Introductie tot Haskell

Leereenheid 1

Introductie tot Haskell

INTRODUCTIE

De auteur van het tekstboek bij deze cursus legt een aantal concepten van programmeertalen uit aan de hand van voorbeelden. In sommige voorbeelden gebruikt hij de talen C, C++, Ada en Java. Deze voorbeelden zult u goed kunnen volgen als u enige programmeerervaring heeft in een imperatieve taal als Pascal of een objectgeoriënteerde taal als Java. In andere voorbeelden wordt de functionele taal Haskell gebruikt. Om deze voorbeelden te kunnen waarderen, is het noodzakelijk om van de basisprincipes van functioneel programmeren op de hoogte te zijn.
Deze eerste leereenheid vormt een kennismaking met een functionele taal. Haskell en ML zijn twee moderne functionele programmeertalen. In deze cursus is gekozen voor de taal Haskell, de taal die in het tekstboek wordt gebruikt.
In deze inleidende leereenheid ligt de nadruk op het gebruik van de Haskell-interpretator WinHugs (een Windows-versie van Hugs 98 die in het cursuspakket bijgeleverd is) en enkele veel voorkomende taalconstructies. Later zult u deze constructies herkennen als ze in het tekstboek gebruikt worden en kunt u deze desgewenst ook zelf uitproberen met de Haskell-interpretator.

De functionele programmeerstijl en de taal Haskell in het bijzonder worden in blok 3 (leereenheden 10 t/m 12) diepgaander behandeld. In die leereenheden zult u ook gevoel ontwikkelen voor wat grotere functionele programma's. In deze leereenheid gaat het echter slechts om een kennismaking met de functionele programmeerstijl en de bijbehorende notaties.
Overigens kunt u – als u dit wilt – zonder veel problemen na deze eerste leereenheid de leereenheden 10 en 11 uit blok 3 bestuderen. U heeft dan wat meer ervaring met functioneel programmeren voordat u begint met de leereenheden 2 t/m 9, waarin regelmatig voorbeelden in een functionele programmeertaal worden gegeven. Met leereenheid 12 kunt u beter wachten tot na de blokken 1 en 2 van de cursus.

LEERDOELEN
Na het bestuderen van deze leereenheid wordt verwacht dat u
- de Haskell-interpretator WinHugs kunt gebruiken
- enkele verschijningsvormen van functies in eigen woorden kunt omschrijven: recursieve functies, hogere-ordefuncties, polymorfe functies, constructor-functies en 'gecurryde' functies
- het verschil en de overeenkomst kent tussen functies en operatoren
- verschillende vormen van functiedefinities kunt schrijven: functiedefinities met voorwaarden, functiedefinities met patronen, recursieve functiedefinities en functiedefinities die gebruik maken van hogere-ordefuncties
- het type van eenvoudige functies kunt bepalen

– het begrip 'datastructuur' kent en functies kunt definiëren die op een bepaalde datastructuur werken
– het verschil kent tussen typedefinities en datadeclaraties.

Studeeraanwijzingen
Deze leereenheid is bedoeld als kennismaking met de belangrijkste begrippen en notaties. Voor een goed begrip kunt u de gegeven voorbeelden het best direct uitproberen met behulp van de Haskell-interpretator. Nog beter is het om zelf kleine variaties aan te brengen in de voorbeelden, om te kijken of de interpretator zo reageert als u verwacht.

Installatie EditPlus en WinHugs

Voordat u deze leereenheid bestudeert, moet u de tekst-editor EditPlus en de Haskell-interpretator WinHugs op uw computer installeren. Informatie over de installatie vindt u op Studienet (Blackboard).

De studielast van deze leereenheid bedraagt circa 8 uur.

LEERKERN

1 Functioneel programmeren

1.1 COMPUTERS EN BEREKENINGEN

In de jaren veertig van de vorige eeuw 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 ingelezen, waardoor de eerste *programmeertalen* hun intrede deden.
Omdat destijds het gebruik van een computer vreselijk duur was, lag het voor de hand dat een programmeertaal zo veel mogelijk aansloot bij de architectuur van de computer. Een computer bestaat uit een besturingseenheid en een veranderbaar geheugen. Daarom bestaat een programma uit instructies om het geheugen te veranderen. Deze instructies worden uitgevoerd door een besturingseenheid. Zo is de *imperatieve programmeerstijl* ontstaan. Imperatieve programmeertalen, zoals Pascal en Ada, kenmerken zich – evenals objectgeoriënteerde talen – door de aanwezigheid van *toekenningsopdrachten* (assignments), die na elkaar worden uitgevoerd. Een toekenningsopdracht is een opdracht, waarmee een waarde toegekend kan worden aan een variabele die een geheugenlocatie representeert.

Ook voordat er computers bestonden, zijn er natuurlijk al wiskundige 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.

Imperatieve programmeerstijl

Bij elke berekening hangt het 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

Functionele programmeerstijl

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 is 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.

1.2 FUNCTIONELE TALEN

De theoretische basis voor het imperatief programmeren is al in de jaren dertig gelegd door Alan Turing (in Engeland) en John von Neumann (in de VS). Ook de theorie van functies als berekeningsmodel stamt uit de jaren twintig en dertig. Grondleggers zijn onder andere Moses Schönfinkel in Duitsland en Haskell Curry in Engeland.
Het heeft tot het begin van de jaren vijftig geduurd voordat iemand op het idee kwam om deze theorie daadwerkelijk als basis voor een programmeertaal te gebruiken. De taal LISP (een afkorting van 'list programming') van John McCarthy was de eerste functionele programmeertaal en is ook jarenlang de enige gebleven. Hoewel LISP nog steeds wordt gebruikt, is dit niet een taal die voldoet aan de huidige opvattingen over eigenschappen van programmeertalen. Met het toenemen van de complexiteit van computerprogramma's wordt de behoefte aan een sterkere controle van het programma door de computer steeds groter. Het gebruik van *typering* speelt daarbij een grote rol. In de jaren zeventig en tachtig zijn dan ook een groot aantal getypeerde functionele programmeertalen ontstaan: ML, Scheme (een aanpassing van LISP), Hope en Miranda zijn een paar voorbeelden.

Typering

Op den duur gaat elke onderzoeker zijn eigen taal ontwikkelen. Om deze wildgroei in talen enigzins te beteugelen, is door een aantal vooraanstaande onderzoekers een nieuwe taal ontworpen, waarin getracht is al het goede uit de verschillende talen te verenigen. De eerste implementaties van deze taal, Haskell genaamd, zijn in het begin van de jaren negentig gemaakt. Door de goede beschikbaarheid – onder andere op pc's – is deze taal snel populair geworden.

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 is door de moeilijke verkrijgbaarheid – het is een commercieel product – niet erg gangbaar geworden.

2 Werken met Haskell

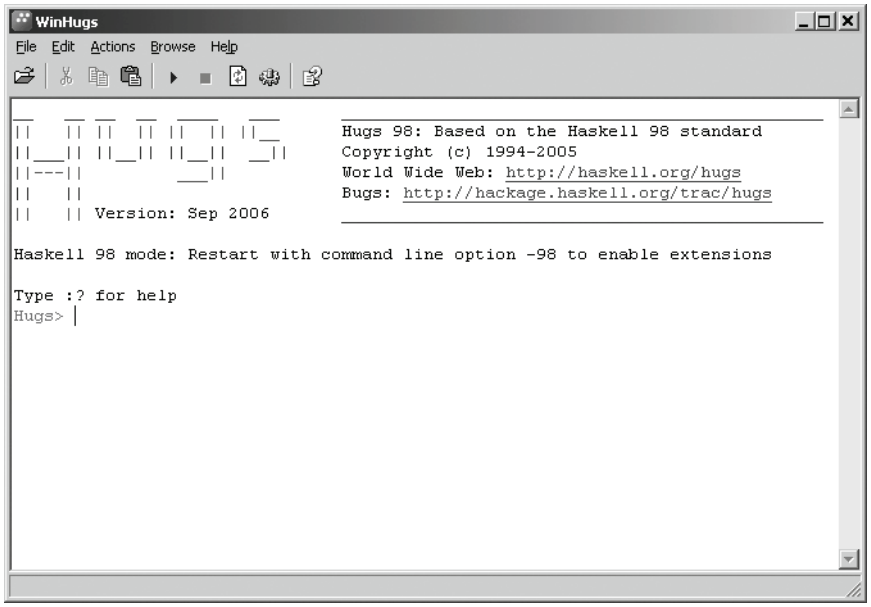
2.1 START VAN DE INTERPRETATOR

In een functionele programmeertaal worden functies geïntroduceerd door middel van functiedefinities. Deze functies zijn bedoeld om te gebruiken in expressies waarvan de waarden uitgerekend moeten

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 *interpretator*.
Voor de taal Haskell die we in deze cursus gebruiken, is de Haskell-interpretator WinHugs beschikbaar.

Interpretator

Start de interpretator. Op het scherm verschijnt het venster van figuur 1.1.



FIGUUR 1.1 De Haskell-interpretator WinHugs

Notatieafspraak

Om het antwoord van de interpretator te onderscheiden van de ingetikte tekst, zullen we de tekst die de interpretator afdrukt in het cursusboek onderstrepen.

Prelude

Bij de start laadt de Haskell-interpretator een module met standaardfuncties, de zogenaamde standaardprelude in het bestand Prelude.hs. Dit bestand bevindt zich in de submap\packages\base\ in de installatiemap van WinHugs. Prelude betekent letterlijk 'voorspel'. Alle functies die in de standaardprelude zijn gedefinieerd, kunnen nu gebruikt worden.

Ten slotte meldt de interpretator dat u ':?' kunt tikken voor een korte gebruiksaanwijzing:

```
Type :? for help
Hugs>
```

Hugs> op de nieuwe regel is de prompt van Haskell: deze geeft aan dat de interpretator nu klaarstaat om de waarde van een door u in te tikken expressie te berekenen. Voor het berekenen van de expressie worden de functiedefinities in de module Prelude gebruikt.

2.2 UITREKENEN VAN EXPRESSIES

Doordat in de prelude onder andere rekenkundige functies gedefinieerd zijn, is de interpretator nu te gebruiken als rekenmachine:

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

De interpretator berekent de waarde van de ingetikte expressie, waarbij * vermenigvuldiging aanduidt. Hierna wordt het resultaat (11) meegedeeld. De prompt geeft aan dat de interpretator klaarstaat voor de volgende expressie.

De van de rekenmachine bekende functies kunt u ook gebruiken in een expressie:

```
Hugs> sqrt 2
1.4142135623731
Hugs>
```

De functie sqrt berekent de vierkantswortel van een getal. Omdat functies in een functionele taal zoveel gebruikt worden, hoeven er geen haakjes om 2 te staan: het is zo ook al duidelijk dat de functie sqrt wordt aangeroepen met 2 als parameter. In ingewikkelder expressies scheelt dat een heleboel haakjes en dat maakt zo'n expressie een stuk overzichtelijker.

Lijst

Grote hoeveelheden getallen kunnen in Haskell in een *lijst* worden geplaatst. Lijsten noteert u met behulp van rechte haken. Er zijn een aantal standaardfuncties die op lijsten werken:

```
Hugs> sum [1,3,7,5,6]
22
Hugs>
```

2.3 FUNCTIES DEFINIËREN

Functiedefinities opgenomen in een module en opgeslagen in een bestand

Zoals in vrijwel alle talen is het in Haskell mogelijk om zelf nieuwe functies te definiëren. De functies kunnen daarna – samen met de standaardfuncties uit de prelude – gebruikt worden in expressies. Definities van een functie zijn altijd opgenomen in een module. Zo'n module is niets anders dan een verzameling functiedefinities die is opgeslagen in een bestand. Dit bestand kunt u aanmaken met een tekst-editor naar keuze. Normaal gesproken is dit EditPlus, maar via het menu File | Options kunt u een andere tekstverwerker kiezen. De tekstverwerker wordt gestart door ':edit' in te tikken, gevolgd door de naam van een bestand, bijvoorbeeld:

```
Hugs> :edit D:\Examples\Nieuw.hs
```

Door de dubbele punt aan het begin van de regel weet de interpretator dat edit geen functie is die uitgerekend moet worden, maar een 'huishoudelijke mededeling'. Het is een conventie dat bestanden met Haskell-programma's de extensie 'hs' hebben. Naast de interpretator wordt de tekstverwerker gestart en u krijgt de mogelijkheid om een

bestand aan te maken met de opgegeven naam in de opgegeven directory. In het bestand Nieuw.hs kunt u bijvoorbeeld de definitie zetten van de kwadrateringsfunctie met de naam kwadraat:

```
module Nieuw where
kwadraat x = x*x
```

De eerste regel bevat de naam van de module, deze moet altijd beginnen met een hoofdletter. Het is een conventie om deze modulenaam te laten overeenkomen met de bestandsnaam. De vetgedrukte woorden zijn gereserveerde woorden van de taal Haskell. De tweede regel bevat de definitie van de functie kwadraat. Als de functie is ingetikt, kunt u de tekst opslaan voordat u teruggaat naar WinHugs.
Voordat de nieuwe functie kan worden gebruikt, moet Haskell weten dat het nieuwe bestand functiedefinities bevat. Dat kunt u vastleggen met de huishoudelijke mededeling ':load', dus:

```
Hugs> :load D:\Examples\Nieuw.hs
Nieuw>
```

Eenvoudiger gaat het laden van een module via de menu-optie File | Open.

Na het analyseren van het nieuwe bestand krijgt de prompt de naam van de geladen module. Naast de functies uit Prelude.hs kunnen nu ook de functies die in het bestand Nieuw.hs gedefinieerd zijn, gebruikt worden. De nieuw gedefinieerde functie kan nu in expressies gebruikt worden:

```
Nieuw> kwadraat 4
16
Nieuw>
```

Het is mogelijk om later functiedefinities aan het laatstgebruikte bestand toe te voegen. Het is dan voldoende om alleen ':edit' te tikken, de naam van het bestand hoeft u dan niet meer te noemen.

OPGAVE 1.1
Definieer een functie derde die de derde macht van een getal berekent. Plaats de functiedefinitie in een bestand en laat dit bestand analyseren door de Haskell-interpretator. Bereken met de interpretator de waarde van $17^3 - 4^3$.

Op Studienet vindt u een zip-bestand met daarin alle voorbeeldprogramma's uit de cursus. De programma's uit deze leereenheid zijn opgenomen in het bestand Le01.hs in de directory Examples. De Haskell-programma's uit leereenheid 2 staan bijvoorbeeld in het bestand Le02.hs. Het zip-bestand moet u in een installatiemap uitpakken. Daarna kunt u de functies die in een van de bestanden worden gedefinieerd, gebruiken door het bestand in WinHugs te laden via het menu File | Open.

2.4 OPDRACHTEN AAN DE INTERPRETATOR

Opdrachten interpretator beginnen met dubbele punt

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

De volgende opdrachten worden het meest gebruikt:
`:?`
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 (u hoeft niet alles te weten, als u maar weet hoe u het kunt vinden).
`:quit`
Met deze opdracht wordt een Haskell-sessie afgesloten.
`:load` *bestand(en)*
Na deze opdracht kent Haskell de functies die in de modulen in de gespecificeerde bestanden zijn gedefinieerd. Met :load zonder bestandsnamen erachter vergeet Haskell alles, behalve de prelude.
`:edit` *bestand*
Dit is een opdracht om het genoemde bestand te creëren of te veranderen. Als u de bestandsnaam weglaat, wordt het bestand bewerkt dat het laatst is geladen – met de naam van de prompt – en na het afsluiten van de tekstverwerker automatisch opnieuw geladen.
`:find` *naam*
Met deze opdracht wordt de tekstverwerker gestart met het bestand waarin de genoemde naam is gedefinieerd.
`:set` ± *letter*
Met deze opdracht kunnen een aantal opties in- of uitgeschakeld worden. Opties zijn bijvoorbeeld:
– s: statistische informatie (aantal reducties en cellen die een indicatie geven van tijd- en geheugengebruik) na elke berekening
– t: geef het type van elke uitkomst van een berekening (zie paragraaf 6.1).
De opdrachten mogen worden afgekort tot hun eerste letter, bijvoorbeeld :q voor :quit.

3 Functies en operatoren

3.1 INGEBOUWDE EN VOORGEDEFINIEERDE FUNCTIES

In de prelude worden ruim tweehonderd standaardfuncties en -operatoren gedefinieerd. Het grootste deel van de prelude bestaat uit gewone functiedefinities, zoals u die ook zelf had kunnen schrijven. De functie sum bijvoorbeeld is alleen maar in de prelude opgenomen omdat deze zo vaak gebruikt wordt; als deze hierin niet gedefinieerd zou zijn, dan had u er zelf een definitie voor kunnen schrijven. Een functiedefinitie is te bekijken met de opdracht :find (zie paragraaf 2.4). 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 u natuurlijk wel weten wat de standaardfunctie foldl' doet, maar ook dat is op te zoeken ...

Primitieve functies

Voorgedefinieerde functies

Andere functies, zoals de functie primPlusInt die twee gehele getallen optelt, kunt u niet zelf definiëren; ze zitten op een 'magische' manier ingebouwd in de interpretator. Dit soort functies noemen we *primitieve functies* (primitive functions). Het aantal primitieve functies in de prelude is zo klein mogelijk gehouden. De meeste standaardfuncties zijn gewoon in de taal Haskell gedefinieerd. Deze functies noemen we *voorgedefinieerde functies* (predefined functions).

3.2 NAMEN VAN FUNCTIES

In de functiedefinitie

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

is kwadraat de naam van een functie die gedefinieerd wordt en x de naam van de parameter daarvan.

Notatie

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, de apostrof (') en het liggend streepje (_) (underscore). Kleine letters en hoofdletters worden als verschillende letters beschouwd. Een paar voorbeelden van mogelijke functie- of parameternamen zijn:

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

Gereserveerde woorden

Namen die met een hoofdletter beginnen, worden voor speciale functies en constanten gebruikt, de zogenaamde *constructor-functies*. Wat dat zijn, beschrijven we in paragraaf 7.1.

Er zijn namen die we niet voor functies of variabelen mogen gebruiken. Deze *gereserveerde woorden* hebben een speciale betekenis voor de interpretator. Dit zijn de gereserveerde woorden in Haskell:

```
case        class       data        default
deriving    do          else        if
import      in          infix       infixl
infixr      instance    let         module
newtype     of          then        type
where       as          qualified   hiding
```

Ook het liggend streepje '_' maakt deel uit van de gereserveerde woorden. In programmafragmenten zullen we deze woorden vet afdrukken. De betekenis van de meeste gereserveerde woorden komt later in de cursus aan de orde.

3.3 OPERATOREN

Operator versus functie

Naast functies (die gebruikmaken van prefixnotatie) bestaan er in Haskell ook *operatoren* (die gebruikmaken van infixnotatie). Er is een aantal primitieve operatoren, maar operatoren kunt u ook zelf definiëren. Er zijn maar drie verschillen tussen een operator en een functie:
– de naam van een operator wordt tussen de parameters geschreven in plaats van ervoor

– een operator wordt aangeduid met symbolen in plaats van met letters en cijfers
– een operator (behalve de negatie '–') heeft altijd precies twee parameters, een functie kan daarnaast ook nul, een of meer dan twee parameters hebben.

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:

Gereserveerde operatoren

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

Enkele operatoren in prelude

```
+  ++  &&  ||  <=  ==  /=  .
```

De operatoren op de eerste van deze twee regels zijn gereserveerde operatoren in Haskell. Op de tweede regel staan enkele operatoren die in de prelude zijn gedefinieerd. Operatoren die met een dubbele punt beginnen, zijn bestemd voor zogeheten constructor-operatoren, die we in paragraaf 7 bespreken.

Gereserveerde symbolen-combinaties

De gereserveerde operatoren en de symboolcombinatie '--' mogen we niet als operator gebruiken, omdat ze een speciale betekenis hebben in Haskell, zoals we de gereserveerde woorden niet als functienaam mogen gebruiken.
Er blijven echter genoeg combinaties van symbolen over om nieuwe operatoren te definiëren.

Operator als functie of andersom

Soms is het gewenst om de naam van een operator toch vóór de parameters te schrijven, of een functienaam juist 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 apostrof. Op sommige pc-toetsenborden komt dit symbool niet voor; het is dan in te tikken door de Alt-toets in te drukken en tegelijkertijd een 9 en een 6 te tikken op het numerieke toetsenblok.

Het is dus toegestaan (+) 1 2 te schrijven in plaats van 1 + 2. 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 3 `totDeMacht` 5 leest nu eenmaal makkelijker dan totDeMacht 3 5. Dit kan natuurlijk alleen als de functie minstens twee parameters heeft.

4 Functiedefinities

4.1 FUNCTIES EN OPERATOREN OP GETALLEN

Getallen

Er zijn twee soorten letterlijke *getalwaarden* (numeric literals, getal in fixed- of floating point notatie) beschikbaar in Haskell:

– integers (gehele getallen), zoals 17, 0 en –3
– floats (drijvende-puntgetallen), zoals 2.5, –7.81, 0.0, 1.2e3 en 0.5e–5.

Rekenkundige operatoren

De vier *rekenkundige operatoren* optellen (+), aftrekken (–), vermenigvuldigen (*) en delen (/) zijn zowel op gehele getallen als op drijvende-puntgetallen te gebruiken. De twee typen letterlijke waarden kunnen door elkaar worden gebruikt:

```
Hugs> 5-12
-7
Hugs> 2.5*3
7.5
Hugs> 19/4
4.75
Hugs> 4/0.75
5.33333333333333
Hugs>
```

Gehele deling

Voor *gehele deling* zijn twee functies beschikbaar: quot en div. Deze functies zijn alleen toe te passen op gehele getallen. Door de naam tussen back quotes te zetten kunnen we de functie als operator gebruiken. Bij gehele deling wordt het gedeelte achter de punt verwaarloosd. Als de deler negatief is, wordt door quot in de richting van 0 afgerond en door div in de richting van oneindig:

```
Hugs> 23`quot`-3
-7
Hugs> -23`quot`-3
7
Hugs> 23`div`-3
-8
Hugs> -23`div`-3
8
Hugs>
```

Bij een positieve deler geven quot en div hetzelfde resultaat:

```
Hugs> -23`quot`3
-7
Hugs> -23`div`3
-7
Hugs>
```

Operator <

De *operator* < is een vergelijkingsoperator die kijkt of een getal kleiner is dan een ander getal. Ook bij deze operator kunnen we de twee typen letterlijke waarden door elkaar gebruiken. De uitkomst is de constante True (als dat inderdaad zo is) of de constante False (als dat niet het geval is):

```
Hugs> 1.5 < 2
True
Hugs> 1 < 0.5
False
Hugs>
```

Daarnaast zijn er nog drie ordeningsoperatoren: >, <= en >=. Bovendien is er een operator die bepaalt of twee getallen gelijk zijn (==) en een die bepaalt of twee getallen ongelijk zijn (/=).

Waarheidswaarden

De waarden True en False zijn de enige elementen van de verzameling *waarheidswaarden* (truth values). Functies (en operatoren) die zo'n waarde opleveren, heten Boolean functies omdat ze van het type Boolean zijn. Uitkomsten van dergelijke functies kunnen gecombineerd worden met de operatoren && ('en') en || ('of').

Operator &&

De *operator* && geeft alleen True als resultaat, als links én rechts een ware uitspraak staat:

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

Operator ||

Bij de *of-operator* hoeft maar een van de twee uitspraken waar te zijn, maar allebei mag ook. Er is ook een functie not:

```
Hugs> not(1<2)
False
Hugs>
```

Verder is er een functie even die bepaalt of een geheel getal een even getal is of niet:

```
Hugs> even 7
False
Hugs>
```

4.2 DEFINITIE VAN FUNCTIES

De eenvoudigste manier om functies te definiëren, is gebruikmaken van andere functies, vooral standaardfuncties of -operatoren uit de prelude:

```
kwadraat x  =  x*x
negatief x  =  x < 0
oneven x    =  not (even x)
```

De functie 'kwadraat' heeft een getal als resultaat; de functies 'negatief' en 'oneven' een waarheidswaarde. Functies kunnen ook meer dan één parameter krijgen:

```
abcFormule a b c  =  [ (-b+sqrt(b*b-4*a*c)) / (2*a)
                     , (-b-sqrt(b*b-4*a*c)) / (2*a)
                     ]
```

Deze functie heeft als resultaat een lijst met twee getallen.

Gevalsonderscheid middels voorwaarden

Soms is het nodig om in de definitie van een functie meer gevallen te onderscheiden. De absolute-waardefunctie abs is hiervan een voorbeeld: voor een negatieve waarde van de parameter is het functievoorschrift anders dan voor een positieve waarde. In Haskell worden de voorwaarden in een functiedefinitie 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
```

In de laatste definitie is de operator == de test op gelijkheid. Hiervoor wordt een dubbel gelijkteken gebruikt, omdat het enkele gelijkteken al de betekenis 'is gedefinieerd als' heeft.

De voorwaarden worden door de interpretator geprobeerd in de volgorde waarin ze beschreven staan. Als een functie voor alle waarden van de parameter gedefinieerd moet worden, kan voor de laatste voorwaarde ook True geschreven worden. In de prelude is gedefinieerd:

```
otherwise  =  True
```

Otherwise

Met gebruik van *otherwise* (een 'functie' met nul parameters) kan de functie signum ook geschreven worden als:

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

OPGAVE 1.2
Definieer een functie aantalOpl die, gegeven drie getallen a, b en c, het aantal oplossingen van de vergelijking $ax^2 + bx + c = 0$ bepaalt.

4.3 RECURSIEVE DEFINITIES

Recursieve definitie

In de definitie van een functie mogen we standaardfuncties en elders gedefinieerde functies gebruiken. Maar ook de te definiëren functie zelf mogen we in de eigen definitie gebruiken! Zo'n definitie heet een *recursieve definitie*; recursie betekent letterlijk 'terugkeer': de naam van de functie keert terug in haar 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 expressie rechts van =. 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.

Recursieve functies zijn echter wel zinvol onder de volgende twee voorwaarden:
– de parameter van de recursieve aanroep is (in een of andere zin) kleiner dan de parameter van de te definiëren functie
– voor een basisgeval is er een niet-recursieve definitie.
U moet er zelf op letten dat aan deze voorwaarden voldaan is; het al of niet zinvol zijn van een recursieve definitie wordt door de Haskell-interpretator niet gecontroleerd.

Een recursieve definitie van de machtsverheffunctie is de volgende:

```
power x n  |  n==0  =  1
           |  n>0   =  x * power x (n-1)
```

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

Gevalsonderscheid middels patronen

Een andere manier om deze twee gevallen – het basisgeval en het recursieve geval – te onderscheiden, is de volgende:

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

Patronen

In deze definitie worden de parameters van de functie niet aangeduid met namen, maar met zogenaamde *patronen*. Patronen kunnen bestaan uit namen (zoals x), constanten (zoals 0 en True) en operatoren (zoals +) en hebben bijvoorbeeld de vorm: naam + constante (zoals n + 1). Bij aanroep van een functie die met behulp van patronen is gedefinieerd, zoekt de interpretator waarden voor de namen in de patronen, zodat het patroon past bij de meegegeven parameter. Zo zal bij de aanroep 'power 2 4 ' de naam x de waarde 2 gaan aanduiden, en de naam n de waarde 3.

Bij een recusieve definitie met gebruikmaking van patronen moet ook weer gelden dat de parameter van de recursieve aanroep (n in het voorbeeld) kleiner is dan de (patroon)parameter van de te definiëren functie (n + 1 in het voorbeeld).

Operatordefinitie

Op deze manier sluit de definitie nog nauwer aan op de wiskundige definitie van machtsverheffen, vooral als de functie als operator wordt gedefinieerd:

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

Inductieve definitie

In de wiskunde wordt een recursieve definitie meestal een *inductieve definitie* genoemd.

OPGAVE 1.3

De faculteit van n kan berekend worden door de getallen 1 t/m n te vermenigvuldigen. De faculteit van 0 is gedefinieerd als 1. Schrijf op twee manieren een functie faculteit:

a met gevalsonderscheid door middel van voorwaarden
b met gevalsonderscheid door middel van patronen.

4.4 LOKALE DEFINITIES

In de definitie van de functie abcFormule komen de expressies (2 * a) en sqrt (b * b – 4 * a * c) twee keer voor. Behalve dat dat veel tikwerk geeft, kost het uitrekenen van zo'n expressie onnodig veel tijd: de identieke deelexpressies worden tweemaal uitgerekend. Om dat te voorkomen, is het in Haskell mogelijk om deelexpressies een naam te geven. De verbeterde definitie luidt dan als volgt:

```
abcFormule' a b c  =  [ (-b+d)/n
                      , (-b-d)/n
                      ]
                      where d = sqrt (b*b-4*a*c)
                            n = 2*a
```

We hebben de verbeterde functie een nieuwe naam gegeven door er een apostrof aan toe te voegen.

Lokale definities

De definities van d en n heten *lokale definities*, omdat we ze alleen binnen de definitie van abcFormule' mogen gebruiken en niet in de rest van het programma. Behalve constanten kunnen ook functies lokaal gedefinieerd worden. In de volgende paragraaf geven we daarvan een voorbeeld. Voor lokale definities wordt de where-clausule gebruikt. De where-clausule moet direct en inspringend worden opgenomen onder de definitie waarop de clausule betrekking heeft.

4.5 FUNCTIES ALS PARAMETER

De parameters van functies kunnen zelf ook weer functies zijn. Een voorbeeld hiervan is het differentiëren van een functie: differentiëren zelf kunnen we als functie beschouwen. De parameter is de functie die gedifferentieerd moet worden. Het resultaat is ook een functie: 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 kunnen we echter verkrijgen door voor h een zeer kleine waarde in te vullen; niet té klein, want dan geeft de deling onacceptabele afrondingsfouten. De definitie in Haskell kan luiden:

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

Hogere-ordefunctie

Een functie met een functie als parameter en/of als resultaat heet een *hogere-ordefunctie*.

OPGAVE 1.4
Geef nog een paar voorbeelden van hogere-ordefuncties in de wiskunde.
Het is niet nodig om de bijbehorende definitie in Haskell te geven.

4.6 PRIORITEIT EN ASSOCIATIE VAN OPERATOREN

Bij het uitrekenen van expressies gebruikt de interpretator de regel ‘vermenigvuldigen gaat voor optellen’. Zo heeft bijvoorbeeld de expressie 2 * 3 + 4 * 5 de waarde 26 en niet 50, 46 of 70. Er zijn in Haskell nog meer prioriteitsniveaus. De vergelijkingsoperatoren, zoals < en ==, hebben een lagere prioriteit dan de rekenkundige operatoren. Dus heeft 3 + 4 < 8 de betekenis die we ervan zouden verwachten: 3 + 4 wordt met 8 vergeleken (resultaat True), en niet: 3 wordt opgeteld bij het resultaat van 4 < 8 (dat zou een typeringsfout opleveren).

Prioriteiten

In totaal zijn er negen *prioriteitsniveaus* mogelijk. De operatoren in de prelude hebben de volgende prioriteit:

niveau	*operator*
8	`^ ^^ **`
7	`* / `quot` `div``
6	+ –
5	: ++ (zie paragraaf 5.2)
4	vergelijkingsoperatoren
3	&&
2	||

Vermenigvuldigen en delen hebben dus dezelfde prioriteit, evenals optellen en aftrekken. De operatoren ^, ^^ en ** kunnen we alle drie gebruiken voor machtsverheffen, zij zijn echter alle drie anders gedefinieerd.
Om af te wijken van de geldende prioriteiten kunnen we in een expressie haakjes plaatsen rond de deelexpressies die eerst uitgerekend moeten worden: in 2 * (3 + 4) * 5 wordt wél eerst 3 + 4 uitgerekend.

Het aanroepen van een functie heeft een prioriteit die hoger is dan alle operatoren. De expressie

```
kwadraat 3 + 4
```

berekent dus eerst het kwadraat van 3 en telt daar 4 bij op. Om het kwadraat van 7 te bepalen, zijn haakjes nodig om de hogere prioriteit van functieaanroep te doorbreken: kwadraat (3 + 4).

```
Le01> kwadraat 3+4
13
Le01> kwadraat (3+4)
49
Le01>
```

Haakjes om patronen

Ook bij het definiëren van functies met gebruik van patronen (zie paragraaf 4.3) is het van belang te bedenken dat functieaanroep altijd voor gaat. In de definitie

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

zijn de haakjes rond n+1 essentieel: zonder haakjes zou dit immers opgevat worden als (power x n) + 1 en dat is geen geldig patroon.

Met de prioriteitsregels ligt nog steeds niet vast wat er moet gebeuren met operatoren van gelijke prioriteit. Voor optellen 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)?

Associatie

Voor elke operator wordt in Haskell vastgelegd in welke volgorde de expressie waarin hij voorkomt, uitgerekend moet worden: van links naar rechts of van rechts naar links. Dit heet de *associatievolgorde* van de operator. Voor de operatoren in de prelude is de keuze overeenkomstig de wiskundige traditie gemaakt.

Links-associatieve operatoren

De volgende operatoren zijn *links-associatief* (worden van links naar rechts uitgerekend):
– de operator –
dus de waarde van 8 – 5 – 1 is 2, zoals gebruikelijk in de wiskunde
– de operatoren +, /, `quot` en `div`.
Overigens maakt het bij de operatoren achter het tweede opsommingstreepje niet uit in welke volgorde deze worden uitgerekend (ga dit na!); in de prelude is voor associatie naar links gekozen.

Ook het toepassen van een functie op een parameter – de 'onzichtbare' operator die tussen f en x staat in de expressie f x – associeert naar links. Dus f x y betekent (f x) y. De reden hiervoor bespreken we in paragraaf 6.5.

Rechts-associatieve operatoren

De volgende operatoren zijn *rechts-associatief* (worden van rechts naar links uitgerekend):
– de operatoren ^, ^^ en ** (machtsverheffen)
dus de waarde van 2 ^ 2 ^ 3 is $2^8 = 256$, zoals gebruikelijk in de wiskunde, en niet $4^3 = 64$
– de operator : (de lijstconstructor)
zie hiervoor paragraaf 5.2.

De volgende operatoren mogen niet tweemaal naast elkaar gebruikt worden, zonder met haakjes aan te geven wat de bedoelde volgorde is:
– de vergelijkingsoperatoren ==, <, enzovoort
Het heeft meestal toch geen zin om a == b == c te schrijven. Wilt u testen of x tussen 2 en 8 ligt, schrijf dan niet 2 < x < 8, maar 2 < x && x < 8. Ga na dat daarbij geen haakjes nodig zijn.

Infixdeclaratie

Wie zelf een operator definieert, moet daarbij aangeven wat de prioriteit is en op welke manier de associatie plaatsvindt. In de prelude is met een *infixdeclaratie* aangegeven dat ^, ^^ en ** prioriteitsniveau 8 hebben en rechts-associatief zijn:

```
infixr 8  ^, ^^, **
```

Voor operatoren die links-associatief zijn, dient het gereserveerde woord infixl, en voor operatoren waarvoor de associatie niet is gedefinieerd, het woord infix:

```
infixl 6  +, -
infix  4  ==, /=, <, <=, >=, >
```

Door een slimme keuze voor de prioriteit te maken, kunnen we haakjes in expressies zo veel mogelijk vermijden.

4.7 GEBRUIKMAKEN VAN ANDERE FUNCTIEDEFINITIES

In een functiedefinitie kunnen standaardfuncties uit de prelude en andere functiedefinities uit dezelfde module gebruikt worden. De module Nieuw kan bijvoorbeeld als volgt worden gedefinieerd:

```
module Nieuw where
kwadraat x = x * x
derde x = x * kwadraat x
```

Hier wordt in de definitie van de functie derde de functie kwadraat uit module Nieuw gebruikt en de operator * uit de prelude.

Het is ook mogelijk om functiedefinities uit andere modulen te gebruiken. Als voorbeeld noemen we de module Ratio waarin breuken van type Rational worden gedefinieerd door de operator % toe te passen op twee integers. Daarbij worden de breuken zoveel mogelijk vereenvoudigd. Als module Nieuw gebruik wil maken van de operator % uit module Ratio, dan moet als volgt deze module worden geïmporteerd:

```
module Nieuw where
import Ratio
breuk x y = x % y
```

In de interpretator kan dan de functie breuk als volgt worden aangeroepen:

```
Nieuw> breuk 7 14
1 % 2
Nieuw>
```

5 Datastructuren in Haskell

5.1 STANDAARDTYPEN

Haskell is een getypeerde taal. Dat wil zeggen dat niet alle operatoren en functies op alle waarden kunnen werken. Stel, dat we in een module Nieuw (in het bestand Nieuw.hs) bijvoorbeeld de volgende definities opnemen:

```
module Nieuw where
n = 5
f = n + True
```

Als de interpretator de expressie n + True leest, dan meldt hij bij het analyseren daarvan:

```
Le01> :load "D:\\Examples\\Nieuw.hs"
ERROR file:.\Nieuw.hs:3 - Type error in application
*** Expression     : n + True
*** Term           : n
*** Type           : Integer
*** Does not match : Bool

Hugs>
```

De deelexpressie (term) n heeft het type Integer (geheel getal). Zo'n integer-waarde kan niet worden opgeteld bij waarheidswaarde True, die van het type Bool is (een afkorting van Boolean).

Standaardtypen

In Haskell zijn de volgende *standaardtypen* beschikbaar:
– *Int*, gehele getallen met waardebereik $[-2^{31}, 2^{31} - 1]$
– *Integer*, gehele getallen met willekeurig waardebereik
– *Float*, enkele precisie drijvende-puntgetallen
– *Double*, dubbele precisie drijvende-puntgetallen

– *Bool*, de waarheidswaarden False en True
– *Char*, ISO Latin-1 symbolen
– *String*, strings van symbolen van type Char.
De interpretator beschouwt een geheel getal (een letterlijke getalwaarde zonder decimale punt) als type Integer en een drijvende-puntgetal (een letterlijke getalwaarde met decimale punt) als type Double. Op de consequenties hiervan komen we terug in paragraaf 6.4.

5.2 LIJSTEN

Zoals in imperatieve talen samengestelde typen gemaakt kunnen worden in de vorm van *arrays* en *records*, zo kent Haskell de samengestelde typen *lijst* en *tupel*. Lijsten worden gebruikt om een willekeurig aantal elementen te groeperen. Die elementen moeten van hetzelfde type zijn. Voor elk type bestaat er een type 'lijst-van-dat-type'. Er bestaan dus bijvoorbeeld lijsten-van-integers, lijsten-van-floats en lijsten-van-chars. 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-waarheidswaarden, enzovoort.

Lijst

Het type van een *lijst* wordt aangegeven door het gemeenschappelijke type van de elementen tussen rechte haken te zetten. De hiervoor genoemde typen kunnen we dus aangeven met:

```
[Integer]
[Float]
[Char]
[[Integer]]
[[[Bool]]]
```

Lijsten kunnen we maken door de elementen op te sommen. Enkele voorbeelden van lijstopsommingen met hun type zijn:

```
[1, 3, 7, 3, 8]     :: [Integer]
[True, False, True] :: [Bool]
[[1,2,3], [1,2]]    :: [[Integer]]
```

Het symbool : : kan uitgesproken worden als 'heeft het type'; zie paragraaf 6.

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 hetzelfde type [Integer]. Daarom mogen de lijsten [1,2,3] en [1,2] in het derde voorbeeld op hun beurt elementen zijn van één lijst-van-lijsten-van-integers.
Omdat het aantal elementen van een lijst niet vastligt, kan een lijst ook bestaan uit maar één element:

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

Singleton-lijst

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 heeft: de lijst [1,2,3].

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

Lege lijst []

Het aantal elementen van een lijst kan ook nul zijn. Een lijst met nul elementen heet de *lege lijst* en wordt genoteerd als `[]`.

Operator :

Een andere manier om een lijst te maken is het gebruik van de *operator* : Deze operator zet een element vooraan in een lijst, en maakt zo een langere lijst. Deze operator kunnen we het beste uitspreken als 'op kop van'. Prolog- en LISP-kenners zullen misschien 'cons' prefereren. Als bijvoorbeeld xs de lijst [3, 4, 5] is, dan is 1:xs de lijst [1, 3, 4, 5]. Gebruikmakend van de lege lijst en de operator : is elke lijst te construeren. Zo staat bijvoorbeeld

```
1:(2:(3:[]))
```

voor dezelfde lijst als

```
[1,2,3]
```

Voor de naam van een lijst zullen we in deze cursus vaak een naam kiezen die op een s eindigt, bijvoorbeeld xs. Dit is bedoeld als geheugensteuntje: xs is een soort meervoudsvorm van x. Lijsten van lijsten kunnen we dan bijvoorbeeld aanduiden met xss. Voor de interpretator maakt dit allemaal niets uit: het enige waaraan die in voorgaand voorbeeld kan zien dat xs een lijst is, is het feit dat xs de rechter parameter van : is.

In feite is de opbouw van een lijst met : 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.

OPGAVE 1.5

Tussen twee lijsten is de operator ++ gedefinieerd. Ga met behulp van de interpretator na wat daarvan de werking is. Wat is het verschil met de operator : ?

5.3 STRINGS

Strings

Net als in veel andere talen kunnen we in Haskell *strings* gebruiken om teksten te manipuleren. Waar in andere talen een string vaak een array is, is een string in Haskell een bijzonder geval van een *lijst*.
Strings noteren we tussen dubbele 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 "sqrt" is een string die uit vier tekens bestaat, maar sqrt is de naam van een functie. Om een string moeten we daarom aanhalingstekens schrijven.

```
Hugs> "sqrt"
"sqrt"
Hugs>
```

Een string is een lijst waarvan de elementen het type Char hebben. Het standaardtype String is equivalent aan het volgende lijsttype:

```
[Char]
```

Let op het verschil tussen de drie soorten aanhalingstekens die in Haskell gebruikt worden en die elk een geheel verschillende betekenis hebben:

`"f"`	een lijst symbolen (string) die slechts één element bevat
`'f'`	een symbool van type Char
`` `f` ``	de functie `f` als operator beschouwd.

De notatie met dubbele aanhalingstekens voor strings betekent precies hetzelfde als een opsomming van een lijst met characters. Zo staat

```
"hallo"
```

voor dezelfde lijst als

```
['h','a','l','l','o']
```

en dat is weer hetzelfde als

```
'h':'a':'l':'l':'o':[]
```

Uit het volgende voorbeeld blijkt dat een string inderdaad een lijst is: de operator ++ (concatenatie) die op lijsten werkt, is ook op strings toe te passen:

```
Hugs> "zon"++"dag"
"zondag"
Hugs>
```

5.4 TUPELS

In Haskell moet elk element van een lijst hetzelfde type hebben. Het is niet mogelijk om in één lijst zowel een integer als een string op te nemen. Toch is het soms nodig om gegevens van verschillende typen te groeperen. De gegevens in een bevolkingsregister bestaan bijvoorbeeld uit een string (naam), een waarheidswaarde (geslacht) en drie integers (geboortedatum). Deze gegevens horen bij elkaar, maar kunnen niet samen in één lijst voorkomen.

Tupel

Voor dit soort gevallen is er, naast lijstvorming, nog een andere manier om samengestelde typen 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 gebruiken we in Haskell waar we in een imperatieve taal *records* gebruiken.

Tupels worden genoteerd met ronde haken rond de elementen, in plaats van de rechte haken die bij lijsten 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 waarheidswaarde True en de integer 2.

Voor elke combinatie van typen vormt de tupel daarvan een apart type. Daarbij is ook de volgorde van belang. Het type van tupels wordt geschreven door de typen van de elementen op te sommen tussen ronde haken. De twee hiervoor genoemde expressies zijn dus als volgt getypeerd:

```
(1, 'a')          ::  (Integer, Char)
("aap", True, 2)  ::  (String, Bool, Integer)
```

Paar

Een tupel met twee elementen noemen we een 2-tupel, of ook wel een *paar*. Tupels met drie elementen heten 3-tupels, enzovoort. Er bestaan geen 1-tupels: de expressie (7) is gewoon een integer; om elke expressie mogen we immers haakjes zetten. Wel bestaat er een 0-tupel: de waarde () die () als type heeft.

5.5 FUNCTIES OP LIJSTEN EN TUPELS

Sommige standaardfuncties werken op lijsten. Bijvoorbeeld:

```
Hugs> tail [1,2,3]
[2,3]
Hugs>
```

Er zijn ook operatoren die op lijsten werken. Bijvoorbeeld de operator ++, die twee lijsten samenvoegt tot één lange lijst:

```
Hugs> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Hugs>
```

Functies en operatoren die op lijsten werken, kunnen we net als functies op getallen, zelf definiëren door gebruik te maken van andere functies. Bijvoorbeeld:

```
ergLang xs  =  length xs > 100
```

Zo nodig kunnen we daarbij voorwaarden en een where-constructie gebruiken:

```
grootte xs  |  n<10       =  "kort"
            |  n<100      =  "lang"
            |  otherwise  =  "erg lang"
    where n = length xs
```

Functies op lijsten definiëren we vaak recursief. Net als bij recursieve functies op getallen moet één van de gevallen niet-recursief zijn. De functie length, die hiervoor al is gebruikt, kunnen we bijvoorbeeld recursief definiëren:

```
length xs  |  null xs    =  0
           |  otherwise  =  1 + length (tail xs)
```

De functie null levert hierbij True op als een lijst leeg is.

In de meeste programmeertalen moet de formele parameter van een functie (zoals xs in voorgaande voorbeelden) een enkelvoudige naam zijn. In Haskell zijn er echter meer mogelijkheden: de formele parameter mag ook hier weer een patroon zijn.
Naast de in paragraaf 4.3 genoemde patronen zijn namelijk ook de volgende constructies toegestaan als patroon: lijstopsommingen met patronen als elementen (zoals [x,y,z]), tupelopsommingen (zoals (x,y)) en de operator : met patronen aan de linker- en rechterkant (zoals x:xs).

Lijstpatronen

Een voorbeeld van een functiedefinitie waarin we *patronen* als formele parameter gebruiken, is:

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

De functie telOp kunnen we toepassen op lijsten met nul, een, twee of drie elementen. In alle gevallen worden de elementen opgeteld. Bij aanroep van de functie kijkt de interpretator of de parameter past bij een van de gevallen van de definitie: de aanroep telOp [3,4] past bijvoorbeeld bij de derde regel. De 3 komt daarbij overeen met de x in de definitie en de 4 met de y. De functie is niet gedefinieerd op lijsten met meer dan drie elementen.

Functies head en tail

Met de operator : kunnen we lijsten opbouwen. De expressie x:xs betekent immers 'zet element x op kop van de lijst xs'. Door de operator : in een patroon te zetten, wordt het eerste element van een lijst juist afgesplitst. Daarmee zijn in de prelude twee standaardfuncties gedefinieerd:

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

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'. Omdat we deze functies veel gebruiken, zijn ze in de prelude gedefinieerd. De *functies head en tail* kunnen we op bijna alle lijsten toepassen, ze zijn alleen niet gedefinieerd op de lege lijst (een lijst zonder elementen): die heeft immers geen eerste element, laat staan een staart.
De haakjes in het patroon (x : xs) zijn noodzakelijk, omdat het toepassen van een functie een hogere prioriteit heeft dan de operator :. Er staan hier geen rechte haakjes, zoals in het vorige voorbeeld, omdat er hier geen sprake is van een lijstopsomming, de lijst wordt nu immers opgebouwd door de operator :

Recursieve functie op lijsten

Het gebruik van patronen komt, net als bij functies op getallen, goed van pas in recursieve definities. Een lijst is óf een lege lijst, óf heeft een eerste element x en een staart xs. De functie length, waarin het gevalsonderscheid eerder door middel van voorwaarden is gemaakt, is dus ook met patronen te definiëren:

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

In de meeste gevallen is een gevalsonderscheid door middel van patronen duidelijker dan door middel van voorwaarden, omdat de verschillende onderdelen in het patroon direct een naam kunnen krijgen (zoals x en xs in de functie length). In de eerdere definitie van length zonder patronen is de standaardfunctie tail nodig om de onderdelen uit de lijst te halen. In de definitie van die functie wordt bovendien alsnog een patroon gebruikt.

OPGAVE 1.6

Definieer een functie som die de som van de elementen van een lijst gehele getallen bepaalt. De functie moet werken op een lijst die van elke lengte kan zijn. Voor de som van nul elementen kan 0 genomen worden. Maak het gevalsonderscheid tussen een lege en een niet-lege lijst op twee manieren:

a door middel van voorwaarden
b door middel van patronen.

Ook functies op tupels zijn met behulp van patronen te definiëren. Een voorbeeld hiervan zijn de functies fst en snd die in de prelude zijn gedefinieerd:

```
fst (x,y)  =  x
snd (x,y)  =  y
```

Deze functies werken op 2-tupels en selecteren daaruit het eerste, respectievelijk het tweede element.

5.6 HOGERE-ORDEFUNCTIES OP LIJSTEN

De volgende twee functies bepalen van alle getallen in een lijst het kwadraat, respectievelijk de faculteit:

```
kwadraten []        =  []
kwadraten (x:xs)    =  kwadraat x : kwadraten xs
faculteiten []      =  []
faculteiten (x:xs)  =  faculteit x : faculteiten xs
```

Zo zijn er natuurlijk nog veel meer functies te schrijven. Het is echter handiger om één functie te schrijven die het algemene principe ‘doe iets met alle elementen van een lijst’ beschrijft. Wat er met alle elementen van de lijst gedaan moet worden, wordt als extra parameter aan deze ‘universele’ functie meegegeven.

Functie map

In de prelude is deze functie gedefinieerd. Zij heet *map*. De functie map past haar functieparameter toe op alle elementen van haar lijstparameter:

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

Functies die een functie als parameter hebben (zoals map) en/of een functie als resultaat, noemen we hogere-ordefuncties. Hier is, in tegenstelling tot wat deze aparte naam suggereert, niets bijzonders aan; het zijn functies die we gewoon in Haskell kunnen definiëren.

De definitie van map lijkt sterk op die van kwadraten en faculteiten. Waar die functies echter de eerder gedefinieerde functies kwadraat en faculteit gebruiken, gebruikt map haar functieparameter f:

```
map f []      =  []
map f (x:xs)  =  f x : map f xs
```

Bruikbare hogere-ordefuncties kunnen we op het spoor komen door de overeenkomst in functiedefinities op te sporen. Bekijk bijvoorbeeld de definitie van de functies som (die de som van een lijst getallen berekent), product (die het product van een lijst getallen berekent) en and (die kijkt of een lijst waarheidswaarden allemaal True zijn):

```
som     []      =  0
som     (x:xs)  =  x + som 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, zijn 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 &&).

Functie foldr

Door deze twee verschillen als parameter mee te geven, ontstaat een algemeen bruikbare hogere-ordefunctie:

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

Gegeven deze functie, zijn de andere drie functies te definiëren door de algemene functie te parametriseren:

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

In dit voorbeeld zien we nogmaals dat we een operator kunnen meegeven op de plaats waar een functie met twee parameters wordt verwacht, door haakjes om de operator te zetten.

De *functie foldr* is in veel meer gevallen bruikbaar, daarom is deze als standaardfunctie in de prelude gedefinieerd. De standaardfunctie foldr knapt het 'vuile werk' op en de andere functies zien er overzichtelijker uit.

De naam van foldr laat zich verklaren door te kijken naar een voorbeeld:

```
            xs = [1 ,  2 ,  3 ,  4 ,  5]
                  |    |    |    |    |
foldr (+) 0 xs = (1 + (2 + (3 + (4 + (5 + 0))))) = 15
```

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-ordefuncties zoals map en foldr, spelen in functionele talen de rol die controlestructuren (zoals for en while) in imperatieve talen spelen. In imperatieve talen zijn de controlestructuren echter ingebouwd, terwijl functies zoals map en foldr zelf gedefinieerd kunnen worden. Dit maakt functionele talen flexibel: er is weinig ingebouwd, maar we kunnen alles zelf maken.

OPGAVE 1.7

Definieer de functie eenVanDe, die bepaalt of minstens één element van een lijst waarheidswaarden True is. Geef drie versies van deze functie:

a een recursieve definitie met gevalsonderscheid door middel van voorwaarden

b een recursieve definitie met gevalsonderscheid door middel van patronen

c een definitie met gebruik van foldr.

6 Typering

6.1 BEPALEN VAN HET TYPE

Bij het analyseren van een bestand meldt de interpretator alle typeringsfouten in een programma. Niet alleen het optellen van integers en waarheidswaarden vallen daaronder, maar ook bijvoorbeeld het toepassen van een functie die op lijsten behoort te werken en die op iets anders dan een lijst werkt, bijvoorbeeld sum True. Om deze controle te kunnen uitvoeren, moet de interpretator van elke expressie het type kunnen bepalen.

Opdracht :type

Het is ook mogelijk om de interpretator expliciet te verzoeken het type van een bepaalde expressie te bepalen. Dat is bijvoorbeeld handig tijdens het ontwikkelen van een nieuwe functie. Dit gebeurt met de *interpretatoropdracht :type*. Achter :type staat de expressie die getypeerd moet worden. Bijvoorbeeld:

```
Hugs> :type ['a','b','c']
['a','b','c'] :: [Char]
Hugs>
```

De symbolencombinatie : : is te lezen als ‘heeft het type’. De expressie wordt met de opdracht :type niet uitgerekend, alleen het type wordt bepaald.

Een andere mogelijkheid om het type van expressies te laten bepalen, is het geven van de opdracht :set + t. Daarna wordt van alle uitgerekende expressies naast de waarde ook het type gegeven.

```
Hugs> :set +t
Hugs> map even [1,2,3]
[False,True,False] :: [Bool]
Hugs>
```

In leereenheid 7 zullen we zien hoe het mogelijk is dat van elke expressie het type is af te leiden.

6.2 TYPE VAN EEN FUNCTIE

Functietype

Alle expressies en deelexpressies in Haskell hebben een type. Dat betekent dat ook functies een type hebben. Het type van een functie ligt vast door het type van de parameters en het type van het resultaat. De functie isDigit, die gedefinieerd is in de module Char, bepaalt bijvoorbeeld of een symbool een cijfer is. Het *type van deze functie* is:

```
Hugs> :load Char
Char> :type isDigit
isDigit :: Char -> Bool
Char>
```

De functie isDigit werkt op characters en heeft als resultaat een waarheidswaarde. Het symbool -> in het type van de functie scheidt het parametergedeelte van het resultaatgedeelte. Het symbool is gekozen als 'toetsenbordbenadering' van het symbool → dat we in de wiskunde gebruiken. In handschrift kunnen we het symbool natuurlijk ook gewoon als pijltje schrijven. Andere voorbeelden van typen van functies zijn:

```
not :: Bool -> Bool
ord :: Char -> Int
```

Zo'n regel kunnen we uitspreken als 'ord heeft het type Char naar Int' of als 'ord is een functie van Char naar Int'.

Omdat functies een type hebben, kunnen ze gebruikt worden als 'waarden', net als getallen, waarheidswaarden en lijsten. Het is dus bijvoorbeeld 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:

```
Char> :type [isDigit, isLower, isUpper]
[isDigit,isLower,isUpper] :: [Char -> Bool]
Char>
```

De drie functies isDigit, isLower en isUpper, die gedefinieerd zijn in de module Char, zijn allemaal functies 'van Char naar Bool'. Ze kunnen dus in een lijst gezet worden, die dan het type 'lijst van functies van Char naar Bool' heeft.

De taal Haskell is zo ontworpen dat de interpretator in de meeste gevallen zelf het type van een expressie of een functie kan bepalen. Dit gebeurt dan ook bij het controleren van de typering van een programma. Desondanks is het toegestaan en aan te bevelen, om het type van een functie in een programma erbij te schrijven. De definitie van de functie som van paragraaf 5.6 ziet er dan bijvoorbeeld als volgt uit:

```
som    ::  [Integer] -> Integer
som xs  =  foldr (+) 0 xs
```

Typespecificatie

Hoewel zo'n *typespecificatie* hier in principe overbodig is, heeft deze twee voordelen:
– er wordt gecontroleerd of de functie inderdaad het type heeft dat ervoor is gespecificeerd
– de typespecificatie maakt het voor een menselijke lezer eenvoudiger om een functie te begrijpen.

De typespecificatie hoeft niet direct voor de functiedefinitie te staan. We zouden bijvoorbeeld een programma kunnen beginnen met de specificaties van de typen van alle functies die erin worden gedefinieerd. De typespecificaties dienen dan als een soort inhoudsopgave.

Bij functies met meer dan één parameter staat er tussen de parameters onderling en tussen de laatste parameter en het resultaat een pijltje. De functie abcFormule heeft bijvoorbeeld drie drijvende-puntgetallen als parameter en een lijst met drijvende-puntgetallen als resultaat. De typespecificatie luidt daarom:

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

De reden dat we een pijltje niet alleen gebruiken om de parameters van het resultaat te scheiden, maar ook om de parameters onderling te scheiden, bespreken we in paragraaf 6.5.

6.3 POLYMORFIE

Voor sommige functies op lijsten maakt het niet uit wat het type van de elementen van die lijst is. De functie length bijvoorbeeld, kan de lengte bepalen van een lijst integers, maar ook van een lijst waarheidswaarden, en –waarom niet – van een lijst functies. Het type van de functie length noteren we als volgt.

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

Typevariabele

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 geven we aan door een *typevariabele* , in het voorbeeld a. Typevariabelen schrijven we, in tegenstelling tot de vaste typen (zoals Int, Integer, Bool, enzovoort) met een kleine letter.

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

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

Net als length werkt deze functie 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 typevariabele gebruikt als voor het type van de elementen van de lijst.

Polymorf type

Polymorfie

Een type waar typevariabelen in voorkomen, heet een *polymorf type* (letterlijk: veelvormig type). Functies met een polymorf type heten polymorfe functies. Het verschijnsel zelf heet *polymorfie*. Deze begrippen bespreken we uitgebreid in leereenheid 7.
Polymorfe functies, zoals length en head, hebben met elkaar gemeen dat ze alleen de *structuur* van de lijst gebruiken. Een niet-polymorfe functie, zoals som, gebruikt ook eigenschappen van de *elementen* van de lijst, zoals optelbaarheid.

Niet alleen functies die op lijsten werken zijn vaak polymorf, ook functies die op functies werken – hogere-ordefuncties dus – zijn vaak polymorfe functies.

OPGAVE 1.8
Wat zou het type zijn van de functie map? Bepaal met behulp van de interpretator het type van map om uw antwoord te controleren.

Veel functies die in de prelude worden gedefinieerd zijn polymorf. Polymorfe functies, zoals map, foldr en length zijn namelijk zeer algemeen toepasbaar en zijn daarom in veel programma's van nut.

Samenstellen van functies

Een ander voorbeeld van een polymorfe functie die in de prelude wordt gedefinieerd, is de operator die twee *functies samenstelt*. Deze operator duiden we aan met een punt (.) en kan als volgt worden gedefinieerd.

```
f . g  =  h
   where  h x = f (g x)
```

In woorden is dit te omschrijven met 'de samenstelling van de functies f en g is de functie h die – als zij op een parameter x wordt toegepast – eerst g op x toepast en daarna f op het resultaat daarvan'.
Een voorbeeld van het gebruik van deze operator is de definitie van de functie oneven op de volgende manier.

```
oneven n = (not . even) n
```

Het polymorfe type van de functiesamenstelling is:

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

De operator heeft dus twee parameters: een functie van het type b->c en een functie van het type a->b, en heeft als resultaat een functie van het type a->c. De tweede parameter g wordt immers op een waarde van een willekeurig type a toegepast en levert een, mogelijk ander, type b. De functie f op haar beurt heeft een waarde van type b als parameter, en maakt daar iets van type c van. De samenstelling van de twee functies is een functie die van een waarde van type a direct iets van type c maakt.

Welke typen a, b en c precies voorstellen maakt voor de functie (.) niet uit, mits het resultaattype van de rechter parameter maar hetzelfde is als het parametertype van de linker parameter.

De typevariabelen in een polymorf functietype kunnen bij elk gebruik van de functie een ander type voorstellen. Zo stelt de typevariabele a in het type van length bijvoorbeeld het type Char voor in de aanroep length "aap", terwijl in de aanroep length [1,2,3] de typevariabele a het type Integer voorstelt.

6.4 OVERLOADING

Overloading

We spreken over *overloading* als verschillende functies of operatoren die werken op een aantal geheel verschillende typen dezelfde naam of hetzelfde symbool hebben. De operator + kan bijvoorbeeld worden toegepast op type Int, maar ook op de typen Integer, Float en Double. In feite zijn dit vier verschillende operaties. Dit komt tot uiting in de typespecificatie van +:

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

Overloaded functie of operator

Zo op het eerste gezicht lijkt het of (+) een polymorfe functie is omdat er een typevariabele in de typespecificatie staat. Voor deze typevariabele mogen we echter niet ieder willekeurig type gebruiken maar alleen een type dat behoort tot de klasse Num. De tekst 'Num a => 'moeten we lezen als 'type a moet behoren tot klasse Num'. De typen die tot deze klasse behoren zijn Int, Integer, Float en Double en voor typevariabele a mogen we dus één van deze typen ingevuld denken. Vullen we Integer in, dan kunnen we het type van (+) beschouwen als

```
Integer -> Integer -> Integer
```

en gebruiken we de operator voor het optellen van twee waarden van type Integer met een resultaat van type Integer. Vullen we Double in, dan is het type

```
Double -> Double -> Double
```

en gebruiken we dezelfde operator voor een andere operatie, namelijk het optellen van twee waarden van type Double met een resultaat van type Double. We zeggen dat de *operator* + *overloaded* is voor het uitvoeren van de verschillende opteloperaties voor de verschillende typen uit klasse Num. Overloading is een beperkte vorm van polymorfie en wordt ook wel aangeduid als *ad-hocpolymorfie*.

Ad-hocpolymorfie

Gezien de bovenstaande typespecificatie is het duidelijk dat we het symbool + kunnen gebruiken voor het optellen van twee waarden van type Integer of twee waarden van type Int. Volgens de typespecificatie mogen we deze beide gehele typen in een optelling niet door elkaar gebruiken. Ook de beide floating-point typen Float en Double mogen niet door elkaar worden gebruikt. De volgende regels in het bestand Nieuw.hs

```
x :: Int
y :: Integer
x = 1
y = 2
z = x + y
```

resulteren in de foutmelding:

```
ERROR file:.\Nieuw.hs:6 - Type error in application
*** Expression     : x + y
*** Term           : x
*** Type           : Int
*** Does not match : Integer
```

Als we + toepassen op de *letterlijke waarden* van verschillend type, bijvoorbeeld een geheel getal en een drijvende-puntgetal of een geheel getal en een breuk, blijkt de optelling wel goed te gaan.

```
Ratio> 5+3.75
8.75
Ratio> 5+15%4
35 % 4
Ratio>
```

De integer 5 wordt eerst geconverteerd naar respectievelijk type Double en Rational, het type van de breuk, waarna de optelling – met respectievelijk de Double-versie en de Rational-versie van + – wordt uitgevoerd. We bekijken hoe dit in Haskell mogelijk is gemaakt.

Overloaded getallen

Als we aan de interpretator vragen wat het type is van het symbool 'A' of de string "Aap", dan krijgen we het antwoord dat we verwachten:

```
Hugs> :type 'A'
'A' :: Char
Hugs> :type "Aap"
"Aap" :: String
Hugs>
```

Als we vragen wat het type is van het gehele getal 5, krijgen we het volgende antwoord:

```
Hugs> :type 5
5 :: Num a => a
Hugs>
```

De letterlijke waarde 5 is niet van type Integer zoals we verwachten, maar kan een van de typen uit klasse Num aannemen, dus Int, Integer, Float of Double. Als 5 wordt gebruikt in een context waarin niet type Integer is te gebruiken (zoals in de optelling 5 + 3.75), converteert de interpretator automatisch de waarde 5 naar het juiste type. We spreken in dit verband over *overloaded getallen* (overloaded literals).
Als we vragen wat het type is van de breuk 15%4, krijgen we het volgende antwoord:

```
Ratio> :type 15%4
15 % 4 :: Integral a => Ratio a
Ratio>
```

Een breuk heeft kennelijk het type Ratio a uit klasse Integral waar a staat voor het type van de teller en de noemer. In een context waarin breuken van type Ratio a worden gebruikt converteert de interpretator de waarde 5 zelfs naar dit type. In leereenheid 12 gaan we dieper in op de techniek die hierbij wordt toegepast.

OPGAVE 1.9

a Wat is het type van de functies quot en div? Verklaar de typespecificatie in eigen woorden.
b Op welke typen zouden de functies quot en div kunnen worden toegepast? Mogen we deze typen door elkaar gebruiken?
c Wat is het type van de letterlijke waarde 7.3?
d Welke typen zouden deel uitmaken van de klasse Fractional?

6.5 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 mogen we ook *minder* parameters aan een functie meegeven. Als plus maar één parameter krijgt, bijvoorbeeld plus 1, dan is het resultaat een functie die nog één parameter verwacht. Deze functie kunnen we bijvoorbeeld gebruiken om een andere functie te definiëren:

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

Op het moment dat de functie opvolger wordt aangeroepen met één parameter, wordt die parameter naast 1 als tweede parameter aan plus doorgespeeld.

Partieel parametriseren

Het aanroepen van een functie met minder parameters dan deze verwacht, zoals dat in dit voorbeeld met de functie plus gebeurde, heet *partieel parametriseren*.

OPGAVE 1.10

De functie opvolger is te definiëren als

```
opvolger x  =  plus 1 x
```

maar met behulp van partiële parametrisatie van plus ook als

```
opvolger  =  plus 1
```

In paragraaf 5.6 is de functie som gedefinieerd met

```
som xs  =  foldr (+) 0 xs
```

Herschrijf ook deze definitie door gebruikmaking van partiële parametrisatie.

OPGAVE 1.11

Definieer de functie oneven door gebruik te maken van partiële parametrisatie van de operator (.).

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

```
Le01> map (plus 5) [1,2,3]
[6,7,8]
Le01>
```

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

OPGAVE 1.12
Geef een expressie die de lijst van kwadraten van de getallen 1 t/m 10 berekent, zonder daarbij eerst een functie kwadraat te definiëren.

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

```
plus  ::  Integer -> (Integer->Integer)
```

Door af te spreken dat -> *rechts-associatief* is, zijn de haakjes hierin overbodig:

```
plus  ::  Integer -> Integer -> Integer
```

Dit is precies de notatie voor het type van een functie met twee parameters, die eerder is besproken.
Het verklaart ook waarom er niet alleen tussen de parameters en het resultaat een pijltje staat, maar ook tussen de parameters onderling. Eigenlijk is er in Haskell namelijk helemaal geen apart concept 'functies met twee parameters'. Er zijn alleen maar functies met één parameter, die desgewenst een functie op kunnen leveren. Die functie heeft op haar beurt een parameter, zodat het lijkt alsof de oorspronkelijke functie twee parameters heeft.

Gecurryde functie

Deze truc – het simuleren van functies met meer parameters door een functie met één parameter die een functie oplevert – noemen we currying, naar de Engelse wiskundige Haskell Curry. De functie zelf heet een '*gecurryde*' *functie*. Dit eerbetoon is overigens niet helemaal terecht, want de methode is eerder gebruikt door Moses Schönfinkel.

OPGAVE 1.13
In de wiskunde definiëren we met $f(x, y) = x$ een functie met twee parameters: in een functiedefinitie moeten altijd haakjes staan en de parameters zijn gescheiden door komma's. In een Haskell-programma heeft deze definitie echter een andere betekenis. Welke?

6.6 BESPARING VAN HAAKJES

We zouden kunnen zeggen dat er in de expressie f x een onzichtbare operator 'pas toe op' staat tussen f en x. Deze onzichtbare operator associeert naar links. Dat wil zeggen: de expressie plus 1 2 vat de interpretator op 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 haar beurt een integer kan verwerken (2 in het voorbeeld).

Associatie van functietoepassing naar rechts zou onzin zijn: in de expressie plus (1 2) zou eerst 1 op 2 worden toegepast (maar dat is onzin) 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
```

waarbij E het type van het eindresultaat van de functie is. Als we hierin alle haakjes zouden schrijven – die weggelaten mogen worden omdat afgesproken is dat -> rechts-associatief is – krijgen we:

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

Zonder haakjes is dit veel overzichtelijker. De associatie van -> en functietoepassing is daarom zo gekozen, dat currying geruisloos verloopt: functietoepassing associeert naar links en -> associeert naar rechts. Anders gezegd: als er geen haakjes staan, staan ze zo, dat currying werkt.

OPGAVE 1.14

Welke haakjes zijn overbodig in de volgende expressies? Raadpleeg in geval van twijfel de interpretator.

a `(plus 3) (plus 4 5)`
b `sqrt(3) + (sqrt 4)`
c `(a->b)->(c->d)`

Als de interpretator een type afdrukt, bijvoorbeeld als antwoord op een opdracht :type, dan is hij daarbij zo zuinig mogelijk met haakjes. Zo wordt het type van de functiesamenstellingsoperator bijvoorbeeld als volgt afgedrukt:

```
Hugs> :type (.)
(.) :: (a -> b) -> (c -> a) -> c -> b
Hugs>
```

Het is niet nodig om haakjes om c->b te schrijven, omdat -> naar rechts associeert.

Als we er op deze manier tegenaan kijken, zouden we kunnen zeggen dat (.) een functie met drie parameters is: twee functies van type a->b, respectievelijk c->a, en een waarde van type c.

OPGAVE 1.15
Bekijk nog eens de definitie van de functie oneven, waarbij we de operator (.) ditmaal als prefix-functie gebruiken:

```
oneven n  =  (.) not even n
```

Wat zijn de typen van not, even, n en oneven n? Vergelijk dit met de typen van de drie parameters en het resultaat van de operator (.).
We kunnen oneven ook definiëren met partiële parametrisatie:

```
oneven  =  (.) not even
```

Vergelijk ook nu de typen van de parameters not, even en het resultaat oneven met het polymorfe type van (.).

Soms is het handiger om het op deze manier te bekijken en soms is het overzichtelijker om (.) te beschouwen als een functie met twee functieparameters en een functie als resultaat, zoals we in de vorige paragraaf deden. Omdat (.) een gecurryde functie is, is de situatie in werkelijkheid nog anders: (.) is een functie met één functieparameter, die een functie oplevert, die op haar beurt een functieparameter heeft en dan wéér een functie oplevert, die uiteindelijk de derde parameter opneemt. Haskell heet met recht een functionele programmeertaal.

6.7 TYPEDEFINITIES

Typespecificaties van functies kunnen ingewikkeld zijn als de argumenten complex samengestelde typen zijn, zoals bij veelvuldig gebruik van lijsten en tupels. Een punt in het vlak kan bijvoorbeeld worden gerepresenteerd door twee Floats. De typen van de eenvoudigste functies die men op punten zou willen definiëren, zijn nog wel te overzien:

```
afstand_oorsprong  ::  (Float,Float) -> Float
afstand  ::  (Float,Float) -> (Float,Float) -> Float
```

Maar lastiger wordt het bij lijsten van punten en vooral bij hogere-ordefuncties. Een veelhoek is bijvoorbeeld voor te stellen als een lijst van punten en zo'n veelhoek zou getransformeerd kunnen worden door elk hoekpunt door een bepaalde functie op een ander punt af te beelden:

```
opp_veelhoek     :: [(Float,Float)] -> Float
transf_veelhoek  :: ((Float,Float)->(Float,Float))
                    -> [(Float,Float)] -> [(Float,Float)]
```

Typedefinitie

In zo'n geval komt een *typedefinitie* van pas. Met een typedefinitie is het mogelijk om een duidelijkere naam te geven aan een type, bijvoorbeeld:

```
type Punt  =  (Float,Float)
```

Na deze typedefinitie zijn de typespecificaties eenvoudiger te schrijven:

```
afstand_oorsprong  :: Punt -> Float
afstand            :: Punt -> Punt -> Float
opp_veelhoek       :: [Punt] -> Float
transf_veelhoek    :: (Punt->Punt) -> [Punt] -> [Punt]
```

Nog beter is het om ook voor 'veelhoek' een typedefinitie te maken:

```
type Veelhoek     =  [Punt]
opp_veelhoek      ::  Veelhoek -> Float
transf_veelhoek   ::  (Punt->Punt) -> Veelhoek -> Veelhoek
```

Enkele opmerkingen bij het gebruik van een typedefinitie:
– 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 *typespecificatie* specificeert het type van een functie; een *typedefinitie* definieert een nieuwe naam voor een type.

De nieuw gedefinieerde naam beschouwt de interpretator puur als afkorting. Bij het typeren van een expressie krijgen we gewoon weer (Float,Float) te zien in plaats van Punt. Als we twee verschillende namen aan één type geven, dan mogen we die namen door elkaar gebruiken, bijvoorbeeld:

```
type Punt     =  (Float,Float)
type Complex  =  (Float,Float)
```

Punt is hetzelfde als Complex, is hetzelfde als (Float,Float).

Type String

Het standaardtype *String* is in de prelude op onderstaande wijze gedefinieerd. String is hetzelfde als een lijst van characters.

```
type String = [Char]
```

In paragraaf 7 beschrijven we een methode om een echt *nieuw* type te definiëren.

7 Nieuwe datastructuren

7.1 DATADECLARATIES

Datatype

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 declareren.
Een datatype is een type dat gekenmerkt is door de manier waarop elementen kunnen worden opgebouwd. Het begrip 'lijst' is een datatype. We kunnen een lijst als volgt recursief definiëren. Een lijst is:
– de lege lijst
of:
– een element op kop van een lijst.

Met behulp van deze definitie is met de operator : iedere lijst te construeren. Bijvoorbeeld de lijst [3] is 3 op kop van de lege lijst:

```
3 : []
```

Lijst [1, 2, 3] is 1 op kop van de lijst [2,3]:

```
1 : (2 : (3 : []))
```

De definitie komt terug in de patronen van definities van functies op lijsten, bijvoorbeeld:

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

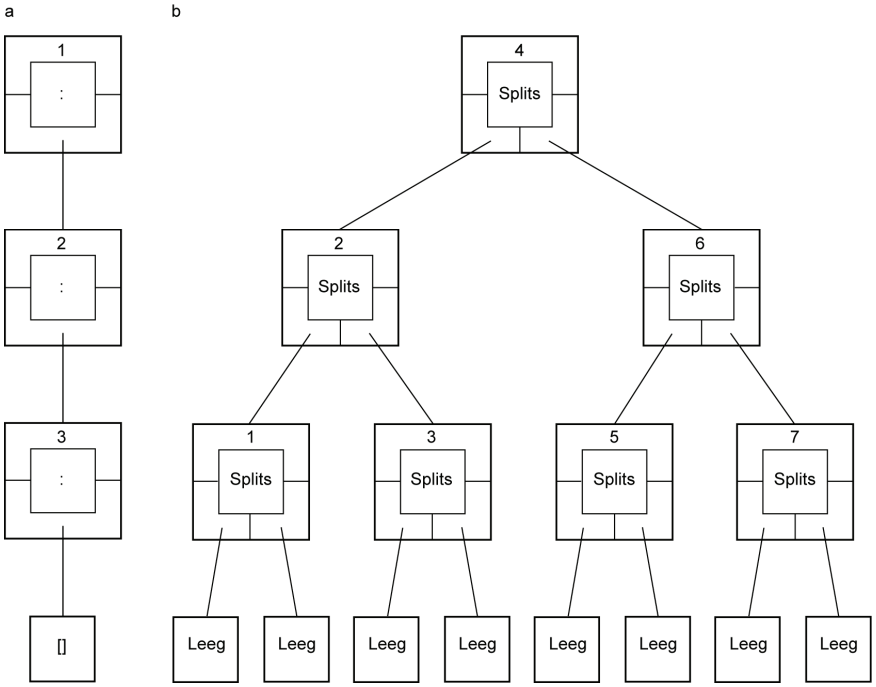
Door de functie te definiëren voor de patronen 'lege lijst' en 'een element op kop van een lijst' is de functie geheel gedefinieerd.

Lineaire structuur

Boomstructuur

Binaire boom

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 figuur 1.2 vergelijken we een lijst met een boom die zich steeds in tweeën splitst: een *binaire boom*.



FIGUUR 1.2 Voorbeelden van een liststructuur (a) en een boomstructuur (b)

In kleine vierkantjes is in de figuur aangegeven hoe de structuur is opgebouwd. In het geval van de lijst is dat gedaan door middel van de operator : met twee parameters – in de figuur gesymboliseerd door aansluitende u-vormige blokken rond het kleine vierkantje – of door middel van [ ] zonder parameters. De boom is niet opgebouwd met operatoren, maar met een speciaal soort functies: Splits (met drie

parameters) of Leeg (zonder parameters). Analoog aan de definitie van een lijst kunnen we een binaire boom als volgt recursief definiëren.
Een binaire boom is:
– de lege binaire boom
of:
– een element met een linker binaire boom en een rechter binaire boom.

Constructor-functies en operatoren

Functies waarmee een datastructuur wordt opgebouwd, heten *constructor-functies*. De constructor-functies van de boom zijn Splits en Leeg. Namen van constructor-functies beginnen met een hoofdletter, om ze te onderscheiden van 'gewone' functies. Constructor-functies mogen we ook als operator schrijven door ze te voorzien van back quotes. Als we constructor-operatoren aanduiden met symbolen (zoals +, /, %) moeten ze met een dubbele punt beginnen. De constructor-operator : van lijsten is daar een voorbeeld van.

Datatypedeclaratie

Welke constructor-functies voor een nieuw type gebruikt kunnen worden, geven we aan met een *datatypedeclaratie*, of kortweg *datadeclaratie*. Daarin staan ook de typen van de parameters van de constructor-functies en er staat of het nieuwe type polymorf is. De datadeclaratie voor bomen zoals hiervoor besproken luidt bijvoorbeeld als volgt:

```
data Boom a  =  Leeg
             |  Splits a (Boom a) (Boom a)
```

Let erop dat het symbool | hier in een geheel andere betekenis is gebruikt dan in functiedefinities met voorwaarden, hoewel het ook in dit geval uitgesproken kan worden als 'of'.

Het woord data is weer een van de gereserveerde woorden die niet als functienaam gebruikt mogen worden, omdat ze een vaste betekenis hebben. U kunt deze declaratie als volgt omschrijven: 'Een boom met elementen van type a kan op twee manieren worden opgebouwd: door de functie Leeg te gebruiken, of door de functie Splits toe te passen op drie parameters (één van type a en twee van type boom met elementen van a).'

Anders dan bij typedefinities wordt door datadeclaraties dus een geheel nieuw type gedeclareerd, compleet met een opsomming van de manieren waarop de elementen te construeren zijn. Een typedefinitie is daarentegen niet meer dan een manier om een naam te geven aan een bepaald type.

7.2 FUNCTIES OP BOMEN

Bomen kunnen worden opgebouwd door de constructor-functies in een expressie te gebruiken. De boom uit figuur 1.2b wordt bijvoorbeeld weergegeven door de volgende expressie:

```
Splits 4 (Splits 2 (Splits 1 Leeg Leeg)
                   (Splits 3 Leeg Leeg)
         )
         (Splits 6 (Splits 5 Leeg Leeg)
                   (Splits 7 Leeg Leeg)
         )
```

Het is niet verplicht om deze expressie zo mooi over de regels te spreiden. Voor de interpretator is het volgende ook acceptabel:

```
Splits 4(Splits 2(Splits 1 Leeg Leeg)(Splits 3 Leeg Leeg))
(Splits 6(Splits 5 Leeg Leeg)(Splits 7 Leeg Leeg))
```

Voor een menselijke lezer is een duidelijke lay-out van expressies echter wel wenselijk.

Functies op een boom kunnen we definiëren door voor elke constructor-functie een patroon te maken. Naast de in paragraaf 4.3 en 5.5 genoemde mogelijkheden, mogen we namelijk ook constructor-functies in patronen gebruiken. De volgende (polymorfe) functie bepaalt bijvoorbeeld het aantal Splits-constructies in een boom:

```
omvang  ::  Boom a -> Int
omvang Leeg                    =  0
omvang (Splits x lBoom rBoom)  =  1 + omvang lBoom
                                   + omvang rBoom
```

Vergelijk deze functie met de functie length op lijsten. De functie is toe te passen op binaire bomen met ieder elementtype, bijvoorbeeld:

```
Le01> omvang (Splits "aap" (Splits "noot" Leeg Leeg) Leeg)
2
Le01>
```

OPGAVE 1.16
Schrijf een functie somBoom die de som berekent van de gehele getallen die in een Boom zijn opgeslagen. Geef ook het type van deze functie.

7.3 DATADECLARATIES VOOR SPECIALE TYPEN

Behalve voor de constructie van bomen van allerlei vorm, is een datadeclaratie nog op een paar speciale manieren te gebruiken. Het datatypemechanisme is zo universeel bruikbaar, dat er ook dingen mee te maken zijn waarvoor in andere talen vaak aparte constructies nodig zijn. Voorbeelden hiervan zijn: opsommingstypen (enumeration types) en de disjuncte vereniging van typen (disjoint unions). Ook het type Rational van module Ratio wordt met behulp van een datadeclaratie gemaakt. We zullen deze drie voorbeelden nu bespreken.

Opsommingstype

Een *opsommingstype* ontstaat als alle constructor-functies in een datadeclaratie nul parameters hebben. Dat dat mogelijk is, bleek al eerder: de constructor-functie Leeg van het type Boom had bijvoorbeeld geen parameters. Als *geen enkele* constructor-functie parameters heeft, wordt daarmee een type gedefinieerd dat precies zoveel elementen bevat als er constructor-functies zijn: een opsommingstype. De constructor-functies dienen als constanten om deze elementen aan te duiden. Een voorbeeld:

```
data Richting  =  Noord | Oost | Zuid | West
```

Functies op dit soort typen zijn gewoon met behulp van patronen te schrijven, bijvoorbeeld:

```
stap  ::  Richting -> (Int,Int) -> (Int,Int)
stap Noord (x,y)   =  (x,y+1)
stap Oost  (x,y)   =  (x+1,y)
stap Zuid  (x,y)   =  (x,y-1)
stap West  (x,y)   =  (x-1,y)
```

De voordelen van zo'n opsommingstype boven een codering met integers of characters zijn:
– functiedefinities zijn duidelijker doordat we de namen van de elementen kunnen gebruiken, in plaats van obscure coderingen
– het typesysteem geeft foutmeldingen als we richtingen per ongeluk zouden optellen; als de richtingen door integers gecodeerd zijn, geeft dit geen foutmelding, met alle vervelende gevolgen van dien.

Type Bool

Opsommingstypen zijn niets nieuws: in feite is het type *Bool* in de prelude op deze manier gedefinieerd:

```
data Bool  =  False | True
```

Dit is ook de reden dat we False en True met een hoofdletter moeten schrijven: het zijn de constructor-functies van Bool.

Disjuncte vereniging van typen

Een tweede speciaal geval waarbij we datadeclaraties kunnen gebruiken, is de constructie van de *disjuncte vereniging van typen*. 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 willen we echter een lijst maken, waarvan bijvoorbeeld sommige elementen integers zijn en andere elementen characters.

Met een datadeclaratie is het mogelijk een type IntOfChar te maken, dat als elementen zowel de integers als de characters heeft:

```
data IntOfChar  =  EenInt Int
                |  EenChar Char
```

Hiermee kunnen we een gemengde lijst maken:

```
xs ::  [IntOfChar]
xs  =  [EenInt 1, EenChar 'a' , EenInt 2, EenInt 3]
```

De enige prijs die we moeten betalen, is dat we elk element moeten markeren met de constructor-functie EenInt of EenChar. Deze functies zijn te beschouwen als conversiefuncties:

```
Le01> :type EenInt
EenInt :: Int -> IntOfChar
Le01> :type EenChar
EenChar :: Char -> IntOfChar
Le01>
```

De typespecificaties van constructor-functies mogen overigens niet als zodanig in een Haskell-programma staan; hun type is immers al impliciet bepaald door de datadeclaratie.

Type Rational

Een derde toepassing van datadeclaraties is de constructie van het type *Rational*. Met een datadeclaratie is het mogelijk om het volgende type te maken:

```
data Ratio  =  Rat Integer Integer
```

Om een breuk te construeren met een teller 3 en een noemer 5 moeten we schrijven: Rat 3 5. Net als bij de disjuncte vereniging kan Rat worden beschouwd als conversiefunctie met twee Integer-parameters naar Ratio. Op dit type kunnen we bijvoorbeeld de functies teller en noemer definiëren die respectievelijk de teller en noemer van een breuk opleveren:

```
teller  ::  Ratio -> Integer
teller (Rat x y) = x
noemer  ::  Ratio -> Integer
noemer (Rat x y) = y
```

Constructor-operator

In plaats van een constructor-functie kunnen we ook een *constructor-operator* gebruiken, bijvoorbeeld door Rat met behulp van back quotes als operator te schrijven of een symbool te gebruiken voorafgegaan door een dubbele punt:

```
data Ratio  =  Integer :% Integer
```

De breuk met teller 3 en noemer 5 moeten we nu schrijven als 3 : % 5 en de functies teller en noemer noteren we in dit geval als:

```
teller  ::  Ratio -> Integer
teller (x:%y) = x
noemer  ::  Ratio -> Integer
noemer (x:%y) = y
```

In de module Ratio is type Ratio als volgt gedefinieerd:

```
data Integral a => Ratio a  =  a :% a
```

Net als bij type Boom uit paragraaf 7.1 staat er in de datadeclaratie een typevariabele. Voor deze variabele a mogen we echter alleen de typen gebruiken die tot de klasse Integral behoren, dus Int en Integer. Het type Ratio is *overloaded* voor deze typen.

Het type *Rational* verkrijgen we vervolgens door in een typedefinitie de naam Rational gelijk te stellen aan de overloaded versie van Ratio voor Integer:

```
type Rational  =  Ratio Integer
```

Op type Ratio zijn in de module Ratio onder andere de functies gedefinieerd met de volgende typespecificaties:

```
(%)          ::  Integral a => a -> a -> Ratio a
numerator    ::  Integral a => Ratio a -> a
denominator  ::  Integral a => Ratio a -> a
```

De functies zijn overloaded voor de typen Int en Integer. Met de operator % hebben we al kennis gemaakt: deze maakt uit twee gehele getallen een nieuwe breuk en vereenvoudigt deze zo mogelijk. De functie numerator geeft de teller van een breuk en de functie denominator geeft de noemer van een breuk. Ook de rekenkundige operatoren +, –, * en / en de vergelijkingsoperatoren zijn toepasbaar op type Ratio en dus op type Rational.

ZELFTOETS

1 Omschrijf in eigen woorden de volgende begrippen:
   a hogere-ordefunctie
   b 'gecurryde' functie
   c polymorfe functie
   d overloaded functie
   e constructor-functie

2 a Leg het verschil uit tussen een typedefinitie en een datadeclaratie.
   b Beschrijf de verschillen en overeenkomsten tussen operatoren en functies.

3 Geef drie verschillende definities van de functie product, die alle elementen van een lijst van gehele getallen met elkaar vermenigvuldigt:
   a een recursieve definitie met gevalsonderscheid door voorwaarden
   b een recursieve definitie met gevalsonderscheid door patronen
   c een definitie met gebruik van foldr.
   Kies daarbij een zinvolle waarde voor het product van alle getallen in een lege lijst.

4 Schrijf een (gecurryde) functie eersteDieIs met twee parameters: een functie a->Bool en een lijst [a]. De functie moet het eerste element van de lijst opleveren waarvoor de parameterfunctie True oplevert. Deze functie zou als volgt gebruikt kunnen worden:

```
Le01> eersteDieIs even [1,3,6,10,15]
6
Le01> eersteDieIs isLower "Abcdefghijklmn"
'b'
Le01>
```

   U mag ervan uitgaan dat in de lijst een element aanwezig is dat aan de parameterfunctie voldoet. Geef ook het type van de functie.

5 Met de volgende datadeclaratie worden binaire bomen beschreven:

```
data Boom a  =  Leeg
             |  Splits a (Boom a) (Boom a)
```

   Schrijf een functie boomElementen die een lijst oplevert van elementen die in een boom zijn opgeslagen. Geef ook het type van boomElementen.

TERUGKOPPELING

1 Uitwerking van de opgaven

1.1 Start de Haskell-interpretator. Tik in

```
Hugs> :edit D:\Examples\Nieuw.hs
```

(of kies een andere bestandsnaam of pad). Zet in het bestand met behulp van de tekstverwerker de volgende tekst:

```
module Nieuw where
derde x = x*x*x
```

Sla het bestand op. Geef, indien dat nog niet eerder is gebeurd, het commando

```
Hugs> :load D:\Examples\Nieuw.hs
Nieuw>
```

Gebruik de nieuwe functie door te laten uitrekenen:

```
Nieuw> derde 17 - derde 4
```

Het antwoord van de interpretator is:

```
4849
Nieuw>
```

1.2 Dit kan met behulp van de functie signum:

```
aantalOpl a b c  =  signum (b*b-4*a*c) + 1
```

1.3 a Een definitie van de faculteitsfunctie, waarbij het niet-recursieve en het recursieve geval door middel van gevalsonderscheid apart worden behandeld, is

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

b De gevallen zijn ook apart te behandelen door middel van patronen:

```
faculteit' 0      =  1
faculteit' (n+1)  =  (n+1) * faculteit' n
```

1.4 Integreren van een functie tussen twee grenzen, bepalen van de inverse van een functie, bepalen van de eventuele nulpunten van een functie, enzovoort.

1.5 De operator ++ koppelt twee lijsten aan elkaar (concatenatie), zoals bijvoorbeeld blijkt uit de aanroep

```
Le01> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Le01>
```

Het verschil met : is dat : werkt op een element en een lijst, terwijl ++ werkt op twee lijsten.

1.6 Recursieve definitie van de functie som:
a met gevalsonderscheid door middel van voorwaarden:

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

b met gevalsonderscheid door middel van patronen:

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

1.7 Hier volgen drie definities van de functie eenVanDe.
a De eerste maakt het gevalsonderscheid door middel van voorwaarden:

```
eenVanDe xs  |  xs==[]     =  False
             |  otherwise  =  head xs ||
                              eenVanDe (tail xs)
```

b De tweede definitie maakt het gevalsonderscheid door middel van patronen. Daarmee zijn de functies head en tail overbodig geworden.

```
eenVanDe' []      =  False
eenVanDe' (x:xs)  =  x || eenVanDe' xs
```

c De definitie in b heeft dezelfde structuur als de definitie van functies als som en product, en is dus ook te schrijven als aanroep van foldr.

```
eenVanDe'' xs  =  foldr (||) False xs
```

1.8 Het type van de functie map is:

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

De eerste parameter is de type van een functie, met parameter a voor het type van de parameter en parameter b voor het type van het resultaat van de functie. De tweede parameter is een lijst waarvan de elementen hetzelfde type (a) moeten hebben als de parameter van de functieparameter. Het resultaat is een lijst waarvan de elementen hetzelfde type (b) hebben als het resultaat van de parameterfunctie.

1.9 a Het type van de functies div en quot is:

```
div  ::  Integral a => a -> a -> a
quot ::  Integral a => a -> a -> a
```

Anders gezegd: het type van div en quot is a -> a -> a, waarbij type a een type moet zijn uit de klasse Integral. De functies quot en div zijn dus overloaded voor de typen uit deze klasse.
b Dit zijn de typen uit klasse Integral, te weten Int en Integer. Deze typen mogen niet door elkaar worden gebruikt. De functies zijn gedefinieerd voor ofwel type Int, danwel type Integer:

```
Int -> Int -> Int
Integer -> Integer -> Integer
```

In de interpretator kunt u dit als volgt testen door het type van de getallen expliciet op te geven:

```
Hugs> (4::Int) `div` (2::Int)
2
Hugs> (4::Integer) `div` (2::Int)
ERROR - Type error in application
*** Expression     : 4 `div` 2
*** Term           : 4
*** Type           : Integer
*** Does not match : Int

Hugs>
```

c Het type van 7.3 is:

```
7.3 :: Fractional a => a
```

De letterlijke waarde 7.3 kan dus een van de typen uit de klasse Fractional aannemen.
d De typen Float en Double maken deel uit van klasse Fractional.

1.10 De functie foldr kan in de definitie van som partieel geparametriseerd worden:

```
som  =  foldr (+) 0
```

Partiële parametrisatie van een functie met meer parameters is overigens alleen mogelijk wanneer de parameters aan het begin (linkerkant) van de parameterlijst aanwezig blijven. Het is bijvoorbeeld niet mogelijk om de functie foldr wel alvast de lijstparameter te geven, maar nog niet de operator en de startwaarde. Wel is het mogelijk om meer dan één parameter aan het eind weg te laten, bijvoorbeeld de startwaarde en de lijst.

1.11 In de definitie van oneven uit paragraaf 6.3

```
oneven n  =  (not . even) n
```

is het niet nodig om de parameter n een naam te geven. We kunnen eenvoudigweg definiëren:

```
oneven  =  not . even
```

1.12 Door de machtsverhefoperator partieel te parametriseren met 2 als rechter parameter kunnen we de kwadraatfunctie bouwen zonder haar een naam te geven. Als we een operator partieel parametriseren met één parameter, blijft er een functie met één parameter over. Deze functie kunnen we meegeven aan map, om haar te laten toepassen op alle elementen van een lijst:

```
Hugs> map (^2) [1,2,3,4,5,6,7,8,9,10]
[1,4,9,16,25,36,49,64,81,100]
Hugs>
```

1.13 In Haskell is de functie f een functie met één parameter. De definitie vindt plaats met behulp van een patroon, dat aangeeft dat de parameter een tupel met twee elementen moet zijn. Een wiskundige die niet weet wat tupels zijn, kan overigens gewoon blijven denken dat het een functie met twee parameters is, zonder dat dat voor het resultaat veel uitmaakt. De tupelmethode voor het maken van functies met twee parameters heeft echter het nadeel dat partiële parametrisatie niet mogelijk is, iets wat met de curry-methode wel kan.

1.14 a Het eerste paar haakjes is overbodig, omdat functietoepassing links-associatief is. Het tweede paar is echter wel nodig, omdat anders de eerste plus vier parameters zou krijgen.
b Beide haakjesparen zijn overbodig. Het eerste omdat haakjes om losse parameters niet nodig zijn. Wel moet er in dit geval een spatie tussen sqrt en 3, omdat er anders de naam sqrt3 zou staan. Het tweede haakjespaar is overbodig omdat functietoepassing een hogere prioriteit heeft dan optelling.
c Het tweede paar haakjes is overbodig, omdat -> naar rechts associeert. Het eerste paar is wel nodig, omdat het hier een functie met een functieparameter betreft en niet een functie met drie parameters.

1.15 Het type van not is

```
Bool -> Bool
```

Dit komt overeen met het type a -> b. Het type van even is

```
Integral a => a -> Bool
```

Dit komt overeen met het type c -> a. De variabele n heeft het type

```
Integral a => a
```

Dit komt overeen met het type c. Het resultaat, oneven n, heeft het type

```
Bool
```

Dit komt overeen met het type b. In deze definitie heeft (.) dus het type:

```
Integral a => (Bool -> Bool) -> (a -> Bool) -> a -> Bool
```

In de partieel geparametriseerde definitie is er geen derde parameter n, maar is het resultaat de functie oneven, die het type

```
Integral a => a -> Bool
```

heeft. Dit komt overeen met het type c -> b in het polymorfe type van (.).

1.16 Deze functie lijkt sterk op de recursieve definitie van som. Alleen is er nu niet één recursieve aanroep (voor de staart van de lijst), maar twee (voor de linker en de rechter deelboom):

```
somBoom  ::  Boom Int -> Int
somBoom Leeg                    =  0
somBoom (Splits n lBoom rBoom) =  n + somBoom lBoom
                                    + somBoom rBoom
```

2 Uitwerking van de zelftoets

1 Mogelijke omschrijvingen zijn:
a Een hogere-ordefunctie is een functie die een functie als parameter heeft en/of een functie als resultaat oplevert.
b Een gecurryde functie is een functie met meer dan één parameter, die gesimuleerd wordt door middel van een hogere-ordefunctie. De gecurryde functie heeft dan één parameter en levert een functie op die de overige parameters als argument verwacht om het resultaat te berekenen.
c Een polymorfe functie is een functie die op verschillende typen kan werken, mits deze typen dezelfde structuur hebben, bijvoorbeeld lijsten, bomen of functies. De polymorfe functie kan geen gebruik maken van specifieke eigenschappen van de elementen van deze structuur, zoals 'optelbaarheid'.

d We spreken over een overloaded functie als dezelfde functienaam wordt gebruikt voor verschillende operaties op een aantal verschillende typen. In feite is er sprake van verschillende functies met dezelfde naam. Overloading is een beperkte vorm van polymorfie en noemen we wel ad-hocpolymorfie.
e Een constructor-functie is een functie waarmee datastructuren kunnen worden opgebouwd. Constructor-functies definiëren we niet met een functiedefinitie, maar zijn impliciet gedefinieerd door een datadeclaratie.

2 a In een typedefinitie wordt een nieuwe naam geïntroduceerd voor een al bestaand type. Deze naam is als afkorting te gebruiken voor het oorspronkelijke type. In een datadeclaratie wordt een geheel nieuw type gecreëerd, ongelijk aan al bestaande typen. Daartoe worden alle manieren opgesomd om een waarde van het nieuwe type te construeren.
b Een operator heeft een (of meer) parameters, een functie kan ook een of zelfs nul parameters hebben. Een operator wordt tussen de operanden (parameters) geschreven, een functie ervoor. Een operator wordt aangeduid met één of meer symbolen, een functie met letters (en na de eerste letter eventueel cijfers, apostrofs en liggende streepjes). De overeenkomsten zijn dat zowel functies als operatoren een definitie hebben en door de programmeur zelf te definiëren en aan te roepen zijn.

3 a Als product van een lege lijst gebruiken we de waarde 1, omdat 1 het neutrale element van vermenigvuldigen is. Hier volgen drie definities van de functie product. De eerste maakt het gevalsonderscheid door middel van voorwaarden:

```
product1 xs  |  xs==[]     =  1
             |  otherwise  =  head xs * product1 (tail xs)
```

b De tweede definitie maakt het gevalsonderscheid door middel van patronen. Daarmee zijn de functies head en tail overbodig geworden.

```
product2 []      =  1
product2 (x:xs)  =  x * product2 xs
```

c De definitie heeft dezelfde structuur als de functie foldr en is dus te schrijven als aanroep van foldr.

```
product3 xs  =  foldr (*) 1 xs
```

Desgewenst kan de functie foldr daarbij partieel geparametriseerd worden:

```
product4  =  foldr (*) 1
```

4 De functie eersteDieIs is als volgt gedefinieerd:

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

5 De functie boomElementen is als volgt te definiëren:

```
boomElementen Leeg                     =  []
boomElementen (Splits x lBoom rBoom)   =  boomElementen lBoom
                                          ++ [x] ++
                                          boomElementen rBoom
```

Het type wordt gespecificeerd door:

```
boomElementen  ::  Boom a -> [a]
```

Dit is een polymorfe functie die op bomen werkt met elementen van een willekeurig type en lijsten oplevert met elementen van datzelfde type.