Functioneel Programmeren 1

Informatica 1e jaar, periode 4

J. Foppele

Opleiding Informatica
NHL Hogeschool

Juli 2016

Inhoud

1 Functioneel programmeren

Functionele programmeertalen onderscheiden zich van ‘gewone’ talen doordat ze wiskundige functies als uitgangspunt nemen i.p.v. het model van de computer. Bij dit vak gebruiken we de functionele taal Haskell.

1.1 Haskell

Functioneel programmeren zullen we doen in de taal Haskell. Hiervoor heeft men de GHCi (Glasgow Haskell Compiler interactive environment) nodig. Deze is te downloaden op https://www.haskell.org/platform/. Voor diegene die niet weten hoe met GHCi te werken staat later in dit hoofdstuk een tutorial.
Haskell is een pure functionele taal, wat het zeer geschikt maakt voor het leren van functioneel programmeren. Tijdens dit vak zal alleen de basis worden behandeld. Geavanceerde constructies zoals bijvoorbeeld monads zullen niet aan de orde komen.

1.1.1 Lazy

Een belangrijke eigenschap van Haskell is, is dat het een lazy taal is. Dit wil zeggen dat Haskell pas iets gaat doen als het ook daadwerkelijk de opdracht krijgt om iets te gaan doen. Doordat Haskell lazy is zijn constructies mogelijk die in andere talen ongebruikelijk zijn, zoals oneindige lijsten. Een voorbeeld van lazy met een voorbeeld erbij is het volgende:
Stel je voor dat je een lijst hebt met de elementen [1,4,7,21,-3,8,2] en een functie genaamd dubbel die de elementen van een lijst verdubbelt. En je roept dan het volgende aan:
dubbel(dubbel(dubbel(dubbel([1,4,7,21,-3,8,2]))))
Een eager taal zal direct aan de gang gaan met deze constructie en zal in de binnenste dubbel() alle elementen in de lijst gaan vermenigvuldigen. Daarna geeft de eager taal dit terug aan de aanroepende functie, die vervolgens weer alle elementen van de teruggegeven lijst zal gaan vermenigvuldigen. Hierdoor ontstaan dus loops in loops.
Een lazy taal zoals Haskell kijkt hier heel anders naar. Haskell zal pas iets gaan doen als om een antwoord wordt gevraagd. Dus als de voorgaande constructie wordt aangeroepen zal de binnenste dubbel() niets doen voordat er een antwoord gevraagd wordt. De buitenste dubbel() zal daardoor een antwoord vragen aan de binnen gelegen dubbel(). De binnenste dubbel wordt dan uiteindelijk gevraagd om een resultaat, en deze zal dan het eerste element verdubbelt returnen (dus 2). Waarna de dubbel() daarvoor weer een resultaat geeft (4). Zo zullen alle getallen 1 voor 1 vier maal verdubbelt worden en wordt er maar een maal door de lijst heen gelopen.

1.2 Tutorial GHCi

GHCi is de compiler voor Haskell en werkt vanuit de command line. Na installatie is GHCi op te starten door in de command prompt `ghci` uit te voeren. Indien GHCi succesvol is geïnstalleerd krijg je nu ongeveer het volgende te zien:

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

Na de prelude kunnen nu commando's worden gegeven die door Haskell worden uitgevoerd. Als bijvoorbeeld `1 + 2` wordt ingevoerd, zal Haskell met het antwoord `3` komen.

```
Prelude> 1 + 2
3
```

1.2.1 Haskell files

Nu kan gestart worden met een eerste Haskell file. Deze kun je in je favoriete tekst editor maken. Als je een file maakt met daarin de regel `verdubbel x = x + x` en deze opslaat met naam test.hs dan kun je daarna deze eerste file compileren met GHCi. Belangrijk is wel dat GHCi gestart wordt in de map waar deze file staat.
Indien je nu het commando `:l test` geeft verschijnt er iets in het scherm als:

```
[1 of 1] Compiling Main             ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Hier is te zien dat het bestand gecompileerd is en dat de functie in het bestand aangeroepen kan worden.

```
*Main> verdubbel 21
42
```

Nu is duidelijk hoe Haskell gebruikt kan worden en wat GHCi kan.

1.2.2 Functies

De basis van functioneel programmeren is een functie. In de voorgaande paragraaf hebben we al een functie gezien, namelijk:

```
verdubbel x = x + x
```

De functie verdubbel heeft als argument x en heeft als resultaat x + x. Een belangrijke regel bij functies is dat deze, net als argumenten, altijd moeten beginnen met een kleine letter. Functies kunnen ook meerdere argumenten bevatten, deze worden dan gescheiden door spaties. Een voorbeeld met meerdere argumenten is:

```
kleinerDan x y = x < y
```

Deze functie kijkt of argument x kleiner is dan argument y.

1.2.3 Functies in functies

Functies zoals hierboven beschreven bevatten rekenkundige expressies. Maar naast rekenkundige expressies kunnen functies ook andere dingen zoals het uitvoeren van andere functies. Hieronder een voorbeeld van een functieaanroep door een andere functie:

```
maalVier x = verdubbel verdubbel x
```

De functie maalVier roept de eerder gemaakte verdubbel twee maal aan.

1.2.4 Functies met condities

Een functie kan ook conditioneel een resultaat geven. Als voorbeeld een functie hieronder waarbij het resultaat afhangt van of een positief of negatief getal wordt meegegeven.

```
absoluut x = if x < 0 then x * (-1)
                      else x
```

De absoluut functie zal ieder negatief getal dat wordt meegegeven vermenigvuldigen met -1, terwijl ieder positief getal gewoon teruggegeven wordt zonder bewerking.

1.2.5 Where

Soms komt het voor dat een berekening meerdere malen in één formule moet worden uitgevoerd. Zo komen er bij het uitvoeren van de abc-formule 2 verschillende antwoorden uit de berekening.
Hiervoor kunnen we de volgende formule maken:

```
abc1 a b c = ((-b - sqrt (b^2 - 4 * a * c)) / (2 * a),
              (-b + sqrt (b^2 - 4 * a * c)) / (2 * a))
```

Hierin zien we dat er twee keer wortelgetrokken wordt. Dit kunnen we versimpelen met een where clausule zoals hier beneden.

```
abc2 a b c = ((-b - d) / (2 * a),
              (-b + d) / (2 * a))
                    where d = sqrt (b^2 - 4 * a * c)
```

Door deze where clausule te gebruiken wordt de berekening van de wortel maar één maal uitgevoerd en wordt de code ook nog beter leesbaar.

1.3 Practicum opdrachten

Opgave 1
Voer de volgende commando's uit in GHCi en evalueer wat er gebeurt:

a. 3 * 6
b. 2 + 4 * 7
c. (2 + 4) * 7
d. 3 / 2
e. 3 `div` 2
f. div 3 2
g. 3 `mod` 2
h. 2^10
i. 2+pi
j. [1..10]
k. sum [1..10]
l. length [1..10]
m. [1..10] !! 2
n. map (*3) [1..10]
o. 1 == 2
p. 1 /= 2
q. "Haskell" > "C#"
r. (1 == 2) == False
s. [1..10] ++ [1..10]
t. take 3 [1..10]
u. drop 3 [1..10]
v. head [1..10]
w. tail [1..10]
x. take 5 [1..]
y. head [1..]

Opgave 2
Download van blackboard de file Werkcollege1.hs en run deze file. Bekijk wat de functies gem, verhoog en ing doen en beschrijf dit.

Opgave 3
Breid de file Werkcollege1.hs uit met de volgende opdrachten:

a. Maak de functie `cirkelOppervlak r`, deze functie moet de oppervlakte van een cirkel bereken aan de hand van de meegegeven straal. De formule hiervoor is $\pi * r^2$.
b. Maak de functie `opbrengst b n r`, deze functie moet de opbrengt na n jaar bij een rente r bereken over het startbedrag b. De formule hierbij is $b * (1 + \frac{r}{100})^n$.
c. Maak de functie `alleGelijk a b c`. alleGelijk geeft True als resultaat als alle 3 argumenten dezelfde waarde hebben, anders False.
d. Maak de functie `alleVerschillend a b c`. alleVerschillend geeft True als resultaat als alle 3 argumenten een verschillende waarde hebben, anders False
e. Maak de functie `derde xs` die het derde element uit een lijst selecteert zonder !! te gebruiken.
f. Maak de functie `plak3 xs ys zs` die drie lijsten aan elkaar plakt.
g. Hoe selecteer je uit een lijst het derde t/m het negende element?

2 Geavanceerde functies en lijstcomprehensies

In voorgaand hoofdstuk is de basis van Haskell behandeld. In dit hoofdstuk gaan we dieper op belangrijke onderwerpen van functioneel programmeren in, zoals pattern matching en lijstcomprehensies.

2.1 Typering

Haskell is van nature statisch getypeerd. Dat houdt in dat iedere expressie en functie een type heeft. Door de statische typering kunnen veel programmeerfouten al tijdens het compilen opgespoord worden en zal de code altijd typecorrect zijn.

2.1.1 Type-inferentie

In Haskell hoeven parameters, expressies en functies niet voorzien te worden van typering. De compiler bepaalt zelf van welk type iets is. Dit wordt type-inferentie genoemd.

2.1.2 Typen

Haskell kent een groot aantal verschillende typen. Deze komen veelal overeen met bekende typen uit andere programmeertalen, zo zijn er:

- Int
- Char
- Bool
- Float
- Double

Daarnaast zijn er nieuwe typen zoals:

- Integer, een Int alleen dan zonder limiet
- Num, een verzameling van alle numerieke datatypen
- Integral, een verzameling van alle numerieke datatypen voor gehele getallen
- Fractional, een verzameling van alle numerieke datatypen voor komma getallen
- Ord, een verzameling van typen die geordend kunnen worden
- Eq, een verzameling van typen die op gelijkheid getest kunnen worden

Er zijn nog enkele andere typen, deze worden echter tijdens dit vak niet behandelt.

2.1.3 Haskell type commando

Omdat Haskell statisch getypeerd is, kan van iedere expressie het type worden opgevraagd. Dit kan worden gedaan met het commando `:t`. Indien we bijvoorbeeld het type willen weten van een getal, dan typen we het onderstaande in GHCi.

```
*Main> :t 'j'
'j' :: Char
```

Hier zien we dat de letter j door Haskell gezien wordt als iets van het type Char, dit wordt aangegeven met de dubbele dubbelepunt (::).

Nu kunnen we ook het type van andere elementen achterhalen. Als we nu de volgende functie implementeren:

```
plus1 x = x + 1.0
```

Nu kunnen we van deze functie met behulp van `:t` de type definities achterhalen.

```
*Main> :t plus1
plus1 :: Fractional a => a -> a
```

We zien hier een op het eerste gezicht ingewikkelde regel, bestaande uit meerdere elementen. Belangrijk om als eerste te weten is dat, net als bij het simpele eerste voorbeeld, de type definitie pas volgt na de dubbele dubbelepunt (::). Hierna staat `Fractional a` gevolgd door het teken `=>`. Alles wat voor het `=>` teken staat zijn definities die in de latere type definitie gebruikt worden. In dit geval word dus het type a gedefinieerd en dit is van type Fractional. Na het `=>` teken volgt de daadwerkelijke type definitie. De type definitie is `a -> a`, wat niets meer betekend dan er gaat een parameter in deze functie van het type a en het resultaat is ook van datzelfde type.
Mocht de functie meerdere parameters hebben, dan worden de parameters ook gescheiden door het teken `->`.
Als laatste voorbeeld maken we nu de volgende functie:

elementen x y = [x,y,'a']

Als we van deze functie het type opvragen, krijgen we:

```
*Main> :t elementen
elementen :: Char -> Char -> [Char]
```

Hier lezen we nu dus simpel af dat elementen een functie is met twee parameters van het type Char en dat het functieresultaat van het type `[Char]` is, oftewel een lijst van Char. Een uitgebreide uitleg over lijsten volgt in paragraaf 2.3.

2.1.4 Type definities bij functies

Het is bij het maken van functies ook mogelijk om zelf de typedefinitie te definiëren. Niet alleen wordt dit gezien als een goede eigenschap van programmeurs, daarnaast kan het ook nog goed zijn om bepaalde typen af te dwingen en dus Haskell te bepreken in de keuze. We hebben in voorgaande paragraaf gezien bij de `plus1` functie dat Haskell niet wist welk type kommagetal gebruikt moest worden, en daarom pakte Haskell alle typen kommagetallen. Indien we echter de volgende functie definiëren dan ontnemen we Haskell deze vrijheid.

```
plus2 :: Double -> Double
plus2 x = x + 2.0
```

Hier maken we, voordat we de eigenlijke functie schrijven, eerst de typedefinitie aan. Hierbij dwingen we nu af dat Haskell bij deze functie altijd Double zal gebruiken voor de parameter en voor het functieresultaat. We zien nu ook dat het antwoord op `:t` aangepast is.

```
*Main> :t plus2
plus2 :: Double -> Double
```

Natuurlijk hadden we ook dezelfde definitie als dat Haskell al had gemaakt bij `plus1` kunnen gebruiken.

2.2 Lijsten

Een belangrijke datastructuur binnen een functionele programmeertaal is de lijst. Een lijst wordt geschreven met blokhaken en bevat elementen van hetzelfde type.
In voorgaand hoofdstuk hebben we al enkele voorbeelden van lijsten gezien. Een simpele lijst zou kunnen zijn:

```
[1,5,9,3,-7]
```

Deze lijst bevat allemaal elementen van het type Num. We kunnen ook simpeler een oplopende lijst maken door het volgende te gebruiken:

```
[1..10]
```

Hiermee genereren we een lijst die loopt van 1 t/m 10 met stappen van 1. We kunnen ook andere stappen nemen, door het volgende te gebruiken:

```
[1,3..10]
```

Dit levert een lijst op van 1 t/m 10, maar nu met stappen van 2. 10 is dus geen onderdeel van de resulterende lijst.
Ook is het mogelijk om oneindige lijsten te maken. Zo levert het volgende een lijst op die bij 1 begint en altijd doorloopt met stappen van 1:

```
[1..]
```

Ondanks dat de meeste programmeertalen dit niet accepteren, kan Haskell hier gewoon mee om gaan omdat het een lazy taal is.
Tot nu toe zijn alle voorbeelden met cijfers geweest, maar een lijst kan ieder type bevatten.
[‘h’,’a’,’l’,’l’,’o’]
Dit is een lijst met karakters. Het volgende is echter voor Haskell precies hetzelfde:

```
"hallo"
```

Een string is voor Haskell dus niets anders dan een lijst met karakters. Naast karakters en cijfers, kan een lijst ook andere typen bevatten, zoals andere lijsten. Het enige wat geldt is dat alle elementen van een lijst van hetzelfde type moeten zijn.

2.3 Tuples

Een tuple is een soort van lijst. Echter mogen er in tuples elementen van verschillend typen staan. Omdat er verschillende typen in een tuple kunnen zitten, zijn er minder functies beschikbaar voor tuples. Een voorbeeld van een tuple:

```
(42,True,”Hello”)
```

2.4 Lijstcomprehensies

Nu we lijsten kennen, willen we natuurlijk ook door een lijst heen kunnen lopen. Uit andere programmeertalen zoals C, C# of Java kennen we hiervoor for en while lussen. In Haskell kan dit met behulp van een lijstcomprehensie.
Om te zien wat een lijstcomprehensie kan kijken we naar het volgende voorbeeld.

```
Prelude> [2*x| x <- [3,4,1,6]]
[6,8,2,12]
```

Hierin zien we dat door de lijst [3,4,1,6] heen wordt gelopen en dat een lijst wordt teruggegeven waarin ieder element met 2 is vermenigvuldigd.
Een lijstcomprehensie bevat twee gedeeltes, gescheiden door een |. Voor het | teken staat het resultaat van de lijstcomprehensie en na het | teken staan condities en generatoren. In bovenstaand voorbeeld staat alleen één generator. Generatoren en condities staan hieronder uitgelegd.

2.4.1 Generator

Een lijstcomprehensie bevat altijd minstens één generator.

```
Prelude> [2*x| x <- [3,4,1,6]]
[6,8,2,12]
```

In dit eerder geziene voorbeeld zien we de generator `x <- [3,4,1,6]`. Met een generator kan een variabele gevuld worden met de elementen uit een lijst. In bovenstaande gevallen is dit een lijst van integers, maar dit kan iedere lijst zijn. Zo wordt in onderstaand voorbeeld door een lijst met karakters heen gelust.

```
Prelude> [fromEnum x| x <- "hallo"]
[104,97,108,108,111]
```

Er kunnen ook meerdere generatoren in en lijstcomprehensie zitten.

```
Prelude> [(x,y)| x <- [1..3], y <- [7..9]]
[(1,7),(1,8),(1,9),(2,7),(2,8),(2,9),(3,7),(3,8),(3,9)]
```

Zoals in bovenstaand voorbeeld te zien, is de volgorde van belang indien er meerdere generatoren zijn. Eerst word namelijk in de eerste generator de eerste waarde in de variabele gezet, waarna de volgende generator geheel wordt doorlopen voordat in de eerste generator de volgende waarde wordt genomen. Dit is te vergelijken met een geneste for-lus in talen zoals Java of C#.

2.4.2 Conditie

In een lijstcomprehensie kunnen condities voor komen. Een voorbeeld van een conditie staat hier onder.

```
Prelude> [x| x <- [1..10], mod x 2 == 0]
[2,4,6,8,10]
```

In bovenstaand voorbeeld staat de conditie `mod x 2 == 0`. Indien deze conditie wordt voldaan, dat wil zeggen, indien x modulo 2 0 is, dan wordt het antwoord teruggegeven en anders niet.
Net zoals bij een generator, kunnen er ook meerdere condities in een lijstcomprehensie staan. Hieronder een voorbeeld van een lijstcomprehensie die getallen van 1 tot 10 teruggeeft die deelbaar zijn door 2 en niet door 3.

```
Prelude> [x| x <- [1..10], mod x 2 == 0, mod x 3 /= 0]
[2,4,8,10]
```

2.4.3 Zip

Soms is het nodig om door meerdere lijsten tegelijkertijd heen te lussen, en dus niet één voor één met meerdere generatoren. Hiervoor is zip geschikt. Een voorbeeld van een lijstcomprehensie met zip staat hieronder.

```
Prelude> [(x,y)| (x,y) <- zip [1..3] [7..9]]
[(1,7),(2,8),(3,9)]
```

Dit is handig als we lijsten willen vergelijken, of een index aan een lijst willen koppelen. Zo kunnen we bijvoorbeeld de index zoeken van de letter l in een tekst.

```
Prelude> [i| (c,i) <- zip "hello world" [0..], c == 'l']
[2,3,9]
```

Hierbij zien we dat we de tekst koppelen aan een oneindige lijst startend bij nul. Omdat Haskell lazy is gaat dit prima, Haskell gaat hiermee door zolang er karakters in de tekst zitten en daarna stopt hij.
We kunnen ook zoeken op dezelfde opeenvolgende letters. Dit kunnen we doen door een tekst aan dezelfde tekst minus het eerste karakter te koppelen. Ook dit kan met zip.

```
Prelude> [c1|(c1,c2) <- zip "hello" (tail "hello"), c1 == c2]
"l"
```

Soms is het nodig om niet twee maar drie lijsten aan elkaar te zippen. Hiervoor is de functie `zip3` beschikbaar. Deze werkt op dezelfde manier.
Nu kunnen we dit nog afmaken door het geheel in een functie te zetten.

```
opeenvolgendeElementen :: (Eq a) => [a] -> [Int]
opeenvolgendeElementen xs = [i|(x,y,i) <- zip3 xs (tail xs) [0..], x == y]
```

Nu hebben we een functie gemaakt die de index van opeenvolgende elementen teruggeeft.

2.4.4 Let

Soms is het handig om in een lijstcomprehensie een tussenvariabele aan te maken. Dit kan door gebruik te maken van het keyword `let`. Een simpel voorbeeld hiervan staat hieronder.

```
Prelude> [y|x <- [1..10], let y = x + 3]
[4,5,6,7,8,9,10,11,12,13]
```

Hierin zien we dat de variabele y wordt aangemaakt en teruggegeven. Dit kan gebruikt worden om constructies waarin eenzelfde berekening vaker dan één keer wordt gedaan te versimpelen en te versnellen.

2.5 Pattern matching

Pattern matching, of in het Nederlands patroonvergelijking, is het herkennen van een specifieke structuur in data.
Door pattern matching in functies te gebruiken, kan men op eenvoudige manier verschillende stukken code maken voor verschillende invoer. Als voorbeeld de volgende code:

```
patternMatch :: String -> String
patternMatch "Functioneel Programmeren" = "Awesome"
patternMatch x = "Niet zo awesome"
```

In deze voorbeeldcode kunnen we zien hoe het aanroepen van de functie met de String “Functioneel Programmeren” een ander resultaat oplevert dan wanneer we dezelfde functie aanroepen met een willekeurige andere string.
Als pattern matching gebruikt wordt dan wordt van boven naar beneden gematched en zal de eerst mogelijke match worden uitgevoerd. Bij pattern matching is de volgorde daarom van belang.

2.6 Practicum opdrachten

Opgave 1
Definieer m.b.v. een lijstcomprehensie een functie `verhoog xs`, die alle elementen van de lijst xs van getallen met 1 verhoogt.

Opgave 2
Definieer m.b.v. een lijstcomprehensies de functie `sumsqr` die het volgende berekent:

```
sumsqrs n = 1^2 + 2^2 + .. + n^2
```

Opgave 3
Definieer m.b.v. een lijstcomprehensie een functie die alle getallen uit een lijst vervangt door drietallen, 1 * getal, 2 * getal, 3 * getal.
Voorbeeld:

```
*Main> fie [1,3,4,5]
[1,2,3,3,6,9,4,8,12,5,10,15]
```

Opgave 4

a. Schrijf m.b.v. een lijstcomprehensie een functie `delers n`, die een lijst van alle delers van n oplevert. Een getal (d) van n is een deler, als de deling n / d geen rest heeft (dus als n `mod` d == 0).
b. Een getal is een priemgetal als het getal maar twee delers heeft (1 en zichzelf). Schrijf nu een functie `priemgetal n`, die bepaalt of n een priemgetal is (maak gebruik van de functie delers uit de vorige opgave).
c. Een getal heet perfect als het gelijk is aan de som van zijn delers (kleiner dan het getal zelf). Bv. 6 is een perfect getal (6 = 1 + 2 + 3). Probeer het volgende perfecte getal te vinden.
d. De grootste gemene deler (ggd) van twee getallen is het grootste getal dat beide getallen deelt. Definieer een functie `ggd n m`, die de grootste gemene deler van n en m bepaalt (maak gebruik van delers en maximum).

Opgave 5
Definieer een functie `pythagoras n` die voor gegeven n een lijst van alle pythagorische drietallen (x,y,z) met x,y,z tussen 1 en n berekent. Voor x,y,z moet dus gelden x^2 + y^2 = z^2. Denk om de efficiency, elk drietal mag maar een keer voorkomen.

Opgave 6
Definieer een functie die twee teksten vergelijkt en de indices teruggeeft waarop beide teksten dezelfde letter bevatten. Hierbij begint de index te tellen vanaf 0. Voorbeeld:

```
*Main> dezelfdeLetter "hello world" "hallo wereld"
[0,2,3,4,5,6,8]
```

Opgave 7
Schrijf een functie die kijkt of een meegegeven string een weekdag is (maandag … zondag) en die bij iedere dag een beschrijving geeft. Gebruik hierbij pattern matching. Voorbeeld:

```
*Main> weekdag "zondag"
"Weekend!"
```

3 Backtracking

Backtracking is een algoritme waarbij gezocht wordt naar oplossingen voor een probleem. Dit zoeken gebeurd door deeloplossingen te evalueren en een deeloplossing te schrappen wanneer deze de evaluatie niet doorstaat.
Met backtracking kunnen alle mogelijke oplossingen bij een probleem gevonden worden, of kan gezocht worden naar de eerste oplossing.
Bekende problemen die opgelost kunnen worden met backtracking zijn het acht koninginnen probleem en het inkleuren van een landkaart. Maar ook andere puzzels zoals sudoku's of doolhoven kunnen opgelost worden met backtracking.

3.1 Backtracking algoritme

De basis van een backtracking algoritme lijkt op brute force alles uitproberen. Echter wordt bij brute force, in tegenstelling tot backtracking, alle mogelijkheden uitgeprobeerd. Alle mogelijkheden proberen is zeer inefficiënt en daarom is een backtracking algoritme vele malen sneller. Later in dit hoofdstuk worden brute force en backtracking tegen elkaar afgezet om het verschil tussen beide inzichtelijk te maken.
De basis van backtracking komt neer op zo snel mogelijk na het maken van een keuze deze keuze te evalueren. Dit zorgt er voor dat het algoritme niet bezig is met het maken van keuzes die toch niet kunnen.

3.2 Magisch vierkant

Een magisch vierkant is een vierkant van n bij n getallen waarbij de som van iedere rij, kolom en diagonaal hetzelfde getal is. Een bekend voorbeeld hiervan is hieronder afgebeeld.

Tabel 1: Een bekend magisch vierkant

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Dit magische vierkant heeft in iedere rij en in iedere kolom getallen met als som 34.

3.2.1 Brute force 3x3 vierkant

Om oplossingen van magische vierkanten te vinden kan een functie geschreven worden in Haskell. Als voorbeeld nemen we het volgende zoekprobleem.
Vul een 3x3 vierkant met de getallen 1 t/m 9 zo dat elke rij, kolom en diagonaal als som 15 heeft.
Om dit zoekprobleem op te lossen moet eerst een nummering worden afgesproken, bijvoorbeeld als volgt:

Tabel 2: Nummering van een 3x3 vierkant

x1	x2	x3
x4	x5	x6
x7	x8	x9

Nu kan er in Haskell code worden geschreven die brute-force naar een oplossing zoekt.

```
vierkantBruteForce :: [(Int,Int,Int,Int,Int,Int,Int,Int,Int)]
vierkantBruteForce = [(x1, x2, x3, x4, x5, x6, x7, x8, x9) |
   x1 <- [1..9],
   x2 <- [1..9],
   x3 <- [1..9],
   x4 <- [1..9],
   x5 <- [1..9],
   x6 <- [1..9],
   x7 <- [1..9],
   x8 <- [1..9],
   x9 <- [1..9],
   x1 /= x2, x1 /= x3, etc...  -- alle getallen moeten verschillend zijn
   x1 + x2 + x3 == 15,         -- de som van rij 1 is 15
   x4 + x5 + x6 == 15,         -- de som van rij 2 is 15
   x7 + x8 + x9 == 15,         -- de som van rij 3 is 15
   x1 + x4 + x7 == 15,         -- de som van kolom 1 is 15
   x2 + x5 + x8 == 15,         -- de som van kolom 2 is 15
   x3 + x6 + x9 == 15,         -- de som van kolom 3 is 15
   x1 + x5 + x9 == 15,         -- de som van de diagonaal is 15
   x3 + x5 + x7 == 15          -- de som van de diagonaal is 15
   ]
```

Een probleem van de brute-force aanpak is de lange rekentijd die nodig is voordat de oplossingen gevonden zijn.

3.2.2 Backtracking 3x3 vierkant

Om backtracking toe te passen op een magisch vierkant wordt bovenstaande brute-force code veranderd in onderstaande code.

```
vierkantBacktracking :: [(Int,Int,Int,Int,Int,Int,Int,Int,Int)]
vierkantBacktracking = [(x1, x2, x3, x4, x5, x6, x7, x8, x9) |
   x1 <- [1..9],
   x2 <- [x|x<-[1..9], x /= x1],
   x3 <- [x|x<-[1..9], x /= x1, x /= x2],
   x1 + x2 + x3 == 15,          -- de som van rij 1 is 15
   x4 <- [x|x<-[1..9], x /= x1, x /= x2, x /= x3],
   x7 <- [x|x<-[1..9], x /= x1, x /= x2, x /= x3, x /= x4],
   x1 + x4 + x7 == 15,          -- de som van kolom 1 is 15
   x5 <- [x|x<-[1..9], x /= x1, x /= x2, x /= x3, x /= x4, x /= x7],
   x3 + x5 + x7 == 15,          -- de som van de diagonaal is 15
   x6 <- [x|x<-[1..9], x /= x1, x /= x2, x /= x3, x /= x4, x /= x7,
                                                            x /= x5],
   x4 + x5 + x6 == 15,          -- de som van rij 2 is 15
   x9 <- [x|x<-[1..9], x /= x1, x /= x2, x /= x3, x /= x4, x /= x7,
                                                   x /= x5, x /= x6],
   x3 + x6 + x9 == 15,          -- de som van kolom 3 is 15
   x1 + x5 + x9 == 15,          -- de som van de diagonaal is 15
   x8 <- [x|x<-[1..9], x /= x1, x /= x2, x /= x3, x /= x4, x /= x7,
                                             x /= x5, x /= x6, x /= x9],
   x7 + x8 + x9 == 15,          -- de som van rij 3 is 15
   x2 + x5 + x8 == 15           -- de som van kolom 2 is 15
   ]
```

In bovenstaande code is te zien hoe backtracking werkt. Er worden 3 keuzes gemaakt (x1, x2 en x3) en direct daarna worden deze keuzes geëvalueerd. Ook valt op dat tijdens het maken van de keuze direct niet mogelijke keuzes verwijderd worden met een lijstcomprehensie. Dit zorgt er voor dat er later niet hoeft gecontroleerd te worden of een getal meerdere malen is gekozen. Deze lijstcomprehensie zorgt er echter wel voor dat de code erg lang wordt.

We kunnen dit nog verbeteren door een hulpfunctie te maken die de dubbelingen voorkomt. Dit resulteert in onderstaande code.

```
del :: Eq a => [a] -> [a] -> [a]
del xs ys = [y| y <- ys, not(elem y xs)]

vierkantBacktracking :: [(Int,Int,Int,Int,Int,Int,Int,Int,Int)]
vierkantBacktracking = [(x1, x2, x3, x4, x5, x6, x7, x8, x9) |
   x1 <- [1..9],
   x2 <- (del [x1] [1..9]),
   x3 <- (del [x1,x2] [1..9]),
   x1 + x2 + x3 == 15,         -- de som van rij 1 is 15
   x4 <- (del [x1,x2,x3] [1..9]),
   x7 <- (del [x1,x2,x3,x4] [1..9]),
   x1 + x4 + x7 == 15,         -- de som van kolom 1 is 15
   x5 <- (del [x1,x2,x3,x4,x7] [1..9]),
   x3 + x5 + x7 == 15,         -- de som van de diagonaal is 15
   x6 <- (del [x1,x2,x3,x4,x5,x7] [1..9]),
   x4 + x5 + x6 == 15,         -- de som van rij 2 is 15
   x9 <- (del [x1,x2,x3,x4,x5,x6,x7] [1..9]),
   x3 + x6 + x9 == 15,         -- de som van kolom 3 is 15
   x1 + x5 + x9 == 15,         -- de som van de diagonaal is 15
   x8 <- (del [x1,x2,x3,x4,x5,x6,x7,x9] [1..9]),
   x7 + x8 + x9 == 15,         -- de som van rij 3 is 15
   x2 + x5 + x8 == 15          -- de som van kolom 2 is 15
   ]
```

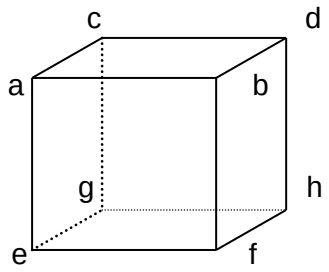
3.3 Practicum opdrachten

Opdracht 1
Vind m.b.v. backtracking het (unieke) getal n bestaande uit 9 cijfers met de volgende eigenschappen:

- de cijfers 1 t/m 9 komen precies één keer voor
- de eerste k cijfers van n zijn deelbaar door k voor k loopt van 1 t/m 9

Als voorbeeld: we maken een getal van 9 cijfers waarbij alle cijfers één maal voorkomen:
n = 123456789
Nu gaan we kijken of n ook voldoet aan de tweede regel.
1 (het eerste cijfer van n) moet deelbaar zijn door 1. Klopt!
12 (de eerste 2 cijfers van n) moet deelbaar zijn door 2. Klopt!
123 (de eerste 3 cijfers van n) moet deelbaar zijn door 3. Klopt!
1234 moet deelbaar zijn door 4. Klopt niet. Dus het getal 123456789 is niet het unieke getal.

Opdracht 2
Bij een magische kubus zijn de getallen 1 t/m 8 verdeeld over de hoekpunten zodat de som van elk vlak 18 is.



Bijvoorbeeld met a=1, b=4, c=8, d=5, e=6, f=7, g=3 en h=2 hebben we een magische kubus want alle vlakken hebben een som van 18.
Schrijf nu een functie met behulp van backtracking die een lijst van alle magische kubussen oplevert.

4 Recursie

Recursie is een methode om problemen op te lossen door het probleem in kleinere deelproblemen op te delen die op dezelfde manier opgelost kunnen worden.
Binnen informatica wordt dit toegepast door functies te maken die zichzelf aanroepen om tot een antwoord te komen. Om een recursieve functie altijd tot een goed einde te laten komen moet hierbij goed nagedacht worden over hoe en wanneer een recursieve functie tot een einde komt.

4.1 Recursie binnen Haskell

Binnen Haskell worden recursieve functies gedefinieerd door eerst te definiëren wanneer een functie tot een einde moet komen en daarna de recursieve functie te definiëren. De recursieve definitie bevat een definitie waarbij de recursieve functie opnieuw wordt aangeroepen, maar nu met parameters die dichter bij het einde liggen. Alle codevoorbeelden in dit hoofdstuk zijn ook te downloaden van blackboard, zie daarvoor de file Hoofdstuk3.hs.

4.1.1 Faculteit

Als voorbeeld van een recursieve functie nemen we een functie om de faculteit te berekenen:

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n – 1)
```

Hierbij zien we dat de functie tot een einde zal komen zodra de parameter 0 wordt bereikt. Als we nu uitwerken wat er gebeurd als we een ander getal meegeven, dan zien we het volgende:

```
fac 4 = 4 * fac 3
      = 4 * 3 * fac 2
      = 4 * 3 * 2 * fac 1
      = 4 * 3 * 2 * 1 * fac 0
      = 4 * 3 * 2 * 1 * 1
      = 24
```

Nu zien we de kracht van recursie. Door 1 functie te gebruiken kunnen we een ingewikkeld probleem zoals het berekenen van een faculteit oplossen.

4.1.2 Lijsten

Recursie kan ook goed gebruikt worden om door datastructuren zoals lijsten heen te lopen. Als voorbeeld een functie om het aantal elementen in een lijst te tellen:

```
aantalEl :: (Num a) => [t] -> a
aantalEl [] = 0
aantalEl (x:xs) = 1 + aantalEl xs
```

Ten eerste zien we dat deze recursieve functie tot een einde komt als geprobeerd wordt het aantal elementen van een lege lijst te tellen, wat natuurlijk 0 oplevert. Daarnaast is de constructie `(x:xs)` een opmerkelijke constructie. Wat dit doet is het volgende: het eerste element van een meegegeven lijst komt in variabele `x`, en de rest van de lijst (wat ook een lege lijst kan zijn) komt in `xs`. Het effect van deze functie is dus als volgt:

```
aantalEl [7,1,0,2] = 1 + aantalEl [1,0,2]
                   = 1 + 1 + aantalEl [0,2]
                   = 1 + 1 + 1 + aantalEl [2]
                   = 1 + 1 + 1 + 1 + aantalEl []
                   = 1 + 1 + 1 + 1 + 0
                   = 4
```

4.2 Recursief sorteren

Zoals al eerder gezegd zijn recursieve functies zeer geschikt om door lijsten heen te lopen. Door deze eigenschap kunnen we met behulp van recursieve functies heel gemakkelijk de sortering van een lijst evalueren. Ook kunnen we een ongesorteerde lijst sorteren.

4.2.1 Evalueren

Om te kijken of een lijst gesorteerd is kunnen we een recursieve functie definiëren. Deze kan bijvoorbeeld zijn:

```
isSorted :: (Ord a) => [a] -> Bool
isSorted [] = True
isSorted [x] = True
isSorted (x:y:xs) = x <= y && isSorted (y:xs)
```

Hier valt ten eerste natuurlijk op dat deze functie niet 1, maar 2 eindes heeft. Zowel bij een lijst van 1 element, als een lege lijst, zal True teruggegeven worden. Daarnaast is de constructie `(x:y:xs)` te zien, welke een uitbreiding is op de hierboven genoemde `(x:xs)` constructie waarbij ditmaal de eerste 2 elementen uit de lijst zullen worden genomen en in aparte variabelen worden gestopt. Ook is te zien dat deze constructie op een andere manier gebruikt kan worden, namelijk in de recursieve aanroep waarbij `(y:xs)` staat voor de lijst bestaande uit het element `y` gevolgd door de lijst `xs`. Deze worden dus aan elkaar geplakt. Deze functie zal op de volgende manier evalueren of een lijst gesorteerd is:

```
isSorted [1,2,4,3,5] = 1 <= 2 && isSorted [2,4,3,5]
                     = isSorted [2,4,3,5]
                     = 2 <= 4 && isSorted [4,3,5]
                     = isSorted [4,3,5]
                     = 4 <= 3 && isSorted [3,5]
                     = False
```

Hier zien we dat een recursieve functie dus ook kan stoppen zonder de functies te bereiken die bedoeld zijn voor het einde. Dit komt omdat een recursieve functie soms niet helemaal doorlopen hoeft te worden. In dit geval kan worden gezegd dat de lijst niet gesorteerd is indien er 2 opeenvolgende elementen kunnen worden gevonden die niet gesorteerd staan. Is dit het geval, dan hoeft niet de rest van de lijst ook nog doorzocht te worden omdat het antwoord toch al vast staat.

4.2.2 Sorteren

Naast kijken of een lijst gesorteerd is, kan een lijst ook gesorteerd worden met behulp van een recursieve functie. Een voorbeeld functie hiervan is:

```
sort :: (Ord a) => [a] -> [a]
sort [] = []
sort [x] = [x]
sort (x:xs) = sort [y| y <- xs, y <= x] ++ [x] ++
              sort [y| y <- xs, y > x]
```

We zien hier een recursieve functie die gebruik maakt van 2 lijstcomprehensies om tot een antwoord te komen. De lijstcomprehensies doen niets anders dan de meegegeven lijst opdelen in 2 lijsten waarbij 1 lijst alle elementen bevat die groter zijn dan het huidige element, en de andere lijst de overige elementen bevat. Als we een lijst willen sorteren, dan zien we het volgende:

```
sort [4,3,7,2] = sort [y| y <- [3,7,2], y <= 4] ++ [4] ++
                   sort [y| y <- [3,7,2], y > 4]
                 = sort [3,2] ++ [4] ++ sort [7]
                 = sort [y| y <- [2], y <= 3] ++ [3] ++
                   sort [y| y <- [2], y > 3] ++ [4] ++ [7]
                 = sort [2] ++ [3] ++ sort [] ++ [4] ++ [7]
                 = [2] ++ [3] ++ [] ++ [4] ++ [7]
                 = [2,3,4,7]
```

4.3 Opsparende parameter

Soms kan het bij recursieve functies handig zijn om tussenresultaten te onthouden terwijl de functie gebruikt wordt. Dit kan door gebruik te maken van een opsparende parameter. Als voorbeeld van een opsparende parameter hier een functie die alle getallen in een lijst bij elkaar optelt:

```
mySum :: (Num a) => [a] -> a
mySum xs = add 0 xs
add s [] = s
add s (x:xs) = add (s+x) xs
```

In het voorbeeld is te zien dat er een hulpfunctie gebruikt wordt die een extra parameter heeft, in dit geval `s`. Deze parameter wordt gebruikt voor het bijhouden van de optelling die tot dan toe gebeurd is. Oftewel:

```
mySum [2,5,2,7] = add 0 [2,5,2,7]
                = add (0+2) [5,2,7]
                = add (0+2+5) [2,7]
                = add (0+2+5+2) [7]
                = add (0+2+5+2+7) []
                = 0+2+5+2+7 = 16
```

Een opsparende parameter hoeft niet per se een getal te zijn, maar kan ook bijvoorbeeld een lijst zijn.

4.4 Practicum opdrachten

Opgave 1
Definieer een functie `remove n xs` die alle voorkomens van n uit xs verwijdert.

```
> remove 3 [4,3,5,6,3,9]
[4,5,6,9]
```

Definieer ook een functie `removeOnce n xs` die alleen het eerste voorkomen van n uit xs verwijdert.

```
> removeOnce 3 [4,3,5,6,3,9]
[4,5,6,3,9]
```

Opgave 2
Definieer een functie `subset xs ys` die True teruggeeft als xs een deelverzameling is van ys. Maak hierbij gebruik van de gedefinieerde functie `elem x xs` die vaststelt of een element in een lijst voorkomt.

```
> subset [1,4,2] [1,2,3,5,4]
True
```

Opgave 3
Definieer de functie `myLast` die het laatste element van een niet lege lijst teruggeeft.

Opgave 4
Definieer een functie `langstebeginplateau xs` die van een lijst het langste beginstuk geeft van elementen die gelijk zijn aan het eerste element.

```
> langstebeginplateau [1,2,3]
[1]
> langstebeginplateau [1,1,1,1,2,3]
[1,1,1,1]
```

Opgave 5
Definieer de functies `bintonum` en `numtobin` voor conversie tussen decimale en binaire getallen. Een binair getal wordt gerepresenteerd door een lijst van nullen en enen.

Opgave 6
Gegeven is dat de functie `insert` een getal op de juiste plaats in een gesorteerde lijst invoegt:

```
> insert 3 [1,2,5,6,7,8]
[1,2,3,5,6,7,8]
> insert 3 []
[3]
```

Geef zelf een recursieve definitie van insert. Vul hiertoe onderstaande definitie aan:

```
insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = ..
```

We kunnen een lijst nu sorteren door te beginnen met een lege lijst er daar één voor één de elementen van de te sorteren lijst in te inserteren. Vul nu de onderstaande definitie weer aan:

```
isort :: (Ord a) => [a] -> [a]
isort [] = []
isort (x:xs) = .. (isort xs)
```

Opgave 7
Mergen is het samenvoegen van twee gesorteerde lijsten tot één gesorteerde lijst.

```
> merge [1,3,5,6,8,9] [2,4,5,6,8]
[1, 2, 3, 4, 5, 5, 6, 6, 8, 8, 9]
```

Voltooi de onderstaande definitie van merge:

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge xs [] = xs
merge [] xs = xs
merge (x:xs) (y:ys) = if x < y then ...
                               else ...
```

Bij msort wordt de lijst steeds in tweeën verdeeld waarna de twee helften worden gesorteerd en daarna gemerged. Het sorteren van de twee helften gebeurt ook weer door deze in tweeën te verdelen en daarna de helften weer te sorteren en te mergen. Dit in tweeën delen gaat net zolang door tot de overgebleven lijsten lengte 0 of 1 hebben gekregen. Aan dit soort lijsten valt natuurlijk niet veel te sorteren. Voltooi nu de onderstaande definitie van msort:

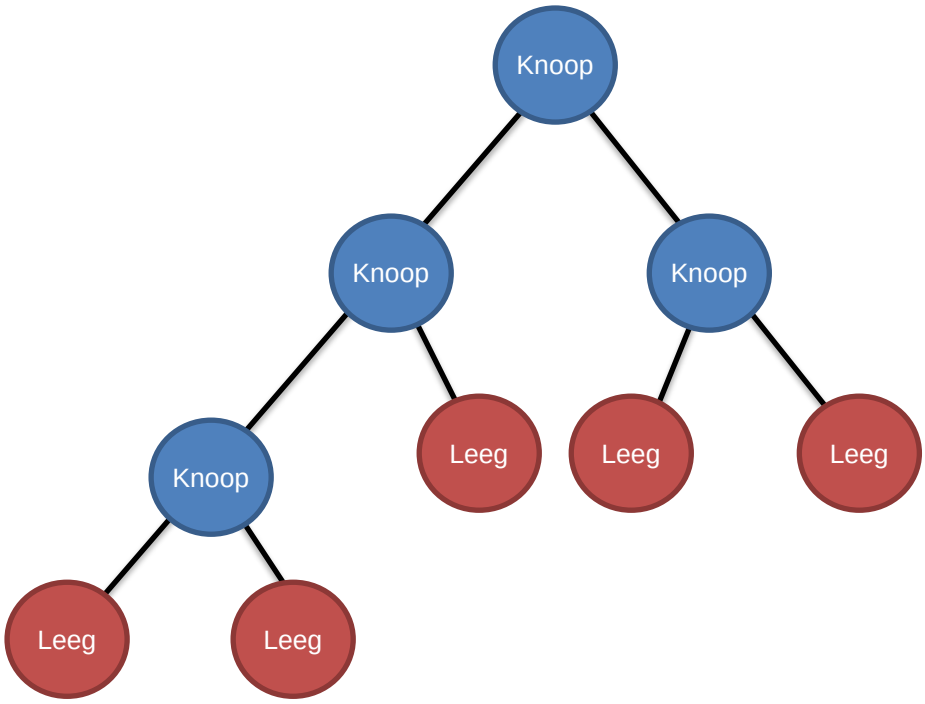
```
msort [] = []
msort [x] = [x]
msort xs = … (take m xs) … (drop m xs)
  where m = …
```

5 Recursieve datastructuren

Binnen Haskell is het mogelijk om naast de bekende datastructuren ook eigen datastructuren te definiëren. Deze functionaliteit kunnen we gebruiken om onze eigen recursieve datastructuur te definiëren. Recursieve datastructuren kennen we al van linked list, maar de structuur die we hier gaan behandelen is de veelvoorkomende binaire boomstructuur.

5.1 Binaire boom

Een binaire boom is een veelgebruikte datastructuur binnen Informatica, en kan bijvoorbeeld tegengekomen worden in de gebieden van graphics of algoritmiek.
Een boom bestaat uit 2 verschillende dingen, knopen en leeg. En iedere knoop heeft weer 2 subbomen onder zich. In figuur 1 staat hiervan een voorbeeld.



Figuur 1: Voorbeeld van een binaire boom

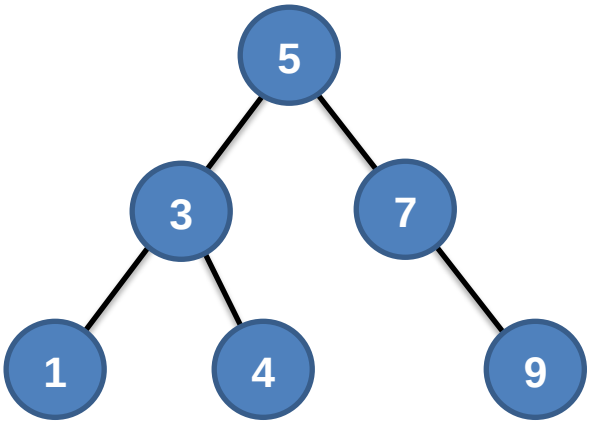
In de meeste gevallen worden de rode, lege, cirkels niet getekend om de boom overzichtelijker te maken.
In een binaire boom bevatten de knopen data.

5.2 Binaire zoekboom

Een binaire zoekboom is een speciale binaire boom. In deze boom definiëren we voorwaardes waaraan item moeten voldoen die in de linker en rechter subboom terecht komen. De meest gebruikte regel is als volgt:

- De waardes in de linker subboom hebben een waarde kleiner dan de knoop
- De waardes in de rechter subboom hebben een waarde groter dan de knoop

De meeste binaire zoekbomen bevatten hierdoor ook geen dubbele waarden.



Figuur 2: Voorbeeld van een binaire zoekboom

5.3 Datastructuur Tree

Zoals begin van dit hoofdstuk vermeld staat, is het mogelijk om een eigen recursieve datastructuur op te zetten. Voor een binaire boom gevuld met integers in de knopen (in het Engels “nodes”) kunnen we de volgende definitie aanhouden:

```
data Tree = Node Int Tree Tree | Empty deriving (Show)
```

We zien hier dat we de datastructuur met het keyword data beginnen en daarna een naam geven (altijd met een hoofdletter). Daarna zien we de definitie van de Node (knoop) met hierbij de data (Int) en 2 subbomen. Ook zien we dat een Tree ook Empty kan zijn, waarbij Empty geen data bevat.
Hierna staat nog vermeld deriving (Show). Dit is zodat we de data ook op het scherm kunnen tonen.

5.4 Recursieve functies op bomen

De in de vorige paragraaf gemaakte datastructuur kunnen we nu gaan gebruiken om functies te maken. Zo kunnen we een functie maken die van een lijst met getallen een binaire zoekboom maakt.

```
buildTree :: [Int] -> Tree
buildTree [] = Empty
buildTree [x] = Node x Empty Empty
buildTree (x:xs) = Node x (buildTree [y | y<-xs, y < x])
                          (buildTree [y | y<-xs, y > x])
```

Nu we een functie hebben die een lijst in een boom kan omzetten, kunnen we simpele een functie maken die een boom weer converteert naar een lijst. Dit kunnen we doen op de zogenaamde inorder manier. Dit betekend dat we eerst de elementen in de linker subboom afdrukken, daarna de waarde van de knoop en als laatste de elementen van de rechter subboom.

```
inorder :: Tree -> [Int]
inorder Empty = []
inorder (Node g l r) = inorder l ++ [g] ++ inorder r
```

Zodra we deze inorder functie uitvoeren op een binaire zoekboom zien we een gesorteerde lijst. Dit is een van de voordelen van een zoekboom boven een lijst. Een binaire zoekboom zal namelijk altijd gesorteerd zijn wanneer deze inorder wordt weergegeven.

5.4.1 Toevoegen aan een binaire zoekboom

Om een binaire zoekboom in stand te houden moeten we nieuwe elementen op de juiste positie in de boom toevoegen. Hiervoor kunnen we een functie maken.

```
addTree :: Tree -> Int -> Tree
addTree Empty n = Node n Empty Empty
addTree (Node g l r) n = if n < g then Node g (addTree l n) r
                                  else if n > g then Node g l (addTree r n)
                                                else Node g l r
```

We zien hier dat per knoop wordt gekeken of dat het toe te voegen getal groter of kleiner is dan de waarde in de knoop en dan wordt daarna alleen het getal toegevoegd aan de juiste subboom. Zoals eerder gezegd kunnen dubbele waardes niet voorkomen, bij het toevoegen van een getal dat al in de boom staat zal er dus niets met de boom gebeuren.
Door dit mechanisme is toevoegen aan een binaire zoekboom veel sneller dan toevoegen aan en gesorteerde lijst.

5.4.2 Guards

In de voorgaande paragraaf hebben we de addTree functie gezien. Deze heeft een geneste if structuur. Om deze structuur beter leesbaar te maken zijn binnen Haskell guards beschikbaar. Dit is te vergelijken met het switch case statement van C#. Hieronder is de functie addTree2 te zien, welke is gebaseerd op addTree maar waar de geneste if is vervangen door guards.

```
addTree2 :: Tree -> Int -> Tree
addTree2 Empty n = Node n Empty Empty
addTree2 (Node g l r) n | n < g = Node g (addTree2 l n) r
                        | n > g = Node g l (addTree2 r n)
                        | otherwise = Node g l r
```

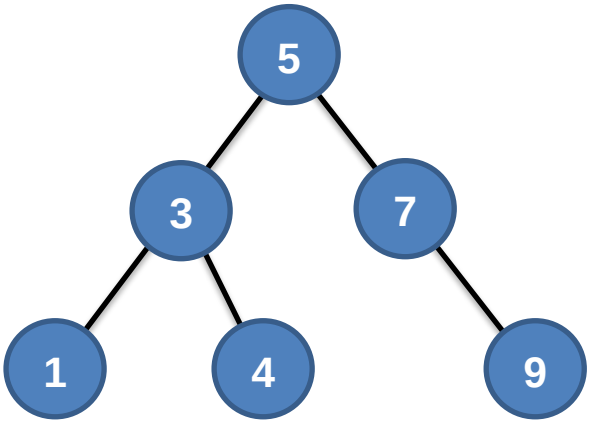
We kunnen zien dat de initiele = is vervangen door |, end at na iedere case nu = staat.
Zo kunnen we diverse cases mooi onder elkaar zetten zonder gebruik te hoeven maken van de if … then … else structuur. Het otherwise keywoord mag hierbij ook gebruikt worden en deze staat voor in alle niet hierboven genoemde gevallen. Otherwise zal dus altijd onderaan komen.

5.5 Practicum opdrachten

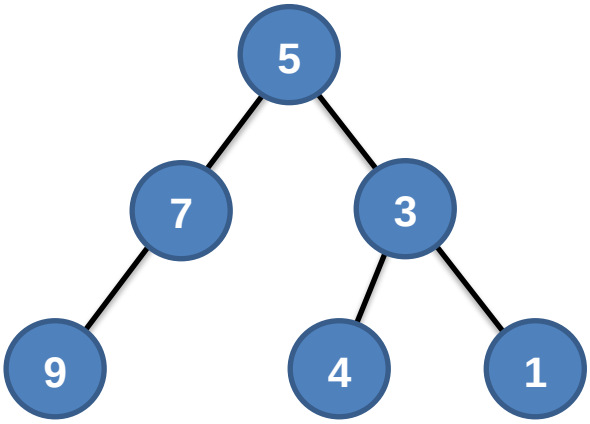
Opdracht 1
Maak de functie `depth` die in een binaire zoekboom op een efficiënte manier vaststelt op welke diepte een cijfer voorkomt. Maak hierbij gebruik van de Tree die in voorgaand hoofdstuk is gedefinieerd. Indien een getal niet voorkomt in de boom, moet -1 teruggegeven worden.
Gegeven is de definitie voor deze functie.

```
depth :: Int -> Tree -> Int
```

Opdracht 2
Maak de functie `spiegelboom` die een boom spiegelt. Een voorbeeld van een boom en diens spiegeling:



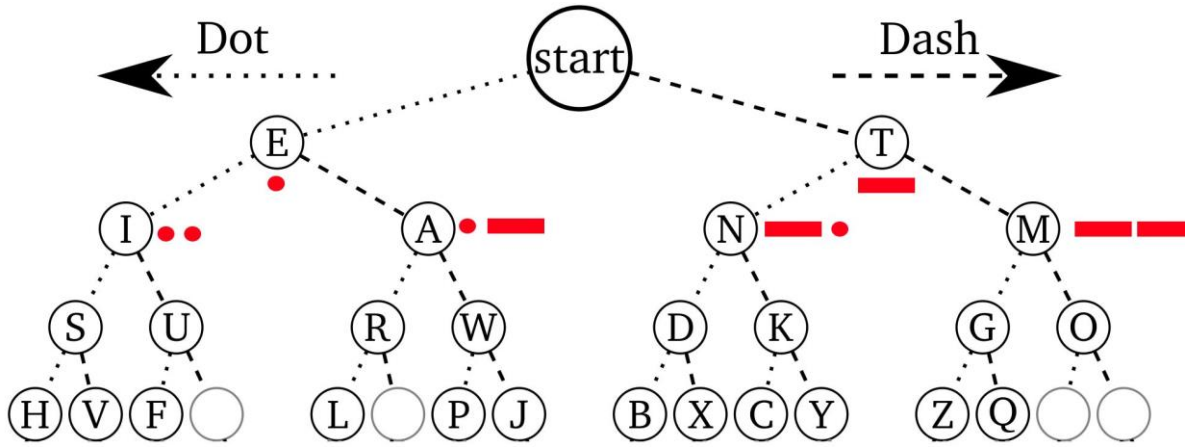
Figuur 3: Originele boom



Figuur 4: Gespiegelde boom

Opdracht 3
Maak een eigen datastructuur boom met daarin chars i.p.v. ints. Noem deze datastructuur CharTree.

Opdracht 4
Vul je eigengemaakte CharTree met alle karakters van het alfabet. Doe dit op zo'n manier dat de boom snel morsecodes kan coderen en decoderen (zie onderstaande afbeelding).



Opdracht 5
Maak de methode `charCount :: CharTree -> Int` die telt hoeveel karakters in een boom zitten. (Voer je dit uit op je morseboom, dan moet er uiteraard 26 uit komen).

Opdracht 6
Maak de methode `morseToChar` die een morsestring om kan zetten naar het juiste karakter.

Opdracht 7
Maak de methode `charToMorse` die de morsestring teruggeeft die hoort bij een meegegeven karakter. (Test dit door `morseToChar` aan te roepen met het resultaat van `charToMorse`).

Opdracht 8
Maak de methode `nameToMorse` die een string omzet naar een morsestring door alle afzonderlijke tekens om te zetten naar morsetekens. Wat is jouw naam in morse?

6 Tentamen

Tijdens het tentamen mogen de volgende hulpmiddelen gebruikt worden:

- Dit dictaat
- Powerpoint sheets (uitgeprint)
- Literatuur
- Aantekeningen
- (Grafische) rekenmachine

Communicatiemiddelen zijn echter verboden.