

Functioneel Programmeren

© Copyright 1992–2002 Informatica-instituut, Universiteit Utrecht

Deze tekst mag voor educatieve doeleinden gereproduceerd worden op de volgende voorwaarden:

- de tekst wordt niet veranderd of ingekort;
- in het bijzonder wordt deze mededeling ook gereproduceerd;
- de kopieën worden niet met winstoogmerk verkocht

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

1e druk (informatica-versie) september 1992

2e druk (informatica-versie) februari 1993

3e druk (informatica-versie) september 1993

4e druk (informatica-versie) september 1994

5e druk (informatica-versie) september 1995

6e druk (CKI-versie) oktober 1995

7e druk (informatica-versie) september 1996

8e druk (CKI-versie) oktober 1996

9e druk (CKI-versie) oktober 1997

10e druk (CKI-versie) oktober 1998

11e druk (CKI-versie) november 2002

Inhoudsopgave

1	Functioneel Programmeren	1
1.1	Functionele talen	1
1.1.1	Functies	1
1.1.2	Talen	1
1.2	De Helium-interpreter	2
1.2.1	Expressies uitrekenen	2
1.2.2	Functies definiëren	3
1.2.3	Opdrachten aan de interpreter	4
1.3	Standaardfuncties	4
1.3.1	Ingebouwd/voorgedefinieerd	4
1.3.2	Namen van functies en operatoren	5
1.3.3	Functies op getallen	6
1.3.4	Boolese functies	7
1.3.5	Functies op lijsten	7
1.3.6	Functies op functies	8
1.4	Functie-definities	8
1.4.1	Definitie door combinatie	8
1.4.2	Definitie door gevalsonderscheid	9
1.4.3	Definitie door patroonherkenning	9
1.4.4	Definitie door recursie of inductie	11
1.4.5	Layout en commentaar	11
1.5	Typering	12
1.5.1	Soorten fouten	12
1.5.2	Typering van expressies	14
1.5.3	Polymorfie	15
1.5.4	Functies met meer parameters	16
	Opgaven	16
2	Getallen en functies	19
2.1	Operatoren	19
2.1.1	Operatoren als functies en andersom	19
2.1.2	Prioriteiten	19
2.1.3	Associatie	20
2.1.4	Definitie van operatoren	21
2.2	Currying	21
2.2.1	Partieel parametriseren	21
2.2.2	Haakjes	22
2.3	Functies als parameter	23
2.3.1	Functies op lijsten	23
2.3.2	Iteratie	24
2.3.3	Samenstelling	25
2.3.4	De lambda-notatie	26
2.4	Numerieke functies	26
2.4.1	Rekenen met gehele getallen	26
2.4.2	Numeriek differentiëren	29
2.4.3	Zelfgemaakte wortel	30
2.4.4	Nulpunt van een functie	30

2.4.5	Inverse van een functie	31
	Opgaven	32
3	Datastructuren	35
3.1	Lijsten	35
3.1.1	Opbouw van een lijst	35
3.1.2	Functies op lijsten	36
3.1.3	Hogere-orde functies op lijsten	39
3.1.4	Lijsten vergelijken en ordenen	40
3.1.5	Lijsten sorteren	42
3.2	Speciale lijsten	44
3.2.1	Strings	44
3.2.2	Characters	44
3.2.3	Functies op characters en strings	45
3.2.4	Oneindige lijsten	46
3.2.5	Lazy evaluatie	47
3.2.6	Functies op oneindige lijsten	48
3.2.7	Lijst-comprehensies	50
3.3	Tupels	51
3.3.1	Gebruik van tupels	51
3.3.2	Type-definities	53
3.3.3	Rationale getallen	53
3.3.4	Tupels en lijsten	55
3.3.5	Tupels en Currying	55
3.4	Bomen	56
3.4.1	Data-definities	56
3.4.2	Zoekbomen	58
3.4.3	Speciaal gebruik van data-definities	61
	Opgaven	63
4	Algoritmen op lijsten	67
4.1	Combinatorische functies	67
4.1.1	Segmenten en deelrijen	67
4.1.2	Permutaties en combinaties	69
4.1.3	De @-notatie	71
4.2	Polynomen	71
4.2.1	Representatie	71
4.2.2	Vereenvoudiging	72
4.2.3	Rekenkundige operaties	73
	Opgaven	75
5	Algoritmen op bomen	77
5.1	Expressiebomen	77
5.1.1	Rekenkundige expressies	77
5.1.2	Symbolisch differentiëren	77
5.1.3	Andere expressiebomen	78
5.1.4	Stringrepresentatie van een boom	79
5.2	Parser combinators	79
5.2.1	The type ‘Parser’	80
5.2.2	Elementary parsers	80
5.2.3	Trivial parsers	81
5.2.4	Parser combinators	82
5.2.5	Parser transformers	83
5.2.6	Matching parentheses	83
5.2.7	More parser combinators	85
5.2.8	Analyzing options	86
5.2.9	Arithmetical expressions	88
5.2.10	Generalized expressions	88
	Opgaven	89

A	Lisp voor Helium-kenners	91
A.1	Expressies	91
A.1.1	Functie-aanroep	91
A.1.2	Operatoren	91
A.1.3	Lijsten	91
A.2	Functies op lijsten	92
A.3	Functiedefinitie	92
A.3.1	Simpele definitie	92
A.3.2	Definitie met gevalsonderscheid	92
A.3.3	Definitie met patronen	93
A.3.4	Evaluatie	93
A.4	Typering	93
A.4.1	Statisch versus dynamisch	93
A.4.2	Types van lijsten	94
A.4.3	Overloading	94
A.4.4	Polymorfie	94
A.5	Hogere-ordefuncties	95
A.5.1	Map / Mapcar	95
A.5.2	Foldr / Reduce	95
A.5.3	Currying / Lambda-notatie	95
A.6	Filosofie	95
A.6.1	Helium: referentieel transparant	95
A.6.2	Lisp: meta-circulair	96
A.7	Helium en Prolog	97
A.7.1	Lijsten	97
A.7.2	Functies en relaties	97
A.7.3	Patronen	97
A.7.4	Richtingloze definities	98
A.7.5	Constructorfuncties	98
B	ISO/ASCII tabel	99
C	Helium syntax	100
D	Helium standaardfuncties	104
E	Literatuur	107
F	Woordenlijst	109

Hoofdstuk 1

Functioneel Programmeren

1.1 Functionele talen

1.1.1 Functies

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

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

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

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

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

1.1.2 Talen

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

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

kan dus ook zo geschreven worden:

```
? sqrt 2.0
1.41421
```

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

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

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

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

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

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

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

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

Er zijn diverse standaardfuncties die lijsten manipuleren. De namen spreken vaak voor zich: `length` bepaalt bijvoorbeeld de lengte van een lijst, en `replicate` maakt een lijst met een aantal kopieën van een waarde:

```
? length [1,5,9,3]
4
? replicate 10 3
[3,3,3,3,3,3,3,3,3,3]
```

In één expressie kunnen meerdere functies gecombineerd worden. Zo is het bijvoorbeeld mogelijk om eerste een lijst te maken met behulp van `replicate`, en die vervolgens om te draaien

```
? reverse (replicate 5 2)
[2,2,2,2,2]
```

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

1.2.2 Functies definiëren

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

Gebruikelijk is dat de filenaam van het programma de extensie `.hs` heeft (Haskell-script of Helium-script). In de eerste regel van de file moet vermeld worden wat de naam van de module is, bijvoorbeeld:

```
module Nieuw where
```

Het keyword `where` duidt aan dat er nu functie-definites volgen. In de file ‘`nieuw.hs`’ kan bijvoorbeeld de definitie worden gezet van de faculteit-functie. De faculteit van een getal n (vaak

genoteerd als $n!$) is het product van de getallen van 1 tot en met n , bijvoorbeeld $4! = 1*2*3*4 = 24$. In Helium kan de definitie van de functie `fac` er als volgt uitzien:

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

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

Voordat de nieuwe functie kan worden gebruikt, moet Helium weten dat de nieuwe file functie-definities bevat. Dat kan hem meegedeeld worden met het commando `:l` (afkorting van ‘load’) dus:

```
? :l nieuw.hs
```

Daarna kan de nieuwe functie gebruikt worden:

```
? fac 6
720
```

Het is mogelijk om later definities aan een file toe te voegen.

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

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

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

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

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

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

```
? boven 10 3
120
```

1.2.3 Opdrachten aan de interpreter

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

De volgende opdrachten zijn mogelijk:

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

`:q` (quit) Met deze opdracht wordt een Helium-sessie afgesloten.

`:l file(s)` (load) Na deze opdracht kent Helium de functies die in de gespecificeerde file(s) zijn gedefinieerd.

blz. 12 `:t expressie` (type) Door deze opdracht wordt het type (zie sectie 1.5) van de gegeven expressie bepaald.

`:m` (module) Met deze opdracht kun je zien welke modules op dit moment beschikbaar zijn.

1.3 Standaardfuncties

1.3.1 Ingebouwd/voorgedefinieerd

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

```
pi = 3.1415926
```

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

blz. 3

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

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

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

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

Er is slechts een klein aantal functies die je niet zelf had kunnen definiëren, zoals de optellingsoperator. Deze functies worden door de prelude op hun beurt geïmporteerd uit de module `PreludePrim`. Van die module is de source niet te bekijken; deze functies zijn op magische wijze ingebouwd in de interpreter. Dit soort functies wordt *ingebouwde functies* genoemd; hun definitie zit ingebouwd in de interpreter (in het Engels heten ze *primitive functions*). Het aantal ingebouwde functies in de prelude is zo klein mogelijk gehouden. De meeste standaardfuncties zijn gewoon in Helium gedefinieerd. Deze functies worden *voorgedefinieerde* functies genoemd.

1.3.2 Namen van functies en operatoren

In de functie-definitie

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

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

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

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

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

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

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

blz. 56

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

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

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

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

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

Toegestane operatoren zijn bijvoorbeeld:

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

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

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

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

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

1.3.3 Functies op getallen

Er zijn twee soorten getallen beschikbaar in Helium:

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

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

De vier rekenkundige operatoren optellen (+), aftrekken (-), vermenigvuldigen (*) en delen (/) werken op gehele getallen; voor floating-pointgetallen zijn er de operatoren +., *., -. en /..

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

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

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

Een aantal functies die echt ingebouwd zijn, zijn:

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

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

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

Gehele getallen moeten kleiner blijven dan 2^{31} , anders treedt er *overflow* op:

```
? 3 * 100000000
300000000
? 3 * 1000000000
exception at LvmLang.*: integer overflow
```

Ook floating-point getallen kennen een maximum waarde (ongeveer 10^{308}), en een kleinste positieve waarde (10^{-308}). Bovendien is de rekennauwkeurigheid beperkt tot 15 significante cijfers.

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

1.3.4 Boolese functies

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

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

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

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

```
? 2+3 > 1+2
True
? 5 /= 1+4
False
```

Voor floating-point getallen is er de functie `eqFloat`:

```
? eqFloat (sqrt 2.0) 1.5
False
```

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

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

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

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

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

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

1.3.5 Functies op lijsten

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

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

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

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

```
[1, 2, 3, 4, 5]
```

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

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

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

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

1.3.6 Functies op functies

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

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

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

blz. 19

1.4 Functie-definities

1.4.1 Definitie door combinatie

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

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

Functies kunnen ook meer dan één parameter krijgen:

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

Functies met nul parameters worden meestal ‘constanten’ genoemd:

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

Elke functie-definitie heeft dus de volgende vorm:

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

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

```
negatief x = x < 0
```

```
positief x = x > 0
isnul    x = x == 0
```

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

In de definitie van de functie `abcFormule` komen de expressies `sqrt(b*b-4.0*a*c)` en `(2.0*a)` twee keer voor. Behalve dat dat veel tikwerk geeft, kost het uitrekenen van zo'n expressie onnodig veel tijd: de identieke deel-expressies worden tweemaal uitgerekend. Om dat te voorkomen, is het in Helium mogelijk om deel-expressies een naam te geven. De verbeterde definitie wordt dan als volgt:

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

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

blz. 5

1.4.2 Definitie door gevalsonderscheid

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

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

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

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

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

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

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

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

Daarbij bestaat elke 'guarded expressie' uit een |-teken, een Boolese expressie, een =-teken, en een expressie¹. Deze beschrijving is echter ook nog niet volledig...

1.4.3 Definitie door patroonherkenning

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

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

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

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

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

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

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

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

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

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

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

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

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

De volgende constructies zijn toegestaan als patroon:

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

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

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

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

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

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

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

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

1.4.4 Definitie door recursie of inductie

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

```
f x = f x
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.4.5 Layout en commentaar

Op de meeste plaatsen in een programma mag extra witte ruimte staan, om het programma overzichtelijker te maken. In bovenstaande voorbeelden zijn bijvoorbeeld extra spaties toegevoegd,

om de =-tekens van één functie-definitie netjes onder elkaar te zetten. Natuurlijk mogen er geen spaties worden toegevoegd midden in de naam van een functie of in een getal: `len gth` is iets anders dan `length`, en `1 7` iets anders dan `17`.

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

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

```

where
    a = f x y
    b = g z

```

```

where
    a = f x
    y b = g z

```

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

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

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

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

```

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

```

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

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

gelijkwaardige definities moeten evenver worden ingesprongen

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

Commentaar

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

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

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

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

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

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

1.5 Typering

1.5.1 Soorten fouten

Vergissen is menselijk, ook bij het schrijven of intikken van een functie. Gelukkig kan de interpreter waarschuwen voor sommige fouten. Als een functie-definitie niet aan de vorm-eisen voldoet, krijg

je daarvan een melding zodra deze functie geanalyseerd wordt. De volgende definitie bevat een fout:

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

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

```
Parse error "nieuw.hs" (line 9, column 12)
unexpected "="
```

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

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

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

meldt de interpreter:

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

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

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

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

```
Reading script file "nieuw.hs":
ERROR "nieuw.hs" (line 22): Type error in application
*** expression   : 1 + True
*** term         : 1
*** type         : Int
*** expected type : Bool
```

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

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

```
ERROR: Type error in application
*** expression   : length 3
*** term         : 3
*** type         : Int
*** expected type : [a]
```

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

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

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

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

voor waarschuwt.

1.5.2 Typering van expressies

Het type van een expressie kan bepaald worden met de interpreter-opdracht `:t` (afkorting van ‘type’). Achter `:t` staat de expressie die getypeerd moet worden. Bijvoorbeeld:

```
? :t True && False
True && False :: Bool
```

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

Er zijn aantal basis-types:

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

blz. 44

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

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

```
? :t ['a', 'b', 'c']
['a','b','c'] :: [Char]
? :t [True,False]
[True,False] :: [Bool]
? :t [ [1,2], [3,4,5] ]
[[1,2],[3,4,5]] :: [[Int]]
```

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

```
? [1,True]
ERROR: Type error in list
*** expression   : [1,True]
*** term         : True
*** type         : Bool
*** expected type : Int
```

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

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

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

Andere voorbeelden van types van functies zijn:

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

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

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

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

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

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

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

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

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

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

1.5.3 Polymorfie

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

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

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

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

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

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

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

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

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

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

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

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

1.5.4 Functies met meer parameters

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

blz. 4

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

blz. 8

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

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

blz. 8

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

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

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

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

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

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

Opgaven

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

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

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

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

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

1.3 Hoeveel is:

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

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

1.5 Schrijf een functie `aantalOp1` die, gegeven `a`, `b` en `c`, het aantal oplossingen van de vergelijking $ax^2 + bx + c$ oplevert, in twee versies:

- met gevalsonderscheid
- door combinatie van standaardfuncties

blz. 12

1.6 Wat is het voordeel van ‘genest’ commentaar (zie paragraaf 1.4.5)?

- 1.7 Wat is het type van de volgende functies: `tail`, `sqrt`, `pi`, `exp`, `(^)`, `(/=)` en `aantalOp1`? Hoe kan je aan de interpreter vragen om dat type te bepalen, en hoe kun je de types zelf specificeren in een programma?
- 1.8 Stel dat `x` de waarde 5 heeft. Wat is de waarde van de expressies `x==3` en `x/=3`? (Voor wie de programmertaal C kent: Wat is de waarde van deze expressies in de taal C?)
- 1.9 Wat betekent ‘*syntax error*’? Wat is het verschil tussen een *syntax error* en een *type error*?
- 1.10 Bepaal de types van `3`, `even` en `even 3`. Hoe doe je dat laatste? Bepaal nu ook het type van `head [1,2,3]` en `head [1,2,3]`. Wat gebeurt er bij het toepassen van een polymorfe functie op een actuele parameter?
- 1.11 Wat is het type van de volgende expressies (zie appendix D voor de types van de gebruikte standaardfuncties):
- `until even`
 - `until or`
 - `foldr (&&) True`
 - `foldr (&&)`
 - `foldr until`
 - `map sqrt`
 - `map filter`
 - `map map`

1.12

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

blz. 11

```

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

```

- a. Wat gebeurt er bij de aanroep `fac (-3)`?
 - b. Hoe kan je de voorwaarde waaronder een recursieve definitie zinvol is nauwkeuriger formuleren?
- 1.14 Wat is het verschil tussen een *lijst* en het wiskundige begrip *verzameling*?

1.15

- 1.16 Stel dat is gedefinieerd:

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

Wat is dan de waarde van de expressie

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


Hoofdstuk 2

Getallen en functies

2.1 Operatoren

2.1.1 Operatoren als functies en andersom

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

blz. 5

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

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

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

Het is dus toegestaan (+) 1 2 te schrijven in plaats van 1+2. Deze notatie wordt in de prelude ook gebruikt om het type van + te kunnen declareren:

```
(+) :: Int -> Int -> Int
```

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

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

2.1.2 Prioriteiten

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

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

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

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

```

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

```

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

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

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

blz. 9

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

```

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

```

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

2.1.3 Associatie

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

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

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

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

De volgende operatoren associëren naar **links**:

- de 'onzichtbare' operator functie-applicatie, dus $f x y$ betekent $(f x) y$ (de reden hiervoor wordt besproken in sectie 2.2);
- de operator `!!` (zie paragraaf 3.1.2);
- de operator `-`, dus de waarde van $8-5-1$ is 2 (zoals gebruikelijk in de wiskunde) en niet 4.

blz. 21

blz. 38

De volgende operatoren associëren naar **rechts**:

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

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

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

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

(*ambiguous* betekent: ‘dubbelzinnig’, ‘voor meerdere uitleg vatbaar’);

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

blz. 63

De volgende operatoren zijn associatief:

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

blz. 25

2.1.4 Definitie van operatoren

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

```
infixr 8 ^
```

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

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

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

blz. 4

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

of met een zelfbedacht symbool:

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

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

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

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

2.2 Currying

2.2.1 Partieel parametriseren

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

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

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

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

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

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

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

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

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

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

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

blz. 16

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

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

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

2.2.2 Haakjes

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

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

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

```
f a b c d
```

wordt opgevat als

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

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

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

of, als je alle haakjes zou schrijven:

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

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

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

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

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

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

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

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

```
kwadraat (sin pi)
```

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

2.3 Functies als parameter

2.3.1 Functies op lijsten

In een functionele programmeertaal gedragen functies zich in veel opzichten hetzelfde als andere waarden, zoals getallen en lijsten. Bijvoorbeeld:

- functies hebben een *type*;
- functies kunnen door andere functies worden opgeleverd als *resultaat* (waarvan met Currying veel gebruik wordt gemaakt);
- functies kunnen als *parameter* van andere functies worden gebruikt.

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

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

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

De functie `map` kan als volgt worden gedefinieerd:

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

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

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

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

In dat laatste voorbeeld worden de getallen waaraan 10 kleiner is in de lijst opgezocht, niet de getallen die kleiner zijn dan 10! Als de lijstelementen van type `a` zijn, heeft de functie-parameter van `filter` het type `a->Bool`. Ook de definitie van `filter` is recursief:

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

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

Bruikbare hogere-orde functies kun je op het spoor komen door de overeenkomst in functie-definities op te sporen. Bekijk bijvoorbeeld de definities van de functies `sum` (die de som van een lijst getallen

berekent), `product` (die het product van een lijst getallen berekent) en `and` (die kijkt of een lijst Boolese waarden allemaal `True` zijn):

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

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

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

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

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

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

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

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

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

is gelijk aan de waarde van de expressie

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

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

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

2.3.2 Iteratie

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

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

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

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

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

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

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

Zoals alle hogere-orde functies kan `until` goed aangeroepen worden met partieel geparаметriseerde functies. Onderstaande expressie berekent bijvoorbeeld de eerste macht van twee die groter of gelijk

is aan 1000 (begin met 1 en verdubbel net zo lang, tot 1000 kleiner is dan het resultaat):

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

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

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

```
? until ((==)0) ((+)1) 1
ctrl-C
?
```

2.3.3 Samenstelling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.4 De lambda-notatie

blz. 21

In paragraaf 2.2.1 werd opgemerkt dat de functie die je als parameter meegeeft aan een andere functie vaak ontstaat door partiële parametrisatie:

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

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

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

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

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

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

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

\backslash *patroon* \rightarrow *expressie*

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

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

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

2.4 Numerieke functies

2.4.1 Rekenen met gehele getallen

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

$$\begin{array}{r} 1\ 2\ / \ 3\ 4\ 5 \ \ 2\ 8 \\ \underline{2\ 4} \\ 1\ 0\ 5 \\ \underline{9\ 6} \\ 9 \end{array}$$

is het quotiënt 28 en de rest 9.

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

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

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

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

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

Berekenen van een lijst priemgetallen

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

```
deelbaar :: Int -> Int -> Bool
deelbaar t n = t 'rem' n == 0
```

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

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

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

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

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

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

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

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

Bepalen van de dag van de week

Op welke dag valt het laatste oudjaar van deze eeuw?

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

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

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

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

blz. 44

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

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

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

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

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

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

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

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

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

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

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

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

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

Een andere manier om dit te definiëren is:

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

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

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

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

(Einde voorbeelden).

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

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

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

2.4.2 Numeriek differentiëren

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

```
? 10.0 /. 6.0
1.6666667
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.3 Zelfgemaakte wortel

blz. 31

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

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

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

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

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

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

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

blz. 24

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

```
wortel x = until goedGenoeg verbeter 1.0
  where verbeter y = 0.5*(y+.x/.y)
        goedGenoeg y = y*.y ~ = x
```

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

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

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

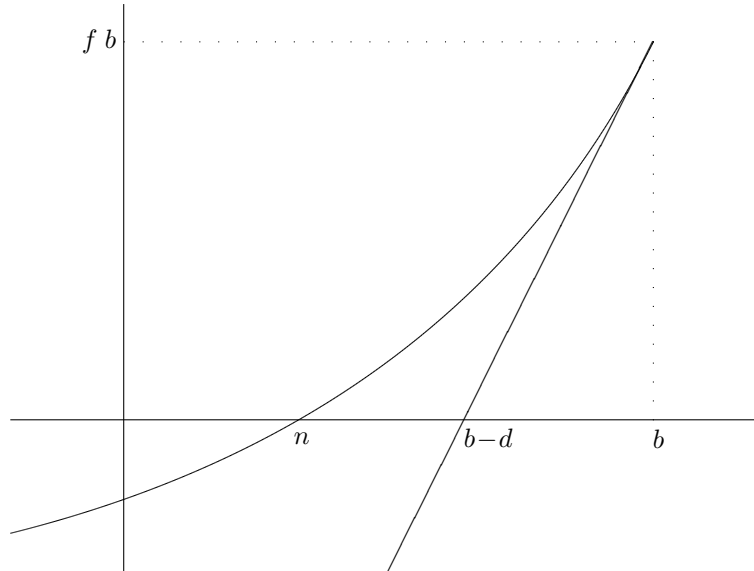
Iets anders dat opvalt aan de definitie van `verbeter` en `goedGenoeg` is dat deze functies, behalve van hun parameter `y`, ook gebruik kunnen maken van `x`. Voor deze functies is het dus alsof `x` een constante is. (Vergelijk de definities van de ‘constanten’ `d` en `n` in de definitie van `abcFormule`’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

blz. 8

2.4.4 Nulpunt van een functie

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

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



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

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

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

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

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

blz. 29

2.4.5 Inverse van een functie

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

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

Ook de derdemachtswortel kan op deze manier berekend worden:

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

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

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

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

```
inverse g a = nulpunt f 0.0
```

blz. 24

```
where f x = g x -. a
```

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

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

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

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

wat ook geschreven kan worden als

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

omdat `->` naar rechts associeert.

blz. 30

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

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

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

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

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

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

blz. 30

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

Opgaven

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

2.2 Welke haakjes zijn overbodig in de volgende expressies?

- (plus 3) (plus 4 5)
- `sqrt(3.0) + (sqrt 4.0)`
- (+) (3) (4)
- `(2*3)+(4*5)`
- `(2+3)*(4+5)`
- `(a->b)->(c->d)`

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

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

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

- 2.6** Geef een voorbeeld van een functie met de volgende types:
- `(Float -> Float) -> Float`
 - `Float -> (Float -> Float)`
 - `(Float -> Float) -> (Float -> Float)`
- 2.7** In paragraaf 2.3.3 wordt beweed dat `na` ook beschouwd kan worden als functie met één parameter. Hoe kun je dat zien aan het type? Geef een definitie van `na` in de vorm `na y = ...` blz. 25
- 2.8** Schrijf een functie die bepaalt hoeveel jaar een bedrag op de bank moet staan om, bij een gegeven rente, een gegeven eindkapitaal te kunnen incasseren.
- 2.9** Geef een definitie van `derdemachtswortel` die geen gebruik maakt van numeriek differentiëren (ook niet indirect via `nulpunt`).
- 2.10** Definiëer de functie `inverse` uit paragraaf 2.4.5 met gebruikmaking van de lambda-notatie. blz. 31
- 2.11** Waarom kunnen de functies `wortel` en `derdemachtswortel` wel worden geschreven zonder gebruik te maken van de functie `diff`, maar is het niet mogelijk om zo een algemene functie `inverse` te schrijven?
- 2.12** Schrijf een functie `integraal`, die de integraal van een functie tussen twee grenzen berekent. De functie werkt door het integratiegebied in een (te specificeren) aantal gebiedjes te verdelen, en op elk gebiedje de functie te benaderen door een lineaire functie. In welke volgorde kunnen de parameters het beste worden meegegeven, om de functie handig partieel te kunnen parameteriseren?

Hoofdstuk 3

Datastructuren

3.1 Lijsten

3.1.1 Opbouw van een lijst

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

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

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

Opsomming

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

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

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

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

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

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

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

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

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

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

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

blz. 15 in een polymorf type waar een willekeurig type ingevuld mag worden, staan type-variabelen (zie paragraaf 1.5.3), dus het type van de lege lijst is `[a]`:

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

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

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

Opbouw met :

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

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

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

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

Numerieke intervallen

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

```
? [1 .. 5]
[1, 2, 3, 4, 5]
? [1, 3 .. 15]
[1, 3, 5, 7, 9, 11, 13, 15]
```

blz. 5 (Hoewel de punt gebruikt mag worden als symbool in operatoren, is `..` geen operator. Het is namelijk één van symbolen-combinaties die in paragraaf 1.3.2 werden gereserveerd voor speciaal gebruik.)

Je kunt je voorstellen dat de waarde van de expressie `[x..y]` zou worden berekend door `enumFromTo x y` aan te roepen, die als volgt gedefinieerd kan worden:

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

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

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

3.1.2 Functies op lijsten

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

blz. 9
blz. 11
blz. 24 Een aantal definities van functies op lijsten zijn al ter sprake gekomen: `head` en `tail` in paragraaf 1.4.3, `sum` en `length` in paragraaf 1.4.4, en `map`, `filter` en `foldr` in paragraaf 2.3.1. Hoewel dit allemaal standaardfuncties zijn die in de prelude worden gedefinieerd, en ze dus niet zelf gedefinieerd hoeven te worden, is het toch belangrijk om hun definitie te bekijken. Ten eerste omdat het goede voorbeelden zijn van definities van functies op lijsten, ten tweede omdat de definitie vaak de duidelijkste beschrijving geeft wat een standaardfunctie doet.

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

Lijsten samenvoegen

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

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

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

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

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

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

De definitie van `concat` is als volgt:

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

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

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

Delen van een lijst selecteren

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

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

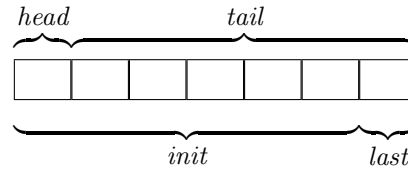
```
head      :: [a] -> a
head (x:xs) = x
tail     :: [a] -> [a]
tail (x:xs) = xs
```

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

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

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

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



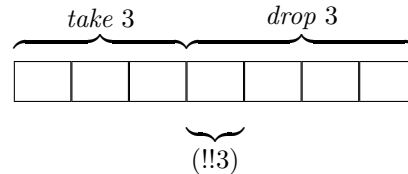
De functie `init` selecteert alles *behalve* het laatste element. Ook daarvoor is weer recursie nodig:

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

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

blz. 28

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



Deze functies zijn als volgt gedefinieerd:

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

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

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

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

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

blz. 27

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

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

Lijsten omdraaien

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

```
reverse [] = []
```

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

Eigenschappen van lijsten

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

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

3.1.3 Hogere-orde functies op lijsten

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

map, filter en foldr

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

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

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

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

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

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

Deze drie standaardfuncties worden in de prelude recursief gedefinieerd. Ze werden eerder besproken in paragraaf 2.3.1.

blz. 24

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

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

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

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

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

takeWhile en dropWhile

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

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

Vergelijk deze definitie met die van `filter`.

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

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

foldl

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

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

$$\begin{array}{ccccccc} \text{xs} & = & & [& 1 & , & 2 & , & 3 & , & 4 & , & 5 &] \\ & & & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\ \text{foldl } (+) \text{ 0 xs} & = & & ((((((0 & + & 1) & + & 2) & + & 3) & + & 4) & + & 5) \end{array}$$

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

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

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

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

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

3.1.4 Lijsten vergelijken en ordenen

Lijsten van integers vergelijken

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

```
eqListInt      :: [Int] -> [Int] -> Bool
eqListInt [] [] = True
eqListInt [] (y:ys) = False
eqListInt (x:xs) [] = False
eqListInt (x:xs) (y:ys) = x==y && eqListInt xs ys
```

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

Lijsten van willekeurige types vergelijken

Behalve lijsten van integers wil je natuurlijk ook wel eens lijsten vergelijken met elementen van een ander type. Je wilt wellicht lijsten vergelijken waarvan de elementen booleans zijn, of strings. In dat geval kun je niet meer de operator `==` gebruiken om de elementen te vergelijken, want die kan alleen gebruikt worden om integers te vergelijken. Maar hoe moet je de elementen dan vergelijken?

We kunnen de functie waarmee de elementen worden vergeleken *als extra parameter* meegeven aan de functie. De vergelijkingsfunctie is zelf ook een functie, maar functies zijn ook zeer wel mogelijk als parameter, zoals dat ook bij bijvoorbeeld `map` het geval was. De definitie wordt dan als volgt:

```
eqList :: (a->a->Bool) -> [a] -> [a] -> Bool
eqList test [] [] = True
eqList test [] (y:ys) = False
eqList test (x:xs) [] = False
eqList test (x:xs) (y:ys) = test x y && eqList xs ys
```

Deze functie is in de prelude inderdaad gedefinieerd, en we kunnen er dus lijsten van elk gewenst type mee vergelijken:

```
? eqList (==) [1,2,3] [1,2,3]
True
? eqList eqBool [True,False] [False,True]
False
```

Door `eqList` partiël te parametriseren kunnen we een vergelijkingsfunctie voor lijsten maken, die we vervolgens weer kunnen meegeven aan `eqList`. Op die manier kun je bijvoorbeeld lijsten van lijsten van integers vergelijken:

```
? eqList (eqList (==)) [[1,2],[3,4]] [[1,2],[4,3]]
False
```

Voor vergelijking van de enkele basistypen zijn in de prelude functies `eqBool`, `eqChar` en `eqString` aanwezig.

Lijsten ordenen

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

Omdat operatoren zoals $<$ al zijn gereserveerd voor het vergelijken van integers, moeten we voor het vergelijken van lijsten een andere naam verzinnen. Of liever gezegd vier namen: voor $<$, \leq , $>$ en \geq . Net als in het geval van `eqList` zouden die functies dan een functie als parameter moeten krijgen voor het vergelijken van de elementen én nog een voor het ordenen van de elementen. Dat kan, maar het wordt een beetje ingewikkeld, en daarom is in de prelude voor een andere aanpak gekozen.

Om te beginnen is er een type `Ordering` beschikbaar met drie mogelijke waarden. Zoals een expressie van type `Bool` de waarden `True` en `False` kan hebben, zo kan een expressie van type `Ordering` de waarden `LT` ('less than'), `EQ` ('equal') of `GT` ('greater than'). Voor de elementaire typen zijn er functies die de onderlinge ligging van twee waarden kunnen aangeven, zoals de functie `ordInt` die als volgt is gedefinieerd:

```
ordInt :: Int -> Int -> Ordering
ordInt x y
  | x < y   = LT
  | x == y  = EQ
  | otherwise = GT
```

Die functie zou in een expressie gebruikt kunnen worden:

```
? ordInt 3 4
LT
```

maar hij is vooral bedoeld om als parameter mee te geven aan de functie `ordList`. Die is in de prelude als volgt gedefinieerd:

```
ordList :: (a->a->Ordering) -> [a] -> [a] -> Ordering
ordList test [] (y:ys) = LT
ordList test [] [] = EQ
ordList test (x:xs) [] = GT
ordList test (x:xs) (y:ys) =
  case test x y of
    GT -> GT
    LT -> LT
    EQ -> ordList test xs ys
```

Met behulp van deze functie kun je nu de onderlinge ligging van twee lijsten bepalen, bijvoorbeeld:

```
? ordList ordInt [2,3] [2,3,4]
LT
```

Net als bij `eqList` kan deze functie partiëel geparametriseerd worden om hem vervolgens aan zichzelf mee te geven. Je kunt op die manier ook lijsten van lijsten van wat je maar wilt te ordenen.

Lidmaatschapstest

Als we een functie hebben om elementen te vergelijken, dan kunnen we een functie maken die bepaalt of een element ergens in een lijst voorkomt. In de prelude is er zo'n functie, genaamd `elemBy` met drie parameters: een vergelijkingstest-functie, een element, en een lijst. De definitie luidt:

```
elemBy :: (a -> a -> Bool) -> a -> [a] -> Bool
elemBy test x [] = False
elemBy test x (y:ys) | test x y = True
                    | otherwise = elemBy test x ys
```

In plaats van met expliciete recursie had deze functie ook gedefinieerd kunnen worden door handig gebruik te maken van `map` en `foldr`:

```
elemBy test e xs = or (map (test e) xs)
```

Nog leuker is om deze functie dan weer te herschrijven met gebruikmaking van de functie-samenstelop-operator `.`:

```
elemBy test e = or . map (test e)
```

3.1.5 Lijsten sorteren

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

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

Toch is het, zeker met hulp van de standaardfuncties, niet moeilijk om een sorteer-functie te schrijven. Er zijn verschillende mogelijkheden om het sorteer-probleem aan te pakken. Deftiger gezegd: er zijn verschillende *algoritmen* mogelijk. Twee algoritmen zullen hier worden besproken. In beide algoritmen is het noodzakelijk dat de elementen van de lijst geordend kunnen worden. Het is dus mogelijk om een lijst integers of een lijst van lijsten van integers te sorteren, maar niet een lijst van functies. Dit komt tot uiting in het feit dat de functie als eerste parameter een functie verwacht, die aangeeft hoe de elementen van de lijst geordend worden:

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

Sorteren door invoegen

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

```

insert          :: (a->a->Ordering) -> a -> [a] -> [a]
insert cmp e [] = [e]
insert cmp e (x:xs) = case e `cmp` x of
                        LT -> e : x : xs
                        _  -> x : insert cmp e xs

```

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

```

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

```

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

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

```

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

```

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

```

isort cmp = foldr (insert cmp) []

```

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

Sorteren door samenvoegen

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

```

merge          :: (a->a->Ordering) -> [a] -> [a] -> [a]
merge cmp []   ys   = ys
merge cmp xs   []   = xs
merge cmp (x:xs) (y:ys) = case x `cmp` y of
                            LT -> x : merge cmp xs (y:ys)
                            _  -> y : merge cmp (x:xs) ys

```

Als één van beide lijsten leeg is, dan is de andere lijst het resultaat. Als beide lijsten niet-leeg zijn, dan komt de kleinste van de twee kop-elementen op kop van het resultaat, en worden de overblijvende elementen samengevoegd door een recursieve aanroep van `merge`.

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

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

```

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

```

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

3.2 Speciale lijsten

3.2.1 Strings

blz. 27

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

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

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

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

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

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

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

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

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

3.2.2 Characters

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

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

Er zijn 256 mogelijke waarden voor het type `Char`:

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

Er is één leesteken dat in een string problemen geeft: het aanhalingsteken. Bij een aanhalingsteken in een string zou de string immers afgelopen zijn. Als er toch een aanhalingsteken in een string

nodig is, moet daar het symbool `\` (een omgekeerde deelstreep of *backslash*) vóór gezet worden. Bijvoorbeeld:

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

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

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

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

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

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

```
? "EEN\nTWEEN\nDRIE"
EEN
TWEEN
DRIE
```

Alle characters zijn genummerd volgens een door de Internationale Standaard Organisatie (ISO) bepaalde codering¹. Er zijn twee (ingebouwde) standaardfuncties, die de code van een character bepalen, respectievelijk het character met een bepaalde code opleveren:

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

Bijvoorbeeld:

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

Een overzicht van alle characters met hun ISO/ASCII-codenummers staat in appendix B. De characters zijn geordend volgens deze codering. Het type `Char` maakt daarmee deel uit van de klasse `Ord`. De ordening komt, wat de letters betreft, overeen met de alfabetische ordening, met dien verstande dat alle hoofdletters vóór de kleine letters komen. Deze ordening werkt ook door in strings; strings zijn immers lijsten, en die zijn lexicografisch geordend gebaseerd op de ordening van hun elementen:

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

3.2.3 Functies op characters en strings

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.4 Oneindige lijsten

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

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

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

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

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

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

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

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

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

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

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

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

3.2.5 Lazy evaluatie

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

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

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

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

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

Het lazy gedrag van de operator `==` wordt veroorzaakt door zijn definitie. De recursieve regel uit de definitie in paragraaf 3.1.4 luidt:

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

blz. 27

blz. 40

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

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

blz. 10 Als de linker parameter de waarde `False` heeft, is de waarde van de rechter parameter niet nodig om het resultaat te berekenen. (Dit is de echte definitie van `&&`. De definitie in paragraaf 1.4.3 is ook goed, maar vertoont niet het gewenste lazy gedrag).

Functies die alle elementen van een lijst nodig hebben, mogen niet op oneindige lijsten worden toegepast. Voorbeelden van zulk soort functies zijn `sum` en `length`. Bij de aanroep `sum (vanaf 1)` of `length (vanaf 1)` helpt zelfs lazy evaluatie niet meer om in eindige tijd het eindresultaat te berekenen. De computer zal in zo'n geval in trance gaan, en nooit met een eindantwoord komen (tenzij het resultaat van de berekening nergens gebruikt wordt, want dan wordt de berekening natuurlijk niet uitgevoerd...).

3.2.6 Functies op oneindige lijsten

In de prelude worden een aantal functies gedefinieerd die oneindige lijsten opleveren.

blz. 46 De functie `vanaf` uit paragraaf 3.2.4 heet in werkelijkheid `enumFrom`. De functie wordt meestal niet als zodanig gebruikt, omdat in plaats van `enumFrom n` ook `[n..]` geschreven mag worden.

blz. 36 (Vergelijk de notatie `[n..m]` voor `enumFromTo n m`, die in paragraaf 3.1.1 werd besproken).

Een oneindige lijst waarin steeds één element herhaald wordt, kan worden gemaakt met de functie `repeat`:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

De aanroep `repeat 't'` levert de oneindige lijst `"tttttttt... op`.

Een oneindige lijst die door `repeat` wordt gegenereerd, kan weer goed gebruikt worden als tussenresultaat door een functie die wel een eindig resultaat heeft. De functie `replicate` bijvoorbeeld maakt een eindig aantal kopieën van een element:

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

Dankzij lazy evaluatie kan `replicate` gebruik maken van het oneindige resultaat van `repeat`. De functies `repeat` en `replicate` worden in de prelude gedefinieerd.

De meest flexibele functie is ook nu weer een hogere-orde functie, dat wil zeggen een functie met een functie als parameter. De functie `iterate` krijgt een functie en een startelement als parameter. Het resultaat is een oneindige lijst, waarin elk volgend element verkregen wordt door de functie op het vorige element toe te passen. Bijvoorbeeld:

```
iterate ((+)1) 3      is [3, 4, 5, 6, 7, 8, ...
iterate ((*)2) 1      is [1, 2, 4, 8, 16, 32, ...
iterate (\x->x/10) 5678 is [5678, 567, 56, 5, 0, 0, ...
```

De definitie van `iterate`, die in de prelude staat, is als volgt:

```
iterate :: (a->a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

blz. 24 Deze functie lijkt een beetje op de functie `until`, die in paragraaf 2.3.2 werd gedefinieerd. Ook `until` krijgt immers een functie en een startelement als parameter, en past de functie herhaald toe op het startelement. Het verschil is, dat `until` stopt als de waarde aan een bepaalde voorwaarde (die ook als parameter wordt meegegeven) voldoet. Bovendien levert `until` alleen de eindwaarde op (die dus aan het meegegeven stopcriterium voldoet), terwijl `iterate` alle tussenresultaten in een lijst stopt. Hij moet wel, want bij oneindige lijsten is er geen laatste element...

Hier volgen twee voorbeelden waarin `iterate` gebruikt wordt om een praktisch probleem op te lossen: de weergave van een getal als string, en het genereren van de lijst van alle priemgetallen.

Weergave van een getal als string

De functie `intString` maakt van een getal een string waarin de cijfers van dat getal zit-

ten. Bijvoorbeeld: `intString 5678` is de string `"5678"`. Dankzij deze functie is het mogelijk om het resultaat van een berekening te combineren met een string, bijvoorbeeld zoals in `intString (3*17)+" gulden"`.

De functie `intString` kan worden samengesteld door na elkaar een aantal functies uit te voeren. Eerst moet het getal met behulp van `iterate` herhaald door 10 gedeeld worden (zoals in het derde voorbeeld van `iterate` hierboven). De oneindige staart nullen is oninteressant, en kan worden afgekapt met `takeWhile`. De gewenste cijfers zijn dan steeds het laatste cijfer van de getallen in de lijst; het laatste cijfer van een getal is de rest bij deling door 10. De cijfers staan nu nog in de verkeerde volgorde, maar dat kan worden opgelost met de functie `reverse`. Tenslotte moeten de cijfers (van type `Int`) nog worden omgezet in het overeenkomstige cijferteken (van type `Char`).

Een schema aan de hand van een voorbeeld maakt dit wat duidelijker:

```

5678
  ↓ iterate (\x->x/10)
[5678, 567, 56, 5, 0, 0, ...]
  ↓ takeWhile (/=0)
[5678, 567, 56, 5]
  ↓ map (\x->x`rem`10)
[8, 7, 6, 5]
  ↓ reverse
[5, 6, 7, 8]
  ↓ map digitChar
['5', '6', '7', '8']

```

De functie `intString` kan simpelweg geschreven worden als samenstelling van deze vijf functies. Let er op dat de functies in omgekeerde volgorde opgeschreven moeten worden, omdat de functie-samenstellings operator `(.)` de betekenis 'na' heeft:

```

intString :: Int -> [Char]
intString = map digitChar
           . reverse
           . map (\x->x`rem`10)
           . takeWhile (/=0)
           . iterate (\x->x/10)

```

Functioneel programmeren is programmeren met functies!

De lijst van alle priemgetallen

In paragraaf 2.4.1 werd een functie `priem` gedefinieerd, die bepaalt of een getal een priemgetal is. De (oneindige) lijst van alle priemgetallen kan daarmee worden berekend door

blz. 27

```

filter priem [2..]

```

De functie `priem` gaat op zoek naar delers van een getal. Als zo'n deler groot is, duurt het dus vrij lang voordat de functie tot de conclusie komt dat een getal geen priemgetal is.

Door handig gebruik te maken van `iterate` is echter een veel snellere methode mogelijk. Deze methode begint ook met de oneindige lijst `[2..]`:

```

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...]

```

Het eerste getal, 2, kan in de lijst van priemgetallen worden gestopt. Nu worden 2 en alle veelvouden daarvan uit de lijst weggestreept. Er blijft dan over:

```

[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, ...]

```

Het eerste getal, 3, is een priemgetal. Dit getal en zijn veelvouden worden uit de lijst weggestreept:

```

[5, 7, 11, 13, 17, 19, 23, 25, 29, 31, ...]

```

Hetzelfde proces wordt weer uitgevoerd, maar nu met 5:

```

[7, 11, 13, 17, 19, 23, 29, 31, 37, 41, ...]

```

En zo kun je doorgaan. De functie ‘streep veelvoud van het eerste element weg’ wordt steeds uitgevoerd op het vorige resultaat. Dit is dus een toepassing van `iterate`, met `[2..]` als startwaarde:

```
iterate streepweg [2..]
where streepweg (x:xs) = filter (not.veelvoud x) xs
      veelvoud x y = deelbaar y x
```

(Het getal `y` is een veelvoud van `x` als `y` deelbaar is door `x`). Doordat de beginwaarde een oneindige lijst is, is het resultaat hiervan een *oneindige lijst van oneindige lijsten*. Die super-lijst is als volgt opgebouwd:

```
[ [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
  , [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, ...
  , [5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, ...
  , [7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, ...
  , [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 51, 53, ...
  , ...
```

Dit ding kun je nooit in zijn geheel te zien krijgen; als je hem probeert te evalueren krijg je alleen het beginstuk van de eerste rij te zien. Maar de complete lijst hoeft niet zichtbaar gemaakt te worden: de gewenste priemgetallen zijn steeds het eerste element van de lijst. De priemgetallen worden dus bepaald door van elke lijst de `head` te nemen:

```
priemgetallen :: [Int]
priemgetallen = map head (iterate streepweg [2..])
      where streepweg (x:xs) = filter (not.veelvoud x) xs
```

Door de lazy evaluatie wordt van elke lijst in de super-lijst precies het gedeelte uitgerekend dat nodig is voor het gewenste deel van het antwoord. Wil je het volgende priemgetal weten, dan wordt elke lijst het noodzakelijke stukje verder uitgerekend.

Het is vaak (zo ook in dit voorbeeld) moeilijk om je precies voor te stellen wat er op welk moment wordt uitgerekend. Maar dat hoeft ook niet: tijdens het programmeren kun je net doen alsof oneindige lijsten echt bestaan; de uitrekensvolgorde wordt door de lazy evaluatie automatisch geoptimaliseerd.

3.2.7 Lijst-comprehensies

In de verzamelingenleer is een handige notatie in gebruik om verzamelingen te definiëren:

$$V = \{ x^2 \mid x \in N, x \text{ even} \}$$

Naar analogie van deze notatie, de zogenaamde verzameling-comprehensie, is in Helium een vergelijkbare notatie beschikbaar om lijsten te construeren. Deze notatie heet dan ook een *lijst-comprehensie*. Een eenvoudig voorbeeld van deze notatie is de volgende expressie:

```
[ x*x | x <- [1..10] ]
```

Deze expressie kan worden uitgesproken als ‘`x` kwadraat voor `x` uit 1 tot 10’. In een lijst-comprehensie staat voor de verticale streep een expressie, waarin een variabele mag voorkomen. Deze variabele (`x` in het voorbeeld) wordt gebonden in het gedeelte achter de verticale streep. De notatie ‘`x <- xs`’ heeft de betekenis: ‘`x` doorloopt alle waarden van de lijst `xs`’. Voor elk van deze waarden wordt de waarde van de expressie voor de verticale streep uitgerekend.

Bovengenoemd voorbeeld heeft dus dezelfde waarde als de expressie

```
map kwadraat [1..10]
```

waarbij de functie `kwadraat` is gedefinieerd als

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

Het voordeel van de comprehensie-notatie is dat de functie die steeds wordt uitgerekend (`kwadraat` in het voorbeeld) niet eerst een naam hoeft te krijgen.

De lijst-comprehensie notatie heeft nog meer mogelijkheden. Achter de verticale streep mag namelijk meer dan één lopende variabele worden gebruikt. De expressie voor de verticale streep wordt dan voor alle mogelijke combinaties uitgerekend. Bijvoorbeeld:

```
? [ (x,y) | x<-[1..2], y<-[4..6] ]
[ (1,4), (1,5), (1,6), (2,4), (2,5), (2,6) ]
```

De laatstgenoemde variabele loopt het snelst: voor elke waarde van `x` doorloopt `y` de lijst `[4..6]`. Behalve definities van lopende variabelen mogen achter de verticale streep uitdrukkingen met de waarde `True` of `False` worden opgenomen. De betekenis daarvan wordt gedemonstreerd door het volgende voorbeeld:

```
? [ (x,y) | x<-[1..5], even x, y<-[1..x] ]
[ (2,1), (2,2), (4,1), (4,2), (4,3), (4,4) ]
```

In de resultaat-lijst worden dus alleen die `x` verwerkt, waarvoor `even x` de waarde `True` heeft.

Door elk pijltje (`<-`) wordt een variabele gedefinieerd, die in de verdere expressies en in de expressie links van de verticale streep gebruikt mag worden. Zo mag de variabele `x` behalve in `(x,y)` gebruikt worden in `even x` en in `[1..x]`. De variabele `y` mag echter alleen maar gebruikt worden in `(x,y)`. Het pijltje is een speciaal voor dit doel gereserveerd symbool, en is dus geen operator!

Strikt genomen is de lijst-comprehensie notatie overbodig. Hetzelfde effect kan bereikt worden door combinaties van `map`, `filter` en `concat`. De comprehensie-notatie is, zeker in ingewikkelde gevallen, echter veel gemakkelijker te begrijpen. Bovenstaand voorbeeld zou anders geschreven moeten worden als

```
concat (map f (filter even [1..5]))
where f x = map g [1..x]
       g y = (x,y)
```

hetgeen veel minder inzichtelijk is.

Een lijst-comprehensie wordt door de interpreter direct vertaald naar een overeenkomstige expressie met `map`, `filter` en `concat`. Net als de notatie voor intervallen is de comprehensie-notatie dus puur bedoeld voor het gemak van de programmeur.

3.3 Tupels

3.3.1 Gebruik van tupels

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

Voor dit soort gevallen is er, naast lijstvorming, nog een andere manier om samengestelde types te maken: *tupelvorming*. Een *tupel* bestaat uit een vast aantal waarden, die tot één geheel zijn gegroepeerd. De waarden mogen van verschillend type zijn (hoewel dat niet verplicht is).

Tupels worden genoteerd met ronde haakjes rond de elementen (waar bij lijsten vierkante haakjes worden gebruikt). Voorbeelden van tupels zijn:

<code>(1, 'a')</code>	een tupel met als elementen de integer 1 en het character 'a';
<code>("aap", True, 2)</code>	een tupel met drie elementen: de string "aap", de boolean <code>True</code> en het getal 2;
<code>([1,2], sqrt)</code>	een tupel met twee elementen: de lijst integers <code>[1,2]</code> , en de float-naar-float functie <code>sqrt</code> ;
<code>(1, (2,3))</code>	een tupel met twee elementen: het getal 1, en het tupel van de getallen 2 en 3.

Voor elke combinatie van types vormt het tupel ervan een apart type. Daarbij is ook de volgorde van belang. Het type van tupels wordt geschreven door de types van de elementen op te sommen tussen ronde haakjes. De vier hierboven genoemde expressies kunnen dus als volgt getypeerd worden:

```
(1, 'a')      :: (Int, Char)
("aap", True, 2) :: ([Char], Bool, Int)
```

```
([1,2], sqrt)    :: ([Int], Float->Float)
(1, (2,3))       :: (Int, (Int,Int))
```

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

In de prelude zijn een paar functies gedefinieerd die op 2-tupels of 3-tupels werken. Deze zijn er meteen een voorbeeld van hoe functies op tupels gedefinieerd kunnen worden: door patroon-analyse.

```
fst          :: (a,b) -> a
fst (x,y)    = x
snd          :: (a,b) -> b
snd (x,y)    = y
fst3         :: (a,b,c) -> a
fst3 (x,y,z) = x
snd3         :: (a,b,c) -> b
snd3 (x,y,z) = y
thd3        :: (a,b,c) -> c
thd3 (x,y,z) = z
```

Deze functies zijn polymorf, maar het is natuurlijk ook mogelijk om functies te schrijven die maar op één specifiek tupel-type werken:

```
f          :: (Int,Char) -> [Char]
f (n,c)    = intString n ++ [c]
```

Als twee waarden van hetzelfde type gegroepeerd moeten worden kan daarvoor een lijst gebruikt worden. In sommige gevallen is een tupel geschikter. Een punt in het platte vlak wordt bijvoorbeeld beschreven door twee `Float` getallen. Zo'n punt kan worden gerepresenteerd door een lijst, of door een 2-tupel. In beide gevallen is het mogelijk om functies te definiëren die op punten werken, bijvoorbeeld 'afstand tot de oorsprong'. De functie `afstandL` is de lijst-versie, `afstandT` de tupel-versie hiervan:

```
afstandL    :: [Float] -> Float
afstandL [x,y] = sqrt (x*x+y*y)
afstandT    :: (Float,Float) -> Float
afstandT (x,y) = sqrt (x*x+y*y)
```

Zolang de functie correct wordt aangeroepen is er geen verschil. Maar het zou kunnen gebeuren dat de functie elders in het programma, door een tikfout of een denkfout, met drie coördinaten wordt aangeroepen. Bij gebruik van `afstandT` wordt daarvoor tijdens de analyse van het programma voor gewaarschuwd: een tupel met drie getallen is een ander type dan een tupel met twee getallen. In het geval van `afstandL` is het programma echter goed getypeerd. Pas als de functie inderdaad gebruikt wordt blijkt dat `afstandL` voor lijsten met drie elementen ongedefinieerd is. Het gebruik van tupels in plaats van lijsten helpt dus in dit geval om fouten zo vroeg mogelijk op te sporen.

Nog een plaats waar tupels van pas komen zijn functies die meer dan één resultaat hebben. Functies met meerdere parameters zijn mogelijk dankzij het Currying-mechanisme; functies die meerdere resultaten hebben, zijn echter alleen mogelijk door die resultaten te 'verpakken' in een tupel. Het tupel in z'n geheel is dan immers één resultaat.

Een voorbeeld van een functie die eigenlijk twee resultaten heeft, is de functie `splitAt` die in de prelude wordt gedefinieerd. Deze functie levert de resultaten van `take` en `drop` in één keer op. De functie zou dus zo gedefinieerd kunnen worden:

```
splitAt     :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

Het werk van beide functies kan echter in één keer worden gedaan, vandaar dat `splitAt` uit efficiëntie-overwegingen als volgt is gedefinieerd:

```
splitAt     :: Int -> [a] -> ([a], [a])
splitAt 0 xs = ([], xs)
splitAt n [] = ([], [])
splitAt n (x:xs) = (x:ys, zs)
                where (ys,zs) = splitAt (n-1) xs
```

De aanroep `splitAt 3 "haskell"` geeft bijvoorbeeld het 2-tupel `("has", "kell")` als resultaat. In de definitie is (bij de recursieve aanroep) te zien hoe zo'n resultaat-tupel gebruikt kan worden: door het te onderwerpen aan een patroon-analyse `((ys, zs)` in het voorbeeld).

3.3.2 Type-definities

Bij veelvuldig gebruik van lijsten en tupels worden type-declaraties vaak nogal ingewikkeld. Bijvoorbeeld bij het schrijven van functies op punten, zoals de functie `afstand` hierboven. De eenvoudigste functies zijn nog wel te overzien:

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

Maar lastiger wordt het bij lijsten van punten, en vooral bij hogere-orde functies:

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

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

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

Na deze type-definitie zijn de type-declaraties eenvoudiger te schrijven:

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

Nog beter is het om ook voor 'veelhoek' een type-definitie te maken:

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

Een paar dingen om in de gaten te houden bij type-definities:

- het woord `type` is een, speciaal voor dit doel, gereserveerd woord;
- de naam van het nieuw gedefinieerde type moet met een hoofdletter beginnen (het is een constante, niet een variabele);
- een *type-declaratie* specificeert het type van een functie; een *type-definitie* definieert een nieuwe naam voor een type.

De nieuw gedefinieerde naam wordt door de interpreter puur beschouwd als afkorting. Bij het typen van een expressie krijg je gewoon weer `(Float,Float)` te zien in plaats van `Punt`. Als er twee verschillende namen aan één type gegeven worden, bijvoorbeeld:

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

dan mogen die namen door elkaar gebruikt worden. Een `Punt` is hetzelfde als een `Complex` is hetzelfde als een `(Float,Float)`. In paragraaf 3.4.3 wordt een methode beschreven hoe `Punt` als een echt *nieuw* type gedefinieerd kan worden.

blz. 62

3.3.3 Rationale getallen

Een toepassing waarbij tupels goed gebruikt kunnen worden is een implementatie van de *rationale getallen*. De rationale getallen vormen de wiskundige verzameling \mathbf{Q} , getallen die als *breuk* te schrijven zijn. Voor het rekenen met rationale getallen kunnen geen `Float` getallen gebruikt worden: het is de bedoeling dat er *exact* gerekend wordt, en dat de uitkomst van $\frac{1}{2} + \frac{1}{3}$ de breuk $\frac{5}{6}$ oplevert, en niet de `Float` 0.833333.

Rationale getallen, oftewel breuken, kunnen worden gerepresenteerd door een teller en een noemer, die allebei gehele getallen zijn. De volgende type-definitie ligt daarom voor de hand:

```
type Ratio = (Int,Int)
```

Een aantal veelgebruikte breuken kunnen een aparte naam krijgen:

```
qNul = (0, 1)
qEen = (1, 1)
```

```

qTwee = (2, 1)
qHalf = (1, 2)
qDerde = (1, 3)
qKwart = (1, 4)

```

Het is de bedoeling om functies te schrijven die de belangrijkste rekenkundige operaties op rationale getallen uitvoeren:

```

qMaal :: Ratio -> Ratio -> Ratio
qDeel :: Ratio -> Ratio -> Ratio
qPlus :: Ratio -> Ratio -> Ratio
qMin  :: Ratio -> Ratio -> Ratio

```

Een probleem is, dat één waarde door verschillende breuken weergegeven kan worden. Een ‘half’ bijvoorbeeld, wordt gerepresenteerd door het tupel (1,2), maar ook door (2,4) en (17,34). Het resultaat van twee maal een kwart (twee-vierde) zou daardoor wel eens kunnen ‘verschillen’ van een half (een-tweede). Om dit probleem op te lossen, is er een functie `eenvoud` nodig, die een breuk kan vereenvoudigen. Door na elke operatie op breuken deze functie toe te passen, wordt een breuk altijd op dezelfde manier gerepresenteerd. Het resultaat van twee maal een kwart kan dan veilig vergeleken worden met een half: het resultaat is `True`.

De functie `eenvoud` deelt de teller en de noemer van een breuk door hun *grootste gemene deler*. De grootste gemene deler (`ggd`) van twee getallen is het grootste getal waardoor beide deelbaar zijn. Daarnaast zorgt `eenvoud` ervoor, dat een eventueel min-teken altijd in de teller van de breuk staat. De definitie is als volgt:

```

eenvoud (t,n) = ( (signum n * t)/d, abs n/d )
               where d = ggd t n

```

Een eenvoudige definitie van `ggd x y` (die alleen werkt als `x` en `y` positief zijn) bepaalt de grootste deler van `x` waardoor `y` deelbaar is, gebruik makend van de functies `delers` en `deelbaar` uit paragraaf 2.4.1:

blz. 27

```

ggd x y = last (filter (deelbaar y') (delers x'))
         where x' = abs x
               y' = abs y

```

(In de prelude wordt een functie `gcd` (*greatest common divisor*) gedefinieerd, die sneller werkt:

```

gcd x y = gcd' (abs x) (abs y)
         where gcd' x 0 = x
               gcd' x y = gcd' y (x `rem` y)

```

Deze methode is erop gebaseerd dat als `x` en `y` deelbaar zijn door `d`, dat dan ook `x`rem`y` ($=x-(x/y)*y$) deelbaar is door `d`).

Met behulp van de functie `eenvoud` kunnen nu de rekenkundige functies gedefinieerd worden. Om twee breuken te vermenigvuldigen, moeten de teller en de noemer vermenigvuldigd worden ($\frac{2}{3} * \frac{5}{4} = \frac{10}{12}$). Daarna kan het resultaat vereenvoudigd worden (tot $\frac{5}{6}$):

```

qMaal (x,y) (p,q) = eenvoud (x*p, y*q)

```

Delen door een getal is vermenigvuldigen met het omgekeerde, dus:

```

qDeel (x,y) (p,q) = eenvoud (x*q, y*p)

```

Voor het optellen van twee breuken moeten ze eerst gelijknamig worden gemaakt ($\frac{1}{4} + \frac{3}{10} = \frac{10}{40} + \frac{12}{40} = \frac{22}{40}$). Als gelijke noemer kan het product van de noemers dienen. De tellers moeten dan met de noemer van de andere breuk worden vermenigvuldigd, waarna ze kunnen worden opgeteld. Het resultaat moet tenslotte vereenvoudigd worden (tot $\frac{11}{20}$).

```

qPlus (x,y) (p,q) = eenvoud (x*q+y*p, y*q)
qMin  (x,y) (p,q) = eenvoud (x*q-y*p, y*q)

```

Het resultaat van berekeningen met rationale getallen wordt als tupel op het scherm gezet. Als dat niet mooi genoeg is, kan er eventueel een functie `ratioString` worden gedefinieerd:

```

ratioString :: Ratio -> String
ratioString (x,y)
  | y'==1    = intString x'
  | otherwise = intString x' ++ "/" ++ intString y'
               where (x',y') = eenvoud (x,y)

```

3.3.4 Tupels en lijsten

Tupels komen vaak voor als elementen van een lijst. Veel gebruikt wordt bijvoorbeeld een lijst van twee-tupels, die als opzoeklijst (woordenboek, telefoonboek enz.) kan dienen. De opzoek-functie is heel eenvoudig te definiëren met behulp van patronen; voor de lijst wordt een patroon gebruikt voor ‘niet-lege lijst waarvan het eerste element een 2-tupel is (en de andere elementen dus ook)’.

```
zoekOp  :: (a->a->Bool) -> [(a,b)] -> a -> b
zoekOp eq ((x,y):ts) z
    | eq x z      = y
    | otherwise  = zoekOp eq ts z
```

De functie is polymorf, dus werkt op lijsten 2-tupels van willekeurig type. Wel moeten de op te zoeken elementen vergeleken kunnen worden, dus het type `a` moet in de klasse `Eq` zitten.

Het op te zoeken element (van type `a`) is opzettelijk als tweede parameter gedefinieerd, zodat de functie `zoekOp` eenvoudig partieel geparametriseerd kan worden met een specifieke opzoeklijst, bijvoorbeeld:

```
telefoonNr = zoekOp eqString telefoonboek
vertaling  = zoekOp eqString woordenboek
```

waarbij `telefoonboek` en `woordenboek` apart als constante gedefinieerd kunnen worden.

Een andere functie waarin lijsten 2-tupels een rol spelen is de functie `zip`. Deze functie wordt in de prelude gedefinieerd. De functie `zip` heeft twee lijsten als parameter, die in het resultaat per element aan elkaar gekoppeld worden. Bijvoorbeeld: `zip [1,2,3] "abc"` geeft de lijst `[(1,'a'),(2,'b'),(3,'c')]`. Als de parameter-lijsten niet even lang zijn, is de lengte van de kortste van de twee bepalend. De definitie is zeer rechtstreeks:

```
zip      :: [a] -> [b] -> [(a,b)]
zip []   ys      = []
zip xs   []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

De functie is polymorf, en kan dus op lijsten met elementen van willekeurige types worden toegepast. De naam *zip* betekent letterlijk ‘rits’: de twee lijsten worden als het ware aan elkaar geritst.

Een hogere-orde variant van `zip` is de functie `zipWith`. Deze functie krijgt behalve twee lijsten ook een functie als parameter, die aangeeft hoe de overeenkomstige elementen aan elkaar gekoppeld moeten worden:

```
zipWith      :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f [] ys      = []
zipWith f xs []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Deze functie past een functie (met twee parameters) toe op alle elementen van twee lijsten. Behalve op `zip` lijkt `zipWith` ook sterk op `map`, die immers een functie (met één parameter) toepast op alle elementen van één lijst.

Gegeven de functie `zipWith` kan `zip` gedefinieerd worden als partiële parametrisatie daarvan:

```
zip = zipWith maak2tupel
    where maak2tupel x y = (x,y)
```

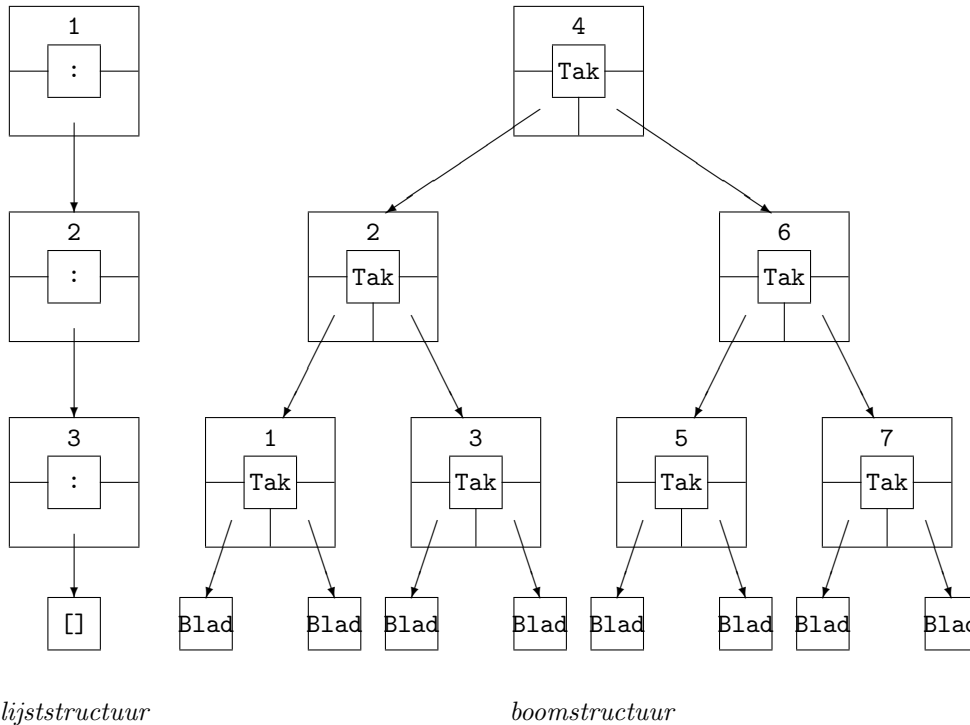
3.3.5 Tupels en Currying

Met behulp van tupels is het mogelijk om functies met meer dan één parameter te schrijven, zonder het Curry-mechanisme te gebruiken. Een functie kan namelijk een tupel als (enige) parameter krijgen, waarmee toch twee waarden naar binnen gesmokkeld worden:

```
plus (x,y) = x+y
```

Deze functiedefinitie ziet er heel klassiek uit. De meeste mensen zouden zeggen dat `plus` een functie is met twee parameters, en dat parameters ‘natuurlijk’ tussen haakjes staan. Maar wij weten inmiddels beter: deze functie heeft één parameter, en wel een tupel; de definitie vindt plaats met behulp van een patroon voor een tupel.

De Curry-methode is overigens vaak te prefereren boven de tupel-methode. Gecurryde functies zijn



immers partieel te parametriseren, en functies met een tupel-parameter niet. Alle standaardfuncties met meer dan één parameter werken dan ook volgens de Curry-methode.

In de prelude wordt een functie-transformatie (functie met functie als parameter en andere functie als resultaat) gedefinieerd, die van een gecurryde functie een functie met tupel-parameter maakt. Deze functie heet `uncurry`:

```
uncurry :: (a->b->c) -> ((a,b)->c)
uncurry f (a,b) = f a b
```

Andersom is er een functie `curry` die van een functie met tupel-parameter een gecurryde functie maakt. Dus `curry plus`, met `plus` zoals hierboven, kan wèl partieel geparametriseerd worden.

3.4 Bomen

3.4.1 Data-definities

Lijsten en tupels zijn twee ‘ingebouwde’ manieren om gegevens te structureren. Mochten deze twee manieren niet geschikt zijn om bepaalde gegevens te representeren, dan is het mogelijk om zelf een nieuw *datatype* te definiëren.

Een datatype is een type dat gekenmerkt wordt door de manier waarop waarden van dat type kunnen worden opgebouwd. Het begrip ‘lijst’ is een datatype. Lijst-waarden kunnen op twee manieren worden opgebouwd:

- als de lege lijst;
- uit een element en een (kleinere) lijst door de operator `:` toe te passen.

Deze twee manieren komen terug in de patronen van definities van functies op lijsten, bijvoorbeeld:

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

Door de functie te definiëren voor de patronen ‘lege lijst’ en ‘operator `:` toegepast op een element en een lijst’ is de functie geheel gedefinieerd.

Een lijst is een lineaire structuur; doordat steeds een element wordt bijgeschakeld, ontstaat een steeds langere keten. Soms is zo’n lineaire structuur niet gewenst, maar voldoet een *boomstructuur* beter. Er zijn verschillende boomstructuren mogelijk. In de volgende figuur wordt een lijst verge-

leken met een boom die zich steeds in tweeën splitst. In kleine vierkantjes is aangegeven hoe de structuur is opgebouwd. In het geval van de lijst is dat door middel van de operator `:` met twee parameters (die eromheen getekend zijn), of door middel van `[]` zonder parameters. De boom is niet opgebouwd met operatoren maar met functies: `Tak` (met drie parameters) of `Blad` (zonder parameters).

Functies waarmee een datastructuur wordt opgebouwd heten *constructor-functies*. De constructor-functies van de boom zijn `Tak` en `Blad`. Namen van constructor-functies beginnen met een hoofdletter, om ze te onderscheiden van ‘gewone’ functies. Als constructor-functies als operator geschreven worden moeten ze met een dubbele punt beginnen. De constructor-operator (`:`) van lijsten is daar een voorbeeld van, en de constructor `[]` is de enige uitzondering.

Welke constructor-functies voor een nieuw type gebruikt kunnen worden, wordt gespecificeerd met een *data-definitie*. Daarin staan ook de types van de parameters van de constructor-functies, en of het nieuwe type polymorf is. De data-definitie voor bomen zoals hierboven besproken luidt bijvoorbeeld als volgt:

```
data Boom a = Tak a (Boom a) (Boom a)
            | Blad
```

Je kunt deze definitie als volgt uitspreken. ‘Een boom met elementen van type `a` (kortweg boom-over-`a`) kan op twee manieren worden opgebouwd: (1) door de functie `Tak` toe te passen op drie parameters (één van type `a` en twee van type boom-over-`a`), of (2) door de constante `Blad` te gebruiken.’

Bomen kunnen worden opgebouwd door de constructor-functies in een expressie te gebruiken. De boom die in de figuur getekend is, wordt bijvoorbeeld weergegeven door de volgende expressie:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad)
        (Tak 3 Blad Blad)
      )
      (Tak 6 (Tak 5 Blad Blad)
        (Tak 7 Blad Blad)
      )
```

Het hoeft niet zo mooi over de regels gespreid te worden; ook toegestaan is:

```
Tak 4(Tak 2(Tak 1 Blad Blad)(Tak 3 Blad Blad))
      (Tak 6(Tak 5 Blad Blad)(Tak 7 Blad Blad))
```

De eerstgenoemde constructie is natuurlijk wel duidelijker. Houd ook de layout-regel uit paragraaf 1.4.5 in de gaten.

blz. 12

Functies op een boom kunnen gedefinieerd worden door voor elke constructor-functie een patroon te maken. De volgende functie bepaalt bijvoorbeeld het aantal `Tak`-constructies in een boom:

```
omvang      :: Boom a -> Int
omvang Blad = 0
omvang (Tak x p q) = 1 + omvang p + omvang q
```

Vergelijk deze functie met de functie `length` op lijsten.

Er zijn nog vele andere mogelijke bomen denkbaar. Een paar voorbeelden:

- Bomen waarbij de informatie in de eindpunten wordt opgeslagen (in plaats van op de splitspunten zoals bij `Boom`):

```
data Boom2 a = Tak2 (Boom2 a) (Boom2 a)
              | Blad2 a
```

- Bomen waarbij de informatie van type `a` in de splitspunten is opgeslagen, en informatie van type `b` in de eindpunten:

```
data Boom3 a b = Tak3 a (Boom3 a b) (Boom3 a b)
                | Blad3 b
```

- Bomen die zich op elk splitspunt in drieën splitsen in plaats van in tweeën:

```
data Boom4 a = Tak4 a (Boom4 a) (Boom4 a) (Boom4 a)
              | Blad4
```

- Bomen waarin het aantal uitgaande takken in een splitspunt variabel is:

```
data Boom5 a = Tak5 a [Boom5 a]
```

In deze boom is geen aparte constructor voor ‘eindpunt’ nodig, omdat daarvoor een splitspunt met nul uitgaande takken gebruikt kan worden.

- Bomen waarin elk splitspunt slechts één uitgaande tak heeft:

```
data Boom6 a = Tak6 a (Boom6 a)
             | Blad6
```

Een ‘boom’ volgens dit type is in feite een lijst: hij heeft een lineaire structuur.

- Bomen met verschillende soorten splitsingen:

```
data Boom7 a b = Tak7a Int a (Boom7 a b) (Boom7 a b)
               | Tak7b Char (Boom7 a b)
               | Blad7a b
               | Blad7b Int
```

3.4.2 Zoekbomen

Een goed voorbeeld van een situatie waarin beter bomen gebruikt kunnen worden dan lijsten, is het zoeken naar (de aanwezigheid van) een waarde in een grote collectie. Daarvoor kunnen *zoekbomen* gebruikt worden.

blz. 42

In paragraaf 3.1.4 werd de functie `elemBy` gedefinieerd, die `True` oplevert als een element in een lijst aanwezig is. Of deze functie nu met behulp van de standaardfuncties `map` en `or` wordt gedefinieerd

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

of direct met recursie

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

maakt voor de efficiëntie ervan niet zo veel uit. In beide gevallen worden de elementen van de lijst één voor één geïnspecteerd. Op het moment dat het element gevonden is, geeft de functie direct een resultaat (dankzij lazy evaluatie), maar als het element niet aanwezig is moet de functie alle elementen van de lijst bekijken om tot die conclusie te komen.

Iets handiger werkt het als de functie mag aannemen dat de te doorzoeken lijst gesorteerd is, dat wil zeggen dat de elementen op stijgende volgorde staan. Het zoekproces kan dan namelijk ook gestopt worden als het gevorderd is tot ‘voorbij’ de gezochte waarde. De prijs is wel dat de elementen nu niet alleen vergelijkbaar moeten zijn met een functie `a->a->Bool`, maar ook ordenbaar met een functie `a->a->Ordering`

```
elem'      :: (a->a->Ordering) -> a -> [a] -> Bool
elem' cmp e []      = False
elem' cmp e (x:xs) = case e 'cmp' x of
                      LT -> False
                      EQ -> True
                      GT -> elem' cmp e xs
```

Een veel grotere verbetering is het echter als de elementen niet in een lijst zijn opgeslagen, maar in een *zoekboom*. Een zoekboom is een soort ‘gesorteerde boom’. Het is een boom die is opgebouwd volgens de definitie van `Boom` uit de vorige paragraaf:

```
data Boom a = Tak a (Boom a) (Boom a)
            | Blad
```

Op elk splitspunt is een element opgeslagen, en twee (kleinere) bomen: een ‘linker’ deelboom en een ‘rechter’ deelboom (zie de figuur op blz. 56). In een zoekboom wordt nu bovendien geëist dat alle waarden in de linker deelboom *kleiner* zijn dan de waarde in het splitspunt, en alle waarden in de rechter deelboom *groter*. De waarden in de voorbeeldboom in de genoemde figuur zijn zo gekozen, dat de afgebeelde boom inderdaad een zoekboom is.

In een zoekboom is het zoeken naar een waarde heel eenvoudig. Als de gezochte waarde gelijk is aan de opgeslagen waarde in een splitspunt: mooi zo. Als de gezochte waarde kleiner is dan de opgeslagen waarde, dan moet doorgezocht worden in de linker deelboom (in de rechter deelboom zitten immers grotere waarden). Andersom, als de gezochte waarde groter is dan de opgeslagen

waarde, moet juist in de rechter deelboom worden doorgezocht. De functie `elemBoom` is dus als volgt:

```
elemBoom  :: (a->a->Ordering) -> a -> Boom a -> Bool
elemBoom cmp e Blad                = False
elemBoom cmp e (Tak x li re) = case e 'cmp' x of
                                EQ -> True
                                LT -> elemBoom cmp e li
                                GT -> elemBoom cmp e re
```

Als de boom evenwichtig is opgebouwd, zal het te doorzoeken aantal elementen bij elke stap ongeveer halveren. Het gezochte element of een `Blad`-eindpunt is dan snel gevonden: een verzameling van duizend elementen hoeft maar 10 keer gehalveerd te worden, en een verzameling van een miljoen elementen 20 keer. Vergelijk dat met de gemiddeld half miljoen stappen die de functie `elem` kost op een verzameling met een miljoen elementen.

In het algemeen kun je zeggen dat het geheel doorzoeken van een verzameling met n elementen met `elem` n stappen kost, maar met `elemBoom` slechts $2 \log n$ stappen.

Zoekbomen zijn goed te gebruiken als een grote hoeveelheid gegevens vaak moet worden doorzocht. Ook in bijvoorbeeld de functie `zoekOp` uit paragraaf 3.3.4 is met behulp van zoekbomen een dramatische snelheidswinst te boeken.

blz. 55

Opbouw van een zoekboom

De vorm van een zoekboom voor een bepaalde collectie gegevens kan ‘met de hand’ bepaald worden. De zoekboom kan vervolgens worden ingetikt als grote expressie met veel constructor-functies. Dat is echter een vervelend werk, dat eenvoudig kan worden geautomatiseerd.

Zoals de functie `insert` een element op de juiste plaats toevoegt aan een gesorteerde lijst (zie paragraaf 3.1.5), voegt de functie `insertBoom` een element toe aan een zoekboom. Het resultaat blijft een zoekboom, dat wil zeggen het element wordt op de juiste plaats ingevoegd:

blz. 42

```
insertBoom :: (a->a->Ordering) -> a -> Boom a -> Boom a
insertBoom cmp e Blad                = Tak e Blad Blad
insertBoom cmp e (Tak x li re) = case e 'cmp' x of
                                LT -> Tak x (insertBoom cmp e li) re
                                _  -> Tak x li (insertBoom cmp e re)
```

In het geval dat het element wordt toegevoegd aan `Blad` (een ‘lege’ boom), wordt een klein boompje gebouwd uit `e` en twee lege boompjes. Anders is de boom niet leeg, en bevat dus een opgeslagen waarde `x`. Deze waarde wordt gebruikt om te beslissen of `e` in de linker- of rechter deelboom ingevoegd moet worden.

Door de functie `insertBoom` herhaald te gebruiken, kunnen alle elementen van een lijst in een zoekboom worden gezet:

```
lijstNaarBoom :: (a->a->Ordering) -> [a] -> Boom a
lijstNaarBoom cmp = foldr (insertBoom cmp) Blad
```

Vergelijk deze functie met de functie `isort` in paragraaf 3.1.5.

blz. 42

Het gebruik van `lijstNaarBoom` heeft het nadeel dat de zoekboom die het resultaat is niet altijd evenwichtig is. Bij gegevens die in een willekeurige volgorde worden ingevoegd valt dat meestal wel mee. Als de lijst die tot boom wordt gemaakt echter al gesorteerd is, is het resultaat een ‘scheefgegroeide’ boom:

```
? lijstNaarBoom (==) [1..7]
Tak 7 (Tak 6 (Tak 5 (Tak 4 (Tak 3 (Tak 2 (Tak 1 Blad Blad)
Blad) Blad) Blad) Blad) Blad) Blad
```

Dit is weliswaar een zoekboom (elke waarde ligt tussen de waardes in de linker- en de rechter zoekboom), maar is helemaal scheefgetrokken zodat een bijna lineaire structuur is ontstaan. De gewenste logaritmische zoektijden zijn in deze boom dan ook niet mogelijk. Een betere (niet-scheve) boom met dezelfde waarden zou zijn:

```
Tak 4 (Tak 2 (Tak 1 Blad Blad)
      (Tak 3 Blad Blad))
      (Tak 6 (Tak 5 Blad Blad)
      (Tak 7 Blad Blad))
```

Sorteren met zoekbomen

De hierboven ontwikkelde functies kunnen worden gebruikt in een nieuw sorteer-algoritme. Daarbij is nog één extra functie nodig: een functie die de elementen van een zoekboom op volgorde in een lijst zet. Deze functie is als volgt:

```
labels      :: Boom a -> [a]
labels Blad = []
labels (Tak x li re) = labels li ++ [x] ++ labels re
```

In tegenstelling tot `insertBoom` doet deze functie een recursieve aanroep op de linker deelboom én de rechter deelboom. Op deze manier wordt elk element in de complete boom bekeken. Doordat de waarde `x` er op de juiste plaats tussen wordt geplakt, is het resultaat een gesorteerde lijst (mits de parameter een zoekboom is).

Een willekeurige lijst kan nu gesorteerd worden door er een zoekboom van te maken met `lijstNaarBoom`, en de elementen vervolgens op volgorde op te sommen met `labels`:

```
sorteer :: (a->a->Ordering) -> [a] -> [a]
sorteer cmp = labels . lijstNaarBoom cmp
```

Weglaten uit zoekbomen

Een zoekboom kan als database gebruikt worden. Naast de operaties opsommen, invoegen en opbouwen, waarvoor al functies geschreven zijn, zou daarbij een functie voor het weglaten van een te specificeren element goed van pas komen. Deze functie lijkt een beetje op de functie `insertBoom`; de functie wordt al naar gelang de aangetroffen waarde recursief aangeroepen op de linker- of de rechter-deelboom.

```
deleteBoom :: (a->a->Ordering) -> a -> Boom a -> Boom a
deleteBoom cmp e Blad = Blad
deleteBoom cmp e (Tak x li re) = case e `cmp` x of
    LT -> Tak x (deleteBoom cmp e li) re
    EQ -> samenvoegen li re
    GT -> Tak x li (deleteBoom cmp e re)
```

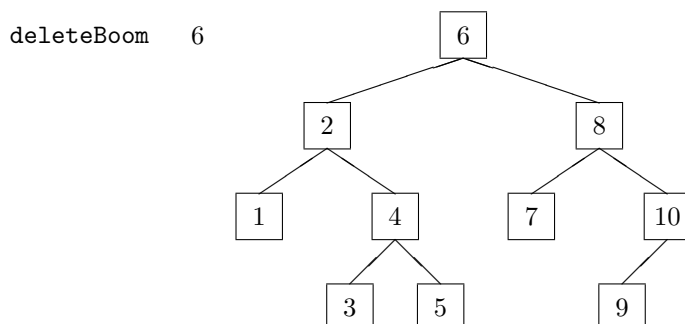
Als de waarde echter in de boom aangetroffen wordt (het geval `e==x`), kan hij niet zomaar worden weggelaten zonder een ‘gat’ achter te laten. Daarom is er een functie `samenvoegen` nodig, die twee zoekbomen samenvoegt. Deze functie werkt door het grootste element uit de linker deelboom te gebruiken als nieuw splitspunt. Als de linker deelboom leeg is, is samenvoegen natuurlijk ook geen probleem:

```
samenvoegen :: Boom a -> Boom a -> Boom a
samenvoegen Blad b2 = b2
samenvoegen b1 b2 = Tak x b1' b2
    where (x,b1') = grootsteUit b1
```

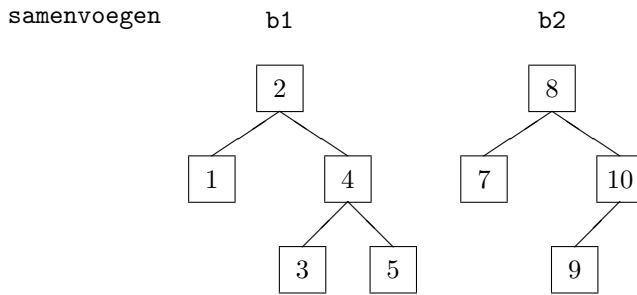
De functie `grootsteUit` levert behalve het grootste element van een boom ook de boom op die ontstaat door dit grootste element te verwijderen. Deze twee resultaten worden in een tupel samengevoegd. Het grootste element kun je vinden door steeds in de rechter deelboom af te dalen:

```
grootsteUit :: Boom a -> (a, Boom a)
grootsteUit (Tak x b1 Blad) = (x, b1)
grootsteUit (Tak x b1 b2) = (y, Tak x b1 b2')
    where (y,b2') = grootsteUit b2
```

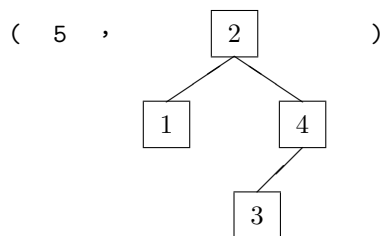
Om de werking van `deleteBoom` te demonstreren bekijken we een voorbeeld, waarbij we voor de duidelijkheid de bomen grafisch voorstellen. Bij de aanroep van



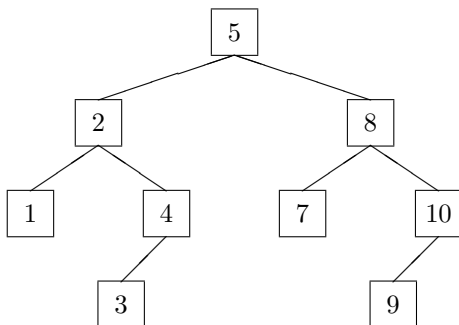
wordt de functie `samenvoegen` aangeroepen met de linker- en de rechter deelboom als parameter:



Door `samenvoegen` wordt de functie `grootsteUit` aangeroepen met `b1` als parameter. Dat levert een tweetupel $(x, b1')$ op:



De bomen `b1'` en `b2` worden als linker- en rechter deelboom gebruikt in een nieuwe zoekboom:



Omdat de functie `grootsteUit` alleen maar wordt aangeroepen vanuit `samenvoegen`, hoeft hij niet gedefinieerd te worden op een `Blad`-boom. Hij wordt immers alleen maar met niet-lege bomen aangeroepen, omdat de lege boom in de functie `samenvoegen` al apart wordt afgehandeld.

3.4.3 Speciaal gebruik van data-definities

Behalve voor de constructie van bomen van allerlei vorm, kan een data-definitie nog op een paar opmerkelijke manieren worden gebruikt. Het datatype-mechanisme is zo universeel bruikbaar, dat er ook dingen mee gemaakt kunnen worden waarvoor in andere talen vaak aparte constructies nodig zijn.

Drie voorbeelden hiervan zijn: eindige types, vereniging van types, en beschermde types.

Eindige types

De constructor-functies in een datatype-definitie mogen ook nul parameters hebben. Dat bleek al eerder: de constructor-functie `Blad` van het type `Boom` had bijvoorbeeld geen parameters.

Het is ook toegestaan dat *geen enkele* constructor-functie parameters heeft. Het resultaat is een type dat precies zoveel elementen bevat als er constructor-functies zijn: een eindig type. De constructor-functies dienen als constanten om deze elementen aan te duiden. Een voorbeeld:

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

Functies op dit soort types kunnen gewoon met behulp van patronen worden geschreven, bijvoorbeeld:

```

move      :: Richtig -> (Int,Int) -> (Int,Int)
move Noord (x,y) = (x,y+1)
move Oost  (x,y) = (x+1,y)
move Zuid  (x,y) = (x,y-1)
move West  (x,y) = (x-1,y)

```

De voordelen van zo'n eindig type boven een codering met integers of characters zijn:

- functie-definities zijn duidelijker doordat de namen van de elementen gebruikt kunnen worden, in plaats van obscure coderingen;
- het type-systeem klaagt als je richtingen per ongeluk zou optellen (als de richtingen door integers gecodeerd werden, dan zou dit geen foutmelding geven, met alle vervelende gevolgen van dien).

Eindige types zijn niets nieuws: in feite kan het type `Bool` op deze manier gedefinieerd worden:

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

Dit is ook de reden dat `False` en `True` met een hoofdletter geschreven moeten worden: het zijn de constructor-functies van `Bool`. (Deze definitie staat overigens niet echt in de prelude. Booleans zijn niet 'voorgedefinieerd' maar 'ingebouwd'. De reden daarvoor is, dat andere ingebouwde taal-constructies de Booleans al moeten 'kennen', zoals gevalsonderscheid met `|` in een functiedefinitie.)

Ook het type `Ordering` is natuurlijk op deze manier gedefinieerd in de prelude:

```
data Ordering = LT | EQ | GT
```

Vereniging van types

Alle elementen van een lijst moeten hetzelfde type hebben. In een tupel mogen waarden van verschillend type worden opgeslagen, maar bij tupels is het aantal elementen weer niet variabel. Soms wil je echter een lijst maken, waarvan bijvoorbeeld sommige elementen integers zijn, en andere elementen characters.

Met een data-definitie is het mogelijk een type `IntOfChar` te maken, die als elementen zowel de integers als de characters heeft:

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

Hiermee kun je een 'gemengde' lijst maken:

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

De enige prijs die je moet betalen, is dat elk element gemarkeerd moet worden met de constructor-functie `EenInt` of `EenChar`. Deze functies zijn te beschouwen als conversiefuncties:

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

waarvan het gebruik vergelijkbaar is met dat van ingebouwde conversiefuncties zoals

```
truncate :: Float -> Int
chr      :: Int -> Char
```

In de prelude is een type `Maybe` gedefinieerd:

```
data Maybe a
  = Nothing
  | Just a
```

dat gebruikt kan worden bij functies die soms een resultaat hebben, en soms niet. Gebruikmakend hiervan is er in de prelude een functie `lookupBy` gemaakt, die een waarde in een lijst van tupels zoekt, als die gevonden wordt `Just` de andere helft van het tupel oplevert, en als de waarde niet in de lijst zit `Nothing` oplevert:

```
lookupBy :: (a->a->Bool) -> a -> [(a,b)] -> Maybe b
lookupBy _ _ []         = Nothing
lookupBy eq k ((x,y):xys)
  | k 'eq' x = Just y
  | otherwise = lookupBy eq k xys
```

Beschermde types

worden gedefinieerd, bijvoorbeeld

```
type Datum = (Int,Int)
type Ratio = (Int,Int)
```

dan kunnen ze door elkaar worden gebruikt. ‘Datums’ kunnen daardoor ineens worden verwerkt alsof het ‘rationale getallen’ zijn, zonder dat dat foutmeldingen van de type-checker oplevert.

Met data-definities is het mogelijk om echte nieuwe types te maken, zodat bijvoorbeeld een `Ratio` niet meer zonder meer uitwisselbaar is met elke andere `(Int,Int)`. In plaats van de type-definitie wordt daartoe de volgende data-definitie gegeven:

```
data Ratio = Rat (Int,Int)
```

Er is dus slechts één constructor-functie. Om een breuk te maken met een teller 3 en een noemer 5, is het nu niet meer voldoende om `(3,5)` te schrijven, maar moet je schrijven `Rat (3,5)`. Net als bij verenigings-types kan `Rat` worden beschouwd als conversiefunctie van `(Int,Int)` naar `Ratio`. Het is eigenlijk wel zo handig om ook constructor-functies te curryen. In dat geval krijgen ze niet een tupel als parameter, maar twee losse waarden. De bijbehorende datatype-definitie is:

```
data Ratio = Rat Int Int
```

Deze methode wordt veel gebruikt om *beschermd type* te maken. Een beschermd type bestaat uit een data-definitie en een aantal functies die op het gedefinieerde type werken (in het geval van `Ratio` bijvoorbeeld `qPlus`, `qMin`, `qMaal` en `qDeel`).

De rest van het programma (dat mogelijkerwijs door een andere programmeur geschreven kan zijn) mag van het type gebruik maken via de daarvoor bedoelde functies. Het mag echter geen gebruik maken van de manier waarop het type is opgebouwd. Dat is te bereiken door de naam van de constructor-functie ‘geheim te houden’. Als later de representatie van rationale getallen om een of andere reden gewijzigd zou moeten worden, hoeven alleen de vier basisfuncties opnieuw geschreven te worden; de rest van het programma blijft gegarandeerd werken.

Als naam voor de constructor-functie wordt vaak dezelfde naam als de naam van het type gekozen, dus bijvoorbeeld

```
data Ratio = Ratio Int Int
```

Daar is niets op tegen; voor de interpreter is er geen verwarring mogelijk (het woord `Ratio` in een type, bijvoorbeeld achter `::`, stelt het type voor; in een expressie is het de constructor-functie).

Opgaven

3.1 Ga na dat `[1,2]++[]` volgens de definitie van `++` inderdaad `[1,2]` oplevert. Hint: schrijf `[1,2]` als `1:(2:[])`.

3.2 Schrijf de functie `concat` als aanroep van `foldr`.

3.3 Welke van de volgende expressies levert `True` op voor alle lijsten `xs`, en welke `False`:

```
[[]] ++ xs == xs
 [[]] ++ xs == [xs]
 [[]] ++ xs == [ [], xs ]
 [[]] ++ [xs] == [ [], xs ]
 [xs] ++ [] == [xs]
 [xs] ++ [xs] == [xs,xs]
```

3.4 De functie `filter` kan gedefinieerd worden in termen van `concat` en `map`:

```
filter p = concat . map box
      where box x =
```

Completeer de definitie van de functie `box` die hierin is gebruikt.

3.5 Schrijf met behulp van de functie `iterate` een niet-recursieve definitie van `repeat`.

3.6 Schrijf een functie met twee lijsten als parameter, die van elk element uit de tweede lijst het eerste voorkomen in de eerste lijst verwijdert. (Deze functie is in de prelude als operator gedefinieerd en heet `\`).

- 3.7** Gebruik de functies `map` en `concat` in plaats van de lijstcomprehensie-notatie om de volgende lijst te definiëren:

```
[ (x,y+z) | x<-[1..10], y<-[1..x], z<-[1..y] ]
```

- 3.8** Het vereenvoudigen van breuken is niet nodig als breuken nooit direct met elkaar vergeleken worden. Schrijf een functie `qEq` die gebruikt kan worden in plaats van `==`. Deze functie levert `True` op als twee breuken dezelfde waarde hebben, ook als de breuken niet vereenvoudigd zijn.

- 3.9** Schrijf de vier rekenkundige functies voor het rekenen met *complexe getallen*. Complexe getallen zijn getallen van de vorm $a + bi$, waarbij a en b reële getallen zijn, en i een ‘getal’ is met de eigenschap $i^2 = -1$. Hint: leid voor de deel-functie eerst een formule af voor $\frac{1}{a+bi}$ door x en y op te lossen uit $(a+bi) * (x+yi) = (1+0i)$.

- 3.10** Schrijf een functie `stringInt`, die van een string de overeenkomstige integer maakt. Bijvoorbeeld: `stringInt "123"` levert de waarde 123. Beschouw daarvoor de string als lijst characters, en bepaal welke operator tussen de characters moet staan. Moet je daarbij van rechts of van links beginnen?

blz. 56

- 3.11** Definieer de functie `curry`, als tegenhanger van `uncurry` uit paragraaf 3.3.5.

- 3.12** Schrijf een zoekboom-versie van de functie `zoekOp`, zoals `elemBoom` een zoekboom-versie is van `elem`. Geef ook het type van de functie.

- 3.13** De functie `map` kan op functies worden toegepast. Het resultaat is ook weer een functie (met een ander type). Er is geen enkele voorwaarde verbonden aan het soort functies waarop `map` toegepast kan worden. Je kunt hem dus ook op de functie `map` zelf toepassen! Wat is het type van de expressie `map map`?

- 3.14** Geef een definitie van `until` die gebruik maakt van `iterate` en `dropWhile`.

- 3.15** Geef een directe definitie van de operator `<` op lijsten. Deze definitie mag dus geen gebruik maken van operatoren zoals `<=` op lijsten. (Als je deze definitie daadwerkelijk met de Helium-interpretator wilt uitproberen, gebruik dan een andere naam dan `<`, omdat de operator `<` al in de prelude wordt gedefinieerd.)

- 3.16** Schrijf de functie `length` als aanroep van `foldr`. (Hint: begin aan de rechterkant met het getal 0, en zorg ervoor dat de operator die achtereenvolgens op alle elementen van de lijst wordt toegepast steeds 1 bij het tussenresultaat optelt, ongeacht de waarde van het lijs-telement.) Wat is het type van de functie die daarbij aan `foldr` wordt meegegeven?

blz. 42

- 3.17** In paragraaf 3.1.5 werden twee sorteermethodes genoemd: de op `insert` gebaseerde functie `isort`, en de op `merge` gebaseerde functie `msort`. Een andere sorteermethode werkt volgens het volgende principe. Bekijk het eerste element van de te sorteren lijst. Neem nu alle elementen van de lijst die kleiner zijn dan deze waarde. In het eindresultaat moeten al deze waarden vóór het eerste element komen. Ze moeten wel eerst (met een recursieve aanroep) gesorteerd worden. De waarden uit de lijst die juist groter zijn dan het eerste element moeten (gesorteerd) erachter komen. (Dit algoritme staat bekend onder de naam *quicksort*). Schrijf een functie die volgens dit principe werkt. Bedenk zelf wat het basisgeval is. Wat is het essentiële verschil tussen deze functie en `msort`?

- 3.18** Beschouw de functie `groepeer` met het volgende type:

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

Deze functie deelt de een gegeven lijst in deel-lijsten (die in een lijst van lijsten worden opgeleverd), waarbij de deel-lijsten een gegeven lengte hebben. Alleen de laatste deel-lijst mag zonnodig wat korter zijn. De functie kan als bijvoorbeeld als volgt gebruikt worden:

```
? groepeer 3 [1..11]
[ [1,2,3], [4,5,6], [7,8,9], [10,11] ]
```

blz. 48

Schrijf deze functie volgens hetzelfde principe als de functie `intString` in paragraaf 3.2.6, dat wil zeggen door samenstelling van een aantal partiële parametrisaties van `iterate`, `takeWhile` en `map`.

3.19 Bekijk bomen van het volgende type:

```
data Boom2 a = Blad2 a
             | Tak2 (Boom2 a) (Boom2 a)
```

Schrijf een functie `mapBoom` en een functie `foldBoom` die op een `Boom2` werken, naar analogie van de functies `map` en `foldr` op lijsten. Geef ook het type van deze functies.

3.20 Schrijf een functie `diepte`, die oplevert uit hoeveel nivo's een `Boom2` bestaat. Geef een definitie met inductie, en een alternatieve definitie waarin je `mapBoom` en `foldBoom` gebruikt.

3.21 Schrijf een functie `toonBoom`, die een aantrekkelijk representatie als string van een boom zoals gedefinieerd op blz. 58 geeft. In de string moet elk blad op een aparte regel komen te staan (gescheiden door `"\n"`); bladeren op een dieper nivo moeten meer zijn ingesprongen dan bladeren op een minder diep nivo.

3.22 Stel dat een boom b diepte n heeft. Wat is het minimale en het maximale aantal bladeren dat b kan bevatten?

3.23 Schrijf een functie die gegeven een gesorteerde lijst een zoekboom oplevert (dus een boom die als je er `labels` op toepast een gesorteerde lijst oplevert). Zorg ervoor dat de boom niet 'scheefgroeit', zoals het geval is als je de functie `lijstNaarBoom` zou gebruiken.

Hoofdstuk 4

Algoritmen op lijsten

4.1 Combinatorische functies

4.1.1 Segmenten en deelrijen

Combinatorische functies werken op een lijst. Ze leveren een lijst van lijsten op, waarbij geen gebruik gemaakt wordt van specifieke eigenschappen van de elementen van de lijst. Het enige wat combinatorische functies kunnen doen, is elementen weglaten, elementen verwisselen, of elementen tellen.

In deze en de volgende paragraaf worden een aantal combinatorische functies gedefinieerd. Omdat ze geen gebruik maken van eigenschappen van de elementen van hun parameter-lijst, zijn het polymorfe functies:

```
inits, tails, segs ::      [a] -> [[a]]
subs, perms       ::      [a] -> [[a]]
combs             :: Int -> [a] -> [[a]]
```

Om de werking van deze functies te illustreren volgen hieronder de uitkomsten van deze functies toegepast op de lijst [1,2,3,4]

inits	tails	segs	subs	perms	combs 2	combs 3
[]	[1,2,3,4]	[]	[]	[1,2,3,4]	[1,2]	[1,2,3]
[1]	[2,3,4]	[4]	[4]	[2,1,3,4]	[1,3]	[1,2,4]
[1,2]	[3,4]	[3]	[3]	[2,3,1,4]	[1,4]	[1,3,4]
[1,2,3]	[4]	[3,4]	[3,4]	[2,3,4,1]	[2,3]	[2,3,4]
[1,2,3,4]	[]	[2]	[2]	[1,3,2,4]	[2,4]	
		[2,3]	[2,4]	[3,1,2,4]	[3,4]	
		[2,3,4]	[2,3]	[3,2,1,4]		
		[1]	[2,3,4]	[3,2,4,1]		
		[1,2]	[1]	[1,3,4,2]		
		[1,2,3]	[1,4]	[3,1,4,2]		
		[1,2,3,4]	[1,3]	[3,4,1,2]		
			[1,3,4]	[3,4,2,1]		
			[1,2]	[1,2,4,3]		
			[1,2,4]	[2,1,4,3]		
			[1,2,3]	[2,4,1,3]		
			[1,2,3,4]	[2,4,3,1]		
				[1,4,2,3]		
				(7 andere)		

Zoals uit de voorbeelden waarschijnlijk al duidelijk is, is de betekenis van deze zes functies als volgt:

- **inits** levert alle *beginsegmenten* van een lijst, dat wil zeggen aaneengesloten stukken van de lijst die aan het begin beginnen. De lege lijst telt ook als beginsegment.
- **tails** levert alle *eindsegmenten* van een lijst: aaneengesloten stukken die tot het eind doorlopen. Ook de lege lijst is een eindsegment.
- **segs** levert *alle segmenten* van een lijst: beginsegmenten en eindsegmenten, maar ook aaneengesloten stukken uit het midden.
- **subs** levert alle *subsequences* (deelrijen) van een lijst. In tegenstelling tot segmenten hoeven de elementen van een deelrij in de originele lijst niet aaneengesloten te zijn. Er zijn dus meer deelrijen dan segmenten.
- **perms** levert alle *permutaties* van een lijst. Een permutatie van een lijst bevat dezelfde elementen, maar mogelijk in een andere volgorde.
- **combs n** levert alle *combinaties van n elementen*, dus alle manieren om n elementen te kiezen uit een lijst. De volgorde is daarbij hetzelfde als in de originele lijst.

Deze combinatorische functies kunnen recursief worden gedefinieerd. In de definitie worden dus steeds de gevallen `[]` en `(x:xs)` apart behandeld. In het geval `(x:xs)` wordt de functie recursief aangeroepen op de lijst `xs`.

Er is een handige manier om op een idee te komen van de definitie van een functie `f`. Kijk bij een voorbeeldlijst `(x:xs)` wat het resultaat is van de recursieve aanroep `f xs`, en probeer het resultaat aan te vullen tot de uitkomst van `f (x:xs)`.

inits

Bij de beschrijving van beginsegmenten hierboven is ervoor gekozen om de lege lijst ook als beginsegment te laten tellen. De lege lijst heeft dus één beginsegment: de lege lijst zelf. De definitie van `inits` voor het geval ‘lege lijst’ is daarom als volgt:

```
inits [] = [ [] ]
```

Voor het geval `(x:xs)` kijken we naar de gewenste uitkomsten voor de lijst `[1,2,3,4]`.

```
inits [1,2,3,4] = [ [], [1], [1,2], [1,2,3], [1,2,3,4] ]
inits [2,3,4]   = [ [], [2], [2,3], [2,3,4] ]
```

Hieruit blijkt dat de tweede t/m vijfde elementen van `inits [1,2,3,4]` overeenkomen met de elementen van `inits [2,3,4]`, alleen steeds met een extra 1 op kop. Deze vier lijsten moeten dan nog worden aangevuld met een lege lijst.

Dit mechanisme wordt algemeen beschreven in de tweede regel van de definitie van `inits`:

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

tails

Net als bij `inits` heeft de lege lijst één eindsegment: de lege lijst. Het resultaat van `tails []` is dus een lijst met als enige element de lege lijst.

Om op een idee te komen voor de definitie van `tails (x:xs)` kijken we eerst weer naar het voorbeeld `[1,2,3,4]`:

```
tails [1,2,3,4] = [ [1,2,3,4], [2,3,4], [3,4], [4], [] ]
tails [2,3,4]   = [ [2,3,4], [3,4], [4], [] ]
```

Bij deze functie zijn het tweede t/m vijfde element dus precies gelijk aan de elementen van de recursieve aanroep. Het enige wat moet gebeuren is uitbreiding met een eerste element (`[1,2,3,4]` in het voorbeeld).

Gebruik makend van dit idee kan de complete definitie geschreven worden:

```
tails []      = [ [] ]
tails (x:xs) = (x:xs) : tails xs
```

De haakjes in de tweede regel zijn essentieel: zonder haakjes zou de typering incorrect zijn, omdat de operator `:` dan naar rechts associeert.

segs

Het enige segment van de lege lijst is weer de lege lijst. De uitkomst van `segs []` is dus, net als bij `inits` en `tails`, een singleton-lege-lijst.

Om op het spoor van de definitie van `segs (x:xs)` te komen, passen we de beproefde methode weer toe:

```
segs [1,2,3,4] = [ [], [4], [3], [3,4], [2], [2,3], [2,3,4], [1], [1,2], [1,2,3], [1,2,3,4] ]
segs [2,3,4]   = [ [], [4], [3], [3,4], [2], [2,3], [2,3,4] ]
```

Als je ze maar op de goede volgorde zet, blijkt dat de eerste zeven elementen van het gewenste resultaat precies overeenkomen met de recursieve aanroep. In het tweede deel van het resultaat (de lijsten die met een 1 beginnen) zijn de beginsegmenten van `[1,2,3,4]` te herkennen (alleen de lege lijst is daarbij weggelaten, want die zit al in het resultaat).

Als definitie van `segs` kan dus genomen worden:

```

segs []      = [ [] ]
segs (x:xs) = segs xs ++ tail (inits (x:xs))

```

Een andere manier om de lege lijst uit de `inits` te verwijderen is:

```

segs []      = [ [] ]
segs (x:xs) = segs xs ++ map (x:) (inits xs)

```

subs

De lege lijst is de enige deelrij van de lege lijst. Voor de definitie van `subs (x:xs)` kijken we weer naar het voorbeeld:

```

subs [1,2,3,4] = [ [1,2,3,4], [1,2,3], [1,2,4], [1,2], [1,3,4], [1,3], [1,4], [1],
                  , [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], [] ]
subs [2,3,4]   = [ [2,3,4], [2,3], [2,4], [2], [3,4], [3], [4], [] ]

```

Het aantal elementen van `subs (x:xs)` (16 in het voorbeeld) is precies twee keer zo groot als het aantal elementen van de recursieve aanroep `subs xs`. De tweede helft van het totaal resultaat is precies gelijk aan het resultaat van de recursieve aanroep. Ook in de eerste helft zijn deze 8 lijsten weer te herkennen, alleen staat er daar steeds een 1 op kop.

De definitie kan dus luiden:

```

subs []      = [ [] ]
subs (x:xs) = map (x:) (subs xs) ++ subs xs

```

De functie wordt tweemaal recursief aangeroepen met dezelfde parameter. Dat is zonde van het werk: beter kan de aanroep maar éénmaal gedaan worden, waarna het resultaat tweemaal gebruikt wordt. Dit levert veel tijdswinst op, want ook voor het bepalen van `subs xs` wordt de functie weer tweemaal recursief aangeroepen, en in die recursieve aanroepen weer... Een veel efficiëntere definitie is dus:

```

subs []      = [ [] ]
subs (x:xs) = map (x:) subsxs ++ subsxs
              where subsxs = subs xs

```

4.1.2 Permutaties en combinaties

perms

Een *permutatie* van een lijst is een lijst met dezelfde elementen, maar mogelijk in een andere volgorde. De lijst van alle permutaties van een lijst kan goed met een recursieve functie worden gedefinieerd.

De lege lijst heeft één permutatie: de lege lijst. Alle 0 elementen zitten daar namelijk in, en in dezelfde volgorde...

Het interessante geval is natuurlijk de niet-lege lijst `(x:xs)`. We kijken eerst weer naar een voorbeeld:

```

perms [1,2,3,4] = [ [1,2,3,4], [2,1,3,4], [2,3,1,4], [2,3,4,1],
                  , [1,3,2,4], [3,1,2,4], [3,2,1,4], [3,2,4,1],
                  , [1,3,4,2], [3,1,4,2], [3,4,1,2], [3,4,2,1],
                  , [1,2,4,3], [2,1,4,3], [2,4,1,3], [2,4,3,1],
                  , [1,4,2,3], [4,1,2,3], [4,2,1,3], [4,2,3,1],
                  , [1,4,3,2], [4,1,3,2], [4,3,1,2], [4,3,2,1] ]
perms [2,3,4]   = [ [2,3,4], [3,2,4], [3,4,2], [2,4,3], [4,2,3], [4,3,2] ]

```

Het aantal permutaties loopt flink op: van een lijst met vier elementen zijn er viermaal zoveel permutaties als van een lijst met drie elementen. In dit voorbeeld is het wat moeilijker om het resultaat van de recursieve aanroep te herkennen. Dit lukt pas door de 24 elementen in 6 groepjes van 4 te verdelen. In elke groepje zitten lijsten met dezelfde waarden als die van één lijst van de recursieve aanroep. Het nieuwe element wordt daar op alle mogelijke manieren tussen gezet.

Bijvoorbeeld, het derde groepje `[[1,3,4,2], [3,1,4,2], [3,4,1,2], [3,4,2,1]]` bevat steeds de elementen `[3,4,2]`, waarbij het element 1 is toegevoegd respectievelijk aan het begin, op de tweede plaats, op de derde plaats, en aan het eind.

Voor het op alle manieren tussenvoegen van één element in een lijst kan een hulpfunctie worden geschreven, die ook weer recursief gedefinieerd is:

```
tussen      :: a -> [a] -> [[a]]
tussen e [] = [ [e] ]
tussen e (y:ys) = (e:y:ys) : map (y:) (tussen e ys)
```

In de definitie van `perms (x:xs)` wordt deze functie, partiëel geparametriseerd met `x`, toegepast op alle elementen van het resultaat van de recursieve aanroep. In het voorbeeld levert dat een lijst met zes lijsten van vier lijstjes. De bedoeling is echter dat er één lange lijst van 24 lijstjes uitkomt. Op de lijst van lijsten van lijstjes moet dus nog de functie `concat` worden toegepast, die immers dat effect heeft.

Al met al wordt de functie `perms` als volgt gedefinieerd:

```
perms [] = [ [] ]
perms (x:xs) = concat (map (tussen x) (perms xs))
               where tussen e [] = [ [e] ]
                     tussen e (y:ys) = (e:y:ys) : map (y:) (tussen e ys)
```

combs

Een laatste voorbeeld van een combinatorische functie is de functie `combs`. Deze functie heeft, behalve de lijst, ook een getal als parameter:

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

De bedoeling is dat in het resultaat van `combs n xs` alle deelrijen van `xs` met lengte `n` zitten. De functie kan dus eenvoudigweg gedefinieerd worden door

```
combs n xs = filter goed (subs xs)
             where goed xs = length xs == n
```

Deze definitie is echter niet zo erg efficiënt. Het aantal deelrijen is namelijk meestal erg groot, dus `subs` kost veel tijd, terwijl de meeste deelrijen door `filter` weer worden weggegooid. Een betere definitie is te verkrijgen door `combs` direct te definiëren, zonder `subs` te gebruiken.

In de definitie van `combs` worden voor de integer-parameter de gevallen 0 en `n` onderscheiden, waarbij door de volgorde dat deze alternatieven geprobeerd worden, duidelijk is dat $n > 0$. In dat geval worden ook voor de lijst-parameter twee gevallen onderscheiden. De definitie krijgt dus de volgende vorm:

```
combs 0 xs = ...
combs n [] = ...
combs n (x:xs) = ...
```

Deze drie gevallen worden hieronder apart bekeken.

- Voor het kiezen van nul elementen uit een lijst is er één mogelijkheid: de lege lijst. Het resultaat van `combs 0 xs` is daarom een singleton-lege-lijst. Het maakt daarbij niet uit of `xs` leeg is of niet.
- Het kiezen van minstens één element uit de lege lijst is onmogelijk. Het resultaat van `combs n []`, in een situatie waarin we weten dat $n > 0$, is dan ook de lege lijst: er is geen oplossing mogelijk. Let wel: hier is dus sprake van een *lege lijst oplossing* en niet van *de lege lijst als enige oplossing* zoals in het vorige geval. Een belangrijk verschil!
- Het kiezen van `n` elementen uit de lijst `x:xs` is wel mogelijk, mits het kiezen van `n-1` elementen uit `xs` mogelijk is. De oplossingen zijn te verdelen in twee groepen: lijstjes waar `x` in zit, en lijstjes waar `x` niet in zit.
 - Voor de lijstjes waar `x` wel in zit, moeten uit de overige elementen `xs` nog `n-1` elementen gekozen worden. Daarin moet dan steeds `x` op kop gezet worden.
 - Voor de lijstjes waar `x` niet in zit, moeten alle `n` elementen uit `xs` gekozen worden.

Voor beide gevallen kan `combs` recursief aangeroepen worden. De resultaten kunnen gecombineerd worden met `++`.

De definitie van `combs` komt er dus als volgt uit te zien:

```
combs 0 xs = [ [] ]
combs n [] = [ ]
combs n (x:xs) = map (x:) (combs (n-1) xs) ++ combs n xs
```

Bij deze functie kunnen niet, zoals bij `subs`, de twee recursieve aanroepen gecombineerd worden. De twee aanroepen hebben hier namelijk verschillende parameters.

4.1.3 De @-notatie

De definitie van `tails` heeft iets omslachtigs:

```
tails []      = [ [] ]
tails (x:xs) = (x:xs) : tails xs
```

In de tweede regel wordt de parameter-lijst door het patroon gesplitst in een kop `x` en een staart `xs`. De staart wordt gebruikt bij de recursieve aanroep, maar de kop en de staart worden ook weer samengevoegd tot de lijst `(x:xs)`. Dat is zonde van het werk, want deze lijst is in feite ook al beschikbaar als parameter.

Een andere definitie van `tails` zou kunnen luiden:

```
tails [] = [ [] ]
tails xs = xs : tails (tail xs)
```

Nu is het opnieuw opbouwen van de parameter-lijst niet nodig, omdat hij helemaal niet gesplitst wordt. Maar nu moet, om de staart bij de recursieve aanroep te kunnen meegeven, expliciet de functie `tail` worden gebruikt. Het leuke van patronen was nu juist, dat dat niet nodig is.

Ideaal zou het zijn om het goede van deze twee definities te combineren. De parameter moet dus zowel als geheel beschikbaar zijn, als gesplitst in een kop en een staart. Voor deze situatie is een speciale notatie beschikbaar. Vóór een patroon mag een naam worden geschreven die het geheel aanduidt. De naam wordt van het patroon gescheiden door het symbool `@`.

Met gebruik van deze constructie wordt de definitie van `tails` als volgt:

```
tails []      = [ [] ]
tails lyst@(x:xs) = lyst : tails xs
```

Hoewel het symbool `@` in operator-symbolen gebruikt mag worden, is een losse `@` speciaal gereserveerd voor deze constructie.

Bij functies die al eerder in dit diktaat gedefinieerd werden, komt een `@`-patroon ook goed van pas. Bijvoorbeeld in de functie `dropWhile`:

```
dropWhile p []      = []
dropWhile p ys@(x:xs)
  | p x      = dropWhile p xs
  | otherwise = ys
```

Dit is ook de manier waarop `dropWhile` in werkelijkheid in de prelude is gedefinieerd.

4.2 Polynomen

4.2.1 Representatie

Een *polynoom* is een som van *termen*, waarbij elke term bestaat uit het product van een reëel getal en een natuurlijke macht van een variabele.

$$\begin{aligned}
 &x^2 + 2x + 1 \\
 &4.3x^3 + 2.5x^2 + 0.5 \\
 &6x^5 \\
 &x \\
 &3
 \end{aligned}$$

De hoogste macht die voorkomt heet de *graad* van het polynoom. In bovenstaande voorbeelden is de graad dus achtereenvolgens 2, 3, 5, 1 en 0. Het lijkt misschien raar om 3 een polynoom te noemen; het getal 3 is echter gelijk aan $3x^0$, en is dus inderdaad een product van een getal en een natuurlijke macht van x .

Met polynomen kun je rekenen: je kunt polynomen bij elkaar optellen, van elkaar aftrekken en met elkaar vermenigvuldigen. Het product van de polynomen $x + 1$ en $x^2 + 3x$ is bijvoorbeeld $x^3 + 4x^2 + 3x$. Als je twee polynomen echter door elkaar deelt is het resultaat niet altijd een polynoom. Het komt nu goed uit dat getallen ook polynomen zijn: zo is het resultaat van het optellen van $x + 1$ en $-x$ het polynoom 1.

In deze paragraaf wordt een datatype `Poly` ontworpen, waarmee polynomen kunnen worden gerepresenteerd. In de volgende paragraaf worden een aantal functies gedefinieerd, die op dat soort

polynomen werken:

```
pPlus  :: Poly -> Poly -> Poly
pMin   :: Poly -> Poly -> Poly
pMaal  :: Poly -> Poly -> Poly
pEq    :: Poly -> Poly -> Bool
pGraad :: Poly -> Int
pEval  :: Float -> Poly -> Float
polyString :: Poly -> String
```

Een mogelijke representatie voor polynomen is ‘functie van float naar float’. Het nadeel daarvan is echter dat je het resultaat van het vermenigvuldigen van twee polynomen niet meer als polynoom kunt inspecteren; je hebt dan een functie die je alleen nog maar op waarden kunt loslaten. Ook is het dan niet mogelijk om een gelijkheids-operator te schrijven; het is dus niet mogelijk om te testen of het product van de polynomen x en $x + 1$ gelijk is aan het polynoom $x^2 + x$.

Het is dus beter om een polynoom te representeren als een datastructuur met getallen. Daarbij ligt het voor de hand om een polynoom voor te stellen als lijst termen, waarbij elke term gekenmerkt wordt door een `Float` (de coëfficiënt) en een `Int` (de exponent). Een polynoom kan dus worden gerepresenteerd als een lijst twee-tupels. We maken er echter meteen maar een *datatype* van met de volgende definitie:

```
data Poly = Poly [Term]
data Term = Term (Float,Int)
```

Let op: de namen `Poly` en `Term` worden dus zowel als naam van het type gebruikt, als als naam van de (enige) constructorfunctie. Dit is toegestaan, want het is uit de context altijd duidelijk welke van de twee bedoeld wordt. Het woord `Poly` is een type in type-declaraties zoals

```
pEq :: Poly -> Poly -> Bool
```

maar het is een constructorfunctie in functie-definities zoals

```
pGraad (Poly []) = ...
```

Een aantal voorbeelden van representaties van polynomen is:

```
3x5 + 2x4  Poly [Term(3.0,5), Term(2.0,4)]
4x2         Poly [Term(4.0,2)]
2x + 1      Poly [Term(2.0,1), Term(1.0,0) ]
3           Poly [Term(3.0,0)]
0           Poly []
```

blz. 53

Net als bij de rationale getallen uit paragraaf 3.3.3 hebben we hier weer het probleem dat er meerdere representaties zijn voor één polynoom. Het polynoom $x^2 + 7$ kan bijvoorbeeld worden gerepresenteerd door de volgende expressies:

```
Poly [ Term(1.0,2), Term(7.0,0) ]
Poly [ Term(7.0,0), Term(1.0,2) ]
Poly [ Term(1.0,2), Term(3.0,0), Term(4.0,0) ]
```

Net als bij rationale getallen is het dus nodig om een polynoom te ‘vereenvoudigen’ nadat er operaties op zijn uitgevoerd. Vereenvoudigen bestaat in dit geval uit:

- sorteren van de termen, zodat de termen met de hoogste exponent voorop staan;
- samenvoegen van termen met gelijke exponent;
- verwijderen van termen met coëfficiënt nul.

Een alternatieve methode is om de polynomen niet te vereenvoudigen, maar dan moet er extra werk gedaan worden in de functie `pEq` waarmee polynomen vergeleken worden.

4.2.2 Vereenvoudiging

Voor het vereenvoudigen van polynomen schrijven we een functie

```
pEenvoud :: Poly -> Poly
```

Deze functie voert de drie genoemde aspecten van het vereenvoudigen uit, en kan dus geschreven worden als functie-samenstelling:

```
pEenvoud (Poly xs) = Poly (eenvoud xs)
  where eenvoud = verwijderNul . samenvExpo . sortTerms
```

Blijft de taak over om de drie samenstellende functies te schrijven. Alledrie werken ze op lijsten van termen.

In paragraaf 3.1.5 werd een functie gedefinieerd die een lijst sorteert. Daarbij moesten de waarden echter ordenbaar zijn, en werd de lijst gesorteerd van klein naar groot. Er is een algemenere sorteer-functie denkbaar, waarbij als extra parameter een criterium wordt meegegeven dat beslist in welke volgorde de elementen komen te staan. Deze functie zou hier goed van pas komen, want hij kan dan gebruikt worden met ‘heeft een grotere exponent’ als sorteer-criterium.

blz. 42

De functie `sortTerms` moet de termen sorteren naar opklimmende exponent. Gelukkig hebben we in paragraaf 3.1.5 al een functie gemaakt die volgens criterium naar believen kan sorteren. Die roepen we nu aan met als sorteer-criterium de gewenste ordening:

blz. 42

```
sortTerms :: [Term] -> [Term]
sortTerms = sort expoGroter
  where Term(c1,e1) 'expoGroter' Term(c2,e2) = ordInt e1 e2
```

De tweede functie die nodig is, is de functie die termen met gelijke exponenten samenvoegt. Deze functie mag er van uitgaan dat de termen al zijn gesorteerd op exponent. Termen met gelijke exponent staan dus naast elkaar. De functie laat lijsten met nul of één element ongemoeid. Bij lijsten met twee of meer elementen zijn er twee mogelijkheden:

- de exponenten van de eerste twee elementen zijn gelijk; de elementen worden samengevoegd, het nieuwe element wordt op kop van de rest gezet, en de functie wordt opnieuw aangeroepen, zodat het nieuwe element eventueel met nog meer elementen samengevoegd kan worden.
- de exponenten van de eerste twee elementen zijn *niet* gelijk; het eerste element komt dan onveranderd in het resultaat, de rest wordt aan een nadere inspectie onderworpen (misschien is het tweede element wel gelijk aan het derde).

Dit alles komt terug in de definitie:

```
samenvExpo :: [Term] -> [Term]
samenvExpo [] = []
samenvExpo [t] = [t]
samenvExpo (Term(c1,e1):Term(c2,e2):ts)
  | e1==e2 = samenvExpo (Term(c1+.c2,e1):ts)
  | otherwise = Term(c1,e1) : samenvExpo (Term(c2,e2):ts)
```

De derde benodigde functie is eenvoudig te maken:

```
verwijderNul :: [Term] -> [Term]
verwijderNul = filter coefNietNul
  where coefNietNul (Term(c,e)) = c/=0.0
```

Desgewenst kunnen de drie functies lokaal gedefinieerd worden in `pEenvoud`:

```
pEenvoud (Poly xs) = Poly (eenvoud xs)
  where eenvoud = vN . sE . sT
        sT = sort expoGroter
        sE [] = []
        sE [t] = [t]
        sE (Term(c1,e1):Term(c2,e2):ts)
          | e1==e2 = sE (Term(c1+.c2,e1):ts)
          | otherwise = Term(c1,e1) : sE (Term(c2,e2):ts)
        vN = filter coefNietNul
        coefNietNul (Term(c,e)) = c/=0.0
        Term(c1,e1) 'expoGroter' Term(c2,e2) = ordInt e1 e2
```

De functie `pEenvoud` verwijdert *alle* termen waarvan de coëfficiënt nul is. Het nul-polynoom wordt dus gerepresenteerd door `Poly []`, een lege lijst termen. Daarmee verschilt het nul-polynoom van andere polynomen waarin de variabele niet voorkomt. Het polynoom ‘3’ wordt bijvoorbeeld gerepresenteerd door `Poly [Term(3.0,0)]`.

4.2.3 Rekenkundige operaties

Het optellen van twee polynomen is eenvoudig. De lijsten van termen kunnen gewoon geconcateneerd worden. Daarna zorgt `pEenvoud` ervoor dat de termen gesorteerd worden, gelijke exponenten samengenomen worden, en nul-termen verwijderd worden:

```
pPlus :: Poly -> Poly -> Poly
```

```
pPlus (Poly xs) (Poly ys) = pEenvoud (Poly (xs++ys))
```

Voor het aftrekken van twee polynomen tellen we het eerste polynoom op bij het tegengestelde van het tweede:

```
pMin :: Poly -> Poly -> Poly
pMin p1 p2 = pPlus p1 (pNeg p2)
```

Blijft natuurlijk de vraag hoe het tegengestelde van een polynoom berekend wordt: daartoe moet het tegengestelde van elke term berekend worden.

```
pNeg :: Poly -> Poly
pNeg (Poly xs) = Poly (map tNeg xs)
tNeg :: Term -> Term
tNeg (Term(c,e)) = Term(-.c,e)
```

Vermenigvuldigen van polynomen is wat moeilijker. Daarvoor moet elke term van het eerste polynoom vermenigvuldigd worden met elke term van het andere polynoom. Dat vraagt om een hogere-orde functie: ‘doe iets met elk element van een lijst in combinatie met elk element van een andere lijst’. Deze functie noemen we `cpWith`. De `cp` staat voor *cross product*, de `With` is naar analogie van de functie `zipWith`. Als de eerste lijst leeg is, valt er niets samen te stellen. Als de eerste lijst de vorm `x:xs` heeft, moet het eerste element `x` met alle elementen van de tweede lijst samengesteld worden, en moet bovendien het cross-product van `xs` en de tweede lijst nog bepaald worden. Dit geeft de definitie:

```
cpWith :: (a->b->c) -> [a] -> [b] -> [c]
cpWith f [] ys = []
cpWith f (x:xs) ys = map (f x) ys ++ cpWith f xs ys
```

Deze definitie kan meteen gebruikt worden bij het vermenigvuldigen van polynomen:

```
pMaal :: Poly -> Poly -> Poly
pMaal (Poly xs) (Poly ys) = pEenvoud (Poly (cpWith tMaal xs ys))
```

Hierin wordt de functie `tMaal` gebruikt, die twee termen vermenigvuldigd. Zoals uit het voorbeeld $3x^2$ maal $5x^4$ is $15x^6$ blijkt, moeten daartoe de coëfficiënten worden vermenigvuldigd, en de exponenten opgeteld:

```
tMaal :: Term -> Term -> Term
tMaal (Term(c1,e1)) (Term(c2,e2)) = Term (c1*.c2,e1+e2)
```

Doordat we polynomen steeds vereenvoudigen, en dus steeds de term met de hoogste exponent voorop staat, is de graad van een polynoom gelijk aan de exponent van de eerste term. Alleen voor het nul-polynoom hebben we een aparte definitie nodig.

```
pGraad :: Poly -> Int
pGraad (Poly []) = 0
pGraad (Poly (Term(c,e):ts)) = e
```

Twee vereenvoudigde polynomen zijn gelijk als alle termen gelijk zijn. Twee termen zijn gelijk als de coëfficiënt en de exponent overeenstemmen. Dit alles laat zich gemakkelijk naar functies vertalen:

```
pEq :: Poly -> Poly -> Bool
pEq (Poly xs) (Poly ys) = length xs==length ys
                        && and (zipWith tEq xs ys)
tEq :: Term -> Term -> Bool
tEq (Term(c1,e1)) (Term(c2,e2)) = eqFloat c1 c2 && e1==e2
```

De functie `pEval` moet een polynoom uitrekenen met een specifieke waarde voor x ingevuld. Daartoe moeten alle termen geëvalueerd worden, en de resultaten opgeteld:

```
pEval :: Float -> Poly -> Float
pEval w (Poly xs) = sum (map (tEval w) xs)
tEval :: Float -> Term -> Float
tEval w (Term(c,e)) = c *. w ^ . e
```

Tenslotte schrijven we een functie voor de weergave van een polynoom als string. Hiervoor moeten de termen worden weergegeven als string, en ertussen moet een `+`-teken komen:

```
polyString :: Poly -> String
polyString (Poly []) = "0"
polyString (Poly [t]) = termString t
```

```
polyString (Poly (t:ts)) = termString t ++ " + "
                        ++ polyString (Poly ts)
```

Bij de weergave van een term laten we de coëfficiënt en de exponent weg als die 1 is. Als de exponent 0 is, wordt de variabele weggelaten, maar de coëfficiënt nooit. De exponent duiden we aan met een \wedge -teken, net zoals dat in Helium-expressies gebruikelijk is. De functie wordt daarmee:

```
termString :: Term -> String
termString (Term(c,0))    = floatString c
termString (Term(1.0,1)) = "x"
termString (Term(1.0,e)) = "x^" ++ intString e
termString (Term(c,e))   = showFloat c ++ "x^" ++ intString e
```

De functie `showFloat` is een preludefunctie die een `Float` omzet naar de bijbehorende stringrepresentatie.

Opgaven

- 4.1 Hoe wordt de volgorde van de elementen van `segs [1,2,3,4]` als de parameters van `++` in de definitie van `segs` worden omgewisseld?
- 4.2 Schrijf `segs` als combinatie van `inits`, `tails` en standaardfuncties. De volgorde van de elementen in het resultaat hoeft niet hetzelfde te zijn als op blz. 67.
- 4.3 Gegeven is een lijst `xs` met `n` elementen. Bepaal het aantal elementen van `inits xs`, `segs xs`, `subs xs`, `perms xs`, en `combs k xs`.
- 4.4 Schrijf een functie `bins :: Int -> [[Char]]` die alle getallen in het tweetalig stelsel (als strings nullen en enen) bepaalt met het gegeven aantal cijfers. Vergelijk de functie met de functie `subs`.
- 4.5 Schrijf een functie `gaps` die alle mogelijkheden geeft om één element uit een lijst weg te laten. Bijvoorbeeld:

```
gaps [1,2,3,4,5] = [ [2,3,4,5] , [1,3,4,5] , [1,2,4,5] , [1,2,3,5] , [1,2,3,4] ]
```

- 4.6 Vergelijk de voorwaarden wat betreft de afmetingen van de matrices bij matrixvermenigvuldiging met het type van de functie-samenstellings-operator `(.)`.
- 4.7 Ga na dat de recursieve definitie van matrix-determinant `det` overeenkomt met de expliciete definitie voor determinanten van 2×2 -matrices uit paragraaf ??.
- 4.8 Schrijf de functie `matInv`.
- 4.9 In paragraaf ?? werd de functie `transpose` beschreven als generalisatie van `zip`. In paragraaf 3.3.4 werd `zip` geschreven als `zipWith maak2tupel`. De functie `cp` wordt nu gedefinieerd als `cpWith maak2tupel`. Schrijf een functie `crossprod` die een generalisatie is van `cp` zoals `transpose` een generalisatie is van `zip`. Hoe kan `length (crossprod xs)` berekend worden zonder `crossprod` te gebruiken?

blz. ??

blz. ??

blz. 55

- 4.10 Een andere mogelijke representatie van polynomen is een lijst van coëfficiënten. De exponenten worden dus niet opgeslagen. De prijs daarvan is dat ‘ontbrekende’ termen als 0.0 opgeslagen moeten worden. Het is het handigst om de termen met de kleinste exponent aan het begin van de lijst op te slaan. Dus bijvoorbeeld:

```
x2 + 2x    [0.0, 2.0, 1.0]
4x3       [0.0, 0.0, 0.0, 4.0]
5          [5.0]
```

Schrijf functies voor de graad van een polynoom en voor de som en het product van twee polynomen in deze representatie.

Hoofdstuk 5

Algoritmen op bomen

5.1 Expressiebomen

5.1.1 Rekenkundige expressies

Data-declaraties worden gebruikt om de vorm van datastructuren te beschrijven. Een veel voorkomende niet-lijstvormige datastructuur is de *ontleedboom*. Een ontleedboom is een symbolische beschrijving van een expressie. Een expressie wordt in een ontleedboom dus in de vorm van een datastructuur opgeslagen.

Numerieke expressies worden bijvoorbeeld beschreven door bomen die zijn opgebouwd volgens de volgende data-declaratie:

```
data Expr = Con Float
          | Var String
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr **: Expr
          | Expr :/: Expr
```

De data-declaratie beschrijft de opbouw van rekenkundige expressies. Voor elk soort expressie (constante, variabele, optelling, aftrekking, vermenigvuldiging en deling) is er een constructor waarmee de representatie van de expressie kan worden opgebouwd.

Twee van de zes constructoren (`Con` en `Var`) hebben één parameter. De andere vier hebben twee parameters. Voor de duidelijkheid schrijven we ze als infix-operator, dus tussen de parameters in plaats van er voor. Operatoren die een constructor voorstellen in plaats van een gewone functie moeten met een dubbele punt beginnen. Voor de symmetrie (het oog wil ook wat) eindigen de operatoren in de gegeven data-declaratie ook op een dubbele punt. De drie symbolen in `:+:` vormen één operator, en moeten dan ook zonder spatie ertussen geschreven worden.

De constructor-operatoren uit de data-declaratie kunnen van een prioriteit en een associatievolgorde worden voorzien met behulp van een infix-declaratie. Deze werd ingevoerd in paragraaf 2.1.4. Het is het handigste om de operatoren van dezelfde prioriteit te voorzien als de gewone rekenkundige operatoren:

```
infixl 7 **:
infix 7 :/:
infixl 6 :+:, :-:
```

Na deze declaraties kan bijvoorbeeld de expressie $3x + 4y$ als datastructuur worden gerepresenteerd door de Helium-expressie

```
Con 3.0 **: Var "x" :+: Con 4.0 **: Var "y"
```

Het is belangrijk om onderscheid te maken tussen expressies in Helium, en rekenkundige expressies in het taaltje dat door de data-declaratie wordt beschreven. Zo is `Var "x"` een Helium-expressie die als waarde de datastructuur heeft, die de rekenkundige expressie x beschrijft.

5.1.2 Symbolisch differentiëren

Het voordeel van de representatie van expressies als datastructuren is dat we functies kunnen schrijven die op expressies werken, en het resultaat kunnen bekijken. Dit wordt *symbolische manipulatie* van expressies genoemd. Een goed voorbeeld van symbolische manipulatie is het *differentiëren*

van een expressie. Als we de expressie `Var "x" :* Var "x"` differentiëren, dan komt daar de expressie `Con 2.0 :* Var "x"` uit, of iets wat daar equivalent aan is.

Het symbolische differentiëren van een expressie biedt veel meer mogelijkheden dan het numeriek differentiëren, waarvoor in paragraaf 2.4.2 de volgende functie werd geschreven:

blz. 29

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

Een numeriek gedifferentieerde functie is immers een functie; het enige wat we met die functie kunnen doen is hem op een parameter toepassen. Het is niet mogelijk om het functievoorschrift van de numeriek gedifferentieerde functie te zien te krijgen. Bij het gebruik van expressie-bomen en symbolisch differentiëren is dat wèl mogelijk.

De symbolische differentieerfunctie heeft behalve een `Expr` als parameter die aangeeft naar welke variabele de expressie gedifferentieerd moet worden (dit is ook iets wat bij numeriek differentiëren niet mogelijk was). In de definitie wordt voor elk van de zes constructoren van expressies aangegeven hoe de expressie gedifferentieerd kan worden. Daarbij kunnen de bekende rekenregels voor differentiëren gevolgd worden: de afgeleide van een som is bijvoorbeeld de som van de afgeleiden, en voor de afgeleide van een product geldt de ‘productregel’ $((fg)' = fg' + gf')$.

```
afg :: Expr -> String -> Expr
afg (Con c) dx = Con 0.0
afg (Var x) dx
  | x==dx      = Con 1.0
  | otherwise  = Con 0.0
afg (f :+: g) dx = afg f dx :+: afg g dx
afg (f :-: g) dx = afg f dx :-: afg g dx
afg (f :* g) dx  = f :* afg g dx :+: g :* afg f dx
afg (f :/: g) dx = ( g :* afg f dx :-: f :* afg g dx )
                  :/: ( g :* g )
```

Uit de definitie blijkt verder dat de afgeleide van een constante de constante 0 is. De afgeleide van een variabele is de constante 1 als het de variabele betreft waarnaar gedifferentieerd wordt (in de definitie suggestief `dx` genoemd). Andere variabelen gedragen zich als constanten.

5.1.3 Andere expressiebomen

Naast de data-declaratie die rekenkundige expressies beschrijft, zijn ook andere soorten expressies te beschrijven met data-declaraties. Onderstaande data-declaratie beschrijft bijvoorbeeld Boolese expressies oftewel *proposities*:

```
data Prop = Cons Bool
          | Vari Char
          | Not Prop
          | Prop :/\: Prop
          | Prop :\/: Prop
          | Prop :->: Prop
```

De propositie $b \vee \neg b$ wordt bijvoorbeeld gerepresenteerd door de datastructuur

```
Vari 'b' :\/: Not (Vari 'b')
```

Functies op het type `Prop` worden natuurlijk weer geschreven door voor alle vijf constructoren een patroon te gebruiken. De functie `verw` bijvoorbeeld, verwijdert alle voorkomens van `:\/:` en `:->:` uit een propositie. Daarvoor worden rekenregels uit de propositielogica, zoals de wet van De Morgan toegepast.

```
verw (Cons b)      = Cons b
verw (Vari x)      = Vari x
verw (Not p)       = Not (verw p)
verw (p :/\: q)    = verw p :/\: verw q
verw (p :\/: q)    = Not (Not (verw p) :/\: Not (verw q))
verw (p :->: q)    = verw (Not p :\/: q)
```

Met data-declaraties kunnen naast expressies ook taalconstructies uit programmeertalen worden beschreven. Statements uit een Pascal-achtige taal kunnen bijvoorbeeld worden beschreven door de volgende data-declaratie:

```

data Stat = Assign Char Expr
         | If Prop Stat Stat
         | While Prop Stat
         | Repeat Stat Prop
         | Compound [Stat]

```

Door functies te schrijven die op dit soort datastructuren werken, is het mogelijk om Pascal-programma's (althans de boom-representaties daarvan) te transformeren volgens 'rekenregels' die daarvoor gelden.

5.1.4 Stringrepresentatie van een boom

We keren weer terug naar de expressiebomen uit paragraaf 5.1.1. De Helium-expressies die nodig zijn om een rekenkundige expressie als boom te representeren, lijken sterk op die expressies zelf. Zo wordt bijvoorbeeld de expressie $x*x+1$ gerepresenteerd door `Var "x":*:Var "x":+:Con 1.0`. Het is alleen jammer dat op elke constante eerst de constructor `Con` moet worden toegepast, op elke variabele `Var`, en dat elke operator van dubbele punten moet worden voorzien. Makkelijker zou het zijn als de expressie direct kan worden ingetikt. De simpelste manier om een datastructuur te maken waar Helium mee uit de voeten kan, is om de rekenkundige expressie in een string te zetten: `"x*x+1"`.

blz. 77

Het is handig om zo'n expressie-boom weer te kunnen geven als string. De functie `weerg` werkt op expressie-bomen, en wordt daarom met zes patronen gedefinieerd voor alle constructoren van `Expr`.

```

weerg      :: Expr -> String
weerg (Con n) = show n
weerg (Var x) = x
weerg (a :+ b) = "(" ++ weerg a ++ "+" ++ weerg b ++ ")"
weerg (a :- b) = "(" ++ weerg a ++ "-" ++ weerg b ++ ")"
weerg (a :* b) = weerg a ++ "*" ++ weerg b
weerg (a :/ b) = weerg a ++ "/" ++ weerg b

```

De functie `show` is een standaardfunctie die een stringrepresentatie geeft van onder andere `Float` waarden. De functie `weerg` wordt waar nodig recursief aangeroepen; de resulterende strings worden samengevoegd tot één lange string, waarin ook nog het symbool voor de betreffende operator wordt opgenomen. Bij het optellen en aftrekken worden er ook nog haakjes in het resultaat gezet, anders zou de boom-expressie `(Var "a":+:Var "b"):(*(Var "c":+:Var "d"))` worden omgezet in de string `"a+b*c+d"`, die de verkeerde interpretatie heeft.

5.2 Parser combinators

This chapter is an informal introduction to writing parsers in a lazy functional language using 'parser combinators'. We will start by motivating the definition of the type of parser functions. Using that type, we will be capable to build parsers for the language of ambiguous grammars. Next, we will introduce some elementary parsers that can be used for parsing the terminal symbols of a language.

In section 5.2.4 the first parser combinators are introduced, which can be used for sequentially and alternatively combining parsers. In section 5.2.5 some functions are defined, which make it possible to calculate a value during parsing. You may use these functions for what traditionally is called 'defining semantic functions': some useful meaning can be associated to syntactic structures. As an example, in section 5.2.6 we construct a parser for strings of matching parentheses, where different semantic values are calculated: a tree describing the structure, and an integer indicating the nesting depth.

In sections 5.2.7 and 5.2.8 we introduce some new parser combinators. Not only these will make life easier later, but also their definitions are nice examples of using parser combinators. A real application is given in section 5.2.9, where a parser for arithmetical expressions is developed. Next, the expression parser is generalized to expressions with an arbitrary number of precedence levels. This is done without coding the priorities of operators as integers, and we will avoid using indices and ellipses.

5.2.1 The type ‘Parser’

The *parsing problem* is: given a string, construct a tree that describes the structure of the string. In a functional language we can define a datatype `Tree`. A parser could be implemented by function of the following type:

```
type Parser = String -> Tree
```

For parsing substructures, a parser could call other parsers, or itself recursively. These calls need not only communicate their result, but also which part of the input string is left unprocessed. As this cannot be done using a global variable, the unprocessed input string has to be part of the result of the parser. The two results can be grouped in a tuple. A better definition for the type `Parser` is thus:

```
type Parser = String -> (String, Tree)
```

The type `String` is defined in the standard prelude as a list of characters. The type `Tree`, however, is not yet defined. The type of tree that is returned depends on the application. Therefore it is better to make the parser type into a polymorphic type, by parameterizing it with the type of the parse tree. Thus we abstract from the type of the parse tree at hand, substituting the type variable `a` for it:

```
type Parser a = String -> (String, a)
```

For example, a parser that returns a structure of type `Oak` now has type `Parser Oak`. For parse trees that represent an ‘expression’ we could define a type `Expr`, making it possible to develop parsers returning an expression: `Parser Expr`. Another instance of a parser is a parse function that recognizes a string of digits, and yields the number represented by it as a parse ‘tree’. In this case the function is of type `Parser Int`.

Until now, we have been assuming that every string can be parsed in exactly one way. In general, this need not be the case: it may be that a single string can be parsed in various ways, or that there is no possible way of parsing a string. As another refinement of the type definition, instead of returning one parse tree (and its associated rest string), we let a parser return a *list* of trees. Each element of the result consists of a tree, paired with the rest string that was left unprocessed while parsing it. The type definition of `Parser` therefore had better be:

```
type Parser a = String -> [(String,a)]
```

If there is just one parsing, the result of the parse function will be a singleton list. If no parsing is possible, the result will be an empty list. In the case of an ambiguous grammar, alternative parsings make up the elements of the result.

This method is called the *list of successes* method by Wadler. It can be used in situations where in other languages you would use backtracking techniques. If only one solution is required rather than all possible solutions, you can take the `head` of the list of successes. Thanks to lazy evaluation, not all elements of the list are determined if only the first value is needed, so there will be no loss of efficiency. Lazy evaluation provides a backtracking approach to finding the first solution.

Parsers with the type described so far operate on strings, that is lists of characters. There is however no reason for not allowing parsing strings of elements other than characters. You may imagine a situation in which a preprocessor prepares a list of tokens, which is subsequently parsed. To cater for this situation, as a final refinement of the parser type we again abstract from a type: that of the elements of the input string. Calling it `a`, and the result type `b`, the type of parsers is defined by:

```
type Parser a b = [a] -> [(a,b)]
```

or if you prefer meaningful identifiers over conciseness:

```
type Parser symbol result = [symbol] -> [(symbol,result)]
```

We will use this type definition in the rest of this article.

5.2.2 Elementary parsers

We will start quite simply, defining a parse function that just recognizes the symbol ‘a’. The type of the input string symbols is `Char` in this case, and as a parse ‘tree’ we also simply use a `Char`:

```
symbola :: Parser Char Char
symbola [] = []
```

```

symbol a (x:xs) | x=='a'    = [ (xs, 'a') ]
                  | otherwise = []

```

The list of successes method immediately pays off, because now we can return an empty list if no parsing is possible (because the input is empty, or does not start with an 'a').

In the same fashion, we can write parsers that recognize other symbols. As always, rather than defining a lot of closely related functions, it is better to abstract from the symbol to be recognized by making it an extra parameter of the function. Also, the function can operate on strings other than characters, so that it can be used in other applications than character oriented ones. The only prerequisite is that the symbols to be parsed can be tested for equality. Therefore, we pass an extra symbol testing function to the function `symbol`.

```

symbol :: (s->s->Bool) -> s -> Parser s s
symbol eq a []          = []
symbol eq a (x:xs)    | a'eq'x    = [ (xs,x) ]
                  | otherwise    = []

```

As usual, there are a number of ways to define the same function. If you like list comprehensions, you might prefer the following definition:

```

symbol eq a []          = []
symbol eq a (x:xs)    = [ (xs,a) | a'eq'x ]

```

In Helium, a list comprehension with no generators but only a condition is defined to be empty or singleton, depending on the condition.

The function `symbol` is a function that, given an equality tester and a symbol, yields a parser for that symbol. A parser in turn is a function, too. This is why not two but three parameters appear in the definition of `symbol`.

We will now define some elementary parsers that can do the work traditionally taken care of by lexical analyzers. For example, a useful parser is one that recognizes a fixed string of symbols, such as 'begin' or 'end'. We will call this function `token`.

```

token :: (s->s->Bool) -> [s] -> Parser s [s]
token eq k xs | eqList eq k (take n xs) = [ (drop n xs, k) ]
              | otherwise                = []
              where n = length k

```

As in the case of the `symbol` function we have parameterized this function with an equality tester, and the string to be recognized, effectively making it into a family of functions. Of course, this function is not confined to strings of characters. The function `token` is a generalization of the `symbol` function, in that it recognizes more than one character.

Another generalization of `symbol` is a function which may, depending on the input, return different parse results. The function `satisfy` is an example of this. Where the `symbol` function tests for equality to a given symbol, in `satisfy` an arbitrary predicate can be specified. Again, `satisfy` effectively is a family of parser functions. It is defined here using the list comprehension notation:

```

satisfy :: (s->Bool) -> Parser s s
satisfy p []          = []
satisfy p (x:xs)    = [ (xs,x) | p x ]

```

5.2.3 Trivial parsers

In books on grammar theory an empty string is often called 'epsilon'. In this tradition, we will define a function `epsilon` that 'parses' the empty string. It does not consume any input, and thus always returns an empty parse tree and unmodified input. A zero-tuple can be used as a result value: `()` is the only value of the type `()`.

```

epsilon :: Parser s ()
epsilon xs = [ ( xs, () ) ]

```

A variation is the function `succeed`, that neither consumes input, but does always return a given, fixed value (or 'parse tree', if you could call the result of processing zero symbols a parse tree...)

```

succeed :: r -> Parser s r
succeed v xs = [ (xs,v) ]

```

Of course, `epsilon` can be defined using `succeed`:

```

epsilon  :: Parser s ()
epsilon  = succeed ()

```

Dual to the function `succeed` is the function `fail`, that fails to recognize any symbol on the input string. It always returns an empty list of successes:

```

fail     :: Parser s r
fail xs  = []

```

We will need this trivial parser as a neutral element for `foldr` later. Note the difference with `epsilon`, which *does* have one element in its list of successes (albeit an empty one).

Do not confuse `fail` with `epsilon`: there is an important difference between returning one solution (which contains the unchanged input as ‘rest’ string) and not returning a solution at all!

5.2.4 Parser combinators

Using the elementary parsers from the previous section, parsers can be constructed for terminal symbols from a grammar. More interesting are parsers for *nonterminal* symbols. Of course, you could write these by hand, but it is more convenient to *construct* them by partially parameterizing higher-order functions.

Important operations on parsers are sequential and alternative composition. We will develop two functions for this, which for notational convenience are defined as operators: `<*>` for sequential composition, and `<|>` for alternative composition. Priorities of these operators are defined so as to minimize parentheses in practical situations:

```

infixr 6 <*>
infixr 4 <|>

```

Both operators have two parsers as parameter, and yield a parser as result. By again combining the result with other parsers, you may construct even more involved parsers.

In the definitions below, the functions operate on parsers `p1` and `p2`. Apart from the parameters `p1` and `p2`, the function operates on a string, which can be thought of as the string that is parsed by the parser that is the result of combining `p1` and `p2`.

To start, we will write the operator `<*>`. For sequential composition, first `p1` must be applied to the input. After that, `p2` is applied to the rest string part of the result. Because `p1` yields a *list* of solutions, we use a list comprehension in which `p2` is applied to all rest strings in the list:

```

(<*>)      :: Parser s a -> Parser s b -> Parser s (a,b)
(p1 <*> p2) xs = [ (xs2, (v1,v2))
                  | (xs1, v1) <- p1 xs
                    , (xs2, v2) <- p2 xs1
                  ]

```

The result of the function is a list of all possible tuples `(v1,v2)` with rest string `xs2`, where `v1` is the parse tree computed by `p1`, and where rest string `xs1` is used to let `p2` compute `v2` and `xs2`.

Apart from ‘sequential composition’ we need a parser combinator for representing ‘choice’. For this, we have the parser combinator operator `<|>`:

```

(<|>)      :: Parser s a -> Parser s a -> Parser s a
(p1 <|> p2) xs = p1 xs ++ p2 xs

```

Thanks to the list of successes method, both `p1` and `p2` yield lists of possible parsings. To obtain all possible successes of choice between `p1` and `p2`, we only need to concatenate these two lists.

The result of parser combinators is again a parser, which can be combined with other parsers. The resulting parse trees are intricate tuples which reflect the way in which the parsers were combined. Thus, the term ‘parse tree’ is really appropriate. For example, the parser `p` where

```

p = symbol 'a' <*> symbol 'b' <*> symbol 'c'

```

is of type `Parser Char (Char, (Char,Char))`.

Although the tuples clearly describe the structure of the parse tree, it is a problem that we cannot combine parsers in an arbitrary way. For example, it is impossible to alternatively compose the parser `p` above with `symbol 'a'`, because the latter is of type `Parser Char Char`, and only parsers of the same type can be composed alternatively. Even worse, it is not possible to recursively combine a parser with itself, as this would result in infinitely nested tuple types. What we need is

a way to alter the structure of the parse tree that a given parser returns.

5.2.5 Parser transformers

Apart from the operators `<*>` and `<|>`, that combine parsers, we can define some functions that modify or *transform* existing parsers. We will develop three of them: `sp` lets a given parser neglect initial spaces, `just` transforms a parser into one that insists on empty rest string, and `<@` applies a given function to the resulting parse trees.

The first parser transformer is `sp`. It drops spaces from the input, and then applies a given parser:

```
sp    :: Parser Char a -> Parser Char a
sp p = p . dropWhile isSpace
```

The second parser transformer is `just`. Given a parser `p` it yields a parser that does the same as `p`, but also guarantees that the rest string is empty. It does so by filtering the list of successes for null rest strings. Because the rest string is the first component of the list, the function can be defined as:

```
just   :: Parser s a -> Parser s a
just p = filter (null.fst) . p
```

The most important parser transformer is the one that transforms a parser into a parser which modifies its result value. We will define it as an operator `<@`, that applies a given function to the result parse trees of a given parser. We have chosen the symbol so that you might pronounce it as ‘apply’; the arrow points away from the function. Given a parser `p` and a function `f`, the operator `<@` returns a parser that does the same as `p`, but in addition applies `f` to the resulting parse tree. It is most easily defined using a list comprehension:

```
infixr 5 <@
(<@)   :: Parser s a -> (a->b) -> Parser s b
(p <@ f) xs = [ (ys, f v)
                | (ys, v) <- p xs
                ]
```

Using this operator, we can transform the parser that recognizes a digit character into one that delivers the result as an integer:

```
digit   :: Parser Char Int
digit   = satisfy isDigit <@ f
      where f c = ord c - ord '0'
```

In practice, the `<@` operator is used to build a certain value during parsing (in the case of parsing a computer program this value may be the generated code, or a list of all variables with their types, etc.). Put more generally: using `<@` we can add *semantic functions* to parsers.

While testing your self-made parsers, you can use `just` for discarding the parses which leave a non-empty rest string. But you might become bored of seeing the empty list as rest string in the results. Also, more often than not you may be interested in just *some* parsing rather than *all* possibilities.

As we have reserved the word ‘parser’ for a function that returns *all* parsings, accompanied with their rest string. Let’s therefore define a new type for a function that parses a text, guarantees empty rest string, picks the first solution, and delivers the parse tree only (discarding the rest string, because it is known to be empty at this stage). The functional program for converting a parser in such a ‘deterministic parser’ is more concise and readable than the description above:

```
type DetPars symbol result = [symbol] -> result
some   :: Parser s a -> DetPars s a
some p = snd . head . just p
```

Use the `some` function with care: this function assumes that there is at least one solution, so it fails when the resulting `DetPars` is applied to a text which contains a syntax error.

5.2.6 Matching parentheses

Using the parser combinators and transformers developed thus far, we can construct a parser that recognizes matching pairs of parentheses. A first attempt, that is not type correct however, is:

```

parens  :: Parser Char ???
parens  = (  symbol '('
            <*> parens
            <*> symbol ')'
            <*> parens
          )
        <|> epsilon

```

This definition is inspired strongly by the well known grammar for nested parentheses. The type of the parse tree, however, is a problem. If this type would be `a`, then the type of the composition of the four subtrees in the first alternative would be `(Char, (a, (Char, a)))`, which is not the same or unifiable. Also, the second alternative (`epsilon`) must yield a parse tree of the same type. Therefore we need to define a type for the parse tree first, and use the operator `<@` in both alternatives to construct a tree of the correct type. The type of the parse tree can be for example:

```

data Tree = Nil
          | Bin (Tree,Tree)

```

Now we can add ‘semantic functions’ to the parser:

```

parens  :: Parser Char Tree
parens  = (  symbol '('
            <*> parens
            <*> symbol ')'
            <*> parens
          )   <@ ( \(_, (x, (_, y))) -> Bin(x,y) )
        <|> epsilon <@ const Nil

```

The rather obscure text `\(_, (x, (_, y)))` is a lambda pattern describing a function with as first parameter a tuple containing the four parts of the first alternative, of which only the second and fourth matter.

In the lambda pattern, underscores are used as placeholders for the parse trees of `symbol '('` and `symbol ')'`, which are not needed in the result. In order to not having to use these complicated tuples, it might be easier to discard the parse trees for symbols in an earlier stage. For this, we introduce two auxiliary parser combinators, which will prove useful in more situations. These operators behave the same as `<*>`, except that they discard the result of one of their two parser arguments:

```

(<*>)  :: Parser s a -> Parser s b -> Parser s a
p <*> q = p <*> q <@ fst
(*>)   :: Parser s a -> Parser s b -> Parser s b
p *> q = p <*> q <@ snd

```

We can use these new parser combinators for improving the readability of the parser `parens`:

```

open    = symbol '('
close   = symbol ')'
parens  :: Parser Char Tree
parens  = (open *> parens <*> close) <*> parens <@ Bin
        <|> succeed Nil

```

By judiciously choosing the priorities of the operators involved:

```

infixr 6 <*> , <*> , *>
infixl 5 <@
infixr 4 <|>

```

we minimize on the number of parentheses needed.

By varying the function used after `<@` (the ‘semantic function’), we can yield other things than parse trees. As an example we write a parser that calculates the nesting depth of nested parentheses:

```

nesting  :: Parser Char Int
nesting  = (open *> nesting <*> close) <*> nesting <@ f
        <|> succeed 0
      where f (x,y) = (1+x) ‘max’ y

```

If more variations are of interest, it may be worthwhile to make the semantic function and the value to yield in the ‘empty’ case into two additional parameters. The higher order function `foldparens` parses nested parentheses, using the given function and constant respectively, after parsing one of the two alternatives:

```

foldparens :: ((a,a)->a) -> a -> Parser Char a
foldparens f e = p
    where p = (open *> p <* close) <*> p <@ f
            <|> succeed e

```

A session in which `nesting` is used may look like this:

```

? just nesting "()()()"
[(2,[])]
? just nesting "()"
[]

```

Indeed `nesting` only recognizes correctly formed nested parentheses, and calculates the nesting depth on the fly.

5.2.7 More parser combinators

Although in principle you can build parsers for any context-free language using the combinators `<*>` and `<|>`, in practice it is easier to have some more parser combinators available. In traditional grammar formalisms, too, additional symbols are used to describe for example optional or repeated constructions. Consider for example the BNF formalism, in which originally only sequential and alternative composition could be used (denoted by juxtaposition and vertical bars, respectively), but which was later extended to also allow for repetition, denoted by asterisks.

It is very easy to make new parser combinators for extensions like that. As a first example we consider repetition. Given a parser for a construction, `many` makes a parser for zero or more occurrences of that construction:

```

many :: Parser s a -> Parser s [a]
many p = p <*> many p <@ list
        <|> succeed []

```

The auxiliary function `list` is defined as the uncurried version of the list constructor:

```
list (x,xs) = x:xs
```

The recursive definition of the parser follows the recursive structure of lists. Perhaps even nicer is the version in which the `epsilon` parser is used instead of `succeed`:

```

many :: Parser s a -> Parser s [a]
many p = p <*> many p <@ (\(x,xs)->x:xs)
        <|> epsilon <@ (\_ ->[] )

```

The order in which the alternatives are given only influences the order in which solutions are placed in the list of successes.

An example in which the `many` combinator can be used is parsing of a natural number:

```

natural :: Parser Char Int
natural = many digit <@ foldl f 0
        where f a b = a*10 + b

```

Defined in this way, the `natural` parser also accepts empty input as a number. If this is not desired, we'd better use the `many1` parser combinator, which accepts one or more occurrences of a construction.

Another combinator that you may know from other formalisms is the `option` combinator. The constructed parser generates a list with zero or one element, depending on whether the construction was recognized or not.

```

option :: Parser s a -> Parser s [a]
option p = p <@ (\x->[x])
         <|> epsilon <@ (\x->[] )

```

For aesthetic reasons we used `epsilon` in this definition; another way to write the second alternative is `succeed []`.

The combinators `many`, `many1` and `option` are classical in compiler constructions, but there is no need to leave it at that. For example, in many languages constructions are frequently enclosed between two meaningless symbols, most often some sort of parentheses. For this we design a parser combinator `pack`. Given a parser for an opening token, a body, and a closing token, it constructs a parser for the enclosed body:

```

pack  :: Parser s a -> Parser s b -> Parser s c -> Parser s b
pack s1 p s2 = s1 *> p <*> s2

```

Special cases of this combinator are:

```

parenthesized p = pack (symbol '(') p (symbol ')')
bracketed p     = pack (symbol '[') p (symbol ']')
compound p     = pack (token "begin") p (token "end")

```

Another frequently occurring construction is repetition of a certain construction, where the elements are separated by some symbol. You may think of lists of parameters (expressions separated by commas), or compound statements (statements separated by semicolons). For the parse trees, the separators are of no importance. The function `listOf` below generates a parser for a (possibly empty) list, given a parser for the items and a parser for the separators:

```

listOf      :: Parser s a -> Parser s b -> Parser s [a]
listOf p s  = p <:*> many (s *> p) <|> succeed []

```

Useful instantiations are:

```

commaList, semicList :: Parser Char a -> Parser Char [a]
commaList p = listOf p (symbol ',')
semicList p = listOf p (symbol ';')

```

A somewhat more complicated variant of the function `listOf` is the case where the separators carry a meaning themselves. For example, an arithmetical expressions, where the operators that separate the subexpressions have to be part of the parse tree. For this case we will develop the functions `chainr` and `chainl`. These functions expect that the parser for the separators yields a function (!); that function is used by `chain` to combine parse trees for the items. In the case of `chainr` the operator is applied right-to-left, in the case of `chainl` it is applied left-to-right. The basic structure of `chainl` is the same as that of `listOf`. But where the function `listOf` discards the separators using the operator `*>`, we will keep it in the result now using `<*>`. Furthermore, postprocessing is more difficult now than just applying `list`.

```

chainl      :: Parser s a -> Parser s (a->a->a) -> Parser s a
chainl p s  = p <*> many (s <*> p) <@ f

```

The function `f` should operate on an element and a list of tuples, each containing an operator and an element. For example, $f(e_0, [(\oplus_1, e_1), (\oplus_2, e_2), (\oplus_3, e_3)])$ should return $((e_0 \oplus_1 e_1) \oplus_2 e_2) \oplus_3 e_3$. You may recognize a version of `foldl` in this (albeit an uncurried one), where a tuple (\oplus, y) from the list and intermediate result x are combined applying $x \oplus y$. If we define

```

ap2 (op,y) x = x 'op' y

```

or even

```

ap2 (op,y) = ('op' y)

```

then we may define

```

chainl      :: Parser s a -> Parser s (a->a->a) -> Parser s a
chainl p s  = p <*> many (s <*> p)
              <@ uncurry (foldl (flip ap2))

```

Dual to this function is `chainr`, which applies the operators associating to the right.

5.2.8 Analyzing options

The `option` function constructs a parser which yields a list of elements: an empty list if the optional construct was not recognized, and a singleton list if it was present. Postprocessing functions may therefore safely assume that the list consists of zero or one element, and will in practice do a case analysis. You will therefore often need constructions like:

```

option p <@ f
  where f [] = a
        f [x] = b x

```

As this necessitates a new function name for every optional symbol in our grammar, we had better provide a higher order function for this situation. We will define a special version `<?@` of the `<@` operator, which provides a semantics for both the case that the optional construct was present and that it was not. The right argument of `<?@` consists of two parts: a constant to be used in absence,

and a function to be used in presence of the optional construct. The new transformer is defined by:

```
p <?@ (no,yes) = p <@ f
               where f [] = no
                   f [x] = yes x
```

For a practical use of this, let's extend the parser for natural numbers to floating point numbers:

```
natural :: Parser Char Int
natural = many digit <@ foldl f 0
        where f n d = n*10 + d
```

The fractional part of a floating point number is parsed by:

```
fract    :: Parser Char Float
fract    = many digit <@ foldr f 0.0
        where f d x = (x + fromInteger d)/10.0
```

But the fractional part is optional in a floating point number.

```
fixed    :: Parser Char Float
fixed    = (integer <@ fromInteger)
          <*>
          (option (symbol '.' *> fract) <?@ (0.0,id))
          <@ uncurry (+)
```

The decimal point is for separation only, and therefore immediately discarded by the operator `*>`. The decimal point and the fractional part together are optional. In their absence, the number `0.0` should be used, in their presence, the identity function should be applied to the fractional part. Finally, integer and fractional part are added.

In the solution of exercise 16 you will find a nice construct, in which the first construct parsed yields a function which is subsequently applied to the second construct parsed. We can use that for yet another refinement of the `chainr` function. It was defined in the previous section using the `many` function. The parser yields a list of tuples (operator,element), which immediately afterwards is destroyed by `foldr`. Why bothering building the list, then, anyway? We can apply the function that is folded with directly during parsing, without first building a list. For this, we need to substitute the body of `many` in the definition of `chainr`. We can further abbreviate the phrase `p <|> epsilon` by `option p`. By directly applying the function that was previously used during `foldr` we obtain:

```
chainr' p s = q
            where q = p <*> (option (s <*> q) <?@ (id,ap2) )
                   <@ flip ap
```

By the use of the `option` and `many` functions, a large amount of backtracking possibilities are introduced. This is not always advantageous. For example, if we define a parser for identifiers by

```
identifier = many1 (satisfy isAlpha)
```

a single word may also be parsed as two identifiers. Caused by the order of the alternatives in the definition of `many`, the 'greedy' parsing, which accumulates as many letters as possible in the identifier is tried first, but if parsing fails elsewhere in the sentence, also less greedy parsings of the identifier are tried – in vain.

In situations where from the way the grammar is built we can predict that it is hopeless to try non-greedy successes of `many`. We can define a parser transformer `first`, that transforms a parser into a parser that only yields the first possibility. It does so by taking the first element of the list of successes.

```
first :: Parser a b -> Parser a b
first p xs | null r = []
          | otherwise = [head r]
          where r = p xs
```

Using this function, we can create a special 'take all or nothing' version of `many`:

```
greedy = first . many
greedy1 = first . many1
```

If we compose the `first` function with the `option` parser combinator:

```
compulsion = first . option
```

we get a parser which must accept a construction if it is present, but which does not fail if it is not present.

5.2.9 Arithmetical expressions

In this section we will use the parser combinators in a concrete application. We will develop a parser for arithmetical expressions, of which parse trees are of type `Expr`:

```
data Expr = Con Int
          | Var String
          | Fun String [Expr]
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :* Expr
          | Expr :/: Expr
```

In order to account for the priorities of the operators, we will use a grammar with non-terminals ‘expression’, ‘term’ and ‘factor’: an expression is composed of terms separated by + or −; a term is composed of factors separated by * or /, and a factor is a constant, variable, function call, or expression between parentheses.

This grammar is represented in the functions below:

```
fact :: Parser Char Expr
fact = integer <@ Con
      <|> identifier
      <*> ( option (parenthesized (commaList expr))
            <?@ (Var,flip Fun))
          <@ ap'
      <|> parenthesized expr
```

The first alternative is a constant, which is fed into the ‘semantic function’ `Var`. The second alternative is a variable or function call, depending on the presence of a parameterlist. In absence of the latter, the function `Var` is applied, in presence the function `Fun`. For the third alternative there is no semantic function, because the meaning of an expression between parentheses is the same as that of the expression without parentheses.

For the definition of a term as a list of factors separated by multiplicative operators we will use the function `chainr`:

```
term :: Parser Char Expr
term = chainr fact
      ( symbol '*' <@ const (:*)
      <|> symbol '/' <@ const (:/)
      )
```

Recall that `chainr` repeatedly recognizes its first parameter (`fact`), separated by its second parameter (an * or /). The parse trees for the individual factors are joined by the constructor functions mentioned after `<@`.

The function `expr` is analogous to `term`, only with additive operators instead of multiplicative operators, and with `terms` instead of `factors`:

```
expr :: Parser Char Expr
expr = chainr term
      ( symbol '+' <@ const (:+)
      <|> symbol '-' <@ const (:-)
      )
```

From this example the strength of the method becomes clear. There is no need for a separate formalism for grammars; the production rules of the grammar are joined using higher-order functions. Also, there is no need for a separate parser generator (like ‘yacc’); the functions can be viewed both as description of the grammar and as an executable parser.

5.2.10 Generalized expressions

Arithmetical expressions in which operators have more than two levels of priority can be parsed by writing more auxiliary functions between `term` and `expr`. The function `chainr` is used in each definition, with as first parameter the function of one priority level lower.

If there are nine levels of priority, we obtain nine copies of almost the same text. This would not be as it should be. Functions that resemble each other are an indication that we should write a generalized function, where the differences are described using extra parameters. Therefore, let's inspect the differences in the definitions of `term` and `expr` again. These are:

- The operators and associated tree constructors that are used in the second parameter of `chainr`
- The parser that is used as first parameter of `chainr`

The generalized function will take these two differences as extra parameters: the first in the form of a list of pairs, the second in the form of a parse function:

```
type Op a = (Char, a->a->a)
gen      :: [Op a] -> Parser Char a -> Parser Char a
gen ops p = chainr p (choice (map f ops))
  where f (s,c) = symbol s <@ const c
```

If furthermore we define as shorthand:

```
multis = [ ('*',(:*)), ('/',(:/)) ]
addis  = [ ('+',(:+)), ('-',(:-)) ]
```

then `expr` and `term` can be defined as partial parametrizations of `gen`:

```
expr = gen addis term
term = gen multis fact
```

By expanding the definition of `term` in that of `expr` we obtain:

```
expr = addis 'gen' (multis 'gen' fact)
```

which an experienced functional programmer immediately recognizes as an application of `foldr`:

```
expr = foldr gen fact [addis, multis]
```

From this definition a generalization to more levels of priority is simply a matter of extending the list of operator-lists.

The very compact formulation of the parser for expressions with an arbitrary number of priority levels was possible because the parser combinators could be used in conjunction with the existing mechanisms for generalization and partial parametrization in the functional language.

Contrary to conventional approaches, the levels of priority need not be coded explicitly with integers. The only thing that matters is the relative position of an operator in the list of 'list with operators of the same priority'. Also, the insertion of new priority levels is very easy.

Opgaven

- 5.1 Breid de data-declaratie van `Expr` uit zodat er vier nieuwe expressievormen ontstaan: de sinus van een expressie, de cosinus van een expressie, de exponentiële functie (e tot de macht een expressie), en de natuurlijke logaritme van een expressie. Breid vervolgens de definitie van `afg` uit voor deze vier nieuwe expressievormen. Verwerk daarin de 'kettingregel' voor het differentiëren: $(f \circ g)' = (f' \circ g) * g'$.
- 5.2 Schrijf een functie `norep` die gegeven een `Stat` alle `Repeat`-statements daaruit verwijdert, door ze te vervangen door een equivalente `While`-statements.
- 5.3 Since `satisfy` is a generalization of `symbol`, the function `symbol` could have been defined as an instance of `satisfy`. How can this be done?
- 5.4 When defining the priority of the `<|>` operator, using the `infixr` keyword we also specified that the operator associates to the right. Why is this a better choice than association to the left?
- 5.5 What would happen if we omit the `just` transformer in the `nesting` examples?
- 5.6 Define the function `just` using a list comprehension instead of the `filter` function.

-
- 5.7 In the function `parens`, why don't we use a four-tuple in the lambda pattern instead of a tuple with as second element a tuple with as second element a tuple?
- 5.8 The parentheses around `open*>parens<*close` in the first alternative, are required in spite of our clever priorities. What would happen if we left them out?
- 5.9 The function `foldparens` is a generalization of `parens` and `nesting`. Write the latter two as an instantiation of the former.
- 5.10 To obtain symmetry in the `many` function, we could also try and avoid the `<@` operator in both alternatives. Earlier we defined the operator `<*` as an abbreviation of applying `<@ fst` to the result of `<*>`. In the function `many`, also the result of `<*>` is postprocessed. Define an utility function `<:*>` for this case, and use it to simplify the definition of `many` even more.
- 5.11 Consider application of the parser `many (symbol 'a')` to the string "aaa". In what order do the four possible parsings appear in the list of successes?
- 5.12 Define the `many1` parser combinator.
- 5.13 As another variation on the theme 'repetition', define a parser `sequence` combinator that transforms a *list of parsers* for some type into a *parser yielding a list* of elements of that type. Also define a combinator `choice` that iterates the operator `<|>`.
- 5.14 As an application of `sequence`, define the function `token` that was discussed in section 5.2.2.
- 5.15 Try to define `chainr`. The definition is beautifully symmetric to `chainl`, but you only experience the beauty when you discover it yourself. . .
- 5.16 Define a parser for a (possibly negative) integer number, which consists of an optional minus sign followed by a natural number.
- 5.17 Let the parser for floating point numbers recognize an optional exponent.

Bijlage A

Lisp voor Helium-kenners

A.1 Expressies

A.1.1 Functie-aanroep

In Lisp kun je, net als in Helium, een functie aanroepen door de naam van de functie en de parameters naast elkaar te schrijven. In Lisp moet echter het geheel nog tussen haakjes geschreven worden.

Het resultaat van een functie-aanroep kan gebruikt worden als parameter van een andere functie. In Helium moet de deel-aanroep dan tussen haakjes gezet worden om aan te geven dat het één parameter betreft; in Lisp staat de deel-aanroep tussen haakjes omdat elke aanroep tussen haakjes moet staan.

Al met al lijken Lisp-expressies sterk op Helium-expressies. Het enige verschil is dat er in Lisp ook om de buitenste aanroep in een expressie haakjes moeten staan.

Helium	Lisp
<code>sqrt 2.0</code> <code>sqrt (exp 1.0)</code>	<code>(sqrt 2.0)</code> <code>(sqrt (exp 1.0))</code>

A.1.2 Operatoren

In Helium zijn er functies en operatoren. De naam van een functie begint met een letter, de naam van een operator bestaat uit symbolen. Functie-namen worden *voor* de parameters geschreven, operatoren er *tussen* (operatoren hebben altijd twee parameters).

In Lisp worden functies en operatoren altijd *voor* de parameter geschreven. Afgezien van de schrijfwijze (letters, resp. symbolen) is er dus geen verschil tussen functies en operatoren.

Prioriteit en associatievolgorde zijn eigenschappen van infix-operatoren. Dit is in Lisp dus geen issue; de berekeningsvolgorde wordt altijd expliciet aangegeven.

Functies kunnen in Lisp een variabel aantal parameters hebben. De functie `+` is daarvan een voorbeeld.

Helium	Lisp
<code>f 1 2</code>	<code>(f 1 2)</code>
<code>1 + 2</code>	<code>(+ 1 2)</code>
<code>x <= y</code>	<code>(<= x y)</code>
<code>x+y*z</code>	<code>(+ x (* y z))</code>
<code>x*(y+z)</code>	<code>(* x (+ y z))</code>
<code>v+w+x+y+z</code>	<code>(+ v w x y z)</code>

A.1.3 Lijsten

In Helium kun je lijsten opschrijven door de elementen, gescheiden door komma's, tussen vierkante haken te zetten. In Lisp wordt voor lijsten dezelfde notatie gebruikt als voor functie-aanroep: de elementen staan naast elkaar, gescheiden door spaties, en het geheel staat tussen ronde haken.

Om te voorkomen dat het eerste element van de lijst wordt opgevat als functie, die moet worden uitgerekend, wordt het geheel voorafgegaan door het symbool ' (spreek uit: quote).

De lege lijst wordt in Lisp aanguid met de naam `nil`, of desgewenst met een leeg, rond haakjespaar. Net als Helium kent Lisp strings, die mogen worden gebruikt als lijst van characters.

Helium	Lisp
<code>[1,2,3]</code>	<code>'(1 2 3)</code>
<code>[]</code>	<code>nil</code>
<code>[]</code>	<code>()</code>
<code>length [1,2,3]</code>	<code>(length '(1 2 3))</code>
<code>length "aap"</code>	<code>(length "aap")</code>

A.2 Functies op lijsten

Zowel in Helium als in Lisp kan een lijst ook worden opgebouwd met de op-kop-van functie. In Helium heet die `:`, in Lisp is dat `cons`. Lijsten kunnen geconcateneerd worden met `++` in Helium, en met `append` in Lisp.

Helium	Lisp
<code>1:2:3:4: []</code>	<code>(cons 1 (cons 2 (cons 3 (cons 4 ()))))</code>
<code>[1,2,3] ++ [4,5,6]</code>	<code>(append '(1 2 3) '(4 5 6))</code>

A.3 Functiedefinitie

A.3.1 Simpele definitie

Ingrediënten van een functiedefinitie zijn de naam van de functie, declaratie van parameters, en een body (expressie waarin de parameters gebruikt mogen worden).

In Helium is er een speciale syntax voor functiedefinities, te herkennen aan het `=` symbool in het midden. Links daarvan staan de naam en namen voor de parameters, rechts staat de body.

In Lisp heeft een functiedefinitie dezelfde vorm als een functie-aanroep: ronde haakjes met daartussen een aantal onderdelen, gescheiden door spaties. De onderdelen zijn achtereenvolgens: het speciale symbool `defun`, de naam van de functie, een lijst met parameternamen, en de body van de functie.

Helium	Lisp
<code>kwadraat x = x*x</code>	<code>(defun kwadraat (x) (* x x))</code>
<code>boven n k = fac n</code>	<code>(defun boven (n k)</code>
<code> / (fac k * fac (n-k))</code>	<code> (/ (fac n</code>
	<code> (* (fac k) (fac (- n k))))</code>

A.3.2 Definitie met gevalsonderscheid

Helium heeft een aparte syntax voor een functiedefinitie met gevalsonderscheid: voor het `=`-teken in een functiedefinitie mag een `|` staan gevolgd door een voorwaarde. In een functiedefinitie kunnen meerdere voorwaarden en bijbehorende functie-bodies staan.

In Lisp worden de gevallen *in* de body van de functie uitgesplitst, door aanroep van de speciale functie `cond`. Deze functie heeft een aantal paren voorwaarde-body als parameter.

Helium	Lisp
<code>signum x x<0 = -1</code>	<code>(defun signum (x)</code>
<code> x==0 = 0</code>	<code> (cond ((< x 0) -1)</code>
<code> x>0 = 1</code>	<code> ((= x 0) 0)</code>
	<code> (> x 0) 1)))</code>

Helium	Lisp
<pre>> 1<='a' Error: Type error in application > 2<=3 1<='a' Error: Type error in application</pre>	<pre>> (<= 1 'a) Error: A is not of type FLOAT > (or (<= 2 3) (<= 1 'a)) T</pre>

A.4.2 Types van lijsten

In Helium moeten alle elementen van een lijst van hetzelfde type zijn. In Lisp hoeft dat niet. Een apart tuple-type voor een vast aantal elementen van verschillend type is in Lisp dus ook niet nodig.

In Helium zijn lijsten van lijsten mogelijk, mits dan ook alle elementen een lijst zijn. In Lisp kunnen sommige elementen van een lijst atomair zijn, en andere elementen een lijst.

Helium	Lisp
<pre>[1,2,3] (1,"aap") [[1,2,3], [4,5,6]] [5, [4,5,6]] -- typeringsfout</pre>	<pre>'(1 2 3) '(1 "aap") '(' (1 2 3) '(4 5 6)) '(5 '(4 5 6))</pre>

A.4.3 Overloading

In Helium's grote broer Haskell kan een functie gedefinieerd worden op meerdere types. Zo'n functie moet dan een member zijn van een class, waarna de verschillende types instance worden gemaakt van die class. door de compiler wordt de juiste versie van de functie gekozen aan de hand van het type.

In Lisp kun je run-time het type van de parameter opvragen, en op die manier een functie op meerdere types toepasbaar maken. De juiste versie wordt met gevalsonderscheid expliciet gekozen.

Haskell	Lisp
<pre>class Seq a where nummer :: a -> Int instance Seq Int where nummer n = n instance Seq Char where nummer c = ord c instance Seq Bool where nummer False = 0 nummer True = 1</pre>	<pre>(defun nummer (x) (cond ((integerp x) x) ((characterp x) (ord x)) (x 1) (t 0)))</pre>

In Helium kun je reeds bestaande overloaded operatoren (zoals ==) nog verder overladen, door een nieuw type ook instance te maken van de betreffende klasse (Eq in het geval van ==). In Lisp is dat onmogelijk.

A.4.4 Polymorfie

Helium is getypeerd volgens het Hindley-Milner typesysteem, waarin bijvoorbeeld onderscheid wordt gemaakt tussen lijsten-van-integer en lijsten-van-lijsten-van-boolean, en tussen functie-van-int-naar-char en functie-van-int-naar-bool. Door middel van type-variabelen is het mogelijk om aan te geven dat een functie bijvoorbeeld het type functie-van-willekeurig-type-naar-datzelfde-type heeft.

In Lisp zijn er types voor getallen (met subtypes voor integer, float, rational etc.), lijsten, en functies. Er kan geen onderscheid gemaakt worden tussen verschillende typen van functies.

A.5 Hogere-ordefuncties

A.5.1 Map / Mapcar

Zowel in Helium als in Lisp is het mogelijk om een functie mee te geven als parameter aan een andere functie. Een bekend voorbeeld is de functie `map`, die in Lisp bekend staat onder de naam `mapcar`. In Lisp moet expliciet worden aangegeven dat de functie die als parameter wordt meegegeven moet worden opgezocht in de verzameling van alle eerder gegeven functie-definities. Daarom staat er in Lisp `#'f` in plaats van `f` als parameter van `map`.

In Lisp heeft de functie `map` een variabel aantal parameters. In plaats van op één lijst kan hij ook op twee lijsten worden toegepast. De functie-parameter moet dan een functie met twee parameters zijn; het effect is hetzelfde als dat van `zipWith` in Helium. Ook generalisaties naar meer lijsten zijn mogelijk.

Helium	Lisp
<code>map f [1,2,3]</code>	<code>(mapcar #'f '(1 2 3))</code>
<code>zipWith (+) [1,2,3] [4,5,6]</code>	<code>(mapcar #'(lambda (x y) (+ x y)) '(1 2 3) '(4 5 6))</code>

A.5.2 Foldr / Reduce

De functie overeenkomstig met Helium's `foldr` heet in Lisp `reduce`. Een neutrale waarde hoeft om mij onduidelijke redenen niet te worden meegegeven; toch kan `reduce` op lege lijsten worden toegepast.

Helium	Lisp
<code>foldr (+) 0 [1,2,3]</code>	<code>(reduce #'(lambda (x y) (+ x y)) '(1 2 3))</code>

A.5.3 Currying / Lambda-notatie

In Lisp zijn functies niet gecurried, en ze kunnen dus ook niet partieel geparametriseerd worden. Het effect kan wel worden bereikt met behulp van de lambda-notatie. Hiermee worden nieuwe, naamloze functies gemaakt, meestal met het doel om ze aan een andere functies mee te geven. De lambda-notatie bestaat ook in Helium, maar is minder vaak nodig omdat in plaats daarvan partiële parametrisatie kan worden gebruikt.

Helium	Lisp
<code>\x -> x*x</code>	<code>(lambda (x) (* x x))</code>
<code>map (\x->x*x) [1,2,3]</code>	<code>(mapcar #'(lambda (x) (* x x)) '(1 2 3))</code>
<code>map (+1) [1,2,3]</code>	<code>(mapcar #'(lambda (x) (+ x 1)) '(1 2 3))</code>
<code>until (>9) (*2) 1</code>	<code>(until #'(lambda (x) (> x 9)) #'(lambda (x) (* x 2)) 1)</code>

A.6 Filosofie

A.6.1 Helium: referentieel transparant

Het resultaat van een functie-aanroep in Helium is geheel bepaald door de waarden van de parameters. Het functieresultaat is de enige manier waarop een functie zijn omgeving kan beïnvloeden. Helium-functies hebben dus geen *neveneffecten*; er zijn geen globale variabelen die veranderd kunnen worden.

Deze eigenschap maakt lazy evaluatie mogelijk: zonder dat het voor het eindresultaat uitmaakt, kan een berekening later, of helemaal niet worden uitgevoerd. Als deze berekeningen neveneffecten gehad zouden hebben, dan kan de uitreken-volgorde het eindresultaat wèl beïnvloeden.

De afwezigheid van neveneffecten maakt het mogelijk om over Helium-programma's te redeneren als waren het wiskundige formules; zo zijn bijvoorbeeld in alle omstandigheden de aanroepen

`map f (xs++ys)` en `map f xs ++ map f ys` gelijk. Als de functie `f` neveneffecten op globale variabelen zou hebben, kun je niet zonder meer op de geldigheid van deze wet vertrouwen.

In Lisp is het wel mogelijk een waarde toe te kennen aan een globale variabele: door aanroep van de functie `setf` wordt een waarde aan een variabele toegekend.

A.6.2 Lisp: meta-circulair

De syntax van Lisp-programma's (functie-aanroepen zoals `(f 1 2)` lijkt sterk op die van Lisp-data (lijsten zoals `'(1 2 3)`). De Lisp-interpreter is als het ware een functie die een lijst, voorstellende een functie-aanroep, interpreteert. Dit maakt het eenvoudig om Lisp-programma's te schrijven die andere programma's manipuleren. Het is zelfs mogelijk om een programma door een programma te laten schrijven, en dat vervolgens uit te voeren.

Dat laatste effect zie je in Helium meestal in de vorm van hogere-ordefuncties. Functies kunnen functies opleveren, die pas later op werkelijke data worden toegepast.

A.7 Helium en Prolog

A.7.1 Lijsten

De notatie van lijsten is enigzins verwarrend als je afwisselend in Helium en Prolog programmeert. Een opsomming gebeurt op dezelfde manier: komma's tussen de elementen en vierkante haken om het geheel. Singletons en de lege lijst worden op dezelfde manier gerepresenteerd. Maar het patroon waarbij een lijst in een kop en een staart wordt verdeeld is verschillend: in Helium wordt de operator `:` gebruikt, in Prolog wordt het symbool `|` tussen kop en staart gezet, maar *bovendien* vierkante haken om het geheel. In Helium moet je die extra vierkante haken vooral *niet* neerzetten: daarmee zou je een lijstnivo toevoegen!

Helium	Prolog
<code>[1,2,3]</code>	<code>[1,2,3]</code>
<code>[5]</code>	<code>[5]</code>
<code>[]</code>	<code>[]</code>
<code>(x:xs)</code>	<code>[x xs]</code>
<code>[x:xs]</code>	<code>[[x xs]]</code>

A.7.2 Functies en relaties

Het belangrijkste verschil tussen Helium en Prolog is dat je in Helium *functies* specificeert, en in Prolog *relaties*. Bij een functie geef je aan wat bij bepaalde invoer de uitvoer is; bij een relatie geef je aan dat invoer en uitvoer met elkaar een relationeel verband hebben. Daarom hebben Prolog-relaties altijd een parameter meer dan de overeenkomstige Helium-functies.

Daar waar je in Helium een aanroep doet van een andere functie, krijgt de Prolog-relatie *voorwaarden*. Je zult merken dat je een extra naam nodig hebt, om het tussenresultaat te kunnen benoemen.

Helium	Prolog
<code>d x = 3</code>	<code>d(x,3) .</code>
<code>f x = h (g x)</code>	<code>f(x,z) :- g(x,y), h(y,z) .</code>

A.7.3 Patronen

Gelukkig kent Prolog wel patronen, waarmee bijvoorbeeld de gevallen 'lege lijst' en 'niet-lege lijst' onderscheiden kunnen worden. Zo kan een relatie worden gedefinieerd waarmee wordt aangeduid dat twee lijsten aan elkaar geplakt een derde lijst opleveren. In Prolog heet deze relatie `append`; de overeenkomstige functie in Helium heet `++`.

Helium	Prolog
<code>[] ++ ys = ys</code>	<code>append([],ys,ys) .</code>
<code>(x:xs) ++ ys = x:(xs++ys)</code>	<code>append([x xs],ys,[x zs]) :- append(xs,ys,zs) .</code>

Merk op dat het resultaat van de recursieve aanroep in de Prolog-versie een aparte naam moet krijgen. Bij wat meer ingewikkelde functies/relaties kan dat tamelijk onoverzichtelijk worden:

Helium	Prolog
<code>ins x [] = [x]</code>	<code>ins(x,[],[x]) .</code>
<code>ins x (y:ys)</code>	<code>ins(x,[y ys],[x y ys])</code>
<code> x<y = x:y:ys</code>	<code>:- x<y .</code>
<code> x>=y = y:ins x ys</code>	<code>ins(x,[y ys],[y zs])</code>
	<code>:- x>=y, ins(x,ys,zs) .</code>

A.7.4 Richtingloze definities

Een voordeel van de Prolog-definities is dat ze ook omgekeerd gebruikt kunnen worden. De `append`-relatie kan behalve voor het samenvoegen van twee lijsten ook worden gebruikt om een lijst in alle mogelijke delen te splitsen. In Helium zou hiervoor een andere functie nodig zijn.

Helium	Prolog
<pre>> [1,2]++[3,4] [1,2,3,4] splits [] = [([],[])] splits (x:xs) = ([],(x:xs)) : map f (splits xs) where f (a,b) = (x:a,b) > splits [1,2,3] [[[], [1,2,3]] , ([1], [2,3]) , ([1,2], [3]) , ([1,2,3], [])]</pre>	<pre>> append([1,2],[3,4],Z). Z = [1,2,3,4] > append(X,Y,[1,2,3]). X=[] Y=[1,2,3] X=[1] Y=[2,3] X=[1,2] Y=[3] X=[1,2,3] Y=[]</pre>

Helaas is dit gebruik van relaties in twee richtingen alleen mogelijk als bij de definitie van relaties geen gebruik is gemaakt van ingebouwde relaties die niet in twee richtingen gebruikt mogen worden, zoals `plus`.

A.7.5 Constructorfuncties

In Prolog kunnen op de parameterplaatsen van predicaten ook weer functies aangeroepen worden. Dit stelt dan echter geen functie-aanroep voor van een elders gedefinieerde functie – dat dient immers te gebeuren met het eerder beschreven mechanisme.

Functies in Prolog komen overeen met wat in Helium constructorfuncties genoemd zou worden.

Helium	Prolog
<pre>data Boom = Tip Int Tak Boom Boom size (Tip x) = 1 size (Tak p q) = size p + size q</pre>	<pre>size(tip(x),1). size(tak(p,q),a+b) :- size(p,a), size(q,b).</pre>

Bijlage B

ISO/ASCII tabel

	0*16+...	1*16+...	2*16+...	3*16+...	4*16+...	5*16+...	6*16+...	7*16+...
...+0	0 NUL	16 DLE	32 SP	48 0	64 @	80 P	96 '	112 p
...+1	1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
...+2	2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
...+3	3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
...+4	4 EOT	20 DC4	36 \$	52 4	68 D	84 T	100 d	116 t
...+5	5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
...+6	6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
...+7	7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
...+8	8 BS	24 CAN	40 (56 8	72 H	88 X	104 h	120 x
...+9	9 HT	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
...+10	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
...+11	11 VT	27 ESC	43 +	59 ;	75 K	91 [107 k	123 {
...+12	12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124
...+13	13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
...+14	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
...+15	15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 DEL

Helium-notatie voor speciale tekens in een string gaat met behulp van een code achter een backslash:

- de *naam* van het speciale teken, bijvoorbeeld "\ESC" voor het escape-teken;
- het *nummer* van het speciale teken, bijvoorbeeld "\27" voor het escape-teken;
- het nummer in het achttallig stelsel (*octaal*), bijvoorbeeld "\o33" voor het escape-teken;
- het nummer in het zestientallig stelsel (*hexadecimaal*), bijvoorbeeld "\x1B" voor het escape-teken;
- door de overeenkomstige letter vier kolommen verder naar rechts, bijvoorbeeld "\^[[" voor het escape-teken;
- een van de volgende codes: "\n" (newline), "\b" (backspace), "\t" (tab), "\a" (alarm), "\f" (formfeed), "\" ("'-symbool), "\'" ('-symbool), en "\\" (\-symbool)

Bijlage C

Helium syntax

Hieronder volgt een samenvatting van de Helium-syntax. Er wordt gebruik gemaakt van de BNF-notatie voor grammatica's, met de volgende conventies:

- grammaticale begrippen staan in het gewone lettertype;
- symbolen die in het programma terecht komen staan in `rechthoeken` en zijn gezet in het `tikmachine lettertype`;
- verticale strepen ‘|’ scheiden de alternatieven;
- {accolades} staan rond dingen die nul of meer keer aanwezig mogen zijn;
- [vierkante haken] staan rond dingen die weggelaten mogen worden.

De inhoud van een file wordt beschreven door het begrip ‘module’. De inhoud van een opdracht aan de interpreter wordt gegeven door het begrip ‘interp’.

In de grammatica is sprake van de symbolen `{` en `}`. Het symbool `{` mag weggelaten worden. De layout-regel gaat dan gelden. Dat wil zeggen: een `;` mag weggelaten worden, als de tekst achter de `;` evenver ingesprongen dan de tekst ervoor. Wordt de tekst minder ingesprongen, dan wordt automatisch een `}` toegevoegd. (Zie ook paragraaf 1.4.5).

blz. 12

De volgende begrippen worden verder niet uitgewerkt (ze staan daarom in *cursief* lettertype):

<code>varid</code>	naam die begint met een kleine letter
<code>conid</code>	naam die begint met een hoofdletter
<code>varop</code>	operator-symbool dat niet met een : begint
<code>conop</code>	operator-symbool dat met een : begint
<code>integer</code>	integer constante
<code>float</code>	floating point constante
<code>char</code>	character constante
<code>string</code>	string constante

Declaraties

module	::=	<code>module</code> <code>Varid</code> <code>where</code> <code>{</code> topdecls <code>}</code>	programma
interp	::=	exp [where]	opdracht aan interpreter
topdecls	::=	topdecls <code>;</code> topdecls	meerdere declaraties
		<code>data</code> typeLhs <code>=</code> constrs	datatype declaratie
		<code>type</code> typeLhs <code>=</code> type	type-synoniem declaratie
		<code>infixl</code> [digit] op { <code>,</code> op}	prioriteits-declaratie
		<code>infixr</code> [digit] op { <code>,</code> op}	
		<code>infix</code> [digit] op { <code>,</code> op}	
		<code>primitive</code> prims <code>::</code> type	declaratie ingebouwde functie
		decls	functie declaratie

typeLhs	::=	<code>conid</code> { <code>varid</code> }	linkerkant typedeclaratie
constrs	::=	constrs <code>conop</code> type	meerdere constructoren infix constructor
		<code>conid</code> {type}	constructor
prims	::=	prims {,} prims	meerdere primitieven
		var <code>string</code>	binding primitieve functie

Types

type	::=	ctype [<code>-></code> type]	functie type
ctype	::=	<code>conid</code> {atype}	datatype of synonym
		atype	
atype	::=	<code>varid</code>	type variabele
		<code>()</code>	nultupel
		<code>(type)</code>	type tussen haakjes
		<code>(type , type {,} type)</code>	tupel type
		<code>[type]</code>	lijst type

Waarde- en functie-declaraties

decls	::=	decls <code>;</code> decls	meerdere declaraties
		var { <code>,</code> var} <code>::</code> sigType	type declaratie
		fun rhs [where]	functiedefinitie
		pat rhs [where]	patroondefinitie
rhs	::=	<code>=</code> exp	eenvoudige rechterkant
		gdRhs {gdRhs}	alternatieven
gdRhs	::=	exp <code>=</code> exp	
where	::=	<code>where</code> { decls }	locale definities
fun	::=	var	functie zonder parameters
		pat varop pat	infix operator
		fun apat	function met parameters
		<code>(fun)</code>	overbodige haakjes

Expressions

exp	::=	\ apat {apat} -> exp	lambda expressie
		let { decls } in exp	locale definitie
		if exp then exp else exp	voorwaardelijke expressie
		case exp of { alts }	case-expressie
		opExp :: sigType	getypeerde expressie
		opExp	
opExp	::=	opExp op opExp	toepassen van operator
		pfxExp	
pfxExp	::=	- appExp	min met één parameter
		appExp	
appExp	::=	appExp atomic	functie-toepassing
		atomic	
atomic	::=	var	variabele
		conid	constructor
		integer	integer constante
		float	floating point constante
		char	character constante
		string	string constante
		()	multupel
		(exp)	expr. tussen haakjes
		(exp op)	secties
		(op exp)	
		[list]	lijst expressie
		(exp , exp { , exp })	tupel
list	::=	[exp { , exp }]	lijst-opsomming
		exp quals	lijst-comprehensie
		exp ..	rekenkundige opsomming
		exp , exp ..	
		exp .. exp	
		exp , exp .. exp	
quals	::=	quals , quals	meerdere qualifiers
		pat <- exp	generator
		pat = exp	locale definitie
		exp	boolean guard
alts	::=	alts , alts	meerdere alternatieven
		pat altRhs [where]	alternatief
altRhs	::=	-> exp	enkel alternatief
		gdAlt gdAlt	guarded alternatieven
gdAlt	::=	exp -> exp	guarded alternatief

Patronen

pat	::=	pat conop pat	operator toepassing
		appPat	
appPat	::=	appPat apat	toepassing
		apat	
apat	::=	var	variabele
		var @ pat	'as'-patroon
		\square	wildcard
		conid	constructor
		\square <i>integer</i>	integer constante
		\square <i>char</i>	character constante
		\square <i>string</i>	string constante
		\square ()	nultupel
		\square (pat)	expressie tussen haakjes
		\square (pat conop)	secties
		\square (conop pat)	
		\square [[pat { , pat }]]	lijst
		\square (pat , pat { , pat })	tupel

Variabelen en operators

var	::=	varid	
		\square (-)	variabele
op	::=	varop	
		conop	
		\square -	operator
varid	::=	\square <i>varid</i>	
		\square (<i>varop</i>)	operator als functie
varop	::=	\square <i>varop</i>	
		\square ' <i>varid</i> '	functie als operator
conid	::=	\square <i>conid</i>	
		\square (<i>conop</i>)	
conop	::=	\square <i>conop</i>	
		\square ' <i>conid</i> '	

Bijlage D

Helium standaardfuncties

Operator-prioriteiten

```

infixr 9  .
infixl 9  !!
infixr 8  ^, ^., **.
infixl 7  *, *., 'quot', 'rem', 'div', 'mod', /., /
infixl 6  +, -, +., -.
infixr 5  ++, :
infix  4  ==, /=, <=, <, >, >=, ==., /=., <=., <., >., >=.
infixr 3  &&
infixr 2  ||
infixr 0  $, $!

```

Functies op Booleans en characters

```

otherwise  :: Bool
not        :: Bool -> Bool
(&&), (||) :: Bool -> Bool -> Bool
and, or    :: [Bool] -> Bool
any, all   :: (a->Bool) -> [a] -> Bool

isAscii, isControl, isPrint, isSpace          :: Char -> Bool
isUpper, isLower,  isAlpha, isDigit, isAlphanum :: Char -> Bool

toUpper, toLower  :: Char -> Char
ord             :: Char -> Int
chr            :: Int -> Char

data Maybe a = Nothing | Just a
maybe :: b -> (a -> b) -> Maybe a -> b
data Either a b = Left a | Right b
either :: (a -> c) -> (b -> c) -> Either a b -> c
data Ordering = LT | EQ | GT

```

Numerieke functies

```

(+), (-), (*), (/), (^)          :: Int -> Int -> Int
(<), (<=), (>), (>=), (==), (/=) :: Int -> Int -> Bool
abs, signum, negate              :: Int -> Int
rem, div, mod, quot              :: Int -> Int -> Int
subtract, gcd, lcm, min, max     :: Int -> Int -> Int
odd, even                        :: Int -> Bool
sum, product, maximum, minimum  :: [Int] -> Int

pi :: Float

```

```
(+.), (-.), (*.), (/.), (^.), (**.)    :: Float -> Float -> Float
(<.), (<=.), (>.), (>=.), (==.), (/=.) :: Float -> Float -> Bool
exp, log, sin, cos, tan                :: Float -> Float
```

```
intToFloat :: Int -> Float
round, truncate, floor, ceiling :: Float -> Int
```

Polymorfe functies

```
undefined :: a
id        :: a -> a
const    :: a -> b -> a
fix      :: (a -> a) -> a
($)      :: (a->b) -> (a->b)
strict   :: (a->b) -> (a->b)

(.)      :: (b->c) -> (a->b) -> (a->c)
uncurry  :: (a->b->c) -> ((a,b)->c)
curry    :: ((a,b)->c) -> (a->b->c)
flip     :: (a->b->c) -> (b->a->c)
until    :: (a->Bool) -> (a->a) -> a -> a
until'   :: (a->Bool) -> (a->a) -> a -> [a]
fst      :: (a,b) -> a
snd      :: (a,b) -> b
```

Functies op lijsten

```
head  :: [a] -> a
last  :: [a] -> a
tail  :: [a] -> [a]
init  :: [a] -> [a]
(!!)  :: [a] -> Int -> a
index :: Int -> [a] -> a

take, drop      :: Int -> [a] -> [a]
splitAt         :: Int -> [a] -> ([a], [a])
takeWhile, dropWhile :: (a->Bool) -> [a] -> [a]
takeUntil      :: (a->Bool) -> [a] -> [a]
span, break     :: (a->Bool) -> [a] -> ([a], [a])

length      :: [a] -> Int
null        :: [a] -> Bool
elemBy, notElemBy :: (a->a->Bool) a -> [a] -> Bool

(++)      :: [a] -> [a] -> [a]
iterate   :: (a->a) -> a -> [a]
repeat    :: a -> [a]
cycle     :: [a] -> [a]
replicate :: Int -> a -> [a]
reverse   :: [a] -> [a]

concat :: [[a]] -> [a]
map     :: (a->b) -> [a] -> [b]
concatMap :: (a -> [b]) -> [a] -> [b]
filter   :: (a->Bool) -> [a] -> [a]

foldl  :: (a->b->a) -> a -> [b] -> a
foldl' :: (a->b->a) -> a -> [b] -> a
```

```

foldr    :: (a->b->b) -> b -> [a] -> b
foldl1   :: (a->a->a) ->      [a] -> a
foldr1   :: (a->a->a) ->      [a] -> a
scanl    :: (a->b->a) -> a -> [b] -> [a]
scanl'   :: (a->b->a) -> a -> [b] -> [a]
scanr    :: (a->b->b) -> b -> [a] -> [b]
scanl1   :: (a->a->a) ->      [a] -> [a]
scanr1   :: (a->a->a) ->      [a] -> [a]

zip      :: [a] -> [b]           -> [(a,b)]
zip3     :: [a] -> [b] -> [c]     -> [(a,b,c)]
zipWith  :: (a->b->c)             -> [a]->[b]->[c]
zipWith3 :: (a->b->c->d)          -> [a]->[b]->[c]->[d]
unzip    :: [(a,b)] -> ([a],[b])
unzip3   :: [(a,b,c)] -> ([a],[b],[c])

```

Functionies op strings

```

type String = [Char]
readInt     :: String -> Int
readUnsigned :: String -> Int
words      :: String -> [String]
lines      :: String -> [String]
unwords    :: [String] -> String
unlines    :: [String] -> String

```

Gelijkheid en ordening

```

eqChar    :: Char -> Char -> Bool
eqBool    :: Bool -> Bool -> Bool
eqString  :: String -> String -> Bool
eqMaybe  :: (a->a->Bool) -> Maybe a -> Maybe a -> Bool
eqList    :: (a->a->Bool) -> [a] -> [a] -> Bool
eqTuple2  :: (a->a->Bool) ->
              (b->b->Bool) -> (a,b) -> (a,b) -> Bool

ordString :: String -> String -> Ordering
ordChar   :: Char -> Char -> Ordering
ordInt    :: Int -> Int -> Ordering
ordList   :: (a->a->Ordering) -> [a] -> [a] -> Ordering

elemBy    :: (a->a->Bool) -> a -> [a] -> Bool
notElemBy :: (a->a->Bool) -> a -> [a] -> Bool
lookupBy  :: (a->a->Bool) -> a -> [(a,b)] -> Maybe b

```

Input/output functies

```

unsafePerformIO :: IO a -> a
return          :: a -> IO a
(>>=)          :: IO a -> (a -> IO b) -> IO b
sequence       :: [IO a] -> IO [a]
sequence_      :: [IO a] -> IO ()
putChar        :: Char -> IO ()
putStr         :: String -> IO ()
putStrLn       :: String -> IO ()
print          :: (a -> String) -> a -> IO ()
getLine        :: IO String

```

Bijlage E

Literatuur

Leerboeken met vergelijkbare stof

- Richard Bird en Philip Wadler: *Introduction to functional programming*. Prentice-Hall, 1988.
- Richard Bird en Philip Wadler: *Functioneel programmeren, een inleiding*. Academic service, 1991.
- A.J.T. Davie: *An introduction to functional programming systems using Haskell*. Cambridge university press, 1992.
- Ian Hoyer: *Functional programming with Miranda*. Pitman, 1991.
- Hudak en Fasel: ‘a gentle introduction to Haskell’. *ACM sigplan notices* **27**, 5 (may 1992) pp.T1–T53.
- Simon Thompson: *Haskell, the craft of functional programming*. Addison-Wesley, 1996.
- Philip Wadler: *Introduction to functional programming, second edition*. Prentice-Hall, 1998.
- Paul Hudak: *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge UP, 2000.

Leerboeken waarin een andere programmeertaal gebruikt wordt

- Rinus Plasmeijer en Marko van Eekelen: *Functional programming and parallel graph rewriting*. Addison-Wesley, 1993.
- Åke Wikström: *Functional programming using standard ML*. Prentice-Hall, 1987.
- Chris Reade: *Elements of functional programming*. Addison-Wesley, 1989.
- Roger Bailey: *Functional programming with Hope*. Ellis-Horwood, 1990.
- Abelson en Sussman: *Structure and interpretation of computer programs*. McGraw-Hill, 1985.

Taalbeschrijvingen

- Hudak, Peyton-Jones, Wadler *et.al.*: ‘Report on the programming language Haskell: a non-strict purely functional language, version 1.2.’ *ACM sigplan notices* **27**, 5 (may 1992) pp.R1–R164.
- Mark P. Jones: *Introduction to Gofer 2.20*. Oxford Programming Research Group, 1992.

Implementatie van functionele programmeertalen

- Mark P. Jones: *The implementation of the Gofer functional programming system*. Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, May 1994.
- Field en Harrison: *Functional programming*. Addison-Wesley, 1989.
- Simon Peyton-Jones: *The implementation of functional programming languages*. Prentice-Hall, 1987.
- Simon Peyton-Jones en David Lester: *Implementing functional languages: a tutorial*. Prentice-Hall, 1992.

Geavanceerd materiaal

- Johan Jeuring en Erik Meijer (eds): *Advanced functional programming*. Springer, 1985 (LN-CS 925).
- Colin Runciman: *Applications of functional programming*. Cambridge university press, 1995.
- *Proceedings of the ...th conference on Functional Programming and Computer Architecture (FPCA)*. Springer, 1992–1995.

Over parse-technieken

- W.H. Burge, ‘Parsing’. In *Recursive Programming Techniques*, Addison-Wesley, 1975.
- Graham Hutton, ‘Higher-order functions for parsing’. *J. Functional Programming* **2**:323–343.
- P. Wadler, ‘How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages’. In *Functional Programming Languages and Computer Architecture*, (J.P.Jouannaud, ed.), Springer, 1985 (LNCS 201), pp. 113–128.
- Philip Wadler, ‘Monads for functional programming’. In: *Advanced Functional Programming*, Tutorial text of the First international spring school on advanced functional programming techniques in Båstad, Sweden, may 1995. (Johan Jeuring and Erik Meijer, eds). Berlin, Springer 1995 (LNCS 925), pp. 24–52.
- Jeroen Fokker, ‘Functional Parsers’. In: *Advanced Functional Programming*, Tutorial text of the First international spring school on advanced functional programming techniques in Båstad, Sweden, may 1995. (Johan Jeuring and Erik Meijer, eds). Berlin, Springer 1995 (LNCS 925), pp. 1–23.
- S.D.Swierstra and Luc Duponcheel, ‘Deterministic, Error-Correcting Combinator Parsers’. In: *Advanced Functional Programming 2*, Tutorial text of the Second international summer school on advanced functional programming techniques, Olympia, Washington USA, august 1996. (John Launchbury, Erik Meijer and Tim Sheard, eds.). Berlin, Springer, 1996 (LNCS 1129), pp. 184–207.

Achterliggende theorie

- Henk Barendregt: *The lambda-calculus: its syntax and semantics*. North-Holland, 1984.
- Richard Bird: *An introduction to the theory of lists*. Oxford Programming Research Group report PRG–56, 1986.
- Luca Cardelli en P.Wegner: ‘On understanding types, data abstraction and polymorphism.’ *Computing surveys* **17**, 4 (1986).
- R. Milner: ‘A theory of type polymorphism in programming.’ *J.computer and system sciences* **17**, 3 (1978).

Historisch materiaal

- Alonzo Church: ‘The calculi of lambda-conversion’. *Annals of mathematical studies* **6**. Princeton university, 1941; Kraus reprint 1971.
- Haskell Curry *et al.*: *Combinatory logic, vol. I*. North-Holland, 1958.
- A. Fraenkel: *Abstract set theory*. North-Holland, 1953.
- M. Schönfinkel: ‘Über die Bausteine der mathematischen Logik.’ *Mathematischen Annalen* **92** (1924).

Tijdschriften

- *Journal of functional programming* (1991–). Cambridge University Press.
- `comp.lang.functional`. Usenet newsgroup.
- `www.haskell.org`. Website.

Bijlage F

Woordenlijst

- !!** operator die een bepaald element van een lijst selecteert. (par. 3.1.2, blz. 36)
- ==** gelijkheidsoperator (par. 1.3.4, blz. 7)
- /=** ongelijkheidsoperator (par. 1.3.4, blz. 7)
- ++** operator die twee lijsten samenvoegt. (par. 3.1.2, blz. 36)
- <=** operator ‘kleiner dan of gelijk aan’. (par. 1.3.4, blz. 7)
- >=** operator ‘kleiner dan of gelijk aan’. (par. 1.3.4, blz. 7)
- Actuele parameter** expressie die bij aanroep van een functie de waarde van de parameter vastlegt. (par. 1.4.3, blz. 9)
- Ambiguous** dubbelzinnig, waarvan de waarde niet bepaald kan worden omdat de *associatievolgorde niet is gedefinieerd. (par. 2.1.3, blz. 20)
- Assignment** toekenningsopdracht, opdracht in een *imperatieve programmeertaal waarmee aan een variabele een bepaalde waarde wordt toegekend. De waarde van de variabele kan later door een andere toekenningsopdracht weer worden veranderd. (par. 1.1.1, blz. 1)
- Associatievolgorde** manier waarop de parameters van twee dezelfde operatoren in een expressie worden gegroepeerd. Associatie naar links: $(a+b)+c$; associatie naar rechts: $a+(b+c)$. (par. 2.1.3, blz. 20)
- Beschermd type** *datatype met slechts één *constructorfunctie, die geheim wordt gehouden waardoor de elementen van het type alleen aan te spreken zijn via daarvoor bedoelde functies. (par. 3.4.3, blz. 62)
- Bool** type met als elementen de *waarheidswaarden True en False. (par. 1.3.4, blz. 7)
- Boomstructuur** element van een *datatype, dat een niet-lineaire structuur heeft. (par. 3.4.1, blz. 56)
- Char** type met als elementen de letters, cijfers en leestekens zoals die op het toetsenbord van de computer voorkomen. (par. 1.5.2, blz. 14)
- Coëfficiënt** getal in een *term van een *polynoom. (par. 4.2.1, blz. 71)
- Comprehensie** *lijstcomprehensie. (par. 3.2.7, blz. 50)
- concat** Helium-functie die een lijst van lijsten samenvoegt tot één lange lijst. (par. 3.1.2, blz. 36)
- Concatenatie** het samenvoegen van lijsten tot één lange lijst; in Helium met de operator **++** (voor twee lijsten) of de functie **concat** (voor een lijst van lijsten). (par. 3.1.2, blz. 36)
- Constructorfunctie** functie waarmee een *datastructuur kan worden opgebouwd. In Helium begint de naam van een constructorfunctie met een hoofdletter, of in het geval van *operatoren met een dubbele punt. (par. 3.4.1, blz. 56)
- Control-C** toetsencombinatie waarmee in Helium oneindig lang durende berekeningen onderbroken kunnen worden. (par. 2.3.2, blz. 24)
- Currying** het simuleren van een functie met meerdere parameters door een functie met één parameter die een functie oplevert. (par. 2.2.1, blz. 21)
- Datastructuur** element van een *datatype. (par. 3.4.1, blz. 56)
- Datadefinitie** definitie waarmee een *datatype wordt geïntroduceerd, door van alle *constructorfuncties de types van de parameters op te sommen. (par. 3.4.1, blz. 56)
- Datatype** type dat gekenmerkt wordt door de manier waarop elementen kunnen worden opgebouwd (door middel van *constructorfuncties). (par. 3.4.1, blz. 56)

- Default-definitie** definitie in Helium voor een overloaded functie die wordt gebruikt indien de functie niet in de instance-declaratie wordt gedefinieerd. De default-definitie wordt gegeven in de klasse-declaratie. (par. ??, blz. ??)
- Derive** afleiden. ‘cannot derive instance’ betekent in Helium dat een overloaded operator wordt gebruikt op een type dat geen instance is van de betreffende klasse. (par. ??, blz. ??)
- Differentiëren** *numeriek of *symbolisch differentiëren.
- Disjoint union** Engels voor *vereniging van typen. (par. 3.4.1, blz. 56)
- Eindig type** *datatype waarin alle *constructorfuncties nul parameters hebben. (par. 3.4.3, blz. 62)
- Expressie** constructie in een programmeertaal, die een waarde heeft. (par. 1.2.1, blz. 2)
- Expressieboom** *ontleedboom van een (rekenkundige of andersoortige) expressie. (par. 5.1.1, blz. 77)
- Faculteit** functie die het product bepaalt van alle natuurlijke getallen tussen 1 en zijn parameter. (par. 1.2.2, blz. 3)
- Float** type met als elementen de *floating-point getallen. (par. 1.3.3, blz. 6)
- Floating-point getallen** deelverzameling van de reële getallen die op een machine met een beperkte reken nauwkeurigheid kunnen worden opgeslagen. In Helium aangeduid met *Float. (par. 1.3.3, blz. 6)
- foldr** Helium-functie die een operator tussen alle elementen van een lijst plaatst, te beginnen aan de rechterkant met een bepaald element. (par. 3.1.3, blz. 39)
- Formele parameter** aanduiding van de veranderlijke van een functie bij de definitie ervan. In de meeste programmeertalen is een formele parameter een variabele, in Helium mag het echter een *patroon zijn. (par. 1.4.3, blz. 9)
- Functie** verband tussen de *parameters en het resultaat van bepaalde processen. (par. 1.1.1, blz. 1)
- Functionele programmeertaal** programmeertaal waarbij een programma bestaat uit de definitie van een aantal *functies, waarvan de resultaatwaarde steeds geheel bepaald wordt door de parameters. (par. 1.1.1, blz. 1)
- Geccurryde functie** functie met één parameter die weer een functie oplevert, en daarom beschouwd kan worden als een functie met meerdere parameters. (par. 2.2.1, blz. 21)
- Gehele getallen** de verzameling Z van (positieve en negatieve) getallen zonder breukdeel. In Helium aangeduid met *Int (begrensd versie) of *Integer (onbegrensd versie). (par. 1.3.3, blz. 6)
- Gereserveerd woord** woord met een speciale betekenis in een programmeertaal, dat daarom niet als naam van een variabele gebruikt mag worden. Voorbeeld in C: ‘if’; voorbeeld in Helium: ‘where’. (par. 1.3.2, blz. 5)
- Gevals onderscheid** methode waarmee een functie in verwschillende gevallen op een verschillende manier gedefinieerd kan worden. Gevals onderscheid is in Helium mogelijk door middel van *voorwaarden of door middel van *patronen. (par. 1.4.2, blz. 9)
- Haskell** *functionele programmeertaal, beschreven in 1992 door het Haskell-comité.
- Helium** Vereenvoudiging van Haskell voor educatief gebruik, ontwikkeld vanaf 2002 aan de universiteit Utrecht.
- Hogere-orde functie** functie met een functie als parameter en/of als resultaat. (par. 2.3.1, blz. 23)
- Hugs** Interpreter voor Haskell, ontwikkeld vanaf in 1992 door Mark Jones. Zie www.cs.nott.ac/~mpj.
- Imperatieve programmeertaal** programmeertaal gekenmerkt door de aanwezigheid van *toekenningsopdrachten, die na elkaar worden uitgevoerd. Voorbeelden zijn Pascal en C. (par. 1.1.2, blz. 1)
- Inductieve definitie** *recursieve definitie. (par. 1.4.4, blz. 11)
- Ingebouwde functie** *primitieve functie.
- Int** type met als elementen een begrensd deelverzameling van de *gehele getallen. (par. 1.3.3, blz. 6)
- Interpreter** computerprogramma dat programma’s in een bepaalde programmeertaal uitvoert. (par. 1.2.1, blz. 2)
- Invoegen** op de juiste plaats zetten, bijvoorbeeld in een gesorteerde lijst (par. 3.1.5, blz. 42) of in een zoekboom (par. 3.4.2, blz. 58)

- iterate** Helium-functie die de oneindige lijst bepaalt waarin een functie steeds vaker op een startwaarde wordt toegepast. (par. 3.2.6, blz. 48)
- Lazy evaluatie** berekeningsvolgorde waarbij de parameter van een functie pas wordt uitgerekend op het moment dat hij nodig is, en niet direct bij aanroep van een functie. (par. 3.2.5, blz. 47)
- Lege lijst** *lijst met nul elementen. (par. 3.1.1, blz. 35)
- Lexicografische ordening** Ordening van lijsten waarbij het eerste element bepalend is, tenzij het eerste element gelijk is; in dat geval beslist het tweede element, tenzij dat ook gelijk is, enzovoorts. Als een van de twee lijsten een beginstuk is van de andere, dan is de lengte bepalend. (par. 3.1.2, blz. 36)
- Lijst** Type waarvan het aantal elementen kan variëren, maar waarvan de elementen allen hetzelfde type hebben. (par. 3.1.1, blz. 35)
- Lijstcomprehensie** notatie waarmee een lijst wordt opgebouwd door een expressie te evalueren in een omgeving waarbij een variabele alle waarden van een lijst doorloopt. (par. 3.2.7, blz. 50)
- Lisp** *functionele programmeertaal, ontwikkeld in 1958 door John McCarthy. Afkorting van ‘list processor’. Vooral populair geworden in het toepassingsgebied ‘kunstmatige intelligentie’. (par. 1.1.2, blz. 1)
- Lokale definitie** definitie van een constante of een functie die alleen gebruikt kan worden binnen een andere functie. (par. 1.4.1, blz. 8)
- map** Helium-functie die een functie toepast op alle elementen van een lijst. (par. 3.1.3, blz. 39)
- Numeriek differentiëren** door middel van numerieke benadering de afgeleide functie evalueren in een bepaald punt. (par. 2.4.2, blz. 29)
- Oneindige lijst** lijst waarvan steeds meer elementen bepaald kunnen worden; het werken met oneindige lijsten is mogelijk vanwege *lazy evaluatie. (par. 3.2.4, blz. 46)
- Ontleden** bepalen van de *ontleedboom van een taalconstructie die als string gegeven is. (par. ??, blz. ??)
- Ontleedboom** *datastructuur waarmee de opbouw van een taalconstructie wordt weergegeven. (par. ??, blz. ??)
- Operator** *functie waarvan de naam tussen zijn twee *parameters wordt geschreven in plaats van ervoor. (par. 2.1.1, blz. 19)
- Paar** *tupel bestaande uit twee elementen. (par. 3.3.1, blz. 51)
- Parameter** veranderlijke van een *functie. Bij de definitie van een functie wordt de veranderlijke aangeduid door een *formele parameter, bij gebruik van de functie wordt de waarde van de veranderlijke vastgelegd door een *actuele parameter. (par. 1.1.1, blz. 1)
- Partieel parametriseren** het aanroepen van een functie met minder parameters dan deze verwacht. (par. 2.2.1, blz. 21)
- Patroon** speciale vorm van *formele parameter, waarmee eenvoudig gevalsonderscheid kan worden gepleegd bij de definitie van een functie. (par. 1.4.3, blz. 9)
- Polymorfe functie** functie met een *polymorf type, die dus op parameters van verschillende typen, die echter een bepaalde structuur gemeen hebben, mag worden toegepast. (par. 1.5.3, blz. 15)
- Polymorf type** type in de aanduiding waarvan een *type-variabele voorkomt. In feite dus een groep typen, die een bepaalde structuur gemeen hebben. (par. 1.5.3, blz. 15)
- Polynoom** som van termen, waarbij elke term bestaat uit een produkt van een reëel getal (de coëfficiënt) en een natuurlijke macht (de exponent) van een variabele. (par. 4.2.1, blz. 71)
- Prelude** verzameling *voorgedefinieerde functies die in alle programma’s mogen worden gebruikt. (par. 1.2.1, blz. 2)
- Primitief type** *type die de *interpreter kent zonder dat er een definitie voor gegeven is. In Helium een van de vier typen *Int, *Float, *Bool en *Char. (par. 1.5.2, blz. 14)
- Primitieve functie** *functie die een *interpreter kent zonder dat er een definitie voor gegeven is. De functie is dus ‘ingebouwd’ in de interpreter. (par. 1.3.1, blz. 4)
- Prioriteit** rangorde die aangeeft in welke volgorde van elkaar verschillende operatoren worden uitgerekend. (par. 2.1.2, blz. 19)
- Rationaal getal** breuk. (par. 3.3.3, blz. 53)

- Recursieve definitie** definitie van bijvoorbeeld een functie of een type, waarbij het te definiëren begrip bij de definitie al wordt gebruikt. (par. 1.4.4, blz. 11)
- Rekenkundige functie** functie die werkt op getallen, bijvoorbeeld ‘optellen’ of ‘vermenigvuldigen’. (par. ??, blz. ??)
- Sectie** notatie waarbij een operator, door hem tussen haakjes te zetten, tot functie gemaakt wordt. (par. ??, blz. ??)
- Singleton-lijst** *lijst met één element. (par. 3.1.1, blz. 35)
- Sorteren** op (opklimmende) volgorde zetten van de elementen (van een lijst). (par. 3.1.5, blz. 42)
- Sqrt** afkorting van ‘square root’, het Engelse woord voor ‘vierkantwortel’. (par. 1.2.1, blz. 2)
- String** *lijst waarvan de elementen het type *Char hebben. (par. 3.2.1, blz. 44)
- Symbolisch differentiëren** door middel van manipulatie van de *ontleedboom van een functiedefinitie de ontleedboom van de afgeleide functie bepalen. (par. 5.1.2, blz. 77)
- Term** deel van een *expressie. (par. 1.5.1, blz. 12) produkt van factoren, bijvoorbeeld in een polynoom (par. 4.2.1, blz. 71) of in een expressie (par. 5.1.1, blz. 77)
- Tupel** samengesteld type bestaande uit een vast aantal waarden, die echter verschillende typen mogen hebben. (par. 3.3.1, blz. 51)
- Type** waardenverzameling waarbinnen een functie zijn waarde kan hebben. (par. 1.5.2, blz. 14)
- Typedefinitie** regel in een programma waarmee ter afkorting een naam gegeven kan worden aan een bepaald type. (par. 3.3.2, blz. 53)
- Typeringsfout** fout die optreedt als in een expressie functies worden toegepast op parameters die niet het juiste type hebben. (par. 1.5.1, blz. 12)
- Typevariabele** variabele in een type-aanduiding, waardoor de type-aanduiding in feite een groep typen wordt aangeduid die allen een bepaalde structuur gemeen hebben (een *polymorf type). (par. 1.5.3, blz. 15)
- until** Helium-functie die net zolang een functie toepast op een startwaarde totdat aan een bepaalde eigenschap is voldaan. (par. 2.3.2, blz. 24)
- Vereenvoudigen** (van *polynoom): sorteren van de termen, samenvoegen van termen met gelijke *coëfficiënt, en verwijderen van termen met coëfficiënt nul. (par. 4.2.2, blz. 72) (van *rationaal getal): delen van teller en noemer van een breuk door hun grootste gemene deler, en verplaatsen van een eventueel minteken naar de teller. (par. 3.3.3, blz. 53)
- Vereniging van typen** *datatype waarin elke *constructorfunctie slechts één, niet-recursieve parameter heeft. Engels: ‘disjoint union’. (par. 3.4.3, blz. 62)
- Voorgedefinieerde functie** *functie die in de *prelude wordt gedefinieerd, en dus in alle programma’s gebruikt mag worden. (par. 1.3.1, blz. 4)
- Voorwaarde** logische expressie, die aangeeft dat een functie alleen op een bepaalde manier wordt gedefinieerd als die expressie de waarde True heeft. Door meerdere voorwaarden in een functiedefinitie te zetten kan *gevalsonderscheid worden gemaakt. (par. 1.4.2, blz. 9)
- Waarheidswaarden** verzameling met de elementen ‘True’ en ‘False’, die de mogelijke uitkomsten vormen van een logische *expressie. In Helium aangeduid met *Bool. (par. 1.3.4, blz. 7)
- zip** Helium-functie die van twee lijsten een lijst tweetupels maakt. (par. 3.3.4, blz. 55)
- Zoekboom** *binaire boom die ‘gesorteerd’ is, in de zin dat alle elementen in de linker deelboom kleiner zijn dan de waarde die op het splitspunt is opgeslagen, en alle elementen in de rechter deelboom groter. (par. 3.4.2, blz. 58)

Index

- () , 81
- (Char, (a, (Char, a))), 84
- (v1, v2), 82
- *>, 86, 87
- ++, 37
- ., 25
- <*, 90
- <*>, 82–86, 90
- <:*>, 90
- &&, 10, 48
- 0.0, 87

- a, 80, 84
- aantalOpl, 16, 17
- abcFormule, 8, 9, 12, 16
- abcFormule', 9, 30
- abs, 6, 9, 54
- actuele parameters, 10
- afg, 78, 89
- afstand, 53
- and, 8, 24, 31, 37
- apostrof, 19
- arccos, 31, 32
- arcsin, 31, 32
- 'as'-patroon, 71
- associatie
 - van :, 36

- b, 80
- back quote, 19
- beginsegment, 67, 68
- Blad, 57, 61
- Bool (type), 13–15, 41, 44, 62
- Boom (type), 57, 58, 61
- Boom2 (type), 65
- boven, 4, 5, 8, 16
- box, 63

- chain, 86
- chainl, 86, 90
- chainr, 86–90
- Char, 80
- Char (type), 14, 44, 45, 49, 78
- choice, 90
- chr, 45, 46, 62
- Church, Alonzo, 1
- Clean, 2
- combinatie, 67, 70
- combinatorische functie, 67–71
- combs, 70
- combs n, 67

- commentaar, 12
- Complex (type), 53
- complexe getallen, 64
- comprehensie, 50
- concat, 37, 51, 63, 64, 70, 109
- concatenatie, 37
- constante, 4
- constructor-functies, 57
- controlestructuur, 24
- cos, 3, 15, 31, 32
- cp, 75
- cpWith, 74
- cross-product, 74
- crossprod, 75
- curry, 56, 64
- Curry, Haskell, 1

- dag, 27, 28
- dagnummer, 27, 28
- data-definitie, 57
- datatype, 56
- deelbaar, 27, 28, 50, 54
- deelij, 67, 69
- delers, 27, 28, 47, 54
- deleteBoom, 60
- derdemachtswortel, 33
- det, 75
- DetPars, 83
- diepte, 65
- diff, 29, 31, 33, 35
- differentiëren
 - symbolisch, 78
- digitChar, 46
- digitValue, 46
- div, 21
- drop, 38, 40, 43, 52
- dropWhile, 40, 64, 71

- e, 8
- eager evaluatie, 47
- EenChar, 62
- EenInt, 62
- eenvoud, 54
- eindsegment, 67, 68
- elem, 59, 64
- elemBoom, 59, 64
- elemBy, 42, 58
- enumFrom, 48
- enumFromTo, 36
- epsilon, 81, 82, 84, 85
- Eq (class), 55

- eq, 40
- eqBool, 41
- eqChar, 41
- eqFloat, 7
- eqList, 41, 42
- eqString, 41
- even, 7, 8, 14, 25
- exp, 6, 8, 17, 32
- Expr, 80, 88
- Expr (type), 79, 89
- expr, 88, 89

- f, 83, 86
- fac, 4, 5, 8, 11, 21
- fact, 88
- fail, 82
- filter, 23, 27, 36, 39, 40, 49–51, 54, 63, 70, 73, 89
- first, 87
- flexDiff, 29
- Float (type), 14, 26, 29, 30, 32, 44, 52, 53, 72, 75, 79
- foldBoom, 65
- foldl, 24, 40, 86
- foldl', 5
- foldparens, 84, 90
- foldr, 24, 31, 36, 38–40, 42, 43, 63–65, 82, 87, 89
- for (keyword), 24
- formele parameters, 10
- fst, 52
- Fun, 88
- functie
 - als operator, 19
 - combinatorische, 67–71
 - op lijsten, 36
 - polymorfe, 15
- functies
 - polymorfe, 15

- gaps, 75
- gcd, 6, 54
- gelijkheid
 - op lijsten, 41
- gen, 89
- gereserveerde woorden, 5
- ggd, 54
- goedGenoeg, 30
- Gofer, 2
- graad, 71
- groepeer, 64
- grootste gemene deler, 54
- grootsteUit, 60, 61
- guards, 9

- head, 10, 11, 15, 36, 37, 50, 80
- hogere-orde functie, 24

- id, 15
- infix (keyword), 21
- infixl (keyword), 21
- infixr, 89
- ingebouwde functies, 5
- init, 37, 38
- inits, 67–69, 75
- insert, 42, 43, 59, 64
- insertBoom, 59, 60
- Int (type), 13–15, 22, 26, 44, 49, 72
- Integer (type), 14
- integraal, 33
- interpreter, 2
- interval-notatie, 36
- IntOfChar (type), 62
- intString, 48, 49, 54, 65, 75
- intToFloat, 6
- inverse, 31–33, 35
- is..., 45
- isnul, 8
- isort, 43, 59, 64
- iterate, 48–50, 63–65

- just, 83, 89

- kwadraat, 8, 23, 50

- labels, 60, 65
- last, 37, 54
- lazy evaluatie, 47
- lege lijst, 35
- length, 3, 7, 11, 13, 15, 36, 39, 43, 48, 56, 57, 64, 70
- lijst, 35–51
 - concateneren, 37
 - gelijkheid, 41
 - interval-notatie, 36
 - lege, 35
 - opsomming, 35, 36
 - singleton, 35
 - type, 35
- lijst-comprehensie, 50
- lijstNaarBoom, 59, 60, 65
- lines, 46
- Lisp, 2
- list, 85, 86
- listOf, 86
- ln, 32
- log, 6
- lokale definities, 9
- lookupBy, 62

- maanden, 28
- many, 85, 87, 90
- many (symbol 'a'), 90
- many1, 85, 90
- map, 8, 16, 22–24, 36, 38, 39, 41, 42, 46, 47, 50, 51, 55, 58, 63–65, 70
- mapBoom, 65
- matInv, 75
- Maybe (type), 62
- MacCarthy, John, 2

- merge, 43, 64
- Miranda, 2
- ML, 2
- mod, 21
- msort, 64

- na, 25, 32, 33
- natural, 85
- negatief, 8
- nesting, 85, 89, 90
- Neuman, John von, 1
- norep, 89
- not, 7, 8, 25
- null, 8
- nulpunt, 31–33

- Oak, 80
- omvang, 57
- oneven, 8, 25
- ontleedboom, 77
- open*>parens<*close, 90
- operator, 5, 19
 - ++, 37
 - ., 25
 - &&, 10, 48
 - als functie, 19
- optellen
 - breuken, 54
 - polynomen, 73
- option, 85–87
- option p, 87
- opvolger, 21, 22
- or, 39, 58
- Ord (class), 45
- ord, 45, 46
- Ordering (type), 41, 62
- ordInt, 41

- p, 82, 83
- p1, 82
- p2, 82
- paar, 52
- pack, 85
- parens, 84, 90
- Parser, 80
- Parser Char (Char, (Char, Char)), 82
- Parser Char Char, 82
- Parser Expr, 80
- Parser Int, 80
- Parser Oak, 80
- pEenvoud, 73
- pEq, 72
- perms, 67, 69, 70
- permuatie, 69–70
- permutatie, 67
- pEval, 74
- pi, 4, 8, 17
- plus, 21, 22
- Poly (type), 71, 72
- polymorf type, 15, 35
- polymorfe functie, 15
- polymorfe functies, 15
- polymorfie, 15
- polynoom, 71–75
 - graad, 71, 74
 - optellen, 73
 - vereenvoudigen, 73
 - vermenigvuldigen, 74
- positief, 8
- priem, 27, 28, 47, 49
- priemgetallen, 27, 28, 50
- primitive functions, 5
- prioriteit, 19–20
- product, 4, 5, 8, 24, 31
- Prop (type), 78
- Punt (type), 53

- qDeel, 63
- qEq, 64
- qMaal, 63
- qMin, 63
- qPlus, 63

- Rat, 63
- Ratio (type), 63
- rationale getallen, 53
- ratioString, 54
- recursieve definitie, 11
- rem, 21, 26–28, 54
- repeat, 48, 63
- replicate, 3, 48
- reverse, 3, 7, 38, 39, 49

- samenvoegen, 60, 61
- satisfy, 81, 89
- Scheme, 2
- Schönfinkel, M., 1
- schrikkel, 28
- segment, 67–69
- segs, 67, 68, 75
- sequence, 90
- show, 79
- showFloat, 75
- signum, 6, 9, 54
- sin, 3, 6, 15, 23, 29, 31, 32
- singleton-lijst, 35
- snd, 52
- som, 11, 13, 36
- some, 83
- sort, 43, 45
- sorteer, 60
- sortTerms, 73
- sp, 83
- splitAt, 52
- sqrt, 2, 3, 6–10, 17, 29, 30, 52
- Stat (type), 89
- String, 80
- string, 44
- stringInt, 64
- subs, 67, 69, 70, 75

- subsequence, 69
- succeed, 81, 82, 85
- succeed [], 85
- sum, 3, 5, 7, 8, 14, 15, 23, 24, 28, 31, 48, 74
- symbol, 81, 89
- symbol '(', 84
- symbol ')', 84
- symbol 'a', 82
- symbolisch differentiëren, 78

- tail, 10, 11, 17, 36, 37, 69, 71
- tails, 67, 68, 71, 75
- Tak, 57
- take, 8, 28, 38, 40, 43, 48, 52
- takeWhile, 40, 47, 49, 65
- tan, 15
- Term (type), 72
- term, 88, 89
- tMaal, 74
- token, 81, 90
- toonBoom, 65
- toUpper, 46
- transpose, 75
- Tree, 80
- truncate, 6, 62
- tupel, 51–56
- Turing, Alan, 1
- tussen, 70
- type
 - polymorf, 35
- type (keyword), 53
- type-definitie, 53
- typevariabele, 15

- uncurry, 56, 64
- unlines, 46
- until, 24, 25, 30–32, 44, 48, 64
- unwords, 46

- v1, 82
- v2, 82
- vanaf, 46, 48
- Var, 88
- veelvoud, 50
- verbeter, 30
- vereenvoudigen
 - breuk, 54
 - polynoom, 73
- vermenigvuldigen
 - breuken, 54
 - polynomen, 74
- verw, 78
- voorgedefinieerde, 5

- weekdag, 27, 28, 38
- weerg, 79
- where (keyword), 9, 12, 13, 25, 26, 29
- while (keyword), 24
- words, 46
- wortel, 30–33

- xs1, 82
- xs2, 82

- zip, 55, 75
- zipWith, 55, 74
- zoekOp, 55, 59, 64