Logisch en Functioneel Programmeren voor Wiskunde D

Wouter Swierstra
Doaitse Swierstra
Jurriën Stutterheim

Technical Report UU-CS-2011-033
Sept 2011

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Inhoudsopgave

1 Inleiding **3**

2 Achtergrond **5**

3 Prolog **11**

4 Hoe werkt Prolog? **25**

5 Een Prolog interpretator in Haskell **33**

6 Beschouwing **39**

Hoofdstuk 1

Inleiding

Iedereen gebruikt computers. Of je nou je e-mail leest, over het web surft, een opstel schrijft, of je muziekcollectie afspeelt—daarvoor gebruik je tegenwoordig een computer. Maar hoe werkt een computer nou? En hoe worden de programma's die je gebruikt ontwikkeld?

Een computer is eigenlijk een slimme rekenmachine. Net als een rekenmachine, kan een computer opdrachten razendsnel uitvoeren. Maar een rekenmachine kan maar één kunstje: heel snel rekenen. Computers, daarentegen, kan je juist nieuwe kunstjes leren. *Programmeren* is het schrijven van opdrachten voor een computer zodat deze nieuwe soorten 'berekeningen' kan uitvoeren. Zo'n 'berekening' kan van alles zijn: van e-mail versturen tot foto's bewerken. Alle programma's die op jouw computer draaien, van je webbrowser tot je tekstverwerker, zijn ooit door een programmeur (of een team van programmeurs) geschreven.

Een programmeur geeft opdrachten aan de computer door middel van een *programmeertaal*. Er bestaan heel veel verschillende programmeertalen. De meeste daarvan zijn gebaseerd op het concept van een "commando": een programma bestaat uit een rij commando's. We noemen dergelijke talen dan ook de *imperatieve* talen (naar het Latijnse werkwoord "imperare", wat iets betekent als "commanderen" of "opdrachten geven"). De meeste talen waar je misschien al wel eens van hebt gehoord, zoals Java, C#, PHP, of Javascript, zijn zulke imperatieve talen.

Naast de imperatieve talen bestaan er ook *declaratieve programmeertalen*. In plaats van een serie opdrachten die de computer moet uitvoeren, *beschrijven* programma's in declaratieve programmeertalen de oplossing van een probleem in termen van functies en relaties. In deze cursus maak je kennis met twee veel gebruikte declaratieve programmeertalen: Prolog en Haskell.

Prolog is een *logische programmeertaal*. Prolog heeft als groot voordeel dat het een vrij *eenvoudige* taal is. Ook al heb je nog nooit eerder geprogrammeerd en weet je niet veel van computers, dan kan je toch wel snel veel leren over Informatica door Prolog te bestuderen.

Haskell is een *functionele* programmeertaal. Haskell is een veel complexere taal dan Prolog, waar je veel meer mee kan. Zo zullen wij hoe je een Haskell programma kan schrijven die een Prolog programma uitvoert. Zo'n programma die een ander programma uitvoert heet ook wel een *interpretator*. Daarmee slaan we twee vliegen in één klap: je krijgt precies te zien hoe een een Prolog interpretator werkt en je leert meer over Haskell.

Naast een Prolog en Haskell kom je in deze cursus ook in aanraking met een aantal belangrijke begrippen uit de Informatica, zoals "zoeken", "bewijzen", "unificatie", en "substitutie".

Al met al is er behoorlijk wat materiaal om door te werken. Als je nog geen ervaring hebt met programmeren, is het misschien verstandig om je tot de eerste vier hoofdstukken te beperken, die alleen over Prolog gaan. Heb je al wel vaker geprogrammeerd, dan kan je de eerste hoofdstukken misschien wat sneller doorwerken. Zo houd je tijd over voor het laatste hoofdstuk over Haskell.

Hoe dan ook, heb je aan het eind van deze cursus een beter beeld van wat Informatica is en hoe leuk programmeren nou eigenlijk kan zijn!

Hoofdstuk 2

Achtergrond

Voordat we op Prolog zelf in gaan, geven we eerst wat achtergrondinformatie over de begrippen *vergelijking* en *substitutie*. We doen dit eerst in termen van de bekende vergelijkingen met getallen en variabelen, zoals je die al kent uit de wiskunde vakken. Later generaliseren we dit, zodat we begrippen als *relaties* en *termen* kunnen definiëren zoals die in de Prolog wereld voorkomen.

2.1 Vergelijkingen oplossen

Als we vragen wat de oplossing van de vergelijking

$$3 + x = 8$$

is, dan zal je waarschijnlijk antwoorden $x = 5$. En als we dan vragen wat de oplossing van de vergelijking

$$5 = x$$

is, dan zal je waarschijnlijk ook antwoorden $x = 5$. En als we dan vragen wat de oplossing van de vergelijking

$$x = 5$$

is, dan kan je niet anders antwoorden dan $x = 5$. Maar is dat niet een beetje vreemd? Waarom is dan de oplossing van de vergelijking $5 = x$ niet gewoon $5 = x$ en van $3 + x = 8$ gewoon $3 + x = 8$? Kennelijk zijn niet alle vergelijkingen gelijk: sommigen vergelijkingen noemen we oplossingen en anderen kennelijk niet.

Deze verwarring vindt zijn oorsprong in de gekozen notatie: traditioneel noteren we een oplossing van een vergelijking ook als een vergelijking, waarbij links van het =-telken één enkele variabele staat en rechts van het =-teken een waarde. De notatie $x = 5$ is dus verwarrend: het kan zowel een vergelijking zijn als een oplossing. Hadden we de oplossingen van de vergelijkingen allemaal genoteerd als $x \leftarrow 5$, dan was de verwarring niet ontstaan. We spreken $x \leftarrow 5$ uit als *vervang x door* 5 of als *substitueer 5 voor x*. Omdat we nu helemaal geen =-teken meer gebruiken om een oplossing aan te duiden, kan er dus niet langer verwarring onstaan. We zullen voortaan alle oplossingen op deze manier representeren. Een uitdrukking als $x \leftarrow 5$ noemen we een substitutie. Iedere *substitutie* beschrijft dus de systematische vervangingen van variabelen door waarden.

Zijn alle substituties nu ook oplossingen? We kijken eens naar de substitutie $x \leftarrow 4$ en de vergelijking $3 + x = 8$. Als we nu de substitutie loslaten op de vergelijking, dan krijgen we $3 + 4 = 8$, en als we dan beide kanten uitrekenen zien we dat daar niet hetzelfde uit komt. Dus lang niet alle substituties zijn oplossingen van een vergelijking.

Wanneer is een substitutie dan wel een oplossing voor een vergelijking? We noemen een substitutie een *oplossing voor een vergelijking* als na het toepassen van de substitutie beide kanten van de vergelijking dezelfde waarde beschrijft. Zo is $x \leftarrow 5$ dus wel een oplossing voor de vergelijking $3 + x = 8$, maar $x \leftarrow 4$ niet.

Opgave 1. Reken uit wat het resultaat is van het toepassen van de substitutie $x \leftarrow 5$ op de volgende vergelijkingen:

1. $x + 7 = 12$
2. $x + x = 8$
3. $x + 12 = 3\,x + 2$
4. $3 = 3$
5. $x^2 = 25$
6. $y + 12 = 13$

Voor welk van deze vergelijkening is $x \leftarrow 5$ een oplossing?

2.2 Eigenschappen van substituties

Hoe veel oplossingen heeft een vergelijking? Heeft iedere vergelijking precies één oplossing? Nee. Kijk maar naar de vergelijking $x^2 + 1 = 0$. Er is geen enkel getal voor x te bedenken die deze vergelijking oplost. Vergelijkingen kunnen ook heel veel oplossingen hebben. Een vergelijking als $x^2 = x$ heeft twee oplossingen: $x \leftarrow 0$ én $x \leftarrow 1$ zijn allebei oplossingen voor deze vergelijking.

Kort samengevat kunnen we dus stellen dat niet iedere vergelijking een oplossing heeft en als een vergelijking wel oplossingen heeft, hoeven deze niet uniek te zijn. Dit lijkt misschien een nogal voor de hand liggende observatie, maar later zullen we zien dat dit toch van belang is.

Meerdere onbekenden Iedere vergelijking heeft een aantal "onbekenden" of *variabelen*. Tot dusver hebben we ons beperkt tot vergelijkingen met daarin hooguit één onbekende, x. Maar niet alle vergelijkingen hebben maar één onbekende. Je kent misschien de volgende formule uit de natuurkundelessen:

$$F = m \cdot a$$

In deze formule staan meerdere onbekenden. Zo zijn a, F, en m allemaal variabelen, waarvoor je ook iets kunt substitueren. We kunnen deze vergelijking op een aantal verschillende manieren gebruiken: kennen we de kracht F op een object met massa m, dan weten we dat de versnelling $F\ /\ m$ is. Omgekeerd weten we dat als een object met massa m versnelt met een versnelling a, er kennelijk een kracht $m \cdot a$ op werkt.

We kunnen deze vergelijking ook als een relatie zien tussen kracht, massa, en versnelling: als we de waarde van twee van de onbekenden weten, kunnen we derde uitrekenen. Deze interpretatie van vergelijkingen als relaties ligt ten grondslag aan Prolog.

Net zoals vergelijkingen meer dan één variabele kunnen bevatten, kunnen substituties ook meer dan één variabele vervangen. We schrijven dan bij voorbeeld $x \leftarrow 5, y \leftarrow 3$ voor de substitutie die *eerst* x door 5 vervangt en daarna y door 3 vervangt. Als we duidelijk willen maken waar een substitutie begint en eindigt, zullen we soms vierkante haken om een substitutie zetten en dus $[x \leftarrow 5, y \leftarrow 3]$ schrijven.

Oplossingen voor vergelijkingen met meerdere onbekenden zijn in het algemeen substituties die meerdere variabelen vervangen. Zo is de substitutie $[x \leftarrow 5, y \leftarrow 5]$ een oplossing van de vergelijking:

$$x + y = 10$$

Hier hebben we dus te maken met een oplossing die twee variabelen substitueert.

Meerdere vergelijkingen De vergelijking $x + y = 10$ heeft een groot aantal oplossingen; en ook de vergelijking $x - y = 2$ heeft een hoop oplossingen. Is er ook een substitutie te bedenken die een oplossing is van *beide* vergelijkingen? Ja natuurlijk! De substitutie $[x \leftarrow 6, y \leftarrow 4]$ is zo'n oplossing.

Er is dus niets op tegen om het begrip oplossing uit te breiden tot een verzameling van vergelijkingen. De oplossingen van een verzameling vergelijkingen is een substitutie, die ook een oplossingen is voor elk van de afzonderlijke vergelijkingen.

Equivalente substituties We hebben al gezien dat sommige vergelijkingen meer dan één oplossing hebben. Zijn al die oplossingen dan echt ook verschillend? Of zijn ze allemaal hetzelfde?

Kijk bijvoorbeeld naar de substituties $[x \leftarrow 3, y \leftarrow 4]$ en $[y \leftarrow 4, x \leftarrow 3]$. Dit zijn allebei oplossingen van de vergelijking $x + 1 = y$. Maar het maakt natuurlijk niet uit op welke volgorde je x en y vervangt door 3 en 4. Deze substituties gedragen zich precies hetzelfde als ze toegepast worden op een vergelijking. We noemen ze daarom *equivalent*. In het algemeen, zijn twee substituties *equivalent* als voor iedere vergelijking, allebei de substituties toepassen op die vergelijking hetzelfde resultaat geeft.

Natuurlijk zijn niet alle substituties equivalent. Kijk naar de vergelijking $x = y$. Het toepassen van de substituties $[x \leftarrow 1, y \leftarrow 2]$ en $[x \leftarrow 3, y \leftarrow 3]$ levert dan ook echt twee andere vergelijkingen op: $1 = 2$ en $3 = 3$. Deze substituties zijn dan ook niet equivalent.

Minimale oplossing Je kan je afvragen of de substitutie $[x \leftarrow 5, y \leftarrow 5]$ ook een oplossing van de vergelijking $x = 5$. Een moment van reflectie doet ons inzien dat het systematisch vervangen van alle y's, op zich geen kwaad kan; ze komen immers helemaal niet voor in de vergelijkingen. De substitutie maakt nog steeds beide kanten aan elkaar gelijk. Dus zeggen we dat deze substitutie ook echt een oplossing is. Om praktische redenen proberen we natuurlijk om geen overbodige ballast mee te nemen in onze oplossingen; het doet geen kwaad, maar we hebben er ook niks aan. We zullen ons dus bij het oplossen van vergelijkingen vooral richten op het vinden van zogenaamde *minimale oplossingen*. Een substitutie is een *minimale oplossing van een vergelijking* als je niet meer een te vervangen variabele weg kan laten, zonder dat het daarna geen oplossing meer is van die vergelijking.

Symbolische substituties We kijken nog eens even naar de vergelijking $x = y$. Wat voor oplossingen heeft deze vergelijking? De substitutie $[x \leftarrow 1, y \leftarrow 1]$ is een oplossing. Maar $[x \leftarrow 2, y \leftarrow 2]$ is dat ook. En zo zijn er natuurlijk oneindig veel van zulke oplossingen te bedenken.

Het is duidelijk dat als we eisen dat er aan de rechterkant van de $\leftarrow$ in een substitutie een getal moet staan, we niet alle oplossingen in eindige tijd kunnen opsommen. Daarom staan we vanaf nu ook toe dat variabelen door een expressie worden vervangen. Zo is $[x \leftarrow y]$ dan een oplossing van de vergelijking hierboven: als we deze substitutie toepassen, krijgen we de vergelijking $y = y$. Aangezien dit een triviale vergelijking is die door elke substitutie wordt waargemaakt, mogen we afleiden dat $[x \leftarrow y]$ ook een oplossing is.

Meer dan alleen getallen Tot dusver zijn we er vanuit gegaan dat je om te kijken of twee kanten van een vergelijking gelijk zijn je nog wel even een beetje mag rekenen. Voor eenvoudige expressies is het makkelijk om even uit je hoofd te bepalen of er links en rechts hetzelfde staat. Maar zijn $2934 \times 917 + 528$ en 2691000 nou hetzelfde of niet? Dat 8 en 8 hetzelfde zijn en dat $3 + 5$ en $3 + 5$ hetzelfde zijn is wellicht nog wel acceptabel. We gaan er vanaf het volgend hoofdstuk van uit dat we niet stiekem nog snel even wat uitrekenen als we twee dingen vergelijken: twee expressies zijn alleen gelijk als ze letter voor letter gelijk aan elkaar zijn en dus niet als de waarden die we krijgen als we de expressies uitrekenen gelijk zijn. We laten daarom de gehele getallen nu achter ons en gaan eens kijken naar een heel ander soort vergelijkingen.

2.3 Opgaven

Opgave 2. Hoe veel oplossingen hebben de volgende vergelijkingen?

1. $x + 3 = 12$
2. $\sqrt{x} = \sqrt{-x}$
3. $x \times 0 = 0$
4. $x + y = y$

Opgave 3. Wat is het resultaat van het toepassen van de volgende substituties op de vergelijking $x + y = z$? Welk van deze substituties is een oplossing?

1. $[x \leftarrow 0, y \leftarrow 3, z \leftarrow 3]$
2. $[x \leftarrow 0, y \leftarrow z]$
3. $[x \leftarrow y, y \leftarrow z, z \leftarrow 3]$
4. $[z \leftarrow 3, x \leftarrow y, y \leftarrow z]$

Opgave 4. Welk van deze paren van substituties zijn equivalent?

1. $[x \leftarrow 3]$ en $[y \leftarrow 3]$
2. $[x \leftarrow y]$ en $[y \leftarrow x]$

3. $[x \leftarrow 3, y \leftarrow 7]$ en $[y \leftarrow 7, x \leftarrow 3]$
4. $[x \leftarrow y, y \leftarrow 3]$ en $[x \leftarrow 3, y \leftarrow x]$

Opgave 5. Welk van de volgende substituties is een minimale oplossing voor de gegeven vergelijking?

1. $[x \leftarrow 5]$ voor de vergelijking $x + 3 = 8$;
2. $[y \leftarrow 5]$ voor de vergelijking $x + 3 = 8$;
3. $[x \leftarrow 5, y \leftarrow 5]$ voor de vergelijking $x + 3 = 8$;
4. $[x \leftarrow y, y \leftarrow 5]$ voor de vergelijking $y + 3 = 8$;

Hoofdstuk 3

Prolog

In dit hoofdstuk introduceren we Prolog, een logische programmeertaal. Prolog is oorspronkelijk ontworpen in 1972 door Alain Colmerauer. De naam Prolog is een samentrekking van "PROgrammation en LOGique". Oorspronkelijk is Prolog ontworpen om computers te leren natuurlijke talen, zoals Nederlands en Engels, te begrijpen en te specificeren hoe zinnen vertaald kunnen worden. Als gevolg hiervan wordt Prolog nog altijd veel gebruikt in gebieden zoals de kunstmatige intelligentie. Maar Prolog wordt ook gebruikt in de besturingssoftware van de N9 generatie mobiele telefoons van Nokia.

We kiezen als eerste programmeertaal om te bestuderen voor Prolog omdat de programmeertaal heel *eenvoudig* is en maar een paar verschillende taalconstructies kent. Toch kunnen we aan de hand hiervan belangrijke begrippen uit de Informatica kunnen introduceren. Ook al worden programmeertalen als C# en Java tegenwoordig meer gebruikt door bedrijven, toch kan het leerzaam zijn om te beginnen met het bestuderen van een programmeertaal als Prolog.

3.1 Inleiding Prolog

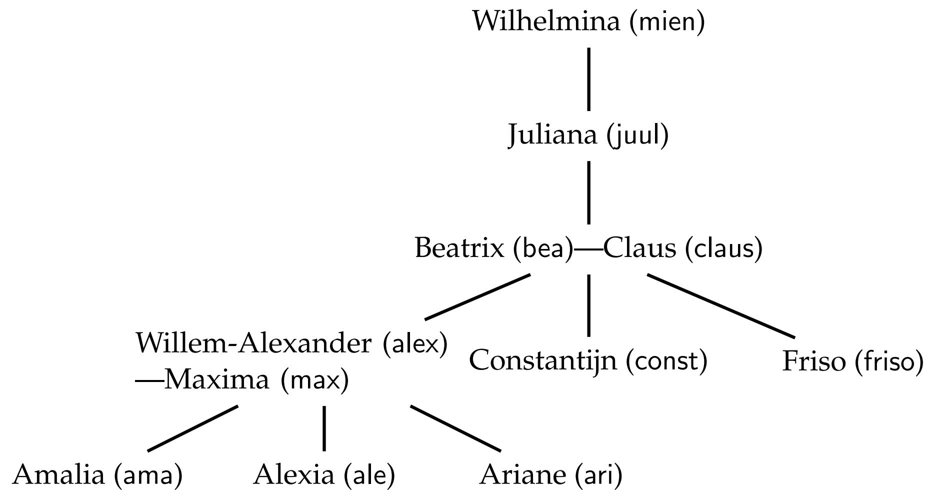
In Prolog beschrijf je *relaties* tusen waarden. Als eerste voorbeeld zullen we laten zien hoe we in Prolog verschillende familierelaties kunnen beschrijven. Nadat je wat meer in Prolog hebt kunnen programmeren, zullen we in de latere hoofdstukken meer vertellen over de achterliggende theorie en de implementatie van de Prolog interpretator die we in deze cursus gebruiken.

Basis waarden en relaties

In het vorige hoofdstuk hebben we vooral gekeken naar vergelijkingen tussen getallen. Nu gaan we verbanden tussen leden van (een deel van) het koningshuis bestuderen. Om een hoop overbodig tikwerk te voorkomen korten we de namen van de personen af. Dat is misschien niet erg eerbiedig, maar wel een stuk makkelijker.

Als waarden hebben we bijvoorbeeld bea, alex, max, ama, ale, ari, claus, juul en mien. Ook de broers van onze kroonprins mogen meedoen: const en friso. Voor diegenen die hier wat minder bekend mee zijn, hebben we in Figuur 3.1 een (stukje) van de stamboom getekend.

Zo kunnen we, om te beginnen, een relatie definiëren die beschrijft dat persoon X de vader is van persoon Y, wat we zullen noteren als $pa\ (X, Y)$. Aan de hand van de stamboom



Figuur 3.1: Nederlandse koningshuis

in Figuur 3.1, kunnen we de volgende opsomming van vaders en kinderen maken:

pa (alex, ama).
pa (alex, ale).
pa (alex, ari).
pa (claus, alex).
pa (claus, const).
pa (claus, friso).

Op dezelfde manier kunnen we ook de moeder-relatie is definiëren:

ma (max, ama).
ma (max, ale).
ma (max, ari).
ma (bea, alex).
ma (bea, const).
ma (bea, friso).
ma (juul, bea).
ma (mien, juul).

Hiermee hebben we onze eerste Prolog relaties gedefineerd. Merk op dat we met deze relaties precies alle informatie uit het plaatje hebben gerepresenteerd. Als je het plaatje kwijt bent, kun je het toch reconstrueren uit de *pa* en *ma* relaties

Gegeven deze basisfeiten kunnen we nu 'vergelijkingen' formuleren:

pa (alex, X)

Zulke vergelijkingen zullen we voortaan een *doel* noemen.

Zoals we bij de vergelijking $3 + x = 8$ ons de vraag stelden voor welke waarde van x de beide kanten van het $=$-teken daadwerkelijk gelijk waren, vragen we ons nu af voor welke waarde(n) van X, alex in de relatie *pa* tot X staat—oftewel van wie is alex allemaal de vader? De volgende drie substituties zijn nu allemaal oplossingen: $X \leftarrow$ ama; $X \leftarrow$ ale; en $X \leftarrow$ ari.

Om een duidelijk onderscheid te maken tussen namen van personen en variabelen gebruiken we voortaan af aan hoofdletters voor variabelen en kleine letters voor de namen van personen. In ons doel pa (alex, X), staat X voor een variabele, maar alex dus niet. De namen van een relatie zullen we cursief schrijven, de namen van personen schrijven we schreefloos. Zo kan je meteen zien dat *pa* en relatie is en alex een persoon.

Prolog vindt automatisch oplossingen voor zulke doelen. Als we de relaties *pa* en *ma* definiëren in een bestand, deze inladen in de Prolog interpretator die we verderop zullen beschrijven, kunnen als volgt vragen naar oplossingen (en alle oplossingen krijgen als antwoord op onze vraag):

```
goals? pa(alex, X)
X <- ama
X <- ale
X <- ari
```

Natuurlijk heeft niet iedere vergelijking een oplossing. Aangezien Amalia geen kinderen heeft, zijn er geen oplossingen voor het doel ma (ama, X). Als we onze Prolog interpretator vragen naar oplossingen krijgen als antwoord dat er geen oplossingen bestaan:

```
goals? ma(ama,X)
No solutions found.
```

We kunnen ook ingewikkeldere doelen formuleren, die gebruik maken van verschillende variabelen. Als we willen weten wie de ouders van ama zijn, moeten we vragen om de volgende twee doelen tegelijkertijd op te lossen:

```
goals? pa(X,ama), ma(Y,ama)
X <- alex, Y <- max
```

Zo leren we dus dat alex en max respectievelijk de vader en moeder van ama zijn.

Opgave 6. Vertaal de volgende vragen naar een of meerdere Prolog doelen:

1. Wie is de vader van Willem-Alexander?
2. Wie zijn de kinderen van Beatrix en Claus?
3. Wie zijn de kinderen van Maxima en Constantijn?
4. Wie zijn de grootouders van Alexia?

Welke van deze doelen hebben een oplossing?

Opgave 7. Bedenk alle oplossingen voor de volgende doelen (als die bestaan):

1. ma (X, ari)
2. pa (ama, X)
3. ma (bea, X)
4. ma (bea, X), pa (X, Y)

Nieuwe relaties definiëren

Stel we willen een nieuwe relatie definiëren, $\mathit{ouder}\ (X, Y)$. We zouden opnieuw een lange opsomming kunnen maken van ouders en hun kinderen:

$\mathit{ouder}\ (\mathsf{alex}, \mathsf{ama})$.
$\mathit{ouder}\ (\mathsf{max}, \mathsf{ama})$.
...

Maar dat schiet niet zo heel erg op.

Het is veel handiger om de *ouder* relatie te definiëren met behulp van de *pa* en *ma* relaties die we al hebben gezien. Zo geldt bijvoorbeeld dat als X een *pa* van Y is, hij ook een *ouder* van Y is. We formuleren een regel van de *ouder* relatie als volgt:

$\mathit{ouder}\ (X, Y) \text{ :- } \mathit{pa}\ (X, Y)$.

De eerste regel, $\mathit{ouder}\ (X, Y) \text{ :- } \mathit{pa}\ (X, Y)$, moet je lezen als: wanneer X de vader van Y is, dan is X een ouder van Y. Soms is het ook handig zo'n regel andersom te lezen: om te laten zien dat X een ouder van Y is, is het voldoende om te laten zien dat X de vader van Y.

Natuurlijk zijn ook alle moeders ouders. Om de *ouder* relatie volledig te definiëren hebben we een tweede regel nodig:

$\mathit{ouder}\ (X, Y) \text{ :- } \mathit{ma}\ (X, Y)$.

Deze twee regels samen definiëren een nieuwe relatie.

Met deze nieuwe relatie, kunnen we nu aan onze Prolog interpretator vragen om de ouders van ama uit te rekenen:

```
goals? ouder(X, ama)
X <- alex
X <- max
```

Nu we een *ouder* hebben is het natuurlijk een fluitje van een cent om de bijbehorende *kind* relatie te definiëren:

$\mathit{kind}\ (X, Y) \text{ :- } \mathit{ouder}\ (Y, X)$.

Voorouders Wat als we nu willen definiëren wanneer een persoon X een voorouder is van een persoon Y? We beginnen met het directe geval, als iemand een *ouder* is dan is deze ook een *voorouder*:

$\mathit{voorouder}\ (X, Y) \text{ :- } \mathit{ouder}\ (X, Y)$.

Deze regel zegt dat je ouders ook je voorouders zijn.

Maar je hebt natuurlijk veel meer voorouders! De ouders van je ouders, dus je opa's en oma's, zijn natuurlijk ook je voorouders. Dat zouden we als volgt kunnen beschrijven:

$\mathit{voorouder}\ (X, Y) \text{ :- } \mathit{ouder}\ (X, Z), \mathit{ouder}\ (Z, Y)$.

Zo'n samengestelde regel kunnen we als volgt lezen: *voor alle* personen X en Y geldt dat wanneer we een persoon Z kunnen vinden zodat X een ouder is van Z en bovendien Z ook een ouder van Y is, dan kunnen we vaststellen dat X een voorouder is van Y.

Zelfs met deze twee regels hebben we 'voorouder van' relatie nog niet helemaal te pakken. De ouders van je opa's en oma's zijn immers ook je voorouders. Om al iemand's voorouders op te sommen, zouden we heel veel regels kunnen definiëren van de vorm:

$$\mathit{voorouder}\ (X, Y) \text{:-}\ \mathit{ouder}\ (X, A), \mathit{ouder}\ (A, Y).$$
$$\mathit{voorouder}\ (X, Y) \text{:-}\ \mathit{ouder}\ (X, A), \mathit{ouder}\ (A, B), \mathit{ouder}\ (B, Y).$$
$$\mathit{voorouder}\ (X, Y) \text{:-}\ \mathit{ouder}\ (X, A), \mathit{ouder}\ (A, B), \mathit{ouder}\ (B, C), \mathit{ouder}\ (C, Y)$$
...

Maar dan hebben we 'oneindig veel' regels nodig. Dit is dus misschien niet zo'n hele goede aanpak.

Het is veel beter om een iets andere definitie te kiezen. Je kunt je eigen voorouders ook beschrijven als je *eigen* ouders en al *hun* de voorouders. In Prolog ziet dit er uit als:

$$\mathit{voorouder}\ (X, Y) \text{:-}\ \mathit{ouder}\ (X, Y).$$
$$\mathit{voorouder}\ (X, Y) \text{:-}\ \mathit{ouder}\ (Z, Y), \mathit{voorouder}\ (X, Z).$$

In woorden, kan je deze laatste regel dus als volgt uitleggen: *voor alle* X en Y geldt dat X een voorouder van Y is als *er een* Z *bestaat* die een ouder is van Y en zelf X als voorouder heeft.

Laten we eens uitproberen of deze definitie ook het verwachte resultaat produceert:

```
goals? voorouder(X, ama)
X <- alex
X <- max
X <- mien
X <- juul
X <- bea
X <- claus
```

Al deze mensen zijn inderdaad voorouders van Amalia—onze definitie van de voorouder relatie klopt kennelijk!

Recursie De voorouder relatie, zoals we die uiteindelijk hebben gedefineerd, heeft een bijzondere eigenschap. Om vast te stellen dat X een voorouder is van Y, maken we weer gebruik maken van de voorouder relatie zelf, die we nu juist aan het definiëren zijn. Zo'n relatie die gedefineerd is in termen van zichzelf noemen, noemen we *recursief*. Bij het geven van een recursieve definitie moet je wel voorzichtig zijn. Een definitie zoals:

$$\mathit{oeps}\ (X) \text{:-}\ \mathit{oeps}\ (Y).$$

is volkomen onzinnig. De Prolog interpretator loopt ook vast als hij een doel als *oeps* (X) moet oplossen. Immers om *oeps* (X) op te lossen, moet er een persoon Y worden gevonden waarvoor geldt dat *oeps* (Y)—maar om *oeps* (Y) te laten zien, moet er weer een persoon Z gevonden worden zodat *oeps* (Z), en zo voorts. De Prolog interpretator blijft zoeken, zonder dat er ooit een oplossing wordt gevonden. Wees dus altijd voorzichtig bij het definiëren van recursieve relaties.

Opgave 8. Vertaal de volgende vragen naar Prolog doelen:

1. Wie zijn de nakomelingen van Beatrix?
2. Wie zijn de voorouders van zowel Amalia als Willem-Alexander?
3. Welke voorouders van Amalia hebben kleinkinderen?

Opgave 9. Definieer een relatie *opa* (X, Y) die beschrijft wanneer persoon X een grootvader is van persoon Y.

Opgave 10. Definieer een relatie *oom* (X, Y) die beschrijft wanneer persoon X een oom is van persoon Y. Wat gaat er mis?

3.2 Programmeren in Prolog

Tot dusverre zijn de waarden die we gebruikt hebben allemaal leden van het koningshuis. Maar als je 'echt' wil programmeren, zal je toch ook ooit moeten werken met andere vormen van gegevens. Hoe kunnen we bijvoorbeeld met getallen rekenen in Prolog?

Natuurlijke getallen

Laten we beginnen door de *natuurlijke getallen* te definiëren. De natuurlijke getallen zijn de getallen $0, 1, 2, 3$, enz. Maar hoe definiëren we die in Prolog? Als we ze één voor één opsommen zijn we oneindig lang bezig. We gebruiken weer *recursie* om een oneindige verzameling van getallen te beschrijven. We doen dat in twee stappen:

- Nul is een natuurlijk getal. We zullen het engelse zero gebruiken om nul aan te duiden.
- Als we al een natuurlijk getal hebben, kunnen we er één bij optellen om een nieuw natuurlijk getal te maken. Dit zullen we schrijven als succ (n), waar n een natuurlijk getal is. De naam succ is een afkorting van het engelse 'successor'.

Hier zijn 0, 1 en 2 geschreven met deze notatie:

```
zero
succ (zero)
succ (succ (zero))
```

We noemen waarden zoals succ, zero, bea, en alex *constanten*. Er is wel een belangrijk verschil tussen succ en de andere constanten die we tot dusver hebben gezien. We zullen altijd succ toepassen op een ander natuurlijk getal. We zeggen daarom dat succ een *parameter* heeft.

Hoe definiëren we nieuwe relaties op onze natuurlijke getallen? Laten we eenvoudig beginnen met twee getallen optellen. We definiëren een nieuwe relatie *plus* (X, Y, Z) die aangeeft wanneer $X + Y$ gelijk is aan Z:

$$\textit{plus}\ (\mathsf{zero}, X, X).$$
$$\textit{plus}\ (\mathsf{succ}\ (X), Y, \mathsf{succ}\ (Z)) \text{ :- } \textit{plus}\ (X, Y, Z).$$

De eerste regel is makkelijk. Die zegt dat $0 + X = X$. De tweede regel is wat moeilijker. Als we regel van links naar rechts lezen, staat er dat om te laten zien dat succ $(X) + Y =$

succ (Z) geldt, is het voldoende om te laten zien dat $X + Y = Z$. Als $x + y = z$ dan $1 + x + y = 1 + z$.

De definitie van *plus* is iets anders dan de andere relaties die we tot nu toe hebben gezien. Een belangrijke uitbreiding die we hier maken is dat we in definities ook constanten met parameters toestaan. Dat zie je in de tweede regel van onze definitie van *plus*. Daar hebben we het over succ (X) en succ (Z).

Met deze twee regels kunnen we eenvoudige vergelijkingen oplossen:

```
goals? plus(zero,zero,Z)
Z <- zero
goals? plus(succ(zero),succ(succ(zero)),Z)
Z <- succ(succ(succ(zero)))
```

Het zal niet als een verbazing komen dat $0 + 0 = 0$ en $1 + 2 = 3$.

Moeten we nou ook aftrekken definiëren? Op zich hoeft dat niet. We kunnen ook onze definitie van *plus* als volgt gebruiken:

```
goals? plus(X, succ(zero), succ(succ(zero)))
X <- succ(zero)
```

Hier vragen we dus aan Prolog om een oplossing voor X te vinden zodat $X + 1 = 2$. Als antwoord krijgen we te horen dat X gelijk moet zijn aan 1.

Als we niet voorzichtig zijn, kunnen we soms wel erg veel oplossingen krijgen voor een bepaald doel. Als we vragen naar oplossingen van de vergelijking $X + 1 = Z$, zijn dat er natuurlijk oneindig veel:

```
goals? plus(X, succ(zero), Z)
X <- zero, Z <- succ(zero)
X <- succ(zero), Z <- succ(succ(zero))
X <- succ(succ(zero)), Z <- succ(succ(succ(zero))),
...
```

Opgave 11. Vertaal de volgende vergelijkingen naar een doel in Prolog. Hoe veel oplossingen verwacht je voor iedere vergelijking?

1. $x + 1 = 2$
2. $x + 2 = 1$
3. $x + y = 0$
4. $x + y = 2$

Opgave 12. Gebruik de *plus* relatie om een relatie *keer* (X, Y, Z) te definiëren. *Hint:* Gebruik de volgende twee feiten

- $0 \times x = 0$;
- $(1 + x) \times y = y + (x \times y)$.

1								6
		6		2		7		
7	8	9	4	5		1		3
			8		7			4
				3				
	9				4	2		1
3	1	2	9	7			4	
	4			1	2		7	8
9		8						

Figuur 3.2: Een voorbeeld sudoku puzzel

Opgave 13. Defineer een *kleiner* (X, Y) relatie, die beschrijft wanneer het getal X kleiner dan of gelijk is aan Y. Gebruik deze om een *minimum* (X, Y, Z) relatie te definiëren zodanig dat de Z het kleinere van de twee getallen X en Y is.

3.3 Mini-sudoku

De toepassingen van Prolog die we tot dusver hebben gezien zijn misschien niet zo spannend: twee getallen optellen of uitzoeken dat Beatrix de oma is van Amalia, dat kan je zelf ook wel. Laten we nu eens kijken naar een wat moeilijker probleem, dat je zelf misschien niet één-twee-drie oplost: sudoku.

Een sudoku puzzel bestaat uit een vierkant veld met 9×9 vakjes. De puzzel moeten we oplossen door in elk vakje een getal tussen de 1 en 9 in te vullen, op zo'n manier dat:

- iedere rij de getallen 1 tot en met 9 bevat;
- ieder kolom de getallen 1 tot en met 9 bevat;
- de negen 3×3 vierkanten de getallen 1 tot en met 9 bevat.

Een voorbeeld sudoku puzzel staat in Figuur 3.2.

Om alle relaties een beetje behapbaar te maken, zullen we voorlopig kijken naar sudoku puzzels met een bord van 4×4 in plaats van 9×9. In iedere rij, kolom, of vierkant zullen we dan alleen de getallen 1 tot en met 4 plaatsen. Een voorbeeld mini-sudoku puzzel staan in Figuur 3.3.

		4	2
2			
3	1	2	

Figuur 3.3: Een voorbeeld mini-sudoku puzzel

We zullen in de komende bladzijdes een relatie *oplossing* beschrijven die gegeven een (niet volledig ingevulde) sudoku puzzel, een substitutie uitrekent waarmee de openstaande vakjes ingevuld worden. Om dat te doen zullen we eerst het sudoku bord modelleren.

Het bord

Waarmee vullen we het bord in? We zouden getallen kunnen gebruiken, maar steeds $\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{zero})))$ in plaats van drie is toch wel vermoeiend. Aangezien we niet verder hoeven te rekenen met de getallen, zullen we dus gewoon een, twee, drie en vier schrijven voor de vier mogelijke waarden voor ieder vakje.

Om het bord zelf te beschrijven zouden we natuurlijk een onbekende kunnen gebruiken voor elk van de zestien vakjes. Vaak is het wel een beetje onhandig om met zo veel variabelen te werken. Het is beter om iets meer structuur vast te leggen.

We zullen het bord beschrijven als een *lijst* van rijen. Iedere rij is zelf ook een lijst van getallen. Hoe is zo'n lijst opgebouwd? Net als voor getallen, zijn er twee manier om lijsten op te bouwen:

1. Als de lijst leeg is schrijven we empty;
2. Als de lijst niet leeg is, is er een eerste element van de lijst X gevolgd door de rest van de lijst XS. Dit zullen we schrijven als $\mathsf{cons}\ (X, XS)$. Om duidelijk onderscheid te maken tussen lijsten en elementen houden we de conventie aan dat variabelen over lijsten altijd met de letter 'S' eindigen.

Laten we even naar een aantal voorbeeld lijsten kijken:

empty	een lege lijst
cons (aap, empty)	een lijst met alleen aap
cons (aap, cons (noot, cons (mies, empty)))	de lijst aap, noot, mies

Net als voor getallen, kunnen we berekeningen over lijsten recursief definiëren. Als voorbeeld, zullen we een *lengte* (XS, Y) relatie definiëren die aangeeft dat de lengte van de lijst XS gelijk is aan Y.

$$\mathit{length}\ (\mathsf{empty}, \mathsf{zero})$$
$$\mathit{length}\ (\mathsf{cons}\ (X, XS), \mathsf{succ}\ (Y)) \mathrel{:\!-} \mathit{length}\ (XS, Y)$$

Het eerste geval is eenvoudig: de lengte van de lege lijst is nul. Het tweede geval is interessanter: daar zeggen we dat als de lengte van de lijst XS gelijk is aan Y, dan heeft de lijst met een element meer als lengte $\mathsf{succ}\ (Y)$.

Opgave 14. Definieer een relatie $append\ (XS, YS, ZS)$ die definiëert wanneer ZS het resultaat is van het achter elkaar plakken van XS en YS. Hier zijn een aantal geldige voorbeelden van de *append* relatie:

- $append\ (\mathsf{cons}\ (\mathsf{drie}, \mathsf{empty}),$
  $\mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty})),$
  $\mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty}))))$

- $append\ (\mathsf{empty},$
  $\mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty})),$
  $\mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty})))$

Met deze notatie voor lijsten, kunnen we nu het sudoku bord modelleren. Laten we eens kijken naar de voorbeeld mini-sudoku puzzel in Figuur 3.3. Op de eerste rij staan twee lege vakjes en twee getallen. We kunnen dat met de volgende lijst representeren:

$$\mathsf{cons}\ (A, \mathsf{cons}\ (B, \mathsf{cons}\ (\mathsf{vier}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty}))))$$

Hier kiezen we twee verschillende variabelen voor de twee lege vakjes, A en B. De twee ingevulde waarden zetten we op de juiste plek in de lijst. Op dezelfde manier kunnen we de drie andere rijen in Prolog opschrijven. Het hele bord bestaat nu uit een lijst van rijen:

$$\begin{array}{l}\mathsf{cons}\ (\mathsf{cons}\ (A, \mathsf{cons}\ (B, \mathsf{cons}\ (\mathsf{vier}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty})))),\\ \quad \mathsf{cons}\ (\mathsf{cons}\ (\mathsf{twee}, \mathsf{cons}\ (C, \mathsf{cons}\ (D, \mathsf{cons}\ (E, \mathsf{empty})))),\\ \quad \mathsf{cons}\ (\mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{cons}\ (F, \mathsf{empty})))),\\ \quad \mathsf{cons}\ (\mathsf{cons}\ (G, \mathsf{cons}\ (H, \mathsf{cons}\ (I, \mathsf{cons}\ (J, \mathsf{empty})))),\\ \quad \mathsf{empty}))))\end{array}$$

Dat lijkt misschien ingewikkeld, maar in feite is het gewoon het uitschrijven van de informatie uit Figuur 3.3. Tot dusver gebeurt er nog niet heel veel spannends.

Een oplossing vinden

Hoe vinden we nu een oplossing van een sudoku puzzel?

Als we nu nauwkeurig beschrijven wanneer een invulling van een bord een echt een oplossing van een sudoku puzzel is, kunnen we de Prolog interpretator aanroepen. De Prolog interpretator zal vervolgens een substitutie bouwen waarmee we de opengebleven vakjes ingevuld worden.

Dus wanneer is een invulling van het sudoku bord correct? Een bord is correct ingevuld als alle rijen, kolommen, en vierkanten de getallen van een tot en met vier bevatten. Daarom definiëren we de *oplossing* relatie als volgt:

$$\begin{array}{l}oplossing\ (BORD)\ \text{:-}\\ \quad rijen\ (BORD, XSS), juist\ (XSS),\end{array}$$

$kolommen\ (BORD, YSS), juist\ (YSS),$
$vierkanten\ (BORD, ZSS), juist\ (ZSS).$

We gebruiken de relatie *juist* (XSS) om te beschrijven wanneer de rijen, kolommen, of vierkanten van een puzzel juist zijn ingevuld.

Ook al hebben we de relaties *juist*, *rijen*, *kolommen*, en *vierkanten* nog niet gedefineerd, hebben we wel alvast het probleem opgedeeld in kleinere stukken. Om onze automatische sudoku oplosser af te maken moeten we dus nog deze vier relaties defineren.

Juist De *juist* relatie neemt als invoer een lijst van lijsten. Iedere afzonderlijke lijst is een kolom, rij, of vierkant uit onze oorspronkelijke sudoku puzzel. Zo'n kolom, rij, of vierkant zullen we een *regio* noemen.

De *juist* relatie loopt de lijst van regio's af, om na te gaan dat ze geen dubbele waardes bevatten. De relatie defineren we weer in twee stappen:

$juist\ (\mathsf{empty}).$
$juist\ (\mathsf{cons}\ (XS, XSS)) \text{:-}\ verschillend\ (XS), juist\ (XSS).$

Als we geen regio's meer hoeven na te lopen, *juist* (empty), dan zijn we klaar. Is onze invoer niet leeg, dan hebben we een regio *XS*, en de overgebleven regio's *XSS*. De overgebleven regio's *XSS* moeten natuurlijk ook *juist* zijn ingevuld; maar ook de huidige regio *XS* moet allemaal verschillende waardes bevatten.

Maar hoe lopen we na of een lijst allemaal verschillende waardes heeft? Een eenvoudige, maar niet zo heel erg slimme, definitie somt de 24 verschillende mogelijke lijsten op:

$verschillend\ (\mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{vier}, \mathsf{empty}))))).$
$verschillend\ (\mathsf{cons}\ (\mathsf{twee}, \mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{vier}, \mathsf{empty}))))).$
$verschillend\ (\mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{vier}, \mathsf{empty}))))).$
...

Rijen, kolommen, en vierkanten We moeten nog drie relaties definiëren voor we de *oplossing* relatie kunnen gebruiken:

- een relatie *rijen* $(BORD, XSS)$, die beschrijft wanneer een sudoku puzzel *BORD* een lijst van rijen *XSS* bevat;
- een relatie *kolommen* $(BORD, XSS)$, die beschrijft wanneer een sudoku puzzel *BORD* een lijst van kolommen *XSS* bevat;
- een relatie *vierkanten* $(BORD, XSS)$, die beschrijft wanneer een sudoku puzzel *BORD* een lijst van vierkanten *XSS* bevat.

Het bord hebben we gemodelleerd als een lijst van rijen. Om een lijst met rijen uit te rekenen hoeven we dus eigenlijk niets te doen:

$rijen\ (XSS, XSS)$

Om de kolommen uit te rekenen is meer werk. We lopen recursief over de lijst van rijen. Als er geen rijen meer zijn om te verwerken, leveren we een lijst van vier lege lijsten op:

$\mathit{kolommen}\ (\mathsf{empty},$
$\quad \mathsf{cons}\ (\mathsf{empty}, \mathsf{cons}\ (\mathsf{empty}, \mathsf{cons}\ (\mathsf{empty}, \mathsf{cons}\ (\mathsf{empty}, \mathsf{empty}))))).$

Dit zijn de vier lege kolommen. De laatste empty geeft aan dat er niet meer kolommen zijn.

Als de lijst van rijen niet leeg is, pakken we de eerste rij van de puzzel, en rekenen de kolommen uit van de overgebleven rijen. Vervolgens voegen we de elementen van de eerste rij één voor één toe aan de kop van de lijst van kolommen. Dat gaat in Prolog als volgt:

$\mathit{kolommen}\ (\mathsf{cons}\ (XS, XSS), ZSS) :\!-$
$\quad \mathit{voegtoe}\ (XS, YSS, ZSS), \mathit{kolommen}\ (XSS, YSS).$

$\mathit{voegtoe}\ (\mathsf{empty}, \mathsf{empty}, \mathsf{empty}).$
$\mathit{voegtoe}\ (\mathsf{cons}\ (X, XS), \mathsf{cons}\ (YS, YSS), \mathsf{cons}\ (\mathsf{cons}\ (X, YS), ZSS)) :\!-$
$\quad \mathit{voegtoe}\ (XS, YSS, ZSS).$

De laatste relatie *vierkanten* ga je zelf definiëren. Om je een beetje op weg te helpen, hebben we de definitie over een aantal opgaven verspreidt die aan het eind van dit hoofdstuk staan.

Als je de definitie van *vierkanten* af hebt, kunnen we eindelijk onze *oplossing* relatie gebruiken om de puzzel uit Figuur 3.3 op te lossen. Met behulp van onze Prolog interpretator krijgen we nu het volgende resultaat:

```
oplossing(cons(cons(A,cons(B,cons(vier,cons(twee,nil)))),
  cons(cons(twee,cons(C,cons(D,cons(E,nil)))),
  cons(cons(drie,cons(een,cons(twee,cons(F,nil)))),
  cons(cons(G,cons(H,cons(I,cons(J,nil)))),nil)))))

substitution: A <- een, B <- drie,
  C <- vier, D <- een, E <- drie,
  F <- vier,
  G <- vier, H <- twee, I <- drie, J <- een

substitution: A <- een, B <- drie,
  C <- vier, D <- drie, E <- een,
  F <- vier,
  G <- vier, H <- twee, I <- een, J <- drie
```

Er zijn dus *twee* oplossing voor de puzzel. De Prolog interpretator vindt ze allebei binnen één seconde—veel sneller dan jij of ik ooit met de hand zouden kunnen!

3.4 Opgaven

Opgave 15. Defineer een relatie $\mathit{splits}\ (XS, YS, ZS)$ die een lijst van vier elementen XS in twee lijsten YS en ZS van lengte twee splitst. Een voorbeeld relatie is dus:

$\mathit{splits}\ (\mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{vier}, \mathsf{empty})))),$
$\qquad \mathsf{cons}\ (\mathsf{een}, \mathsf{cons}\ (\mathsf{twee}, \mathsf{empty})),$
$\qquad \mathsf{cons}\ (\mathsf{drie}, \mathsf{cons}\ (\mathsf{vier}, \mathsf{empty})))$

Opgave 16. Gebruik de *splits* en *append* relaties om de *vierkanten* relatie te definiëren. De relatie *vierkanten* (XSS, YSS) geldt als YSS de lijst van vier vierkanten van het bord XSS is.

Opgave 17. Waarom is de definitie van *juist* niet geschikt voor echte 9x9 sudoku puzzels? Kan je andere definitie bedenken?

Opgave 18. Lastig: Breid deze oplossing voor mini-sudoku puzzels uit naar een oplossing voor 9×9 sudoku puzzels.

Hoofdstuk 4

Hoe werkt Prolog?

Hoe vindt de Prolog interpretator een oplossing voor de gestelde doelen? In dit hoofdstuk leer je hoe de Prolog interpretator werkt. Je leert iets over belangrijke begrippen uit de Informatica, zoals unificatie, bewijzen, en zoeken. In het volgend hoofdstuk leggen we uit hoe je de Prolog interpretator kan programmeren in de functionele programmeertaal Haskell.

4.1 De Prolog server

Nu we een aantal voorbeelden van Prolog programma's hebben gezien staan we even stil bij hoe de Prolog interpretator eigenlijk werkt. Om te beginnen, mag je zelf oefenen met oplossingen zoeken voor doelen. Als je daar eenmaal een beetje ervaring mee hebt, zullen we dan gaan onderzoeken hoe we dit kunnen automatiseren.

Je kan online oefenen met het vinden van oplossingen van Prolog vergelijkingen. Hiervoor is de volgende website beschikbaar: `http://vhost0031.zoo.cs.uu.nl/login`. Als je eenmaal een account hebt aangemaakt en voor het eerst inlogt, krijg je een scherm te zien zoals Figuur 4.1. In eerste instantie zul je nog geen Prolog regels aan de rechter kant van het scherm hebben staan. Deze kan je zelf toevoegen, of je kan voorbeeld regels inladen door op de link aan de linker kant van het scherm te klikken onder het kopje 'Example data'.

Aan de rechterkant van je scherm staan een aantal regels uit het vorige hoofdstuk, zoals $ma\ (\mathsf{bea}, \mathsf{alex})$ of $ouder\ (X, Y)$:- $ma\ (X, Y)$. Links zie je een leeg tekstveld. Hier kan je een doel invoeren, zoals $pa\ (\mathsf{alex}, \mathsf{ama})$ of $ouder\ (X, \mathsf{ama})$.

Vervolgens kan je een *bewijs* construeren voor een doel. Zo'n bewijs bouw je stapje voor stapje op door telkens een regel uit het lijstje rechts naar het doel links te slepen. We kunnen het bewijs dat we zo construeren checken met de knop *Check Proof*. Een voorbeeld zien we in Figuur 4.2.

In het bewijs in Figuur 4.2 willen we laten zijn dat bea een voorouder is van ama. Aangezien bea geen ouder is van ama, zullen we de tweede regel van de *voorouder* relatie gebruiken. Nu moeten we nog een persoon *Z0* vinden die een ouder van ama is en een nakomeling van bea. Om te laten zien dat iemand een *ouder* is moet kunnen we weer twee regels gebruiken. Hier kiezen we voor om de tweede regel toe te passen. Nu zoeken we nog een *Z0* die de vader is van ama en een nakomeling van bea. Als we nu op de *Check Proof* knop drukken zien we dat een aantal onderdelen van het bewijs geel worden: we zijn nog niet klaar. Met behulp van een paar andere regels kan je het bewijs afmaken, zodat het hele bewijs groen wordt.

Logout

Module Functional Programming

Proof Tree

0. voorouder(bea, ama)
0.1. ouder(bea, Z0)
0.1.1. ma(bea, Z0)
0.2. ouder(Z0, ama)

Substitute for Substitute (e.g. substitute bea for X0)

Check Proof Reset Tree

Color coding help

Incorrect rule application
Incomplete proof
Correct rule
Syntax error

Example data

Example data containing the Dutch royal family, the list structure and lookup, and the natural numbers (as discussed in the JCU lecture notes) can be loaded by clicking this link. Beware that this will replace all your existing rules!

Stored Rules

Drag a rule form the list below to a field containing a term in the tree on the left.

ouder(X, Y):-pa(X, Y).
ouder(X, Y):-ma(X, Y).
ma(bea, alex).
ma(max, ale).
ma(max, ama).
ma(max, ari).
pa(alex, ale).
pa(alex, ama).
pa(alex, ari).
voorouder(X, Y) :- ouder(X, Y).
voorouder(X,Y) :- ouder(X,Z), ouder (Z,Y).

Add

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Figuur 4.1: De Prolog server

Opgave 19. Bewijs *voorouder* (bea, ama) met de Prolog server.

Opgave 20. Reken *plus* (succ (zero), succ (succ (zero)), X uit met behulp van de Prolog server.

Opgave 21. Reken *plus* (succ (zero), succ (succ (zero)), X uit met behulp van de Prolog server.

Opgave 22. Reken *keer* (succ (succ (zero)), succ (succ (zero)), X uit. Gebruik de keer definitie uit opgave 12.

4.2 Wat is het probleem?

Nu heb je met de hand kunnen oefenen om handmatig te zoeken naar oplossingen van een Prolog doel. Als je de opgaven hebt gedaan, heb je waarschijnlijk gemerkt dat het al snel vrij saai wordt om dit met de hand te doen. Nu gaan we onderzoeken hoe de Prolog interpretator automatische oplossingen vindt.

Laten we nu even stil staan bij wat voor informatie de Prolog interpretator als invoer heeft:

- regels die elementaire feiten aangeven, en dus geen :- teken bevatten, zoals bijvoorbeeld *pa* (alex, ama).

Module Functional Programming

Logout

Proof Tree

0. voorouder(bea, ama)
0.1. ouder(bea, Z0)
0.1.1. ma(bea, Z0)
0.2. ouder(Z0, ama)

Substitute for Substitute (e.g. substitute bea for X0)

Check Proof Reset Tree

Color coding help

Incorrect rule application
Incomplete proof
Correct rule
Syntax error

Example data

Example data containing the Dutch royal family, the list structure and lookup, and the natural numbers (as discussed in the JCU lecture notes) can be loaded by clicking this link. Beware that this will replace all your existing rules!

Stored Rules

Drag a rule form the list below to a field containing a term in the tree on the left.

ouder(X, Y):-pa(X, Y).
ouder(X, Y):-ma(X, Y).
ma(bea, alex).
ma(max, ale).
ma(max, ama).
ma(max, ari).
pa(alex, ale).
pa(alex, ama).
pa(alex, ari).
voorouder(X, Y) :- ouder(X, Y).
voorouder(X,Y) :- ouder(X,Z), ouder (Z,Y).

Add

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Figuur 4.2: Een voorbeeld bewijs

- regels die een rechterkant hebben. Deze interpreteren we als volgt: *als je de linkerkant wilt bewijzen is het voldoende als je voor elk van de doelen aan de rechterkant een bewijs levert*, zoals bijvoorbeeld

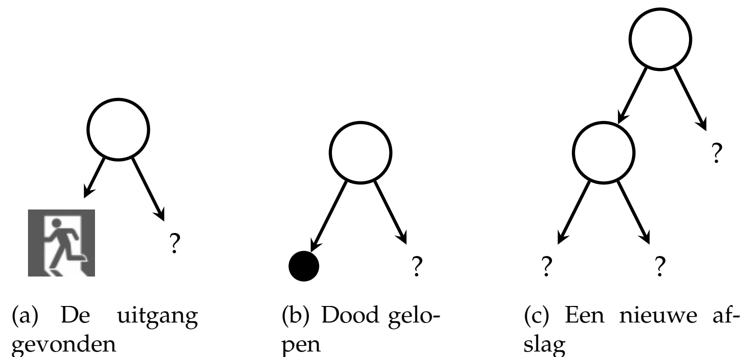
  $\mathit{voorouder}\ (X, Y)$:- $\mathit{ouder}\ (X, Z), \mathit{voorouder}\ (Z, Y)$.

- een lijst van nog te bewijzen doelen. In eerste instantie hebben we maar één doel, maar door het toepassen van regels kunnen we dit probleem misschien reduceren tot een aantal sub-doelen. Daarom houden we steeds een *lijst* van doelen bij.

Als uitvoer moet de Prolog interpretor een *substitutie* opleveren. Wanneer deze substitutie toegepast wordt op het oorspronkelijke doel, levert dat een term op die waar is. Hoe vinden we zo'n substitutie? En hoe weten we wanneer we ons doel bereikt hebben? Laten we even proberen dit probleem in deelproblemen op te breken.

- Soms kunnen er meerdere regels van toepassing zijn. Welke moeten we dan toepassen? Van tevoren weet je niet welke regel 'beter' is—ze zouden allebei oplossingen op kunnen leveren. Daarom moeten we *alle* mogelijke regels toepassen en *zoeken* naar alle mogelijke oplossingen. Dit leggen we in sectie 4.3 verder uit.

- In principe proberen we iedere regel toe te passen. Nu zijn niet alle regels altijd toepasbaar: als we $\mathit{pa}\ (X, \mathsf{ama})$ moeten oplossen, dan kunnen we $\mathit{ma}\ (\mathsf{bea}, \mathsf{alex})$ niet gebruiken. We moeten dus beslissen welke regels wel of niet toepasbaar zijn. Dit proces, gebaseerd op het begrip *unificatie*, leggen we in sectie 4.4 verder uit.

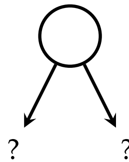


(a) De uitgang gevonden (b) Dood gelopen (c) Een nieuwe afslag

4.3 Zoeken

Laten we eerst kijken naar een eenvoudiger zoekprobleem. Stel je bent in een doolhof, hoe vind je een uitweg? Om het probleem iets te vereenvoudigen, gaan we er van uit dat je bij ieder afslag alleen kan kiezen of je links of rechts wil. Na de afslag loopt de weg dood, vind je de uitgang, of kom je bij een nieuwe afslag.

Aan het begin van het doolhof ziet je situatie er dus als volgt uit:



Hier kan je dus kiezen: wil je naar links of naar rechts? Laten we naar links gaan. Als we geluk hebben, staan we meteen bij de uitgang. Zo niet, loopt de weg dood of komen we bij een nieuwe splitsing aan. Deze drie situaties staan uitgebeeld in Figuur 4.3.

Hoe moeten we nu verder zoeken?

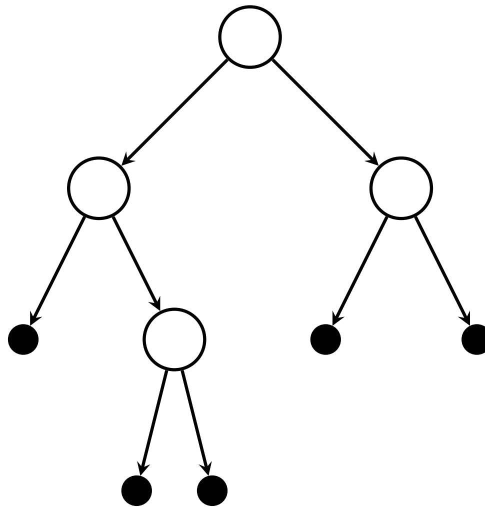
(a) Als we de uitgang hebben bereikt zijn we natuurlijk klaar.

(b) Als we op een dood spoor zitten, gaan we terug naar de vorige afslag, en proberen nu naar rechts te gaan.

(c) Als we een nieuwe afslag bereiken, dan gaan we verder naar links. Pas als we hebben vastgesteld dat de weg naar links doodloopt, zullen we de eerdere afslag naar rechts nemen.

Deze drie eenvoudige regels leggen al precies vast in welke volgorde een doolhof wordt doorlopen. Deze zoekstrategie wordt *depth-first search* genoemd—we lopen eerst meteen zo 'diep' mogelijk door het doolhof door alsmaar links af te slaan. Pas als we dood zijn gelopen, zullen we afslagen naar rechts uitproberen.

Opgave 23. In welke volgorde komen we langs de verschillende afslagen in dit doolhof bij een depth-first search.



Opgave 24. Bedenk een doolhof met minstens vier afslagen, waar je pas op het allerlaatst de uitgang zal vinden met een depth-first search.

Wat heeft dit nou met Prolog te maken? Als je hebt geoefend met de Prolog server, zul je hebben gemerkt dat je op een gegeven moment een aantal doelen hebt waar je oplossingen voor moeten vinden. De regels die we als invoer hebben gegeven zijn 'zetten' aan die we kunnen doen; net als dat we bij een afslag mogen kiezen of we links of rechts gaan, kunnen we kiezen met welke regel we verder gaan. Net als onze oplossing voor het doolhof probleem, gebruikt Prolog een depth-first search om naar een oplossing voor een doel te vinden. Zo probeer je één voor een regels toe te passen totdat je geen openstaande doelen meer hebt of je vastgelopen bent en een onbewijsbaar doel open hebt staan.

4.4 Unificatie

We hebben tot dusver gezien hoe de Prolog interpretator zoekt, maar hoe bouwt de interpretator deze oplossing op? Hoe weet de interpretator wanneer een regel toegepast mag worden? Wat is een geldige zet en wat niet?

Laten we nu een aantal voorbeelden doornemen. Aan de hand van deze voorbeelden zullen we een *unificatie* algoritme defineren, die bepaalt wanneer een regel toegepast mag worden.

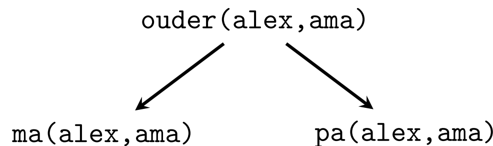
Voorbeeld I Laten we weer teruggaan naar ons allereerste Prolog doel: `pa(alex,X)`. Welke regels kunnen we gebruiken om dit te bewijzen? En welke oplossingen leveren die op? Jij en ik zien meteen dat er drie manieren zijn om `pa(alex,X)` te bewijzen: $X \leftarrow \mathsf{ama}$; $X \leftarrow \mathsf{ale}$; of $X \leftarrow \mathsf{ari}$. Maar *hoe* zien we dat?

Er zijn verschillende regels die iets zeggen over de *pa* relatie, bij voorbeeld $pa\ (\mathsf{alex}, \mathsf{ama})$, $pa\ (\mathsf{alex}, \mathsf{ale})$, en $pa\ (\mathsf{alex}, \mathsf{ari})$. Maar ook regels als $pa\ (\mathsf{claus}, \mathsf{alex})$ en $pa\ (\mathsf{claus}, \mathsf{friso})$. Die laatste regels vertellen ons dat claus een vader is, en niet alex zoals we zouden willen. Die regels vallen dus af. Dan blijven er maar drie regels over.

Elk van die drie regels maakt een andere keuze voor de variabele X, wat dus drie oplossingen oplevert. *Een regel is dus alleen een geldige zet als het de juiste vorm heeft.* Als we

pa (alex, X) proberen te bewijzen zoeken we naar regels met als conclusie de *pa* relatie waarvan het eerste argument alex is.

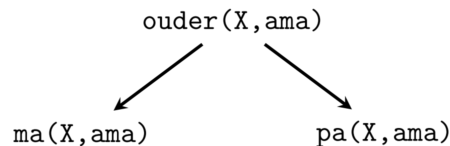
Voorbeeld II Stel ons doel is `ouder(alex,ama)`. Er zijn twee regels die we mogen gebruiken om te laten zien dat iemand een ouder is. Hier zullen we dus moeten kiezen welke kant we op willen:



We proberen eerst 'linksaf' te gaan: ons doel wordt nu om te bewijzen dat *ma* (alex, ama). Dat lukt niet, want er is geen regel die we kunnen toepassen om dit te bewijzen. Als we de 'rechterkant' proberen hebben we meer succes. Omdat *pa* (alex, ama) een gegeven regel is, zijn we meteen klaar.

Zo zie je dus waarom de Prolog interpretator moet zoeken: soms kan er meer dan één regel worden toegepast en het is niet van tevoren duidelijk welke regel een oplossing zal opleveren. Als een keus verkeerd uitpakt, zal de Prolog interpreter dus een stap terug moeten nemen en een andere regel proberen toe te passen.

Voorbeeld III Tot slot kijken we naar het doel `ouder(X,ama)`. Net als in het volgend voorbeeld kunnen we kiezen welke regel we willen toepassen:



Maar nu levert de eerste keus, `ma(X,ama)` wel een antwoord op. Om toch *alle* mogelijke oplossingen te vinden moeten we ook de tweede regel proberen toe te passen.

Uit de voorbeelden die we net hebben gezien kunnen we een aantal lessen trekken. Om een doel als *pa* (alex, X) op te lossen, hoeven we alleen naar die regels te kijken die iets over *pa* relatie zeggen. Een regel als *ma* (bea, alex) is dan niet interessant.

Maar niet alle *pa*-regels zijn toepasbaar. Een regel als *pa* (claus, alex) kunnen we niet gebruiken om *pa* (alex, X) op te lossen. Een regel is alleen toepasbaar als alle elementen van de regel *unificeren* zijn met het doel dat we proberen op te lossen. Wanneer zijn twee termen unificeerbaar?

Als ons doel *pa* (alex, ama) is en we hebben de regel *pa* (alex, ama) gedefineerd, dan is die regel toepasbaar. Blijkbaar kunnen we alex met alex unificeren en ama met ama. Daarentegen kunnen we niet een regel als *pa* (claus, alex) toepassen. Constante termen zonder parameters zijn alleen unificeerbaar met zichzelf.

Als ons doel *pa* (alex, X) is en we hebben de regel *pa* (alex, ama) dan kunnen we alex met alex unificeren en X met ama. Een onbekende is altijd unificeerbaar met iedere andere term. Dat levert dan de substitutie $X \leftarrow$ ama op.

We moeten nog wel een beetje voorzichtig zijn. Stel dat we willen weten of er iemand zijn eigen vader is. Ons doel wordt dan $pa\ (X, X)$. Nu zou je kunnen denken dat $pa\ (\mathsf{alex}, \mathsf{ama})$ toepasbaar is: immers X unificeert met alex en X unificeert met ama. Toch is deze regels niet toepasbaar. Als we eenmaal een substitutie hebben gevonden voor een onbekende, moeten we deze eerst toepassen voor we de rest van de termen unificeren. Als we eenmaal hebben besloten dat $X \leftarrow \mathsf{alex}$, wordt moeten we vervolgens niet X en ama unificeren, maar alex en ama—en dat gaat natuurlijk niet.

4.5 Alle stukjes bij elkaar

Nu kunnen we eindelijk de Prolog interpretator beschrijven. We doen dit nu in het Nederlands, in het volgend hoofdstuk maken we dit meer precies door de Haskell implementatie van de Prolog interpretator uit te leggen

Gegeven het openstaande doel, begint de interpretator met een lege substitutie als 'oplossing'. De interpretator probeert één voor één de linkerkant van alle regels te unificeren met het doel. Lukt dit niet, dan is er geen oplossing. Lukt dit wel, wordt de substitutie uitgebreidt met het resultaat van de unificatie en wordt het huidige doel vervangen door de doelen aan de rechterkant van de regel die wordt toegepast. Dan gaat de Prolog interpretator deze doelen ook proberen te bewijzen. Als een poging mislukt, gaat de Prolog interpretator terug naar de vorige toegepaste regel, en probeert vervolgens of een andere regel wel een oplossing oplevert. Zo gebruikt de interpretator een *depth first search* om een oplossing te vinden.

Nu we weten hoe de Prolog interpretator werkt, gaan we in de komende hoofdstukken deze ideeën uitwerken en implementeren in de functionele programmeertaal Haskell.

Hoofdstuk 5

Een Prolog interpretator in Haskell

Haskell is een *functionele programmeertaal*. Je kan heel veel informatie vinden over Haskell op `www.haskell.org`. De programmeertaal Haskell is een stuk uitgebreider dan Prolog. Gelukkig is er een enorm actieve gemeenschap van mailing lijsten, webfora, en het #haskell IRC chatkanaal waar je voor vragen terecht kan.

Om Haskell te gebruiken zul je de Haskell Platform zelf moeten installeren. Je kan die gratis downloaden van `http://hackage.haskell.org/platform`. Als je alleen wat wil oefenen met een Haskell interpretator kan dat ook online op `http://tryhaskell.org`.

Omdat Haskell een veel 'rijkere' taal is dan Prolog, raden we je alleen aan om Haskell te bestuderen als je zelf al meer programmeerervaring hebt met andere talen dan Prolog. Je kan programma's schrijven in Haskell die meer kunnen dan de eenvoudige Prolog relaties die we tot dusver hebben gezien.

Er zijn diverse tutorials geschreven over Haskell. We zullen de basis niet hier herhalen, maar nemen aan dat je zelf al eens hebt gekeken naar het materiaal dat online beschikbaar is. We raden aan dat je kijkt naar het (Engelse) boek *Learn you a Haskell!*, dat online te lezen is op `http://learnyouahaskell.com`. We gaan er in de rest van dit hoofdstuk er van uit dat je zelfstandig de eerste twee hoofdstukken van *Learn you a Haskell* hebt bestudeerd. Dit boek is ook erg geschikt om na het lezen van dit document nog meer over Haskell te weten te komen.

5.1 Data structuren

Naast lijsten of getallen, gebruiken programma's vaak andere data structuren. De Prolog interpretator moet ook werken met *termen* en *regels*. Laten we eens kijken hoe we deze kunnen modelleren in Haskell.

Onbekenden en constanten representeren we allebei als een *String*. Een onbekende als X wordt gerepresenteerd als `"X"`; een constante als alex wordt `"alex"`. Een term is een onbekende (zoals X) of een constante toegepast op een aantal argument andere termen (zoals alex of succ (succ (zero))). In Haskell kunnen we dit alles als volgt opschrijven:

```
type Onbekende = String
type Constante  = String
data Term = Var Onbekende
          | Fun Constante [Term]
```

De eerste twee regels definiëren twee *type synoniemen*–een andere naam voor *String* is *Onbekende*, en een andere naam voor *String* is *Constante*. Deze definities introduceren nieuwe namen voor het *String* type. Dat kan soms handig zijn om verwarring te voorkomen.

De definitie van *Term* is interessanter. Daar introduceren we een nieuw type, *Term*. Er zijn twee manieren om een waarde van het *Term* type te maken:

- variabelen zoals X zullen we in Haskell schrijven als *Var* `"X"`;
- constantes zoals alex zullen we in Haskell schrijven als *Fun* `"alex"` [ ]. Als de constante wel argumenten heeft, zoals succ of cons, zullen we in deze in een lijst opslaan.

Opgave 25. Geef Haskell representaties van de volgende termen:

1. bea
2. zero
3. succ (zero))
4. cons (een, empty)

Naast termen, werkt de Prolog interpretator ook met regels. Een regel, zoals bij voorbeeld *ouder* (X, Y) :- *pa* (X, Y), bestaat uit een linkerkant en rechterkant. Links staat een enkele term; rechts een lijst van termen. We definiëren een nieuw data type om regels te representeren:

```
data Regel = Term :-[Term]
```

We gebruiken hier precies dezelfde notatie als in Prolog.

Met deze data types kunnen we een Prolog regel als *pa* (alex, ama) als volgt representeren in Haskell:

```
pa_alex_ama :: Regel
pa_alex_ama = Fun "pa" [Fun "alex" [], Fun "ama" []] :-[]
```

Opgave 26. Schrijf de volgende regels als een Haskell waarde van het type *Regel*:

1. *ma* (bea, alex)
2. *ouder* (X, Y) :- *pa* (X, Y)
3. *voorouder* (X, Y) :- *ouder* (Z, Y), *voorouder* (X, Z)

5.2 Unificatie

Een substitutie zouden we kunnen representeren als een lijst van de vorm 'vervang onbekende X door term t'. Het is efficiënter om Haskell's *Data.Map* bibliotheek te gebruiken. We zullen ons beperken tot de volgende functies op substituties:

```
empty :: Subst
voegToe :: Onbekende → Term → Subst → Subst
```

$$
\begin{array}{l}
\mathit{unificeer} :: (\mathit{Term}, \mathit{Term}) \to \mathit{Poging\ Subst} \to \mathit{Poging\ Subst} \\
\mathit{unificeer}\ (t, u)\ \mathit{Mislukt} = \mathit{Mislukt} \\
\mathit{unificeer}\ (t, u)\ (\mathit{Geslaagd\ subst}) = \mathit{uni}\ (\mathit{apply\ subst}\ t)\ (\mathit{apply\ subst}\ u) \\
\quad \textbf{where} \\
\quad \mathit{uni} :: \mathit{Term} \to \mathit{Term} \to \mathit{Subst} \to \mathit{Subst} \\
\quad \mathit{uni}\ (\mathit{Var}\ x)\ y \qquad \mathit{subst} = \mathit{Geslaagd}\ (\mathit{voegToe}\ x\ y\ \mathit{subst}) \\
\quad \mathit{uni}\ x \qquad (\mathit{Var}\ y)\ \mathit{subst} = \mathit{Geslaagd}\ (\mathit{voegToe}\ y\ x\ \mathit{subst}) \\
\quad \mathit{uni}\ (\mathit{Fun}\ x\ \mathit{xs})\ (\mathit{Fun}\ y\ \mathit{ys}) = \\
\qquad \textbf{if}\ x \equiv y \wedge \mathit{length}\ \mathit{xs} \equiv \mathit{length}\ \mathit{ys} \\
\qquad \textbf{then}\ \mathit{unificeerKinderen}\ (\mathit{Geslaagd\ subst})\ \mathit{xs}\ \mathit{ys} \\
\qquad \textbf{else}\ \mathit{Mislukt}
\end{array}
$$

Figuur 5.1: Het unificatie algoritme

De implementatie van deze functies roept de functies uit de *Data.Map* bibliotheek.

Unificatie kan slagen of niet. Het resultaat van unificatie is dus een substitutie (als alles goed gaat), maar misschien ook niet. Daarom zullen we het volgende data type gebruiken om substituties te modelleren:

$$\textbf{data}\ \mathit{Poging} = \mathit{Geslaagd\ Subst} \mid \mathit{Mislukt}$$

Het unificatie algoritme uit het vorige hoofdstuk kunnen we nu implementeren in Haskell. De implementatie van het algoritme kan je nalezen in Figuur 5.1.

Om de *unificeer* functie te begrijpen, is het goed om eerst naar de hulp-functie *uni* te kijken. Gegeven twee termen en de oplossing tot dusver, probeert de *uni* functie de substitutie uit te breiden.

Als één van de termen een variabele is, kunnen we de substitutie uitbreiden. Om een concreet voorbeeld te geven: stel dat je doel $pa\ (\mathsf{alex}, X)$ is en je probeert de regel $pa\ (\mathsf{alex}, \mathsf{ama})$ toe te passen, dan zal je (uiteindelijk) X en ama gaan unificeren. De eerste twee regels van de *uni* functie zorgen er voor dat in zo'n geval de substitutie die tot dusver is opgebouwd, wordt uitgebreid met $X \leftarrow \mathsf{alex}$.

Als we twee termen tegenkomen moeten we meer werk verrichten. Dit gebeurt in de derde regel van de *uni* functie. Om te beginnen, voeren we een aantal checks uit:

- $(x \equiv y)$ We lopen na dat de we hier te maken hebben met dezelfde term. We kunnen alex en ama niet unificeren, maar alex en alex wel. Hier kijken we (nog) niet naar eventuele kinderen: $\mathsf{succ}\ (\mathsf{zero})$ en $\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{zero}))$ zal deze check nog doorstaan.

- $\mathit{length}\ \mathit{xs} \equiv \mathit{length}\ \mathit{ys}$ Allebei de termen moeten evenveel kinderen hebben. Als je bijvoorbeeld een fout in je programma hebt, waardoor de interpreter $\mathsf{succ}\ (\mathit{nul}, \mathit{nul})$ en $\mathit{suc}\ (\mathit{nul})$ probeert te unificeren, zal dat niet lukken.

Als aan deze twee voorwaarden is voldaan, proberen we de kinderen van de termen te unificeren. Dat gebeurt in de functie *unificeerKinderen*, die je zelf zo zal schrijven. Anders leveren we *Mislukt* op.

De functie *unificeer* roept *uni* aan, maar past wel de tot dusver opgebouwde substitutie toe met behulp van de functie *apply*. We zagen al aan het eind van het vorige hoofdstuk

dat dit nodig is: anders zou je twee waardes van X kunnen kiezen tijdens de unificatie van $pa\ (X, X)$ en $pa\ (\mathsf{alex}, \mathsf{ama})$.

Opgave 27. Schrijf de ontbrekende functie *unificeerKinderen*. Het type van deze functie is

$$unificeerKinderen :: Subst \to [Term] \to [Term] \to Subst$$

Gegeven de substitutie tot dusver, unificeert deze functie één voor één alle paren van termen in de twee lijsten. Je mag er van uit gaan dat allebei de lijsten even lang zijn.

5.3 Zoeken

Gedurende het zoeken naar een oplossing hebben we gezien dat we moeten zoeken—er kunnen soms meerdere regels van toepassing zijn. Om onze depth-first search zoek algoritme te definiëren, zullen we eerst een *data type* definiëren dat de boom is waarin we zoeken. Dan kunnen we ons algoritme schrijven door recursief over deze boom te lopen. Tot slot, zullen we laten zien hoe we deze *zoekboom* opbouwen gegeven de regels en doelen van ons Prolog programma.

We definiëren daarom de volgende data structuur om in te zoeken:

```
data Resultaat = Klaar Subst
               | Stap [Resultaat]
```

Als we een oplossing s hebben gevonden, zullen we aangeven met *Klaar s*. Anders hebben we een *Stap* met een lijst van kinderen, die ook weer een *Resultaat* zijn. Als de lijst leeg is, zijn we 'dood gelopen'—en moeten we proberen om andere regels toe te passen om een oplossing te vinden. Is deze lijst niet leeg, dan hebben we een keuze uit verschillende regels die we zouden kunnen toepassen.

Een depth first search zoek algoritme is heel makkelijk te schrijven met behulp van *lijst comprehensies*—deze staan onder meer beschreven in het tweede hoofdstuk van *Learn you a Haskell*.

We definiëren de volgende functie, *dfs*, die door een *Resultaat* boom zoekt en alle oplossingen oplevert:

```
dfs :: Resultaat -> [Subst]
dfs (Klaar oplossing) = [oplossing]
dfs (Stap [])         = []
dfs (Stap (k : ks))   = dfs k ++ dfs (Stap ks)
```

We onderscheiden hier drie gevallen: óf we zijn *Klaar* en hebben een oplossing voor handen; óf we zijn doodgelopen; óf we zijn bij een *Stap* waar we moeten kiezen welke regel we verder gaan toepassen. Het eerste geval is makkelijk: we leveren de oplossing op die we gevonden hebben. Het tweede geval is niet veel moeilijker: als we geen verdere regels kunnen toepassen en we hebben nog geen oplossing gevonden, leveren we de lege lijst op. Het laatste geval is het meest interessant. Daar hebben we een aantal verschillende keuzes, die ieder een andere 'afslag' voorstellen. We beginnen door recursief het eerste kind k te doorzoeken. Maar om *alle* mogelijke oplossingen te vinden moeten we ook de *dfs* functie aanroepen op de rest van de kinderen. Het resultaat van deze twee aanroepen van *dfs* combineren we met de $+\!\!+$ operator van Haskell, die twee lijsten samenvoegt.

Opgave 28. Schrijf met de hand een voorbeeld zoekboom van het type *Resultaat*. Roep de *dfs* functie aan op jou zoekboom. Leg uit wat de uitkomst is van deze functieaanroep. Waarom staan de elementen van de uitkomst lijst in deze volgorde?

5.4 Alle stukjes bij elkaar

De kern van de Prolog interpretator definiëren we door middel van de volgende *solve* functie:

```
solve :: [Regel] → Term → Resultaat
solve regels doel = stappen (Geslaagd empty) [doel]
  where
  stappen :: Poging Subst → [Term] → Resultaat
  stappen Mislukt _   = Stap []
  stappen (Geslaagd e) [] = Klaar e
  stappen e (doel : ds) =
    Stap [stappen (unify (doel, c) e) (cs ++ ds) | c :- cs ← regels]
```

De *solve* functie zelf doet erg weinig, behalve de hulp functie *stappen* aan roepen. De argumenten van de *stappen* functie zijn de lijst van alle regels, de tot dusver opgebouwde substitutie (die in eerste instantie leeg is), en de lijst met openstaande doelen. Om te beginnen is bevat deze lijst één element: het doel waar we een oplossing voor moeten vinden.

Het echte werk gebeurt in de *stappen* functie. De eerste twee gevallen beschrijven de twee mogelijke uitkomsten: de unificatie mislukt, of (als er geen openstaande doelen meer zijn) levert het een oplossing op. Het echte werk gebeurt in het derde geval. In het derde geval gebruiken we een *lijst comprehensie* om ons doel te verfijnen. Je kan meer over lijst comprehensies lezen in Hoofdstuk 2 van *Learn you a Haskell*. Hier zullen we in woorden proberen duidelijk te maken wat er gebeurt. We proberen één voor één alle regels toe te passen door het de uitkomst *c* van iedere regel te unificeren met het volgende openstaande doel. Vervolgens bouwen we recursief de zoekboom verder op, door de *stappen* functie weer aan te roepen met het resultaat van deze unificatie. De doelen die dan nog open staan zijn de nieuwe sub-doelen die rechts van de regel staan die we toepassen, *cs*, gevolgd door de oorspronkelijke doelen *ds*. Door deze boom depth-first te doorzoeken kunnen we een lijst van oplossingen uitrekenen.

5.5 Wat nu?

Ook al hebben we de *solve* functie gedefiniëert, we zijn nog niet helemaal klaar. We moeten nog een bestand met regels openen, die regels inlezen, de gebruiker vragen naar een doel, en met al deze gegevens de *solve* functie aanroepen. De Prolog server moet bovendien allerlei informatie versturen tussen je webbrowser en de server die allerlei berekeningen doet. Er is dus heel wat meer *infrastructuur* nodig om een werkend programma te hebben. Toch is dit vaak niet het moeilijkste gedeelte van een computer programma: als je eenmaal de algoritmes hebt bedacht en uitgewerkt, heb je het moeilijkste achter de rug.

Hoofdstuk 6

Beschouwing

Wat heb je nu geleerd?

Je hebt kennis gemaakt met je eerste *declaratieve* programmeertaal, Prolog. Je hebt kunnen zien hoe je een aantal verschillende problemen met behulp van Prolog kan oplossen, van het beschrijven van familierelaties tot het oplossen van Sudoku puzzels.

Als een computer een Prolog doel oplost, gebeurt er heel wat 'onder de motorkap'. Je hebt kunnen zelf kunnen oefenen met het samenstellen van eenvoudige oplossingen voor Prolog doelen. Bovendien heb je kunnen leren hoe je dit proces kan automatiseren, zodat de computer het met één druk op de knop voor je kan uitrekenen.

Sommige van jullie hebben bovendien iets geleerd over Haskell, een *functionele* programmeertaal. De Prolog server en interpretator die jullie hebben gebruikt zijn allemaal in Haskell geschreven. Je hebt een programma gezien dat oplossingen zoekt voor een Prolog doel. Hiermee leggen we precies vast hoe het algoritme uit de eerdere hoofdstukken werkt.

Maar we hopen vooral dat je hebt geleerd wat Informatica is. Het is geen wiskunde waar je op papier oefent met differentiëren en integreren. Het is geen levenswetenschap, waar je experimenten uitvoert om meer over de wereld te leren. Het is een exacte wetenschap die bestudeert hoe informatie te verwerken. Het is een jonge wetenschap dat overal toepassingen heeft van je mobiele telefoon tot je koelkast. Maar we hopen dat je vooral hebt geleerd dat Informatica heel erg leuk is.