  14 21 28       do  4  11 18 25          do  1  8  15 22 29
vr  1  8  15 22 29       vr  5  12 19 26          vr  2  9  16 23 30
za  2  9  16 23 30       za  6  13 20 27          za  3  10 17 24
zo  3  10 17 24 31       zo  7  14 21 28          zo  4  11 18 25
oktober 1994             november 1994            december 1994
ma     3  10 17 24 31     ma     7  14 21 28       ma     5  12 19 26
di     4  11 18 25        di  1  8  15 22 29       di     6  13 20 27
wo     5  12 19 26       wo  2  9  16 23 30       wo     7  14 21 28
do     6  13 20 27       do  3  10 17 24          do  1  8  15 22 29
vr     7  14 21 28       vr  4  11 18 25          vr  2  9  16 23 30
za  1  8  15 22 29       za  5  12 19 26          za  3  10 17 24 31
zo  2  9  16 23 30       zo  6  13 20 27          zo  4  11 18 25
```

B.3 Formulemanipulatie

Probleemstelling

In deze opgave worden twee types gedefinieerd waarmee *formules gemanipuleerd* kunnen worden. De belangrijkste functies die geschreven moeten worden zijn *symbolisch differentiëren* en het bepalen van de zogenaamde *additieve normaalvorm*. De onderdelen van de opgave zijn hieronder gemarkeerd met een •.

Het type 'Expressie'

In het diktaat wordt een datastructuur gegeven om een *polynoom* te beschrijven, en een aantal functies om polynomen te manipuleren. Een polynoom is een speciaal geval van een *rekenkundige expressie*. Een rekenkundige expressie wordt beschreven door de volgende data-definitie:

```
data Expr  =  Con Int
           |  Var Char
           |  Expr :+: Expr
           |  Expr :-: Expr
           |  Expr :*: Expr
           |  Expr :^: Expr
```

In deze definitie zijn `Con`, `Var`, `:+:`, `:-:`, `:*:` en `:^:` de constructoren. Met deze data-definitie definiëren we eigenlijk een klein programmeertaaltje. In dat taaltje is een expressie een constante integer, een variabele (die uit één letter moet bestaan), of twee expressies die opgeteld, afgetrokken, vermenigvuldigd of tot elkaars macht verheven moeten worden. In het taaltje bestaat er volgens deze data-definitie geen deling.

Voorbeelden van rekenkundige expressies met hun representatie als datastructuur zijn de volgende:

expressie	datastructuur
$3x + 5$	`Con 3 :*: Var 'x' :+: Con 5`
$(x + y)^2$	`(Var 'x' :+: Var 'y') :^: Con 2`
3	`Con 3`

- Definiëer het type `Expr` (kwestie van overschrijven). Gebruik precies dezelfde namen zoals in het voorbeeld, want de 'klant' wil ze zo gebruiken. Voorzie de vier constructor-operatoren van dezelfde prioriteit als de ingebouwde operatoren `+`, `-`, `*` en `^`.

Evalueren van een expressie

Een *environment* koppelt letters aan getallen:

```
type Env = [(Char,Int)]
```

Gegeven een environment kun je de waarde van een expressie bepalen (als alle variabelen uit de expressie tenminste in het environment aan een waarde gekoppeld worden).

- Schrijf een functie
  ```
  eval :: Expr -> Env -> Int
  ```
  die de waarde van een expressie bepaalt in een gegeven environment.

Voorbeeld van een gebruik van deze functie:

```
? eval  (Con 3 :*: Var 'x' :+: Con 5)  [('x',2),('y',4)]
11
```

Als de expressie variabelen bevat die niet in het environment staan, hoeft de functie niet te werken als deze variabelen echt nodig zijn voor het resultaat. Als bij vermenigvuldigen echter de linker parameter de waarde nul heeft, of bij machtsverheffen de rechter, moet de functie wel een resultaat opleveren, ook al bevat de andere parameter variabelen die niet in het environment staan. Bijvoorbeeld:

```
? eval  (Con 0 :*: Var 'x')  []
0
? eval  (Con 3 :*: Var 'x')  []
{...}
```

Symbolisch differentiëren

Bij elke expressie kan de *afgeleide* expressie worden bepaald bij differentiëren 'naar' een bepaalde variabele. Zo kan bijvoorbeeld de afgeleide van de expressie $3x^2$ naar de variabele x de expressie $6x$ zijn (of $3 * 2 * x^1$, of andere equivalente vormen). Voor het differentiëren van een expressie gelden de bekende rekenregels voor afgeleide van som en product. De afgeleide van elke andere variabele dan de variabele 'waarnaar' gedifferentieerd wordt, is nul.

- Schrijf een functie
  ```
  diff :: Expr -> Char -> Expr
  ```
  die de afgeleide functie bij differentiëren naar een bepaalde variabele bepaalt. Voor het differentiëren van een expressie waarin `:^:` voorkomt, is geen rekenregel beschikbaar. In dat geval gaan we daarom als volgt te werk: evalueer de exponent in een leeg environment (we hopen maar dat er geen variabelen in voorkomen), en gebruik dan de rekenregel $(f(x)^n)'$ is $n * f(x)^{n-1} * f'(x)$.

  De resulterende expressie hoeft niet vereenvoudigd te worden.

Het type 'Polynoom'

De datastructuur voor polynomen uit het diktaat kan gegeneraliseerd worden om *polynomen in meerdere variabelen* te representeren. Een 'polynoom' is nog steeds een rij 'termen'. Een 'term' bestaat nu echter niet meer uit een coëfficiënt en een exponent, maar uit een coëfficiënt (een integer) en een rij 'factoren'. Elke 'factor' bestaat uit een variabele-naam (een character) en een exponent (een integer).

- Definieer een type `Poly` waarmee dit soort polynomen gerepresenteerd worden.

Vereenvoudigen van polynomen

- Schrijf een functie die een polynoom vereenvoudigt.

Net als in het diktaat moeten daartoe drie dingen gebeuren:
1. de termen moeten worden gesorteerd, dus $x^2 + xy + x^2$ moet worden $x^2 + x^2 + xy$;
2. termen met gelijke factoren, moeten worden samengenomen, dus $x^2 + 2x^2$ moet worden $3x^2$;
3. termen met coëfficiënt nul moeten worden verwijderd, dus $0x^2 + x$ moet worden x.

Voordat dit kan gebeuren, moeten echter eerst de afzonderlijke termen worden vereenvoudigd. Dat lijkt erg op het vereenvoudigen van complete polynomen:
1. de factoren moeten worden gesorteerd, bijvoorbeeld alfabetisch op de erin voorkomende variabele, dus x^2yx wordt x^2xy;

2. factoren met gelijke variabele moeten worden samengenomen, dus x^2x wordt x^3;
3. factoren met exponent nul moeten worden verwijderd, dus x^0y wordt y.

Als je het handig aanpakt, kun je voor beide vereenvoudigingen dezelfde (hogere-orde) functie gebruiken.

De additieve normaalvorm

Bijna elke rekenkundige expressie kan als polynoom in meerdere variabelen geschreven worden. Bijvoorbeeld de expressie $(x + y)^4$ is equivalent aan het polynoom $x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$. Als dit polynoom vereenvoudigd is, wordt dit de *additieve normaalvorm* van de expressie genoemd.

- Schrijf een functie
  ```
  anf :: Expr -> Poly
  ```
  die de additieve normaalvorm van een expressie bepaalt. In het geval van een `:^`:-expressie mag je, net als bij het symbolisch differentiëren, de exponent eerst evalueren. Het is verstandig om hulpfuncties te schrijven die factoren, termen en polynomen kunnen vermenigvuldigen, analoog aan de functies in het diktaat.

B.4 Predicatenlogica

Deze opgave gaat over *predicaten* uit de *predicatenlogica*. Hieronder zal een datastructuur worden gedefinieerd die een predicaat beschrijft, en de opgave is om hier een aantal functies op te schrijven. Voor het schrijven van deze functies is enige kennis van de predicatenlogica nodig. Het gaat echter alleen over onderwerpen die in het kollege 'Beschrijven en Bewijzen' al aan de orde zijn geweest. Voor de studenten die dit kollege niet volgen (verkort-doctoraal opleiding): de in deze opgave geciteerde stellingen uit BB zijn op een aparte bijlage verkrijgbaar. Mocht je de informatie daarin niet begrijpen, dan wordt het tijd om zelfstandig je logica-kennis bij te spijkeren. Later in de studie komt dat ook goed van pas.

Iets over de terminologie: zoals het gebruik van de operator + een 'optelling' heet, en het gebruik van de operator × een 'vermenigvuldiging', zo hebben ook expressies waarin logische operatoren worden gebruikt een naam:

expressie met $\wedge$	conjunctie
expressie met $\vee$	disjunctie
expressie met $\rightarrow$	implicatie
expressie met $\leftrightarrow$	equivalentie
expressie met $\neg$	negatie
expressie met $\forall$	universele quantificatie
expressie met $\exists$	existentiële quantificatie

Deze woorden zullen we in de opgave verder zonder nadere toelichting gebruiken.

De types

Als je kijkt waaruit een predicaat uiteindelijk is opgebouwd, dan blijken dat *relaties* te zijn die gelden tussen *objecten*. In deze opgave spelen de relaties en objecten die gebruikt worden geen rol; het gaat alleen om hun logische samenhang. Daarom duiden we relaties en objecten aan door een naam. In teksten over propositielogica worden objecten vaak aangeduid met a, b, x, y en dergelijke, en relaties met P, Q en andere letters.

In deze opgave laten we (zoals gebruikelijk in de informatica) ook toe dat namen uit meerdere letters bestaan. Voor de duidelijkheid zijn er twee type-declaraties, zodat deze later zonodig onafhankelijk van elkaar gewijzigd kunnen worden:

```
type RelName  =  String
type ObjName  =  String
```

In het BB-diktaat worden voorbeelden gegeven van relaties tussen objecten. Eén daarvan is $K(x, y)$ (uit te spreken als 'x is een kind van y). Een relatie is blijkbaar een relatienaam toegepast op een aantal objecten. We definiëren daarom het volgende type:

```
type Relation  =  (RelName, [ObjName])
```

De Haskell-expressie waarmee we $K(x, y)$ representeren is dus `("K", ["x","y"])`.

Het type dat we gaan gebruiken om predicaten te beschrijven is een uitbreiding van het type `Prop` in paragraaf 7.1.3. blz. 134

```
data Pred  =   Con Bool
           |   Rel Relation
           |   And [Pred]
           |   Or  [Pred]
           |   Imp Pred Pred
           |   Eqv Pred Pred
           |   Not Pred
           |   All ObjName Pred
           |   Exi ObjName Pred
```

De propositie-variabelen in het type `Prop` zijn dus vervangen door relaties. Een ander verschil is dat we constructoren hebben toegevoegd om quantificaties te beschrijven. Verder is de representatie van conjuncties en disjuncties anders: in plaats van twee parameters hebben we deze operatoren een hele lijst van parameters gegeven. Dit is gedaan omdat de operatoren $\wedge$ en $\vee$ associatief zijn, en we dus geen kunstmatig onderscheid hoeven te maken tussen $a \wedge (b \wedge c)$ en $(a \wedge b) \wedge c$.

In de voorbeelden die hieronder volgen zullen we predicaten voor de duidelijkheid als wiskundige formules schrijven (met cursieve letters). Daarvoor in de plaats moet eigenlijk steeds de representatie als Haskell-datastructuur worden gelezen. Bijvoorbeeld: $P(x) \wedge \neg Q(x)$ staat voor `And [ Rel ("P",["x"]), Neg (Rel ("Q",["x"])) ]`.

De functies

Dan volgt nu een beschrijving van de functies die geschreven moeten worden. Let erop dat je de functie precies dezelfde naam en hetzelfde type geeft als hieronder gespecificeerd, dit in verband met het automatisch testen van de ingeleverde opgaven. Het is wel toegestaan om naast de gevraagde functies nog extra hulpfuncties te schrijven, waarvan je de naam zelf mag kiezen.

free Objectnamen in een predicaat die niet door een quantor worden gebonden heten *vrije variabelen*(BB paragraaf 5.9). De functie `free` moet bepalen welke variabelen vrij voorkomen in een predicaat:

```
free  ::  Pred -> [ObjName]
```

closed Een predicaat zonder vrije variablen heet *gesloten*. De functie `closed` moet bepalen of aan deze eigenschap is voldaan.

```
closed  ::  Pred -> Bool
```

rename De betekenis van een quantificatie hangt niet af van de naam die in de quantificatie gebruikt wordt. Zo zijn $\forall_x\langle P(x)\rangle$ en $\forall y\langle P(y)\rangle$ gelijkwaardig. De functie `rename` voert zo'n her-benoeming van de gebonden variabele uit:

```
rename  ::  ObjName -> Pred -> Pred
```

Als deze functie wordt toegepast op een quantificatie, dan wordt de naam die door die quantificatie gebonden wordt veranderd in de gegeven naam. Andere predicaten blijven ongewijzigd.

objsubst Soms is het nodig om objecten in een predicaat te vervangen door andere objecten. Het is daarbij de bedoeling om alleen de *vrije voorkomens* te vervangen(BB paragraaf 6.7). Als bijvoorbeeld in het predicaat $P(x) \wedge \forall_x\langle Q(x)\rangle$ het object x vervangen wordt door het object y, dan ontstaat het predicaat $P(y) \wedge \forall_x\langle Q(x)\rangle$. De door het symbool $\forall$ gebonden x wordt dus niet vervangen.

De opgave is een functie

```
objsubst  ::  [(ObjName,ObjName)] -> Pred -> Pred
```

te schrijven die in een gegeven predicaat (tweede parameter) alle vrije voorkomens van de variabelen in de linkerhelft van de lijst 2-tupels vervangt door de overeenkomstige rechterhelft.

Als de lijst meer dan één element bevat, moet de substitutie gelijktijdig plaatsvinden. Dus als x door y moet worden vervangen en y door x, dan wordt het predicaat $P(x, y)$ omgezet in $P(y, x)$ en niet in $P(x, x)$.

Er is een complicatie die substitutie nog wat ingewikkelder maakt. Bekijk bijvoorbeeld het predicaat $P(x) \wedge \forall_y\langle Q(x, y)\rangle$. Als hierin het object x wordt vervangen door y dan zou het predicaat $P(y) \wedge \forall_y\langle Q(y, y)\rangle$ ontstaan. De eerste parameter van Q, die vrij was, is hierdoor gebonden geraakt door de $\forall$. Dat is niet de bedoeling. Het gewenste resultaat is $P(y) \wedge \forall_z\langle Q(y, z)\rangle$. Omdat het vervang-object (y) gebonden dreigde te raken, hebben we de variabele van de $\forall$ eerst een andere naam gegeven. (De functie `rename` komt daarbij goed van pas!)

Welke naam je kiest (z in het voorbeeld) mag je zelf bepalen. De voorwaarde is wel dat de naam nog niet vrij voorkomt in het predicaat (x zou geen goede keuze zijn geweest) en niet gelijk is aan vervang-objecten. Het is misschien handig om een functie te maken die een willekeurige naam verzint die niet in een gegeven lijst uitzonderingen voorkomt.

relsubst Naast substitutie van objecten voor objecten bestaat er ook substitutie van predicaten voor relaties. We bekijken eerst een voorbeeld. In het predicaat $P(a, b) \wedge Q(c)$ kan de relatie $P(x, y)$ vervangen worden door $R(x) \wedge S(y)$. Het resultaat is het predicaat $R(a) \wedge S(b) \wedge Q(c)$.

Wat er in dit voorbeeld gebeurt is dat elke relatie P in het originele predicaat wordt vervangen door het predicaat met de R en de S, waarbij de x en y vervangen worden door de in het origineel aangetroffen a en b.

De te schrijven functie is

```
relsubst  ::  Relation -> Pred -> Pred -> Pred
```

De eerste parameter is de relatie die vervangen moet worden (in het voorbeeld $P(x, y)$). De tweede parameter is het predicaat dat daarvoor in de plaats komt (in het voorbeeld $R(x) \wedge S(y)$). De derde parameter is het predicaat waarin vervangen moet worden (in het voorbeeld $P(a, b) \wedge Q(c)$). Bekijk goed de manier waarop in dit voorbeeld de variabelen x en y (die in de tweede component van de `Relation`-parameter stonden opgesomd) worden vervangen door a en b (die achter P stonden in de `Pred`-parameter).

Ook bij deze substitutie moet weer worden gewaakt voor ongewenste bindingen. Een voorbeeld waarin dat gedemonstreerd wordt is de aanroep

`relsubst` $(Q(x))$ $(R(x) \wedge S(y))$ $(\forall_y\langle Q(y) \wedge T(y)\rangle)$

Zou je deze substitutie zondermeer uitvoeren dan ontstaat het predicaat $\forall_y\langle R(y) \wedge S(y) \wedge T(y)\rangle$. Hierbij is de y in $S(y)$ ten onrechte gebonden geraakt. Het gewenste resultaat is $\forall_z\langle R(z) \wedge S(y) \wedge T(z)\rangle$. (Bekijk dit voorbeeld nauwkeurig; het is zo ontworpen dat alle problemen erin gedemonstreerd worden).

prenex In opgave 6.1 in het BB-diktaat wordt gevraagd om predicaten in *prenex*-vorm te schrijven, dat wil zeggen met alle quantoren vooraan bij elkaar. Ook dit proces gaan we automatiseren. Het type van de functie is

```
prenex  ::  Pred -> Pred
```

Voor de werking van de functie kun je gebruik maken van stelling 6.1 en 6.3 in het BB-diktaat. Om aan de voorwaarde uit stelling 6.3 te voldoen, kun je zonodig de functie `rename` gebruiken.

De operator $\leftrightarrow$ wordt in de stellingen niet genoemd. Deze operator kun je 'wegwerken' door gebruik te maken van de gelijkheid tussen $P \leftrightarrow Q$ en $P \rightarrow Q \wedge Q \rightarrow P$.

dnv Elk predicaat zonder quantoren heeft een *disjunctieve normaalvorm*, afgekort *dnv*. De dnv is een formule die bestaat uit een grote disjunctie, waarvan de elementen conjuncties zijn, waarvan de elementen relaties of negaties van relaties zijn. (zie paragraaf 3.7 van het BB-diktaat).

Het type van de functie is

```
dnv  ::  Pred -> Pred
```

Deze functie hoeft niet te werken op predicaten die een quantificatie zijn.

In het BB-diktaat staat een methode genoemd om de dnv te bepalen die gebruik maakt van waarheidstafels. Makkelijker is het om de dnv te bepalen met formule-manipulatie. Daartoe doe je een gevalsonderscheid:

- Is de formule een disjunctie? Schrijf dan alle deelformules (recursief) in dnv, en voeg al deze disjuncties samen tot één grote.
- Is de formule een conjunctie, bijvoorbeeld $P \wedge Q$? Maak dan recursief dnv's van de deelformules. Dit levert bijvoorbeeld $(P_1 \vee P_2) \wedge (Q_1 \wedge Q_2)$. Hiervan kun je één disjunctie maken door hierin de haakjes uit te werken met behulp van BB-stelling 3.5.16.
- Is de formule een implicatie of equivalentie? Schrijf deze dan eerst als conjunctie of disjunctie.
- Is de formule een negatie? Werk die dan eerst 'naar binnen' met BB-stelling 3.5.[1,10-12].

prenexdnv Deze functie zet een predicaat om in zijn prenex-vorm, en zet het binnenstuk in dnv.

```
prenexdnv  ::  Pred -> Pred
```

simple Door al dat gemanipuleer ontstaan erg ingewikkelde formules. De functie `simple` moet een formule kunnen simplificeren met de volgende regels:

- Een conjunctie waarin zowel $P(x)$ als $\neg P(x)$ voorkomt is *False*
- Een disjunctie waarin zowel $P(x)$ als $\neg P(x)$ voorkomt is *True*
- Een conjunctie waarin *False* voorkomt is *False*;
- Een disjunctie waarin *True* voorkomt is *True*;
- De constante *True* kun je verwijderen uit conjuncties;
- De constante *False* kun je verwijderen uit disjunctie;
- Een lege conjunctie is *True*;
- Een lege disjunctie is *False*;
- Een dubbele ontkenning valt weg;
- Quantificaties over variabelen die niet gebruikt worden vallen weg;
- Een conjunctie waarin weer een conjunctie voorkomt kan geschreven worden als één grote conjunctie;
- Idem voor een disjunctie.

Eenvoudiger testen

Om het testen van je programma eenvoudiger te maken zijn er twee functies beschikbaar:

```
showPred  ::  Pred -> String
parsePred ::  String -> Pred
```

waarmee een predicaat omgezet kan worden in een meer leesbare representatie, en andersom. Deze functies staan in de file

```
/praktikum/fp/predicaat.gs
```

Je kunt deze file bij je eigen functiesdefinities bijladen met het Haskell-commando `:a`. Lees het commentaar in deze file voor een beschrijving hoe een predicaat dat door `parsePred` verwerkt wordt opgebouwd moet worden.

Aanwijzingen voor het inleveren

Maak een Haskell-programma waarin de genoemde types (`RelName`, `ObjName`, `Relation` en `Pred`) worden gedefinieerd, en alle gevraagde functies: `free`, `closed`, `rename`, `objsubst`, `relsubst`, `prenex`, `dnv`, `prenexdnv`, en `simple`. Specificeer bij elke functie ook het type. De functies moeten exact dezelfde naam hebben zoals hierboven gespecificeerd, omdat de programma's automatisch getest worden.

B.5 De klasse 'Verzameling'

Probleemstelling

In deze opgave wordt een klasse gedefinieerd waarin de operaties op *verzamelingen* zijn vastgelegd. Vervolgens wordt een tweetal implementaties van de verzamelingsoperaties als instance van deze klasse gemodelleerd.

Alle onderdelen van de opgave zijn aangegeven met een •. Geef alle gevraagde functies, types enz. precies de naam zoals die in de opgave omschreven staat (inclusief hoofdletters). Voor parameters en eventuele hulpfuncties mag je natuurlijk zelf namen verzinnen.

Verzamelingen

Een type is een *verzameling*-type als er functies `doorsnede`, `vereniging` en `complement` op gedefinieerd kunnen worden, en er constanten `leeg` en `universum` zijn.

- Definieer een klasse `Verz`, waarin de genoemde functies gedefinieerd worden. Zet in de klasse een default-definitie voor `universum`.

Eerste implementatie

Lijsten lijken een handige implementatie te zijn van verzamelingen. Het probleem is echter dat de complement-functie niet gedefinieerd kan worden. Wat is immers het complement van de lege verzameling? Voor verzamelingen van natuurlijke getallen zou je nog de lijst `[0..]` kunnen nemen, maar hoe moet dat voor verzamelingen van een willekeurig type...

We kiezen daarom een andere benadering. Van de te representeren verzameling worden òf alle elementen in een lijst opgesomd, òf er wordt opgesomd welke elementen nìet in de verzameling zitten. Deze twee gevallen worden onderscheiden door een constructorfunctie `Alleen`, respectievelijk `Behalve` op de lijst toe te passen.

- Geef een data-declaratie voor `LijstVerz`, waarmee verzamelingen van een willekeurig type op deze manier opgeslagen kunnen worden.
- Maak het zojuist gedefinieerde type tot instance van `Verz`. Denk er aan om hierbij een voorwaarde voor het object-type van de verzameling op te geven.
- Maak het type tot instance van `Eq`. Denk eraan dat het aantal keren dat een element in de verzameling voorkomt niet uitmaakt, dit in tegenstelling tot wat bij lijsten het geval is.
- Maak het type tot instance van `Ord`. De rol van de $\leq$-operator wordt hierbij gespeeld door de *deelverzamelings*-relatie.

Tweede implementatie

Een andere mogelijke implementatie voor verzamelingen zijn functies met `Bool` resultaat. Een element is lid van een gegeven verzameling, als de functie die de verzameling voorstelt `True` oplevert als hij wordt toegepast op dat element.

- Geef een `type`-declaratie voor `FunVerz`, waarmee verzamelingen op deze manier opgeslagen kunnen worden.
- Maak `FunVerz` tot instance van `Verz`.
- Beschrijf in woorden (als commentaar bij het programma) waarom `FunVerz` niet tot instance van `Ord` gemaakt kan worden.
- Schrijf ook in commentaar waarom de tweede implementatie ook voordelen biedt boven de eerste. Geef daartoe aan wat voor soort verzamelingen niet als `LijstVerz` maar wel als `FunVerz` kan worden gerepresenteerd.

Meer functies op verzamelingen

Door gebruik te maken van de functies die in de klasse `Verz` zijn gedeclareerd, is het mogelijk om *overloaded* functies te schrijven die op verzamelingen werken, ongeacht de implementatie daarvan.

- Schrijf een functie `verschil` die het verschil van twee verzamelingen bepaalt. Geef ook het type van deze functie.
- Schrijf een functie `symmVerschil` die het symmetrisch verschil van twee verzamelingen bepaalt.

B.6 Chipknip kaarten

Om enigszins met de moderne tijd mee te gaan, hebben de banken besloten dat betalen met geld achterhaald is en dat iedereen met een chipknip alles moet kunnen doen. Dat wil zeggen geld moet kunnen overmaken van een rekening nummer naar een ander rekening nummer en bedragen van de eigen rekening kunnen opnemen of storten. Omdat men al leuke voiceherkenners kan maken, gaan de banken hierin mee en hebben hardware gekocht om een gesproken woord te vertalen naar een string en andersom.

Nadat de gebruiker de pas heeft ingevoerd en een pincode heeft getoetst, is een actie toegestaan door het inspreken van een boodschap. (Het rekeningnummer van de eigenaar is dan bekend.) Als deze boodschap door de spraakherkenner heen is, is ze in de vorm van een string. Deze moet worden vertaald in een reeks opdrachten.

Deze opdrachten worden tijdelijk opgeslagen in een wachtrij alvorens te worden uitgevoerd. (Vanwege het drukke verkeer op het internet is het niet altijd mogelijk direct de opdracht te verwerken.) Daarna gebeurt hetzelfde voor de volgende klant.

De bedoeling van deze opgave is om een aantal hulpfuncties en een datastructuur te schrijven waarmee de chipknip gerealiseerd wordt.

Het hele proces van het chipknippen kan verdeeld worden in de tracées die gevolgd worden voor elke klant. Het uitvoeren van het programma is dan het herhaaldelijk verwerken van de opdrachten van de klant. Dit gebeurt echter in batch: het fysieke verwerken van de opdracht gebeurt als er geen klanten meer zijn. (Dit levert mogelijk inconsistentie op in de database, maar maakt het iets eenvoudiger.)

1. Allereerst wordt de pincode gelezen en de gesproken boodschap omgezet in een string d.m.v. hardware, microfonen e.d. Als gegeven mag je dus aannemen dat er hierna een string per pincode beschikbaar is. Bij elke pincode is een uniek rekeningnummer gelezen dat in de opdrachten van belang is. Voor de invoer kun je daarom definiëren :

   ```
   type Input = [(Int,String)]
   ```

   Dit is het type van de invoer van het programma.

2. Om de gesproken boodschappen te kunnen verwerken, moet de string uit het tupel worden omgezet in een reeks (of één) boodschap(pen). Om de string te parsen tot een opdracht, kijken we eerst iets precieser naar de vorm van de boodschappen en de opdrachten. De acties die ondernomen kunnen worden met de chipknip kunnen als volgt worden omschreven: (de bedragen zijn in hele guldens)
   - overmaken bedrag rek_nr1 rek_nr2
   - opnemen bedrag rek_nr1
   - storten bedrag rek_nr1

   Hierbij is rek_nr1 het rekening nummer van de chipknip houder. Voor de duidelijkheid is het handig om een extra type te introduceren dat staat voor de rekeningnummers:

   ```
   type Rekening = Int
   ```

3. Definieer een datatype voor `Actie` om deze opdrachten te representeren.

4. Schrijf allereerst een hulpfunctie om het juiste rekeningnummer van de pinhouder in te vullen. Deze heeft type:

```
Actie -> Rekening -> Actie
```

s Deze functie vult dus gegevens in binnen het datatype.

Een actie mislukt als een rekening nummer niet bestaat of het tegoed op een rekening te laag is. Je mag ervan uitgaan dat de chipknip altijd genoeg geld bevat om het gewenste te kunnen storten en dat alle transacties slagen. (Als uitbreiding is het misschien leuk om mislukte acties wel mee te nemen. Je moet dan wel wat aan de types veranderen.)

5. Schrijf nu een functie die gegeven een rekeningnummer een parser oplevert voor de opdrachten van de rekeninghouder:

   ```
   Rekening -> Parser Char Actie
   ```

   (Hint: denk aan het gebruik van de `<$>`-combinator!)

6. Met deze functie is het vrij eenvoudig om een derde functie te schrijven die gegeven een rekeningnummer en een invoerstring een lijst van acties produceert:

   ```
   (Rekening,String) -> [Actie]
   ```

7. Tenslotte moet deze functie voor alle gebruikers uitgevoerd worden. Dat betekent dat de functie bij het vorige onderdeel gebruikt wordt voor een functie van het type:

   ```
   [(Rekening,String)] -> [Actie]
   ```

   Oftewel:

   ```
   Input -> [Actie]
   ```

8. Nu ben je klaar met het herkennen van de acties. Dus voor een stel gegeven opdrachten in string vorm zijn deze nu in gerepresenteerde vorm in een lijst te zetten. Deze lijst moet nu worden verwerkt in de database. Het programma moet nu voor een lijst van acties en de database als input, de acties verwerken in de database en aan de gebruiker teruggeven hoeveel er is opgenomen, gestort etc.

   Definieer het type voor de database:

   ```
   type Database = [(Rekening,Naam,Saldo)]
   ```

   waarbij is gedefinieerd:

   ```
   type Naam = String
   type Saldo = Int
   ```

9. We moeten nu een aantal functies schrijven om de acties te verwerken. Het is een goede gewoonte om eerste een simpel probleem op te lossen en dan dit te generaliseren. Schrijf daarom eerst een functie die één actie verwerkt in de database. Deze heeft als type:

   ```
   Actie -> Database -> Database
   ```

   Schrijf daarna een functie met het type

   ```
   [Actie] -> Database -> Database
   ```

10. De gebruiker moet een boodschap terugkrijgen om te controleren of de spraakherkenner wel een goede vertaling heeft gedaan. Daartoe geef je een vertaling van de abstracte respresentatie van de opdracht naar een string. De boodschappen behorende bij de voorbeeld-opdrachten hierboven zijn:
    - “overmaken vijfennegentig gulden op negen drie vijf drie drie zes twee een nul” (voor het overmaken van 95 gulden op een rek_nr2: 935336210)
    - “opnemen vijfennegentig gulden”
    - “storten vijfennegentig gulden”

    Deze teksten zullen de speaker worden doorgegeven aan de gebruiker.

    We moeten dus een unparsing maken:

    ```
    [Actie] -> String
    ```

    Daarnaast moet de geupdate database worden weergegeven op het scherm. De functie die dit voor je doet, is van het type:

```
Database -> String
```

Hint: scrijf eerst functies om één actie (resp. één rekening) te vertalen.

11. Schrijf tenslotte de functie waar het allemaal om begonnen is, met het type

```
Input -> String
```

Bijlage C

ISO/ASCII tabel

	0*16+...		1*16+...		2*16+...		3*16+...		4*16+...		5*16+...		6*16+...		7*16+...	
...+0	0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	‘	112	p
...+1	1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
...+2	2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
...+3	3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
...+4	4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
...+5	5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
...+6	6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
...+7	7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
...+8	8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
...+9	9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
...+10	10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
...+11	11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
...+12	12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
...+13	13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
...+14	14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
...+15	15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Gofer-notatie voor speciale tekens in een string gaat met behulp van een code achter een backslash:

- de *naam* van het speciale teken, bijvoorbeeld `"\ESC"` voor het escape-teken;
- het *nummer* van het speciale teken, bijvoorbeeld `"\27"` voor het escape-teken;
- het nummer in het achttallig stelsel (*octaal*), bijvoorbeeld `"\o33"` voor het escape-teken;
- het nummer in het zestientallig stelsel (*hexadecimaal*), bijvoorbeeld `"\x1B"` voor het escape-teken;
- door de overeenkomstige letter vier kolommen verder naar rechts, bijvoorbeeld `"\^["` voor het escape-teken;
- een van de volgende codes: `"\n"` (newline), `"\b"` (backspace), `"\t"` (tab), `"\a"` (alarm), `"\f"` (formfeed), `"\""` ("-symbool), `"\’"` (’-symbool), en `"\\"` (\-symbool)

Bijlage D

Haskell syntax

Hieronder volgt een samenvatting van de Haskell-syntax. Er wordt gebruik gemaakt van de BNF-notatie voor grammatica's, met de volgende conventies:

- grammaticale begrippen staan in het gewone lettertype;
- symbolen die in het programma terecht komen staan in rechthoeken en zijn gezet in het `tikmachine lettertype`;
- verticale strepen '|' scheiden de alternatieven;
- {accolades} staan rond dingen die nul of meer keer aanwezig mogen zijn;
- [vierkante haken] staan rond dingen die weggelaten mogen worden.

De inhoud van een file wordt beschreven door het begrip 'module'. De inhoud van een opdracht aan de interpreter wordt gegeven door het begrip 'interp'.

In de grammatica is sprake van de symbolen `{` en `}`. Het symbool `{` mag weggelaten worden. De layout-regel gaat dan gelden. Dat wil zeggen: een `;` mag weggelaten worden, als de tekst achter de `;` evenver ingesprongen dan de tekst ervóór. Wordt de tekst minder ingesprongen, dan wordt automatisch een `}` toegevoegd. (Zie ook paragraaf 1.4.5). blz. 14

De volgende begrippen worden verder niet uitgewerkt (ze staan daarom in *cursief* lettertype):

varid	naam die begint met een kleine letter
conid	naam die begint met een hoofdletter
varop	operator-symbool dat niet met een `:` begint
conop	operator-symbool dat met een `:` begint
integer	integer constante
float	floating point constante
char	character constante
string	string constante

Declaraties

module	::=	`{` topdecls `}`	programma
interp	::=	exp [where]	opdracht aan interpreter
topdecls	::=	topdecls `;` topdecls	meerdere declaraties
	|	`data` typeLhs `=` constrs	datatype declaratie
	|	`type` typeLhs `=` type	type-synoniem declaratie
	|	`infixl` [digit] op {`,` op}	prioriteits-declaratie
	|	`infixr` [digit] op {`,` op}	
	|	`infix` [digit] op {`,` op}	
	|	`primitive` prims `::` type	declaratie ingebouwde functie
	|	class	klasse declaratie
	|	inst	instance declaratie
	|	decls	functie declaratie

typeLhs	::=	*conid* { *varid* }	linkerkant typedeclaratie
constrs	::=	constrs `|` constrs	meerdere constructoren
	|	type *conop* type	infix constructor
	|	*conid* {type}	constructor
prims	::=	prims `,` prims	meerdere primitieven
	|	var *string*	binding primitieve functie

Types

sigType	::=	[context `=>`]type	[voorwaardelijk] type
context	::=	`(` [pred {`,` pred}] `)`	algemene vorm
	|	pred	singleton voorwaarde
pred	::=	*conid* type {type}	vorm van voorwaarde
type	::=	ctype [`->` type]	functie type
ctype	::=	*conid* {atype}	datatype of synonym
	|	atype	
atype	::=	*varid*	type variabele
	|	`()`	nultupel
	|	`(` type `)`	type tussen haakjes
	|	`(` type `,` type {`,` type} `)`	tupel type
	|	`[` type `]`	lijst type

Klasse- en instance-declaraties

class	::=	`class` [context `=>`] pred [cbody]	
cbody	::=	`where` `{` cdecls `}`	
cdecls	::=	cdecls `;` cdecls	meerdere declaraties
	|	var {`,` var} `::` type	onderdeel-declaratie
	|	fun rhs [where]	default definities
inst	::=	`instance` [context `=>`] pred [ibody]	
ibody	::=	where `{` idecls `}`	
idecls	::=	idecls `;` idecls	meerdere declaraties
	|	fun rhs [where]	onderdeel-definitie

Waarde- en functie-declaraties

decls	::=	decls `;` decls	meerdere declaraties
	|	var {`,` var} `::` sigType	type declaratie
	|	fun rhs [where]	functiedefinitie
	|	pat rhs [where]	patroondefinitie
rhs	::=	`=` exp	eenvoudige rechterkant
	|	gdRhs {gdRhs}	alternatieven
gdRhs	::=	`|` exp `=` exp	
where	::=	`where` `{` decls `}`	locale definities
fun	::=	var	functie zonder parameters
	|	pat varop pat	infix operator
	|	`(` pat varop `)`	sectie-notatie
	|	`(` varop pat `)`	
	|	fun apat	function met parameters
	|	`(` fun `)`	overbodige haakjes

Expressies

exp	::=	`\` apat {apat} `->` exp	lambda expressie
	|	`let` `{` decls `}` `in` exp	locale definitie
	|	`if` exp `then` exp `else` exp	voorwaardelijke expressie
	|	`case` exp `of` `{` alts `}`	case-expressie
	|	opExp `::` sigType	getypeerde expressie
	|	opExp	
opExp	::=	opExp op opExp	toepassen van operator
	|	pfxExp	
pfxExp	::=	`-` appExp	min met één parameter
	|	appExp	
appExp	::=	appExp atomic	functie-toepassing
	|	atomic	
atomic	::=	var	variabele
	|	conid	constructor
	|	*integer*	integer constante
	|	*float*	floating point constante
	|	*char*	character constante
	|	*string*	string constante
	|	`()`	nultupel
	|	`(` exp `)`	expr. tussen haakjes
	|	`(` exp op `)`	secties
	|	`(` op exp `)`	
	|	`[` list `]`	lijst expressie
	|	`(` exp `,` exp {`,` exp} `)`	tupel
list	::=	[exp {`,` exp}]	lijst-opsomming
	|	exp `|` quals	lijst-comprehensie
	|	exp `..`	rekenkundige opsomming
	|	exp `,` exp `..`	
	|	exp `..` exp	
	|	exp `,` exp `..` exp	
quals	::=	quals `,` quals	meerdere qualifiers
	|	pat `<-` exp	generator
	|	pat `=` exp	locale definitie
	|	exp	boolean guard
alts	::=	alts `,` alts	meerdere alternatieven
	|	pat altRhs [where]	alternatief
altRhs	::=	`->` exp	enkel alternatief
	|	gdAlt gdAlt	guarded alternatieven
gdAlt	::=	`|` exp `->` exp	guarded alternatief

Patronen

pat	::=	pat conop pat	operator toepasssing
	|	pat `+` *integer*	$(n + k)$ patroon
	|	*integer* `*` pat	$(c * n)$ patroon
	|	appPat	
appPat	::=	appPat apat	toepassing
	|	apat	
apat	::=	var	variabele
	|	var `@` pat	'as'-patroon
	|	`~` pat	'irrefutable' patroon
	|	`_`	wildcard
	|	conid	constructor
	|	*integer*	integer constante
	|	*char*	character constante
	|	*string*	string constante
	|	`()`	nultupel
	|	`(` pat `)`	expressie tussen haakjes
	|	`(` pat conop `)`	secties
	|	`(` conop pat `)`	
	|	`[` [pat {`,` pat}] `]`	lijst
	|	`(` pat `,` pat {`,` pat} `)`	tupel

Variabelen en operators

var	::=	varid	
	|	`(-)`	variabele
op	::=	varop	
	|	conop	
	|	`-`	operator
varid	::=	*varid*	
	|	`(` *varop* `)`	operator als functie
varop	::=	*varop*	
	|	`` ` `` *varid* `` ` ``	functie als operator
conid	::=	*conid*	
	|	`(` *conop* `)`	
conop	::=	*conop*	
	|	`` ` `` *conid* `` ` ``	

Bijlage E

Gofer standaardfuncties

Operator-prioriteiten

```
infixl  9  !!
infixr  9  .
infixr  8  ^
infixl  7  *
infix   7  /, `div`, `quot`, `rem`, `mod`
infixl  6  +, -
infix   5  \\
infixr  5  ++, :
infix   4  ==, /=, <, <=, >=, >
infix   4  `elem`, `notElem`
infixr  3  &&
infixr  2  ||
infixr  0  $
```

Functies op Booleans en characters

```
otherwise   ::  Bool
not         ::  Bool   -> Bool
(&&), (||)  ::  Bool   -> Bool -> Bool
and, or     ::  [Bool] -> Bool
any, all    ::  (a->Bool) -> [a] -> Bool

isAscii, isControl, isPrint, isSpace              ::  Char -> Bool
isUpper, isLower,   isAlpha, isDigit, isAlphanum  ::  Char -> Bool

minChar, maxChar   ::  Char
toUpper, toLower   ::  Char -> Char
ord                ::  Char -> Int
chr                ::  Int  -> Char
```

Numerieke functies

```
even, odd  ::  Int -> Bool
gcd, lcm   ::  Int -> Int -> Int
div, quot  ::  Int -> Int -> Int
rem, mod   ::  Int -> Int -> Int

pi  ::  Float
sin, cos, tan     ::  Float -> Float
asin, acos, atan  ::  Float -> Float
log, log10        ::  Float -> Float
exp, sqrt         ::  Float -> Float
atan2             ::  Float -> Float -> Float
```

```
truncate          ::  Float -> Int

subtract       ::  Num a          =>  a -> a -> a
(^)            ::  Num a          =>  a -> Int -> a
abs            ::  (Num a, Ord a) =>  a -> a
signum         ::  (Num a, Ord a) =>  a -> Int
sum, product   ::  Num a          =>  [a] -> a
sums, products ::  Num a          =>  [a] -> [a]
```

Polymorfe functies

```
undefined ::  a
id        ::  a -> a
const     ::  a -> b -> a
asTypeOf  ::  a -> a -> a
($)       ::  (a->b) -> (a->b)
strict    ::  (a->b) -> (a->b)

(.)       ::  (b->c)     -> (a->b) -> (a->c)
uncurry   ::  (a->b->c)  -> ((a,b)->c)
curry     ::  ((a,b)->c) -> (a->b->c)
flip      ::  (a->b->c)  -> (b->a->c)
until     ::  (a->Bool)  -> (a->a) -> a -> a
until'    ::  (a->Bool)  -> (a->a) -> a -> [a]
```

Functies op tupels

```
fst   ::  (a,b)   -> a
snd   ::  (a,b)   -> b
fst3  ::  (a,b,c) -> a
snd3  ::  (a,b,c) -> b
thd3  ::  (a,b,c) -> c
```

Functies op lijsten

```
head   ::  [a] -> a
last   ::  [a] -> a
tail   ::  [a] -> [a]
init   ::  [a] -> [a]
(!!)   ::  [a] -> Int -> a

take, drop           ::  Int       -> [a] -> [a]
splitAt              ::  Int       -> [a] -> ([a],[a])
takeWhile, dropWhile ::  (a->Bool) -> [a] -> [a]
takeUntil            ::  (a->Bool) -> [a] -> [a]
span, break          ::  (a->Bool) -> [a] -> ([a],[a])

length           ::            [a] -> Int
null             ::            [a] -> Bool
elem, notElem    ::  Eq a  =>  a   -> [a] -> Bool
maximum, minimum ::  Ord a =>  [a] -> a
genericLength    ::  Num a =>  [b] -> a

(++)      ::  [a] -> [a]  -> [a]
iterate   ::  (a->a) -> a -> [a]
repeat    ::  a           -> [a]
cycle     ::  [a]         -> [a]
```

```
copy      ::  Int -> a    -> [a]
reverse   ::  [a]         -> [a]
nub       ::  Eq a => [a] -> [a]
(\\)      ::  Eq a => [a] -> [a] -> [a]

concat    ::  [[a]] -> [a]
transpose ::  [[a]] -> [[a]]

map       ::  (a->b)    -> [a] -> [b]
filter    ::  (a->Bool) -> [a] -> [a]

foldl     ::  (a->b->a) -> a -> [b] -> a
foldl'    ::  (a->b->a) -> a -> [b] -> a
foldr     ::  (a->b->b) -> b -> [a] -> b
foldl1    ::  (a->a->a) ->      [a] -> a
foldr1    ::  (a->a->a) ->      [a] -> a
scanl     ::  (a->b->a) -> a -> [b] -> [a]
scanl'    ::  (a->b->a) -> a -> [b] -> [a]
scanr     ::  (a->b->b) -> b -> [a] -> [b]
scanl1    ::  (a->a->a) ->      [a] -> [a]
scanr1    ::  (a->a->a) ->      [a] -> [a]

insert    ::  Ord a =>  a -> [a] -> [a]
sort      ::  Ord a =>       [a] -> [a]
qsort     ::  Ord a =>       [a] -> [a]
merge     ::  Ord a =>       [a] -> [a] -> [a]

zip   ::  [a] -> [b]                                    -> [(a,b)]
zip3  ::  [a] -> [b] -> [c]                             -> [(a,b,c)]
zip4  ::  [a] -> [b] -> [c] -> [d]                      -> [(a,b,c,d)]
zip5  ::  [a] -> [b] -> [c] -> [d] -> [e]               -> [(a,b,c,d,e)]
zip6  ::  [a] -> [b] -> [c] -> [d] -> [e] -> [f]        -> [(a,b,c,d,e,f)]
zip7  ::  [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [(a,b,c,d,e,f,g)]

zipWith   ::  (a->b->c)                -> [a]->[b]->[c]
zipWith3  ::  (a->b->c->d)             -> [a]->[b]->[c]->[d]
zipWith4  ::  (a->b->c->d->e)          -> [a]->[b]->[c]->[d]->[e]
zipWith5  ::  (a->b->c->d->e->f)       -> [a]->[b]->[c]->[d]->[e]->[f]
zipWith6  ::  (a->b->c->d->e->f->g)    -> [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7  ::  (a->b->c->d->e->f->g->h) -> [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
unzip     ::  [(a,b)] -> ([a],[b])
```

Functies op strings

```
ljustify    ::  Int -> String -> String
cjustify    ::  Int -> String -> String
rjustify    ::  Int -> String -> String
space       ::  Int -> String

words     ::  String   -> [String]
lines     ::  String   -> [String]
unwords   ::  [String] -> String
unlines   ::  [String] -> String
layn      ::  [String] -> String

type ShowS  = String -> String

showChar   ::            Char   -> ShowS
showString ::            String -> ShowS
```

```
shows      ::  Text a => a      -> ShowS
show       ::  Text a => a      -> String
show'      ::            a      -> String

openfile   ::  String -> String
```

Typeklassen

```
class Eq a where
    (==), (/=) ::  a -> a -> Bool

class Eq a => Ord a where
    (<), (<=), (>), (>=) ::  a -> a -> Bool
    max, min             ::  a -> a -> a

class Ord a => Ix a where
    range   ::  (a,a) -> [a]
    index   ::  (a,a) -> a -> Int
    inRange ::  (a,a) -> a -> Bool

class Ord a => Enum a where
    enumFrom       ::  a -> [a]              -- [n..]
    enumFromThen   ::  a -> a -> [a]         -- [n,m..]
    enumFromTo     ::  a -> a -> [a]         -- [n..m]
    enumFromThenTo ::  a -> a -> a -> [a]    -- [n,n'..m]

class (Eq a, Text a) => Num a where
    (+), (-), (*), (/) ::  a -> a -> a
    negate             ::  a -> a
    fromInteger        ::  Int -> a

class Text a where
    showsPrec      ::  Int -> a   -> ShowS
    showList       ::         [a] -> ShowS
```

Instanties van typeklassen

```
instance  Eq   ()
instance  Ord  ()
instance  Text ()
instance  Eq   Bool
instance  Ord  Bool
instance  Text Bool
instance  Eq   Char
instance  Ord  Char
instance  Ix   Char
instance  Enum Char
instance  Text Char
instance  Eq   Int
instance  Ord  Int
instance  Ix   Int
instance  Enum Int
instance  Num  Int
instance  Text Int
instance  Eq   Float
instance  Ord  Float
instance  Enum Float
instance  Num  Float
```

```
instance  Text Float
instance  Eq a             => Eq   [a]
instance  Ord a            => Ord  [a]
instance  Text a           => Text [a]
instance  (Eq a, Eq b)     => Eq   (a,b)
instance  (Ord a, Ord b)   => Ord  (a,b)
instance  (Text a, Text b) => Text (a,b)
```

Input/output functies

```
type Dialogue    =  [Response]  -> [Request]
type SuccCont    =                 Dialogue
type StrCont     =  String      -> Dialogue
type StrListCont =  [String]    -> Dialogue
type FailCont    =  IOError     -> Dialogue

stdin, stdout, stderr, stdecho ::  String

done            ::                                                  Dialogue
readFile        ::  String ->           FailCont -> StrCont     -> Dialogue
writeFile       ::  String -> String -> FailCont -> SuccCont    -> Dialogue
appendFile      ::  String -> String -> FailCont -> SuccCont    -> Dialogue
readChan        ::  String ->           FailCont -> StrCont     -> Dialogue
appendChan      ::  String -> String -> FailCont -> SuccCont    -> Dialogue
echo            ::  Bool   ->           FailCont -> SuccCont    -> Dialogue
getArgs         ::                      FailCont -> StrListCont -> Dialogue
getProgName     ::                      FailCont -> StrCont     -> Dialogue
getEnv          ::  String ->           FailCont -> StrCont     -> Dialogue

abort     ::            FailCont
exit      ::            FailCont
interact  ::            (String->String) -> Dialogue
run       ::            (String->String) -> Dialogue
print     ::  Text a => a                -> Dialogue
prints    ::  Text a => a -> String      -> Dialogue
```

Bijlage F

Literatuur

Leerboeken met vergelijkbare stof

- Richard Bird en Philip Wadler: *Introduction to functional programming.* Prentice-Hall, 1988.
- Richard Bird en Philip Wadler: *Functioneel programmeren, een inleiding.* Academic service, 1991.
- A.J.T. Davie: *An introduction to functional programming systems using Haskell.* Cambridge university press, 1992.
- Ian Hoyler: *Functional programming with Miranda.* Pitman, 1991.
- Hudak en Fasel: 'a gentle introduction to Haskell'. *ACM sigplan notices* **27**, 5 (may 1992) pp.T1–T53.
- Simon Thompson: *Haskell, the craft of functional programming.* Addison-Wesley, 1996.
- Philip Wadler: *Introduction to functional programming, second edition.* Prentice-Hall, 1998.

Leerboeken waarin een andere programmeertaal gebruikt wordt

- Rinus Plasmeijer en Marko van Eekelen: *Functional programming and parallel graph rewriting.* Addison-Wesley, 1993.
- Åke Wikström: *Functional programming using standard ML.* Prentice-Hall, 1987.
- Chris Reade: *Elements of functional programming.* Addison-Wesley, 1989.
- Roger Bailey: *Functional programming with Hope.* Ellis-Horwood, 1990.
- Abelson en Sussman: *Structure and interpretation of computer programs.* McGraw-Hill, 1985.

Taalbeschrijvingen

- Hudak, Peyton-Jones, Wadler *et.al.*: 'Report on the programming language Haskell: a non-strict purely functional language, version 1.2.' *ACM sigplan notices* **27**, 5 (may 1992) pp.R1–R164.
- Mark P. Jones: *Introduction to Gofer 2.20.* Oxford Programming Research Group, 1992.

Implementatie van functionele programmeertalen

- Mark P. Jones: *The implementation of the Gofer functional programming system.* Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.
- Field en Harrison: *Functional programming.* Addison-Wesley, 1989.
- Simon Peyton-Jones: *The implementation of functional programming languages.* Prentice-Hall, 1987.
- Simon Peyton-Jones en David Lester: *Implementing functional languages: a tutorial.* Prentice-Hall, 1992.

Geavanceerd materiaal

- Johan Jeuring en Erik Meijer (eds): *Advanced functional programming.* Springer, 1985 (LNCS 925).
- Colin Runciman: *Applications of functional programming.* Cambridge university press, 1995.
- *Proceedings of the ... th conference on Functional Programming and Computer Architecture (FPCA).* Springer, 1992–1995.

Over parse-technieken

- W.H. Burge, 'Parsing'. In *Recursive Programming Techniques*, Addison-Wesley, 1975.
- Graham Hutton, 'Higher-order functions for parsing'. *J. Functional Programming* **2**:323–343.
- P. Wadler, 'How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages'. In *Functional Programming Languages and Computer Architecture*, (J.P.Jouannaud, ed.), Springer, 1985 (LNCS 201), pp. 113–128.
- Philip Wadler, 'Monads for functional programming'. In: *Advanced Functional Programming*, Tutorial text of the First international spring school on advanced functional programming techniques in Båstad, Sweden, may 1995. (Johan Jeuring and Erik Meijer, eds). Berlin, Springer 1995 (LNCS 925), pp. 24–52.
- Jeroen Fokker, 'Functional Parsers'. In: *Advanced Functional Programming*, Tutorial text of the First international spring school on advanced functional programming techniques in Båstad, Sweden, may 1995. (Johan Jeuring and Erik Meijer, eds). Berlin, Springer 1995 (LNCS 925), pp. 1–23.
- S.D.Swierstra and Luc Duponcheel, 'Deterministic, Error-Correcting Combinator Parsers'. In: *Advanced Functional Programming 2*, Tutorial text of the Second international summer school on advanced functional programming techniques, Olympia, Washington USA, august 1996. (John Launchbury, Erik Meijer and Tim Sheard, eds.). Berlin, Springer, 1996 (LNCS 1129), pp. 184-207.

Achterliggende theorie

- Henk Barendregt: *The lambda-calculus: its syntax and semantics.* North-Holland, 1984.
- Richard Bird: *An introduction to the theory of lists.* Oxford Programming Research Group report PRG–56, 1986.
- Luca Cardelli en P.Wegner: 'On understanding types, data abstraction and polymorphism.' *Computing surveys* **17**, 4 (1986).
- R. Milner: 'A theory of type polymorphism in programming.' *J.computer and system sciences* **17**, 3 (1978).

Tijdschriften

- *Journal of functional programming* (1991–). Cambridge University Press.
- `comp.lang.functional`. Usenet newsgroup.

Historisch materiaal

- Alonzo Church: 'The calculi of lambda-conversion'. *Annals of mathematical studies* **6**. Princeton university, 1941; Kraus reprint 1971.
- Haskell Curry *et al.*: *Combinatory logic, vol. I.* North-Holland, 1958.
- A. Fraenkel: *Abstract set theory.* North-Holland, 1953.
- M. Schönfinkel: 'Über die Bausteine der mathematischen Logik.' *Mathematischen Annalen* **92** (1924).

Bijlage G

Woordenlijst

!! operator die een bepaald element van een lijst selecteert. (par. 3.1.2, blz. 38)

== gelijkheidsoperator (par. 1.3.4, blz. 8)

/= ongelijkheidsoperator (par. 1.3.4, blz. 8)

++ operator die twee lijsten samenvoegt. (par. 3.1.2, blz. 38)

<= operator 'kleiner dan of gelijk aan'. (par. 1.3.4, blz. 8)

>= operator 'kleiner dan of gelijk aan'. (par. 1.3.4, blz. 8)

=> gereserveerd symbool in Haskell, dat gebruikt wordt in samenhang met klassen: in het type van een overloaded operator (par. 1.5.5, blz. 17), in een klasse-declaratie (par. 6.1.1, blz. 113), of in een instance-declaratie (par. 6.2.3, blz. 118).

Actuele parameter expressie die bij aanroep van een functie de waarde van de parameter vastlegt. (par. 1.4.3, blz. 10)

Additieve normaalvorm vorm waarin een expressie geschreven wordt als som van *termen, waarin geen optelling meer voorkomt, dus als *polynoom. (par. B.3, blz. 158)

Ambiguous dubbelzinnig, waarvan de waarde niet bepaald kan worden omdat de *associatievolgorde niet is gedefinieerd. (par. 2.1.3, blz. 22)

Assignment toekenningsopdracht, opdracht in een *imperatieve programmeertaal waarmee aan een variabele een bepaalde waarde wordt toegekend. De waarde van de variabele kan later door een andere toekennigsopdracht weer worden veranderd. (par. 1.1.1, blz. 1)

Associatievolgorde manier waarop de parameters van twee dezelfde operatoren in een expressie worden gegroepeerd. Associatie naar links: (a+b)+c; associatie naar rechts: a+(b+c). (par. 2.1.3, blz. 22)

Beschermd type *datatype met slechts één *constructorfunctie, die geheim wordt gehouden waardoor de elementen van het type alleen aan te spreken zijn via daarvoor bedoelde functies. (par. 3.4.3, blz. 64)

Bool type met als elementen de *waarheidswaarden True en False. (par. 1.3.4, blz. 8)

Boomstructuur element van een *datatype, dat een niet-lineaire structuur heeft. (par. 3.4.1, blz. 57)

Cel maat voor het benodigde geheugen bij het uitrekenen van de waarde van een *expressie in een functionele taal. (par. 5.1.4, blz. 90)

Char type met als elementen de letters, cijfers en leestekens zoals die op het toetsenbord van de computer voorkomen. (par. 1.5.2, blz. 15)

Coëfficiënt getal in een *term van een *polynoom. (par. 4.3.1, blz. 80)

Comprehensie *lijstcomprehensie. (par. 3.2.7, blz. 52)

concat Haskell-functie die een lijst van lijsten samenvoegt tot één lange lijst. (par. 3.1.2, blz. 38)

Concatenatie het samenvoegen van lijsten tot één lange lijst; in Haskell met de operator `++` (voor twee lijsten) of de functie `concat` (voor een lijst van lijsten). (par. 3.1.2, blz. 38)

Constructorfunctie functie waarmee een *datastructuur kan worden opgebouwd. In Haskell begint de naam van een constructorfunctie met een hoofdletter, of in het geval van *operatoren met een dubbele punt. (par. 3.4.1, blz. 57)

Continuatiefunctie Functie die aan een andere functie wordt meegegeven om aan te geven wat er moet gebeuren met een waarde die beschikbaar is gekomen. Wordt gebruikt bij interactieve invoer en uitvoer in Haskell. (par. 7.2.2, blz. 137)

Control-C toetsencombinatie waarmee in Haskell oneindig lang durende berekeningen onderbroken kunnen worden. (par. 2.3.2, blz. 27)

Currying het simuleren van een functie met meerdere parameters door een functie met één parameter die een functie oplevert. (par. 2.2.1, blz. 23)

Datastructuur element van een *datatype. (par. 3.4.1, blz. 57)

Datadefinitie definitie waarmee een *datatype wordt geïntroduceerd, door van alle *constructorfuncties de types van de parameters op te sommen. (par. 3.4.1, blz. 57)

Datatype type dat gekenmerkt wordt door de manier waarop elementen kunnen worden opgebouwd (door middel van *constructorfuncties). (par. 3.4.1, blz. 57)

Default-definitie definitie in Haskell voor een overloaded functie die wordt gebruikt indien de functie niet in de instance-declaratie wordt gedefinieerd. De default-definitie wordt gegeven in de klasse-declaratie. (par. 6.2.1, blz. 116)

Derive afleiden. 'cannot derive instance' betekent in Haskell dat een overloaded operator wordt gebruikt op een type dat geen instance is van de betreffende klasse. (par. 6.2.5, blz. 120)

Dialogue Type in Haskell waarmee interactieve programma's kunnen worden gemodelleerd. (par. 7.2.1, blz. 135)

Differentiëren *numeriek of *symbolisch differentiëren.

Disjoint union Engels voor *vereniging van typen. (par. 3.4.1, blz. 57)

Eindig type *datatype waarin alle *constructorfuncties nul parameters hebben. (par. 3.4.3, blz. 64)

Enum *klasse van typen in Haskell waarop functies zoals `enumFromTo` zijn gedefinieerd. (par. 6.2.4, blz. 119)

Eq *klasse van typen in Haskell waarop (on)gelijkheid is gedefinieerd. (par. 6.1.1, blz. 113)

Expressie constructie in een programmeertaal, die een waarde heeft. (par. 1.2.1, blz. 2)

Expressieboom *ontleedboom van een (rekenkundige of andersoortige) expressie. (par. 7.1.1, blz. 133)

Faculteit functie die het product bepaalt van alle natuurlijke getallen tussen 1 en zijn parameter. (par. 1.2.2, blz. 3)

Float type met als elementen de *floating-point getallen. (par. 1.3.3, blz. 7)

Floating-point getallen deelverzameling van de reële getallen die op een machine met een beperkte rekennauwkeurigheid kunnen worden opgeslagen. In Haskell aangeduid met *Float. (par. 1.3.3, blz. 7)

foldr Haskell-functie die een operator tussen alle elementen van een lijst plaatst, te beginnen aan de rechterkant met een bepaald element. (par. 3.1.3, blz. 42)

Formele parameter aanduiding van de veranderlijke van een functie bij de definitie ervan. In de meeste programmeertalen is een formele parameter een variabele, in Haskell mag het echter een *patroon zijn. (par. 1.4.3, blz. 10)

fromInteger Haskell-functie die een geheel getal converteert naar een ander numeriek type. (par. 6.1.4, blz. 115)

Functie verband tussen de *parameters en het resultaat van bepaalde processen. (par. 1.1.1, blz. 1)

Functionele programmeertaal programmeertaal waarbij een programma bestaat uit de definitie van een aantal *functies, waarvan de resultaatwaarde steeds geheel bepaald wordt door de parameters. (par. 1.1.1, blz. 1)

Gecurryde functie functie met één parameter die weer een functie oplevert, en daarom beschouwd kan worden als een functie met meerdere parameters. (par. 2.2.1, blz. 23)

Gehele getallen de verzameling Z van (positieve en negatieve) getallen zonder breukdeel. In Haskell aangeduid met *Int (begrende versie) of *Integer (onbegrensde versie). (par. 1.3.3, blz. 7)

Gereserveerd woord woord met een speciale betekenis in een programmeertaal, dat daarom niet als naam van een variabele gebruikt mag worden. Voorbeeld in C: 'if'; voorbeeld in Haskell: 'where'. (par. 1.3.2, blz. 6)

Gevalsonderscheid methode waarmee een functie in verwschillende gevallen op een verschillende manier gedefinieerd kan worden. Gevalsonderscheid is in Haskell mogelijk door middel van *voorwaarden

of door middel van *patronen. (par. 1.4.2, blz. 10)

Haskell *functionele programmeertaal, beschreven in 1992 door het Haskell-comité.

Hogere-orde functie functie met een functie als parameter en/of als resultaat. (par. 2.3.1, blz. 25)

Hugs Compiler voor Haskell, ontwikkeld vanaf in 1992 door Mark Jones. Zie `www.cs.nott.ac/~mpj`.

Imperatieve programmeertaal programmeertaal gekenmerkt door de aanwezigheid van *toekenningsopdrachten, die na elkaar worden uitgevoerd. Voorbeelden zijn Pascal en C. (par. 1.1.2, blz. 1)

Inductieve definitie *recursieve definitie. (par. 1.4.4, blz. 12)

Ingebouwde functie *primitieve functie.

Instance type dat lid is van een bepaalde *klasse (par. 6.1.2, blz. 113)

Instance-declaratie declaratie waarmee een type toegevoegd wordt aan een klasse, door de vereiste functies er op te definiëren. (par. 6.1.2, blz. 113)

Int type met als elementen een begrensde deelverzameling van de *gehele getallen. (par. 1.3.3, blz. 7)

Integer type met als elementen (bijna) alle *gehele getallen. (par. 1.3.3, blz. 7)

interact Haskell-functie die een bepaalde functie interactief toepast op de invoer. In tegenstelling tot de functie `run` wordt daarbij steeds gewacht op een hele regel invoer, en wordt deze regel ook op het scherm getoond. (par. 7.2.1, blz. 135)

Interpreter computerprogramma dat programma's in een bepaalde programmeertaal uitvoert. (par. 1.2.1, blz. 2)

Invoegen op de juiste plaats zetten, bijvoorbeeld in een gesorteerde lijst (par. 3.1.4, blz. 44) of in een zoekboom (par. 3.4.2, blz. 59)

iterate Haskell-functie die de oneindige lijst bepaalt waarin een functie steeds vaker op een startwaarde wordt toegepast. (par. 3.2.6, blz. 49)

Klasse groep typen waarop bepaalde *overloaded functies en operatoren kunnen worden toegepast. (par. 6.1.2, blz. 113)

Klasse-declaratie declaratie waarmee gespecificeerd wordt welke operatoren op de typen in een bepaalde klasse moeten worden gedefinieerd. (par. 6.1.2, blz. 113)

Lazy evaluatie berekeningsvolgorde waarbij de parameter van een functie pas wordt uitgerekend op het moment dat hij nodig is, en niet direct bij aanroep van een functie. (par. 3.2.5, blz. 49)

Lege lijst *lijst met nul elementen. (par. 3.1.1, blz. 37)

Lexicografische ordening Ordening van lijsten waarbij het eerste element bepalend is, tenzij het eerste element gelijk is; in dat geval beslist het tweede element, tenzij dat ook gelijk is, enzovoorts. Als een van de twee lijsten een beginstuk is van de andere, dan is de lengte bepalend. (par. 3.1.2, blz. 38)

Lijst Type waarvan het aantal elementen kan variëren, maar waarvan de elementen allen hetzelfde type hebben. (par. 3.1.1, blz. 37)

Lijstcomprehensie notatie waarmee een lijst wordt opgebouwd door een expressie te evalueren in een omgeving waarbij een variabele alle waarden van een lijst doorloopt. (par. 3.2.7, blz. 52)

Lisp *functionele programmeertaal, ontwikkeld in 1958 door John McCarthy. Afkorting van 'list processor'. Vooral populair geworden in het toepassingsgebied 'kunstmatige intelligentie'. (par. 1.1.2, blz. 1)

Lokale definitie definitie van een constante of een functie die alleen gebruikt kan worden binnen een andere functie. (par. 1.4.1, blz. 9)

main In Haskell: waarde van het type *`Dialogue` die gedefinieerd moet worden in een Haskell-programma dat gecompileerd zal worden. (par. 7.2.4, blz. 138)

map Haskell-functie die een functie toepast op alle elementen van een lijst. (par. 3.1.3, blz. 42)

Num *klasse van numerieke typen, waarop operatoren `+`, `-`, `*` en `/` gedefinieerd zijn, en functies `neg` en `fromInteger`.

Numeriek differentiëren door middel van numerieke benadering de afgeleide functie evalueren in een bepaald punt. (par. 2.4.2, blz. 31)

Oneindige lijst lijst waarvan steeds meer elementen bepaald kunnen worden; het werken met oneindige lijsten is mogelijk vanwege *lazy evaluatie. (par. 3.2.4, blz. 48)

Ontleden bepalen van de *ontleedboom van een taalconstructie die als string gegeven is. (par. 7.3.3, blz. 141)

Ontleedboom *datastructuur waarmee de opbouw van een taalconstructie wordt weergegeven. (par. 7.3.3, blz. 141)

Operator *functie waarvan de naam tussen zijn twee *parameters wordt geschreven in plaats van ervoor. (par. 2.1.1, blz. 21)

Ord *klasse van typen in Haskell waarvan de elementen ordenbaar zijn. (par. 6.2.4, blz. 119)

Overlapping instances probleem dat in Haskell optreedt als hetzelfde type op twee manieren tot instance van dezelfde klasse wordt gemaakt. (par. 6.2.5, blz. 120)

Overloading het gebruik van een functie- of operatornaam voor verschillende operaties op verschillende typen. (par. 1.5.5, blz. 17)

Paar *tupel bestaande uit twee elementen. (par. 3.3.1, blz. 53)

Parameter veranderlijke van een *functie. Bij de definitie van een functie wordt de veranderlijke aangeduid door een *formele parameter, bij gebruik van de functie wordt de waarde van de veranderlijke vastgelegd door een *actuele parameter. (par. 1.1.1, blz. 1)

Partieel parametriseren het aanroepen van een functie met minder parameters dan deze verwacht. (par. 2.2.1, blz. 23)

Patroon speciale vorm van *formele parameter, waarmee eenvoudig gevalsonderscheid kan worden gepleegd bij de definitie van een functie. (par. 1.4.3, blz. 10)

Polymorfe functie functie met een *polymorf type, die dus op parameters van verschillende typen, die echter een bepaalde structuur gemeen hebben, mag worden toegepast. (par. 1.5.3, blz. 16)

Polymorf type type in de aanduiding waarvan een *type-variabele voorkomt. In feite dus een groep typen, die een bepaalde structuur gemeen hebben. (par. 1.5.3, blz. 16)

Polynoom som van termen, waarbij elke term bestaat uit een produkt van een reëel getal (de coëfficiënt) en een natuurlijke macht (de exponent) van een variabele. (par. 4.3.1, blz. 80)

Prelude verzameling *voorgedefinieerde functies die in alle programma's mogen worden gebruikt. (par. 1.2.1, blz. 2)

Primitief type *type die de *interpreter kent zonder dat er een definitie voor gegeven is. In Haskell een van de vier typen *Int, *Float, *Bool en *Char. (par. 1.5.2, blz. 15)

Primitieve functie *functie die een *interpreter kent zonder dat er een definitie voor gegeven is. De functie is dus 'ingebouwd' in de interpreter. (par. 1.3.1, blz. 5)

Prioriteit rangorde die aangeeft in welke volgorde van elkaar verschillende operatoren worden uitgerekend. (par. 2.1.2, blz. 21)

Rationaal getal breuk. (par. 3.3.3, blz. 55)

Recursieve definitie definitie van bijvoorbeeld een functie of een type, waarbij het te definiëren begrip bij de definitie al wordt gebruikt. (par. 1.4.4, blz. 12)

Reduction maat voor de benodigde tijd bij het uitrekenen van de waarde van een *expressie in een *functionele taal. (par. 5.1.1, blz. 85)

Rekenkundige functie functie die werkt op getallen, bijvoorbeeld 'optellen' of vermenigvuldigen'. (par. 6.1.1, blz. 113)

run Haskell-functie die een bepaalde functie interactief toepast op de invoer. In tegenstelling tot de functie `interact` wordt de functie direct op een karakter toegepast zodra die beschikbaar is, en worden de ingetypte karakters niet getoond. (par. 7.2.1, blz. 135)

Sectie notatie waarbij een operator, door hem tussen haakjes te zetten, tot functie gemaakt wordt. (par. 2.2.3, blz. 25)

Singleton-lijst *lijst met één element. (par. 3.1.1, blz. 37)

Sorteren op (opklimmende) volgorde zetten van de elementen (van een lijst). (par. 3.1.4, blz. 44)

Sqrt afkorting van 'square root', het Engelse woord voor 'vierkantwortel'. (par. 1.2.1, blz. 2)

String *lijst waarvan de elementen het type *Char hebben. (par. 3.2.1, blz. 45)

Symbolisch differentiëren door middel van manipulatie van de *ontleedboom van een functiedefinitie de ontleedboom van de afgeleide functie bepalen. (par. 7.1.2, blz. 133)

Term deel van een *expressie. (par. 1.5.1, blz. 14) produkt van factoren, bijvoorbeeld in een polynoom (par. 4.3.1, blz. 80) of in een expressie (par. 7.1.1, blz. 133)

Tupel samengesteld type bestaande uit een vast aantal waarden, die echter verschillende typen mogen hebben. (par. 3.3.1, blz. 53)

Type waardenverzameling waarbinnen een functie zijn waarde kan hebben. (par. 1.5.2, blz. 15)

Typedefinitie regel in een programma waarmee ter afkorting een naam gegeven kan worden aan een bepaald type. (par. 3.3.2, blz. 54)

Typeringsfout fout die optreedt als in een expressie functies worden toegepast op parameters die niet het juiste type hebben. (par. 1.5.1, blz. 14)

Typevariabele variabele in een type-aanduiding, waardoor de type-aanduiding in feite een groep typen wordt aangeduidt die allen een bepaalde structuur gemeen hebben (een *polymorf type). (par. 1.5.3, blz. 16)

Unresolved overloading probleem dat in Haskell optreedt als het resultaat van overloaded functies weer gebruikt wordt als parameter van een andere overloaded functie. (par. 6.2.5, blz. 120)

until Haskell-functie die net zolang een functie toepast op een startwaarde totdat aan een bepaalde eigenschap is voldaan. (par. 2.3.2, blz. 27)

Vereenvoudigen (van *polynoom): sorteren van de termen, samenvoegen van termen met gelijke *coëfficiënt, en verwijderen van termen met coëfficiënt nul. (par. 4.3.2, blz. 81) (van *rationaal getal): delen van teller en noemer van een breuk door hun grootste gemene deler, en verplaatsen van een eventueel minteken naar de teller. (par. 3.3.3, blz. 55)

Vereniging van typen *datatype waarin elke *constructorfunctie slechts één, niet-recursieve parameter heeft. Engels: 'disjoint union'. (par. 3.4.3, blz. 64)

Voorgedefinieerde functie *functie die in de *prelude wordt gedefinieerd, en dus in alle programma's gebruikt mag worden. (par. 1.3.1, blz. 5)

Voorwaarde logische expressie, die aangeeft dat een functie alleen op een bepaalde manier wordt gedefinieerd als die expressie de waarde True heeft. Door meerdere voorwaarden in een functiedefinitie te zetten kan *gevalsonderscheid worden gemaakt. (par. 1.4.2, blz. 10)

Waarheidswaarden verzameling met de elementen 'True' en 'False', die de mogelijke uitkomsten vormen van een logische *expressie. In Haskell aangeduid met *Bool. (par. 1.3.4, blz. 8)

zip Haskell-functie die van twee lijsten een lijst tweetupels maakt. (par. 3.3.4, blz. 56)

Zoekboom *binaire boom die 'gesorteerd' is, in de zin dat alle elementen in de linker deelboom kleiner zijn dan de waarde die op het splitspunt is opgeslagen, en alle elementen in de rechter deelboom groter. (par. 3.4.2, blz. 59)

Index