



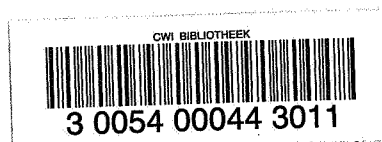
Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.C. van der Gaag, P.J.F. Lucas

Inleiding tot PROLOG

Informatica/Afdeling Programmatuur Notitie CS-N8605 April

Introduction to Prolog
Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam



The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Inleiding tot PROLOG

L.C. van der Gaag, P.J.F. Lucas

Centrum voor Wiskunde en Informatica

Postbus 4079, 1009 AB Amsterdam

De programmeertaal PROLOG geniet zowel vanuit het theoretisch onderzoek als vanuit praktische toepassingen ruime belangstelling. PROLOG vormt enerzijds een praktische weerslag van de principes van programmeren in de logica en heeft hierdoor anderzijds een aantal eigenschappen die de taal geschikt maken voor symboolverwerking. In dit artikel wordt een inleiding gegeven in de grondslagen en het praktische gebruik van PROLOG. Tevens worden enkele gebieden belicht waarbinnen PROLOG inmiddels met succes wordt toegepast.

Opmerking: Dit artikel verschijnt in het Handboek Informatica, een uitgave van Samsom Uitgeverij bv.

69 D 42, 69 K 15

1. INLEIDING

De belangstelling voor de programmeertaal PROLOG neemt de laatste jaren sterk toe, in het bijzonder omdat PROLOG een taal voor kunstmatige intelligentie wordt genoemd. In dit artikel wordt een korte inleiding in de grondslagen van de taal gegeven, maar zal de nadruk vooral op het praktisch gebruik van de taal liggen.

1.1. Geschiedenis

PROLOG is een eenvoudige, maar krachtige programmeertaal, gebaseerd op de principes van de eerste-orde predicaatlogica. De basis voor de taal werd omstreeks 1970 aan de Universiteit van Marseille gelegd door Alain Colmerauer. Toch duurde het nog vier jaar voordat het eerste PROLOG systeem ontwikkeld werd, als resultaat van onderzoek naar programmeren in de logica, met name verricht door Colmerauer en Robert Kowalski, en onderzoek naar het automatisch bewijzen van theorema's (stellingen), waarmee de naam van Phillippe Roussel verbonden is. De nieuwe taal kreeg de naam PROLOG, een afkorting van "PROgramming in LOGic", als verwijzing naar de principes waarop de taal gebaseerd werd. Hiermee was voor het eerst een taal ontwikkeld, die de programmeur in staat stelde zijn programma-ontwerp te specificeren in een vorm, direct geschikt voor verwerking door de computer.

Het gebruik van PROLOG is tot het einde van de jaren zeventig beperkt gebleven tot de universitaire wereld. Met name aan de universiteiten van Marseille en Edinburgh is gebruik gemaakt van PROLOG voor een keur van projecten. Aan de Universiteit van Edinburgh werd in 1975 door David Warren, Fernando Pereira en Lawrence Byrd de eerste PROLOG compiler, voor DEC 10/20 PROLOG, ontwikkeld. Met de constructie van deze compiler doet PROLOG voorzichtig zijn intrede in het milieu buiten het onderwijs. In de daarop volgende jaren worden er verschillende interpreters en compilers voor PROLOG ontworpen, waaronder de PROLOG microcomputer interpreter voor Micro PROLOG.

De belangstelling voor PROLOG neemt op dit ogenblik toe. Vooral in Groot-Brittannië en Hongarije heeft de toepassing van de taal een grote vlucht genomen. De toenemende interesse is niet in de laatste plaats het gevolg van het Vijfde Generatie Computer Project in Japan: het ICOT kiest in 1981 PROLOG en hiervan afgeleide talen als basis voor de ontwikkeling van een nieuwe generatie

Notitie CS-N8605

Centrum voor Wiskunde en Informatica

Postbus 4079, 1009 AB Amsterdam

hardware.

Op een aantal terreinen wordt PROLOG inmiddels met succes gebruikt [COE80]. Hiertoe behoren onder andere:

- natuurlijke-taal verwerking
- compilerconstructie [WAR80]
- de ontwikkeling van expertsystemen [LUC86, SUB85]
- werk op het gebied van de symbolische wiskunde
- databasesystemen [GAL78, LLO83]

Vaak wordt voor het gebruik van PROLOG gekozen, omdat de wijze waarop informatie in deze taal kan worden gerepresenteerd nauw aansluit bij de methoden die men in het betreffende vakgebied hanteert.

1.2. Programmeren in de logica

Bij het programmeren in een conventionele, procedurele programmeertaal als Pascal is een programma een specificatie van een reeks stappen die achtereenvolgens ondernomen moeten worden om tot de oplossing van het beschouwde probleem te komen. De beschrijving van het probleem is impliciet in deze specificatie vastgelegd. Bij het programmeren in de logica worden de beschrijving van het probleem en de methode om tot de oplossing van het probleem te komen expliciet van elkaar gescheiden. Deze scheiding formuleerde Kowalski [KOW79] in de volgende vergelijking:

$$\text{algoritme} = \text{logica} + \text{besturing}$$

In deze vergelijking wordt met de term logica de beschrijvende component van een algoritme aangegeven, dat wil zeggen de beschrijving van het probleem, en met de term besturing de component die de methode om tot een oplossing te komen, beschrijft. De logische component definieert derhalve *wat* het algoritme doet; de besturende component geeft weer *hoe* dit gebeurt.

De splitsing van een algoritme in een logische component en een besturende component heeft diverse voordelen:

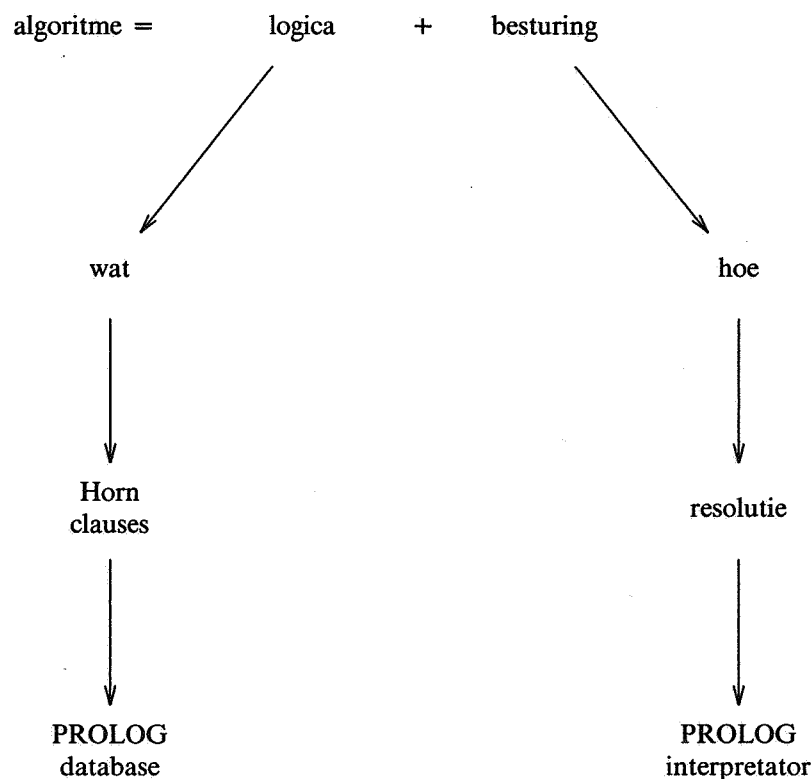
- de twee componenten kunnen los van elkaar ontwikkeld worden
- een logische component kan worden ontwikkeld met de methode van stapsgewijze verfijning, waarbij men slechts de correctheid in ogenschouw behoeft te nemen
- veranderingen in de besturende component hebben alleen invloed op de efficiëntie van het algoritme

De componenten kunnen apart ontwikkeld worden: de logische component wordt ontwikkeld door in het probleem relevante objecten en relaties tussen objecten te onderscheiden en feiten en regels over deze objecten en relaties vast te leggen in een eenduidige formele taal. In de besturende component wordt een methode vastgelegd die aangeeft hoe uit de reeds in de logische component aanwezige informatie nieuwe informatie kan worden afgeleid, die mogelijk tot de oplossing van het probleem kan leiden; een dergelijke methode wordt in het algemeen een *inferentiemethode* genoemd.

Als er eenmaal een besturende component is ontwikkeld, kan deze op verschillende logische componenten toegepast worden om telkens een ander probleem op te lossen. Een omgeving voor logisch programmeren biedt de programmeur een besturende component, zodat voor een concreet probleem alleen de logische component gespecificeerd dient te worden. De ontwikkeling van een toepassingsonafhankelijke, efficiënte inferentiemethode is nog steeds een van de aandachtsgebieden van het onderzoek in de symbolische logica en de kunstmatige intelligentie. In dit onderzoek wordt veelal van resolutie, een methode voor de manipulatie van logische formules, als basis voor inferentie uitgegaan.

1.3. PROLOG

Met de programmeertaal PROLOG wordt de gebruiker een omgeving voor programmeren in de logica geboden. In figuur 1 wordt getoond hoe PROLOG de praktische weerslag is van de ideeën die in de voorgaande paragraaf globaal zijn besproken.



Figuur 1: De relatie tussen PROLOG en programmeren in de logica

Een PROLOG systeem bestaat uit twee onderdelen: een PROLOG *database* en een PROLOG *interpretator*. In de PROLOG database brengt de programmeur de logische component van zijn algoritme onder. Voor het vastleggen van informatie in de logische component wordt in PROLOG gebruik gemaakt van het formalisme van de Horn clause, een speciale vorm van de zogenaamde clausal form of logic. De besturende component maakt standaard onderdeel uit van een PROLOG systeem en wordt de interpretator genoemd. De interpretator is gebaseerd op het resolutie principe van Robinson [ROB65].

Voor de lezer die in de meer theoretische aspecten van PROLOG en programmeren in de logica is geïnteresseerd, is met name Kowalski's "Logic for problem solving" [KOW79] een onontbeerlijke bron van informatie. Minder diepgaand maar een aardige inleiding is [GEN85]. Voor een aankomend PROLOG programmeur is het niet strikt noodzakelijk zich in deze theoretische aspecten te verdiepen; als basis voor het leren programmeren in PROLOG is [CLO81] een juiste keuze. De reeds ervaren PROLOG programmeur zal met [KLU85] en [CAM84] zijn kennis zinvol kunnen uitbreiden.

2. DE PROGRAMMEERTAAL PROLOG

In de rest van dit artikel zal de nadruk sterk liggen op het praktisch gebruik van PROLOG. Na een korte inleiding in de taal, zal uitgaande van een concrete probleemstelling, stapsgewijs een PROLOG programma ontwikkeld worden. Hierbij zullen die aspecten belicht worden, die het programmeren in PROLOG zo doen verschillen van het programmeren in een meer conventionele programmeertaal als Pascal of PL/I. Tenslotte zullen enkele toepassingsgebieden de revue passeren, veelal in de vorm van eenvoudige PROLOG programma's voor weliswaar gesimplificeerde maar toch karakteristieke probleemstellingen.

2.1. Inleiding

Het oplossen van een probleem met behulp van PROLOG begint met het onderscheiden van de objecten die voor het betreffende probleem relevant zijn, en de relaties die er tussen deze objecten bestaan. In een probleem waarin verzamelingen een rol spelen, kunnen bijvoorbeeld constanten en verzamelingen van constanten relevante objecten zijn, en kan de relatie "is een element van" tussen een constante en een verzameling bestaan. Als objecten en relaties onderscheiden zijn, dient gespecificeerd te worden, welke feiten er over deze objecten en hun onderlinge relaties bekend zijn, en welke regels er gelden. In het voorbeeld kan "de constante a is een element van de verzameling S " een belangrijk feit zijn en kan de regel "de verzameling X is een deelverzameling van de verzameling Y , als elk element van X een element van Y is" gelden in het probleemgebied. Als feiten en regels zijn vastgelegd, kan de concrete probleemstelling beschouwd worden als een vraag over de objecten en hun onderlinge relaties.

Samengevat komt het specificeren van de logische component van een algoritme neer op:

- het specificeren van *feiten* betreffende de voor het probleem relevante objecten en relaties tussen objecten
- het specificeren van *regels* betreffende de objecten en relaties tussen objecten
- het stellen van *vragen* over objecten en relaties tussen objecten

2.2. De declaratieve semantiek

Informatie (feiten, regels en vragen) wordt in PROLOG met behulp van het formalisme van de *Horn clause* gerepresenteerd. Een Horn clause heeft de volgende vorm

$$B :- A_1, \dots, A_n.$$

Hierin zijn $B, A_1, \dots, A_n$, ($n \geq 0$), atomaire formules. Een atomaire formule is een expressie van de vorm

$$P(t_1, \dots, t_m)$$

waarin P een predicaat (uitspraak) is afhankelijk van m argumenten, ($m \geq 0$), en $t_1, \dots, t_m$ termen zijn. Een term is een constante, een variabele of een functie van variabelen. Bezien in het licht van het voorgaande geven $B, A_1, \dots, A_n$ relaties tussen objecten weer.

Een Horn clause kan als volgt geïnterpreteerd worden:

$$"B \text{ (is waar) als } A_1 \text{ en } A_2 \text{ en } \dots \text{ en } A_n \text{ (waar zijn)}"$$

$A_1, \dots, A_n$ worden de condities van de clause genoemd en B de conclusie.

Als $n = 0$, dat wil zeggen als de condities A_i in de clause ontbreken, dan zijn er aan de conclusie B geen voorwaarden verbonden en representeert de clause het feit B .

```
element(X,[X|_]).           /*1*/
```

is een voorbeeld van een feit betreffende de relatie met de naam `element`. Een relatienaam wordt ook wel een *predicaat* genoemd.

Als de clause compleet is, dat wil zeggen als de clause zowel een of meer condities als een conclusie bevat, geeft zij een regel weer.

```
element(X,[_|Y]) :- element(X,Y).      /*2*/
```

is een regel betreffende de relatie met de naam `element`. Aan de conclusie `element(X,[_|Y])` is in deze regel slechts één conditie verbonden, de conditie `element(X,Y)`.

Als de conclusie *B* in een clause ontbreekt, bevat de clause alleen een aantal condities en is het aan het systeem een conclusie te trekken. In dat geval kan de clause als een vraag gezien worden.

```
?- element(a,[a,b,c]).          /*3*/
```

is een voorbeeld van een dergelijke vraag.

Aan de hand van de drie bovenstaande voorbeelden, kunnen al een aantal PROLOG conventies getoond worden. In het geval dat een clause een feit representeert, wordt in PROLOG het teken `:-` weggelaten; dit teken wordt vervangen door `?-` in het geval dat de clause een vraag representeert. Commentaar dient in PROLOG tussen `/*` en `*/` weergegeven te worden. Alle constanten hebben in PROLOG een naam die met een kleine letter begint: `element`, `a`, `b` en `c` zijn in deze voorbeelden constanten. Namen die met een hoofdletter of underscore, `_`, beginnen duiden in PROLOG een variabele aan. Een relatie tussen objecten wordt aangegeven met de naam van de relatie gevolgd door, tussen haakjes, de objecten, waarop de relatie betrekking heeft.

De relatie met de naam `element` is een relatie tussen twee objecten, een constante en een lijst van constanten. Een lijst wordt in PROLOG tussen vierkante haken weergegeven. In clause 3 bevat de lijst `[a,b,c]` drie constanten: `a`, `b` en `c`. In een lijst kunnen de kop, het eerste element, en de staart, een lijst van de resterende elementen, expliciet van elkaar gescheiden worden met behulp van een verticale streep: `[X|Y]`. Het deel voor de streep, `X`, geeft de kop van de lijst, en het deel na de streep, `Y`, geeft de staart van de lijst aan. In de lijst `[a,b,c]` kunnen de kop en de staart als volgt gescheiden worden: `[a|[b,c]]`.

Omdat elke clause een apart stukje kennis representeert, kan de betekenis van een clause afgeleid worden zonder de andere clauses van de verzameling in de beschouwing te betrekken. De betekenis van clause 1 is dat er een relatie met de naam `element` tussen een constante en een lijst van constanten bestaat, als de kop van de lijst gelijk is aan die constante. Clause 1 is geen uitspraak over een specifieke constante, maar geeft een algemeen feit weer. Dit wordt tot uitdrukking gebracht in het gebruik van de variabele `X`. Clause 2 geeft de andere mogelijkheid weer dat de constante deel uitmaakt van de staart van de lijst. De laatste clause geeft de vraag weer of de constante `a` tot de lijst met constanten `a`, `b` en `c`, behoort.

De betekenis van een clause als een stukje informatie wordt de *declaratieve semantiek* van een clause genoemd.

2.3. De procedurele semantiek

In de voorgaande paragraaf is het formalisme van de Horn clause beschouwd als een formele (specificatie) taal, als een methode om kennis te representeren. Het formalisme kan echter ook als een programmeertaal beschouwd worden. Deze kijk op Horn clauses wordt de *procedurele semantiek* genoemd.

In de procedurele semantiek wordt een verzameling clauses als een programma beschouwd. Elke clause wordt beschouwd als een procedure. In de clause

$$B :- A_1, \dots, A_n.$$

is de conclusie B de procedure-heading, bestaand uit een procedure-naam en een aantal parameters, en vormt $A_1, \dots, A_n$ de body van de procedure. Elke conditie A_i wordt beschouwd als een procedure-aanroep, zodat de procedure-body bestaat uit een reeks na elkaar uit te voeren procedure-aanroepen. Een clause waarin geen conclusie voorkomt, fungeert hierbij als het hoofdprogramma. In het geval dat er in een verzameling clauses een aantal clauses bestaan met hetzelfde predicaat in de conclusie, worden deze clauses als verschillende entries van eenzelfde procedure gezien.

In het navolgende zal geen strikt onderscheid gemaakt worden tussen de beide vormen van semantiek; afhankelijk van het onderwerp van discussie zal aan de terminologie van de declaratieve semantiek, danwel aan de terminologie van de procedurele semantiek de voorkeur gegeven worden.

2.4. De PROLOG interpreter

De drie voorbeelden uit paragraaf 2.2 vormen samen een compleet PROLOG programma:

```
/* 1*/ element(X,[X|_]).
/* 2*/ element(X,[_|Y]) :-
        element(X,Y).

/* 3*/ ?- element(a,[a,b,c]).
```

Als de PROLOG database de clauses 1 en 2 bevat, en clause 3 als vraag aan het PROLOG systeem gesteld wordt, probeert de PROLOG interpreter een antwoord op deze vraag af te leiden door gebruik te maken van de informatie die in de database is opgeslagen.

2.4.1. Matching van clauses

De PROLOG interpreter begint bij het beantwoorden van een vraag met de eerste conditie in de vraagclause en beschouwt deze als procedure-aanroep. Er wordt nu in de PROLOG database gezocht naar een geschikte entry van de aangeroepen procedure; de interpreter zoekt, beginnend bij de eerste clause in de database, een clause waarvan voor de conclusie een match kan worden verkregen met de betreffende conditie. Er kan een *match* van een conclusie en een conditie worden verkregen, als er substituties bestaan, zodanig dat de conclusie en de conditie (syntactisch) gelijk worden als een dergelijke substitutie wordt toegepast. Een dergelijke substitutie bestaat als in de betreffende conclusie en conditie

- de predicaten gelijk zijn, en
- voor de parameters op overeenkomstige parameterposities gelijke waarden gevonden kunnen worden

De meest algemene substitutie vindt vervolgens zowel in de aanroepende clause waarvan de conditie deel uitmaakt, als in de aangeroepen clause waarvan de conclusie onderdeel is, plaats, waarna de condities van de aangeroepen clause achtereenvolgens als procedure-aanroepen worden behandeld. In

het geval dat de aangeroepen clause geen condities bevat, wordt de volgende conditie in de aanroepende clause beschouwd.

In het voorbeeld probeert de interpretator een match te verkrijgen voor de vraag

```
?- element(a,[a,b,c]).
```

De eerste clause in de database waarvan het predicaat in de conclusie gelijk is aan het predicaat in de vraag, is clause 1:

```
element(X,[X|_]).
```

In de vraag staat op de eerste parameterpositie de constante *a*. In clause 1 staat op dezelfde parameterpositie de variabele *X*. Als voor de variabele *X* de waarde *a* wordt gesubstitueerd, is er een match voor de eerste parameterpositie verkregen.

Het uitvoeren van een substitutie wordt in PROLOG *instantiatie* genoemd. Instantiatie vertoont een sterke gelijkenis met assignment in meer conventionele programmeertalen, maar is echter niet hetzelfde. Een instantiatie is een eenmalige toekenning van een waarde aan een variabele; als een variabele geïnstantieerd is, kan er geen andere waarde meer aan toegekend worden. Een uitdrukking als

```
X := X + 1
```

is bij instantiatie niet zondermeer mogelijk. Voorts heeft een variabele in PROLOG een scope van één clause. Buiten die clause hebben die variabele en de instantiaties aan die variabele geen invloed, zodat PROLOG geen globale variabelen kent.

In het voorbeeld moet de variabele *X* in clause 1 voor een match met de vraag voor de eerste parameterpositie met de waarde *a* geïnstantieerd worden. Ook in de tweede parameter in clause 1 heeft *X* dan de waarde *a*, zodat voor een match voor de tweede parameterpositie de lijsten *[a,b,c]* en *[a|_]* vergeleken dienen te worden.

De variabele die in PROLOG met een underscore wordt aangegeven, is de zogenaamde *don't-care variabele*. Voor een don't-care variabele wordt weliswaar met elke mogelijke waarde een match verkregen, maar daarbij vindt geen instantiatie plaats. Een don't-care variabele wordt op een parameterpositie gebruikt indien er geen referentie aan die parameterpositie plaatsvindt.

In het voorbeeld is derhalve ook voor de tweede parameterpositie overeenstemming bereikt en is er een volledige match van de vraag met clause 1 gevonden. Omdat aan clause 1 geen condities zijn verbonden en ook de vraag geen verdere condities bevat, antwoordt de interpretator met "yes" op de gestelde vraag.

Stel nu dat in plaats van de in het voorgaande behandelde vraag, de vraag

```
?- element(a,[b,a,c]).
```

gesteld wordt. De interpretator probeert eerst weer een match met clause 1 te verkrijgen:

```
element(X,[X|_]).
```

Uit een vergelijking van de respectievelijke eerste parameters volgt ook nu weer dat de variabele *X* met de constante *a* moet worden geïnstantieerd. Ook in de tweede parameterpositie in clause 1 heeft *X* de waarde *a*, zodat voor een match voor de tweede parameterpositie de lijsten *[b,a,c]* en *[a|_]* vergeleken moeten worden. Het is duidelijk dat voor de tweede parameterpositie geen match kan worden gevonden. Aangezien de instantiatie van *X* met *a* de enig mogelijke instantiatie was om gelijke waarden voor de respectievelijke eerste parameters te verkrijgen, kunnen er geen gelijke waarden voor de tweede parameters gevonden worden. De PROLOG interpreter vindt voor de vraag derhalve geen match met clause 1.

Clause 2 wordt nu geprobeerd.

```
element(X,[_|Y]) :-
    element(X,Y).
```

Uit een vergelijking van de eerste parameters volgt weer dat *X* met *a* moet worden geïnstantieerd. Voor de tweede parameterpositie moeten de lijsten *[b,a,c]* en *[_|Y]* vergeleken worden. Aangezien in de tweede lijst de kop een don't-care variabele is, is het slechts nodig de staarten van de respectievelijke lijsten te vergelijken. Voor de tweede parameterpositie kan een match worden verkregen door de variabele *Y* met de lijst *[a,c]* te instantiëren. Er is nu een volledige match van de vraag met de conclusie van clause 2 bereikt en de gevonden instantiaties worden uitgevoerd:

```
element(a,[_|[a,c]]) :-
    element(a,[a,c]).
```

Merk op dat de instantiaties van *X* en *Y* in de gehele clause doorwerken. Aan de conclusie van clause 2 is een conditie verbonden: de conclusie mag pas dan getrokken worden als aan deze conditie is voldaan. De interpreter behandelt de conditie derhalve als nieuwe vraag:

```
?- element(a,[a,c]).
```

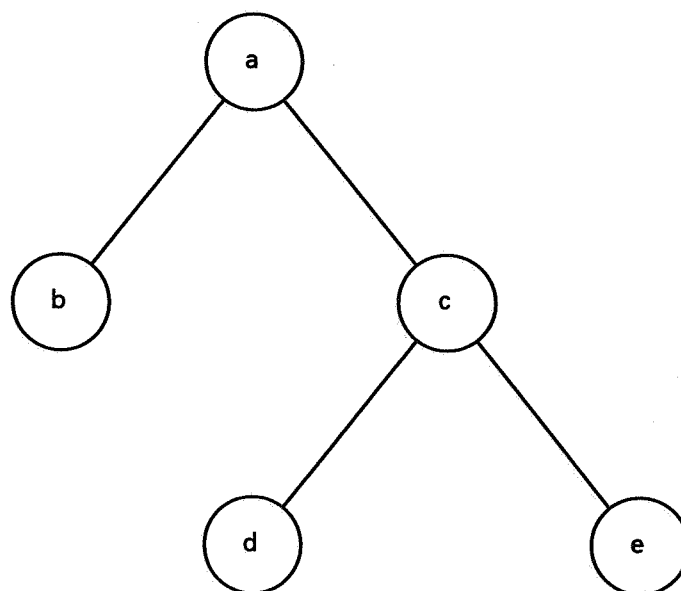
Met behulp van een match met clause 1 op de bovenbeschreven wijze wordt het antwoord "yes" op deze vraag afgeleid. Omdat hiermee aan de conditie van clause 2 is voldaan, mag de bijbehorende conclusie getrokken worden, en reageert de interpreter met het antwoord "yes" op de oorspronkelijke vraag.

2.4.2. Backtracking

Als het systeem er na een aantal instantiaties en matches, op een gegeven moment, niet in slaagt een volgende match te vinden, wordt systematisch naar mogelijke alternatieven voor de eerder gedane instantiaties en matches gezocht. Dit proces wordt *backtracking* genoemd. Beschouw het onderstaande PROLOG programma

```
/*1*/ tak(a,b).
/*2*/ tak(a,c).
/*3*/ tak(c,d).
/*4*/ tak(c,e).
/*5*/ pad(X,X).
/*6*/ pad(X,Y) :-
    tak(X,Z),
    pad(Z,Y).
```

De clauses 1 tot en met 4 representeren de binaire boom van figuur 2, met behulp van het predicaat **tak**: **tak(a,b)** heeft de betekenis "er bestaat een tak van knoop a naar knoop b".



Figuur 2: Een binaire boom

In de clauses 5 en 6 is met behulp van het predicaat **pad** vastgelegd wanneer er een pad bestaat tussen twee knopen.

De definitie van het begrip **pad** is *recursief*; in de beschrijving van het begrip **pad** wordt gebruik gemaakt van datzelfde begrip **pad**. Een recursieve definitie bestaat in het algemeen uit twee delen: een of meer stopcriteria, beschrijvingen van de eenvoudigste situaties waarop het begrip betrekking heeft, en de recursie, waarin wordt aangegeven hoe uit een situatie waarop het begrip betrekking heeft, een nieuwe, eenvoudigere situatie wordt afgeleid waarop dat begrip ook betrekking heeft.

Aan het systeem wordt de vraag

?- pad(a,d).

gesteld. De interpreter probeert voor deze vraag eerst een match met clause 5 te verkrijgen:

pad(X,X).

Voor een match voor de eerste parameterpositie moet de variabele **X** met de knoop **a** geïnstantieerd worden, waarna er geen match meer voor de tweede parameterpositie kan worden gevonden. Clause 6 wordt nu geprobeerd:

pad(X,Y) :- tak(X,Z), pad(Z,Y).

De interpreter slaagt er nu wel in een match voor de vraag te verkrijgen, door de variabele **X** met **a** en **Y** met **d** te instantiëren. Deze instantiaties worden weer in de gehele aangeroepen clause doorgevoerd:

$\text{pad}(a,d) \text{ :- tak}(a,Z), \text{pad}(Z,d).$

zodat aan de conclusie de twee condities $\text{tak}(a,Z)$ en $\text{pad}(Z,d)$ zijn verbonden. Pas als aan deze condities is voldaan mag de conclusie $\text{pad}(a,d)$ getrokken worden. De interpretator werkt de twee nieuwe vragen van links naar rechts af. Voor de vraag

$?- \text{tak}(a,Z).$

vindt de interpretator een match met clause 1

$\text{tak}(a,b).$

door de variabele Z met b te instantiëren. Deze instantiatie wordt niet alleen in de betreffende vraag maar in de gehele clause waarvan de vraag deel uitmaakt, doorgevoerd:

$\text{pad}(a,d) \text{ :- tak}(a,b), \text{pad}(b,d).$

zodat de interpretator nu de conditie $\text{pad}(b,d)$ dient te bekijken. Voor deze vraag wordt weer geen match met clause 5 verkregen, maar wel met clause 6 door de variabele X met b en de variabele Y met d te instantiëren, resulterend in:

$\text{pad}(b,d) \text{ :- tak}(b,Z), \text{pad}(Z,d).$

Deze instantiaties van de variabelen X en Y zijn toegestaan omdat de eerder gedane instantiaties aan X en Y op een andere vraag betrekking hadden. Aan de match van de vraag $\text{pad}(b,d)$ met clause 6 zijn weer twee condities verbonden. Voor de eerste conditie

$?- \text{tak}(b,Z).$

kan geen match gevonden worden: in de boom van figuur 2 loopt er geen tak vanuit knoop b.

De PROLOG interpretator maakt nu de laatst verkregen match ongedaan door de betreffende instantiaties ongedaan te maken, en probeert een nieuwe match voor de oorspronkelijke vraag te verkrijgen. In het voorbeeld is de match van de vraag $\text{pad}(b,d)$ met clause 6 de laatstgevonden match. De instantiaties voor X en Y in clause 6 worden nu ongedaan gemaakt, en de interpretator probeert een nieuwe match voor $\text{pad}(b,d)$ te vinden. De match met clause 6 was echter de enige mogelijke, zodat de interpretator nog een stap verder teruggaat.

De match van $\text{tak}(a,Z)$ met clause 1 wordt ongedaan gemaakt door de instantiatie van Z met b ongedaan te maken. Voor de vraag $?- \text{tak}(a,Z)$ is nu een match met clause 2 te verkrijgen door de variabele Z met c te instantiëren. De vraag $?- \text{tak}(a,Z)$ was ontstaan als gevolg van de match van de vraag $?- \text{pad}(a,d)$ met clause 6:

$\text{pad}(a,d) \text{ :- tak}(a,Z), \text{pad}(Z,d).$

Het ongedaan maken van de instantiatie van Z en vervolgens de nieuwe instantiatie van Z hebben weer invloed op de gehele aanroepende clause:

$\text{pad}(a,d) \text{ :- tak}(a,c), \text{pad}(c,d).$

zodat in plaats van de conditie $\text{pad}(b,d)$ nu de conditie $\text{pad}(c,d)$ moet worden bekeken.

Achtereenvolgens met de clauses 6, 3 en 5 wordt het antwoord "yes" op de vraag `?- pad(c,d)` afgeleid. Aan beide condities aan de match van de oorspronkelijke vraag met clause 6 is nu voldaan, zodat de interpreterator op de vraag `?- pad(a,d)` het antwoord "yes" geeft.

De volgorde waarin de clauses in de PROLOG database gespecificeerd worden, is van essentieel belang. Volgens de principes van programmeren in de logica is de logische component een verzameling clauses, waarin de onderlinge volgorde dus niet van belang is. Maar in het voorgaande zal bij de beschrijving van de werking van de PROLOG interpreterator duidelijk geworden zijn dat de clauses in de database worden bekeken in de volgorde waarin ze opgeslagen zijn, zodat de volgorde van opslag van de clauses wel degelijk van belang is in PROLOG. Dit impliceert dat de programmeur zich hiervan bewust dient te zijn bij het opstellen van een PROLOG programma. Bij het aanbrengen van zo'n ordening is het noodzakelijk de clauses die bij recursie een stopcriterium of anderszins een uitzondering vormen, eerder te specificeren in de database dan de clauses die de algemene regel voor de betreffende relatie weergeven: als in de bovenstaande specificatie van een pad de twee clauses 5 en 6 in omgekeerde volgorde in de database waren opgeslagen, zou de interpreterator nooit op het stopcriterium testen.

3. TOEPASSINGSGEBIEDEN

In de inleiding zijn een vijftal gebieden genoemd, waarop PROLOG met succes wordt toegepast. In deze paragraaf zullen deze toepassingsgebieden nader worden besproken, waar mogelijk in de vorm van eenvoudige voorbeelden.

3.1. Natuurlijke-taal verwerking

Reeds vanaf zijn ontstaan is PROLOG toegepast bij natuurlijke-taal verwerking; Alain Colmerauer en zijn groep hebben veel onderzoek verricht op dit terrein, waarbij zij van PROLOG gebruik hebben gemaakt. Allerlei projecten met uiteenlopende uitgangspunten en vaak ook verschillende doelstellingen, worden onder de noemer natuurlijke-taal verwerking gevat, omdat zij op de een of andere wijze computers een natuurlijke taal willen laten hanteren. Met name voor die projecten die sterk gebaseerd zijn op een syntactische beschrijving van een taal, is PROLOG een voor de hand liggende keuze. Maar ook voor de projecten die de semantiek als uitgangspunt gekozen hebben, is PROLOG als "symboolverwerker" van nut gebleken.

Als basis voor een syntactische beschrijving van een taal wordt veelal een contextvrije grammatica gekozen. Met behulp van onderstaande grammatica wordt bijvoorbeeld een klein deel van de Nederlandse declaratieve zinnen beschreven:

```

zin → naamwoordelijk gezegde
      werkwoordelijk gezegde

naamwoordelijk gezegde → zelfstandig naamwoord
naamwoordelijk gezegde → lidwoord
                           bijvoeglijk naamwoord
                           zelfstandig naamwoord

werkwoordelijk gezegde → werkwoord
                           naamwoordelijk gezegde

```

Een dergelijke syntax-beschrijving laat zich eenvoudig in PROLOG omzetten:

```

zin(X) :- naamwoordelijk_gezegde(X,Y),
          werkwoordelijk_gezegde(Y,[ ]).

naamwoordelijk_gezegde([X|Y],Y) :-
          zelfstandig_naamwoord(X).
naamwoordelijk_gezegde([X1,X2,X3|Y],Y) :-
          lidwoord(X1),
          bijvoeglijk_naamwoord(X2),
          zelfstandig_naamwoord(X3).

werkwoordelijk_gezegde([X|Y],Z) :-
          werkwoord(X),
          naamwoordelijk_gezegde(Y,Z).

```

Dit PROLOG-programma kan gebruikt worden om de syntactische correctheid van bepaalde Nederlandse zinnen te onderzoeken, mits er een woordenboek aan gekoppeld wordt waarin kan worden opgezocht tot welke woordklasse een woord behoort. Een dergelijk woordenboek kan ook in PROLOG geschreven worden. Voor de illustratieve doeleinden van deze paragraaf kan met het onderstaande, zeer primitieve woordenboek worden volstaan:

```

lidwoord(de).
lidwoord(het).
lidwoord(een).

werkwoord(eet).
werkwoord(eten).

bijvoeglijk_naamwoord(grote).
bijvoeglijk_naamwoord(rode).

zelfstandig_naamwoord(jongen).
zelfstandig_naamwoord(appel).

```

De zin

de grote jongen eet een rode appel

kan nu in lijstrepresentatie aan het programma worden aangeboden voor een controle op syntactische correctheid:

```
?- zin([de,grote,jongen,eet,een,rode,appel]).
```

De interpreter antwoordt op deze vraag "yes".

Hoe primitief bovenstaande grammatica ook mag lijken, bij veel onderzoeksprojecten wordt zo'n grammatica als uitgangspunt genomen. Een dergelijk primitieve beschrijving wordt vervolgens verfijnd door (contextgevoelige) syntactische en semantische restricties te specificeren, in de vorm van

metaregels. Ook deze verfijningen blijken zich eenvoudig in PROLOG te laten formuleren.

3.2. Compiler constructie

Een compiler is een programma dat een programma geschreven in een brontaal (meestal een hogere programmeertaal) vertaalt naar een programma in een doeltaal (meestal een assembleertaal). Er bestaat een sterke analogie met het automatisch vertalen van natuurlijke taal, met dit verschil dat een programmeertaal zich in het algemeen beter formeel laat beschrijven dan een natuurlijke taal. Het construeren van een compiler voor een programmeertaal komt neer op het schrijven van een formele specificatie van die taal, en vervolgens het specificeren en implementeren van een verzameling regels om de elementen van die specificatie te vertalen naar stukjes (object) code in de doeltaal.

Bij het gebruik van PROLOG voor de constructie van een compiler is met de formele specificatie van de brontaal en de vertaalregels nagenoeg de implementatie in PROLOG verkregen. Uit het feit dat de specificatie en de implementatie van een compiler in PROLOG zo nauw bij elkaar aansluiten, vloeien een aantal belangrijke voordelen voort:

- na de specificatie van de compiler is de implementatie vrijwel tot een trivialiteit gereduceerd, waarmee de kans op fouten in de compiler zeer klein geworden is.
- de implementatie is goed leesbaar, omdat elke regel in de specificatie ook als afzonderlijk onderdeel in de implementatie te herkennen is.
- uitbreidingen aan de brontaal zijn eenvoudig in de reeds bestaande compiler te verwerken.

Deze voordelen maken dat in PROLOG in betrekkelijk korte tijd compilers kunnen worden ontwikkeld. Daarbij dient wel de kanttekening geplaatst te worden dat de efficiëntie van met PROLOG geconstrueerde compilers in belangrijke mate afhankelijk is van de efficiëntie van het PROLOG systeem zelf. Dit laatste punt nu, staat momenteel volop in de belangstelling.

3.3. Expertsystemen

Expertsystemen zijn computersystemen die advies kunnen verlenen op een niveau dat vergelijkbaar is met dat van menselijke deskundigen in een specifiek vakgebied. Dit wordt gerealiseerd door de probleem-gerichte kennis van een deskundige voor een computer toegankelijk te maken. Een expertstelsysteem bestaat uit twee gescheiden componenten:

- een knowledge base of kennisbank, de component waarin de expertkennis wordt vastgelegd
- een inference engine of inferentiemachine, de component die de kennis in de knowledge base manipuleert

Vervangt men de knowledge base van een expertstelsysteem door een knowledge base op een ander gebied, dan wordt hiermee een ander expertstelsysteem verkregen.

De splitsing van kennis in een knowledge base en oplossingsmethode in een inference engine in een expertstelsysteem vertoont een sterke gelijkenis met de splitsing van kennis in een PROLOG database en oplossingsmethode in een PROLOG interpreterator in een PROLOG systeem.

De vraag in hoeverre PROLOG een geschikte taal is voor de implementatie van een expertstelsysteem, bestaat derhalve uit twee deelvragen:

- de vraag hoeveel vrijheid het PROLOG systeem biedt voor de representatie van expertkennis in de PROLOG database
- de vraag in hoeverre de PROLOG interpreterator als basis voor de inference engine van een expertstelsysteem kan fungeren

In expertsystemen kan van verschillende formalismen gebruik gemaakt worden om kennis te representeren. Een van die representatieformalismen, *productieregels*, sluit zeer nauw aan bij het formalisme van de Horn clause, de methode die in een PROLOG systeem voor kennisrepresentatie wordt gebruikt. Informeel kan een productieregel als volgt worden weergegeven:

als aan een aantal condities is voldaan
dan wordt een conclusie getrokken

Voor een expertsysteem dat gebruikt wordt om dieren te classificeren, is de informatie dat een vogel zich vliegend beweegt en zich voortplant door middel van eieren, bijvoorbeeld in onderstaande informele produktieregel ondergebracht:

als het dier vliegt en eieren legt,
dan is het een vogel.

Ook bij expertsystemen wordt bij de formele weergave van condities en conclusies gebruik gemaakt van predicaten met bijbehorende argumenten. Dit heeft tot gevolg dat met de specificatie van de produktieregels, de implementatie in PROLOG verkregen is.

Bovenstaande produktieregel wordt formeel bijvoorbeeld weergegeven door:

```
dierklasse(Dier,vogel) :-
    beweging(Dier,vliegen),
    voortplanting(Dier,eieren).
```

Een van de basistechnieken die bij expertsystemen voor kennismanipulatie wordt gebruikt, *top-down inference*, vertoont een sterke overeenkomst met de oplossingsmethodiek die in de PROLOG interpreterator geïncorporeerd is. Top-down inference is het redeneren uitgaande van een doel naar de noodzakelijke gegevens, van conclusie naar condities.

Een prototype expertsysteem dat voor de representatie van kennis gebruik maakt van produktieregels en voor het afleiden van nieuwe informatie van top-down inference, blijkt in PROLOG zonder veel inspanning ontwikkeld te kunnen worden, aangezien de essentiële zaken standaard door de PROLOG interpreterator geboden worden. Een nagenoeg compleet op deze principes gebaseerd systeem in PROLOG wordt in [LUC86] beschreven.

Ook als men bij de ontwikkeling van een expertsysteem van een andere oplossingsmethodiek en van een ander representatieformalisme gebruik wenst te maken, kan PROLOG handig gereedschap zijn, hoewel het voordeel dan beduidend is afgenomen.

3.4. Symbolische wiskunde

Een toepassingsgebied waar de mogelijkheid van PROLOG om met symbolen te manipuleren van belang is, is de symbolische wiskunde. Om de eerste afgeleide van een functie te bepalen, bestaan een aantal recursieve regels, zoals

$$\begin{aligned}
 d \frac{x}{dx} &= 1 \\
 d \frac{c}{dx} &= 0, \text{ met } c \text{ constant} \\
 d \frac{(f+g)}{dx} &= d \frac{f}{dx} + d \frac{g}{dx} \\
 d \frac{(f-g)}{dx} &= d \frac{f}{dx} - d \frac{g}{dx} \\
 d \left(f \cdot \frac{g}{dx} \right) &= g \cdot d \frac{f}{dx} + f \cdot d \frac{g}{dx} \\
 d \frac{(-f)}{dx} &= -d \frac{f}{dx}
 \end{aligned}$$

Deze regels zijn eenvoudig in PROLOG-clauses om te zetten:

```

/* 1*/ d(X,X,1) :- !.
/* 2*/ d(C,X,0) :-
        atomic(C).
/* 3*/ d(F+G,X,DF+DG) :-
        !, d(F,X,DF), d(G,X,DG).
/* 4*/ d(F-G,X,DF-DG) :-
        !, d(F,X,DF), d(G,X,DG).
/* 5*/ d(F*G,X,G*DF+F*DG) :-
        !, d(F,X,DF), d(G,X,DG).
/* 6*/ d(-F,X,-DF) :-
        !, d(F,X,DF).

```

In paragraaf 4 wordt op de betekenis van ! en het predicaat `atomic` ingegaan.

Stelt men nu de vraag:

```
?- d(2*x-5,x,Afgeleide).
```

Dan zal de interpretator reageren met:

```
Afgeleide = x*0+2*1-0
```

Door een aantal clauses te ontwikkelen, die overbodige constanten en operatoren verwijderen, kan deze formule vereenvoudigd worden.

3.5. Database systemen

Een database is een gestructureerde gegevensverzameling. Er bestaan verschillende technieken om een gegevensverzameling te structureren, en derhalve verschillende typen databases. De twee bekendste typen zijn momenteel de *netwerk database* en de *relationele database*. Met name de wijze waarop gegevens in een relationele database zijn georganiseerd sluit nauw aan bij de wijze waarop in PROLOG informatie wordt vastgelegd: in een relationele database wordt informatie in relaties vastgelegd.

Stel dat een database ontworpen dient te worden die informatie bevat over geneesmiddelen.

Informatie hiervoor wordt meestal in een tabel weergegeven:

geneesmiddel			
generische naam	merk naam	dosis	bijwerking
paracetamol	Panadol	500 mg	toxische hepatitis
cloxacilline	Orbenin	250 mg	anafylactische shock

De relatie heeft de naam *geneesmiddel*. Een relatie is een verzameling *tupels*, rijen in de tabel, waarin elk tupel een feit weergeeft. De relatie *geneesmiddel* bevat derhalve twee tupels:

<paracetamol, Panadol, 500 mg, toxische hepatitis>
 <cloxacilline, Orbenin, 250 mg, anafylactische shock>

In PROLOG kunnen deze tupels, samen met de relatienaam, als volgt weergegeven worden:

```
geneesmiddel(paracetamol,panadol,'500mg',toxische_hepatitis).
geneesmiddel(cloxacilline,orbenin,'250mg',anafylactische_shock).
```

Met behulp van een vraagtaal, meestal met de Engelse term *query language* aangeduid, kan informatie uit de database worden geselecteerd. In een *database management system* (DBMS) bestaan voor het ontwerp en de manipulatie van databases verschillende talen. Voor de selectie van informatie uit een met PROLOG ontwikkelde database, kan echter van PROLOG gebruik gemaakt worden. Met de onderstaande clauses kan respectievelijk de dosis van een geneesmiddel en de eventuele bijwerkingen opgevraagd worden:

```
/*1*/  dosis(Farmacon,D) :-
        geneesmiddel(Farmacon,_,D,_);
        geneesmiddel(_,Farmacon,D,_).

/*2*/  bijwerking(Farmacon,B) :-
        geneesmiddel(Farmacon,_,_,B);
        geneesmiddel(_,Farmacon,_,B).
```

De dosis van het geneesmiddel paracetamol kan worden gevonden met behulp van de vraag

```
?- dosis(paracetamol,X).
```

De PROLOG interpretator antwoordt:

```
X=500mg
```

4. PROLOG STANDAAPREDICATEN

Tot nu toe zijn in de besproken voorbeelden vrijwel alle predicaten speciaal voor het betreffende probleem gedefinieerd. In elk PROLOG systeem bestaan een aantal predicaten, die reeds gedefinieerd zijn en waarvan een programmeur desgewenst gebruik kan maken. Deze predicaten worden in het algemeen *standaardpredicaten* genoemd, om ze te onderscheiden van de door de programmeur

gedefinieerde predicaten.

In deze paragraaf zullen een aantal standaardpredicaten en hun gebruik besproken worden. Alleen de meest gebruikte predicaten zullen aan bod komen. Voor een compleet overzicht wordt verwezen naar de documentatie van het aangeschafte PROLOG systeem en naar [CLO80]. Deze bespreking is gebaseerd op C-PROLOG, versie 1.5, een in C geschreven PROLOG systeem, door David Warren, David Bowen, Lawrence Byrd en Luis Pereira ontworpen voor het UNIX operating system.

4.1. Het inlezen van programma's

Met behulp van het predicaat `consult` kunnen programma's uit files in de PROLOG database ingelezen worden; de clause

```
?- consult(file).
```

geeft de interpretator opdracht de file met de naam `file` in te lezen. Als er een aantal programma's afkomstig uit verschillende files in de database dienen te worden ingelezen, kan in plaats van

```
?- consult(file1),...,consult(filen).
```

gebruik gemaakt worden van de verkorte lijst-notatie

```
?- [file1,...,filen].
```

4.2. Invoer en uitvoer

Het afdrukken van tekst op het beeldscherm kan plaatsvinden met behulp van

```
write(X).
```

Als voor de uitvoering van `write` de variabele `X` met een term is geïnstantieerd, wordt deze term op het beeldscherm zichtbaar gemaakt. Bijvoorbeeld

```
write(uitvoer).
```

zorgt ervoor dat de term `uitvoer` op het scherm wordt afgedrukt.

```
write('Dit is uitvoer.').
```

leidt tot

```
Dit is uitvoer.
```

In

```
write(Uitvoer).
```

vat de interpretator `Uitvoer` als variabele op en wordt de waarde die de variabele `Uitvoer` op dat moment heeft, op het beeldscherm afgedrukt.

Met het predicaat `read` kunnen termen van het scherm ingelezen worden. In

```
read(X).
```

dient X een ongeïnstantieerde variabele te zijn; X wordt geïnstantieerd met de eerstvolgende term die op het beeldscherm verschijnt, mits deze term met een punt en een carriage return is afgesloten.

De clause

$nl.$

tenslotte, zorgt ervoor dat de volgende in- of uitvoer op een nieuwe regel plaatsvindt.

4.3. Rekenkundige predicaten

PROLOG kent een aantal rekenkundige predicaten. Deze predicaten hebben als argumenten rekenkundige expressies, die geëvalueerd worden. Rekenkundige expressies worden met behulp van infix-operatoren als $+$, $-$, $*$ en $/$ voor respectievelijk optellen, aftrekken, vermenigvuldigen en delen, opgebouwd. Voor de uitvoering van de volgende rekenkundige predicaten dienen alle variabelen in de rekenkundige expressies X en Y , die als argument optreden, geïnstantieerd te zijn; X en Y worden geëvalueerd voordat de aangegeven test plaatsvindt.

$X > Y.$
 $X < Y.$
 $X \geq Y.$
 $X \leq Y.$

Het predicaat is gedraagt zich afwijkend; voor de uitvoering van

$Z \text{ is } X.$

dient X met een rekenkundige expressie geïnstantieerd te zijn; deze expressie wordt geëvalueerd en Z wordt met het resultaat geïnstantieerd, als Z een ongeïnstantieerde variabele is. Als Z reeds geïnstantieerd is, wordt de waarde van Z vergeleken met het resultaat van de evaluatie van X ; als Z geïnstantieerd is met een rekenkundige expressie, wordt deze *niet* eerst geëvalueerd.

4.4. Het onderzoeken van instantiaties

Met een aantal predicaten kan onderzocht worden of een variabele op dat moment geïnstantieerd is, en zo ja, met welk type waarde. Aan de conditie

$var(X).$

is voldaan als X een ongeïnstantieerde variabele is; als X op het moment van de uitvoering van $var(X)$ geïnstantieerd is, is niet aan de conditie voldaan. Het predicaat $nonvar$ in

$nonvar(X).$

heeft precies de tegenovergestelde werking.

Met behulp van

```
atom(X).
atomic(X).
integer(X).
```

wordt respectievelijk getest of de variabele *X* op dat moment met een term, een constante of met een geheel getal is geïnstantieerd.

4.5. Beïnvloeding van de besturing

Er bestaan in PROLOG een aantal predicaten die de programmeur in staat stellen de besturing die door de PROLOG interpretator geboden wordt, expliciet te beïnvloeden.

Als het predicaat *true* als conditie wordt gespecificeerd

```
true.
```

is dit een conditie waaraan altijd voldaan is; aan de conditie

```
fail.
```

is daarentegen nooit voldaan.

Als een conditie *P* als argument optreedt van het predicaat *not*

```
not(P).
```

dan is aan de nieuwe conditie voldaan als niet aan de conditie *P* is voldaan, en omgekeerd.

De *cut*, genoteerd als *!*, is een predicaat zonder argumenten. Als de *cut* als conditie in een clause wordt gebruikt

```
!.
```

is bij het uitvoeren van de clause aan deze conditie voldaan. Bij backtracking kan echter niet opnieuw aan de *cut* voldaan worden. De *cut* oefent in een dergelijk geval aanzienlijke invloed uit op de rest van het backtracking-proces: de interpretator verworpt de clause waarvan de *cut* deel uitmaakt, en bekijkt ook geen alternatieven meer voor de procedure-aanroep die tot de uitvoering van de betreffende clause leed. Als in de clause

$$B :- A_1, A_2, A_3, !, A_4, A_5.$$

voor de conditie *A*₅ geen match meer kan worden gevonden, worden op de gebruikelijke wijze mogelijke alternatieven voor clause *A*₄ bekeken. Bij elk alternatief voor *A*₄ wordt opnieuw geprobeerd een match voor conditie *A*₅ te verkrijgen. Als er geen alternatieven meer voor *A*₄ zijn, zou *A*₃ opnieuw moeten worden bekeken. Tussen *A*₃ en *A*₄ is echter de *cut* gebruikt. Backtracking over de *cut* heeft nu tot gevolg dat voor de condities *A*₁ tot en met *A*₃ geen alternatieven worden gezocht. Ook voor de aanroep van *B* worden geen alternatieven meer bekeken. De laatste procedure-aanroep voor de aanroep van *B* is de eerste die voor een nieuwe beschouwing in aanmerking komt.

Er zijn drie situaties waarin veelal van de cut gebruik wordt gemaakt. De cut wordt in de eerste plaats gebruikt om aan te geven dat de geselecteerde clause de enige is, die van toepassing is op een bepaald (deel) probleem. Stel dat, als aan de conditie A_1 voldaan is, de clause

$$B :- A_1, A_2.$$

de enige clause is die van toepassing is op het beschouwde probleem. Als er nog meer clauses zijn met hetzelfde predicaat in de conclusie, dan worden deze clauses onnodig bij de backtracking betrokken. Door de cut ná de doorslaggevende conditie te specificeren

$$B :- A_1, !, A_2.$$

wordt dit onnodige zoekproces voorkomen.

De cut wordt in de tweede plaats gebruikt om aan te geven dat een bepaalde procedure-aanroep nooit tot succes kan leiden als aan bepaalde voorwaarden is voldaan. In een dergelijke situatie wordt de cut in combinatie met het predicaat `fail` gebruikt. Stel dat de conclusie B nooit kan worden afgeleid als aan de conditie A_1 is voldaan. In

$$B :- A_1, !, \text{fail}.$$

is deze informatie vastgelegd om onnodige pogingen om B af te leiden met andere clauses te voorkomen.

De derde situatie waarin de cut wordt gebruikt, is de situatie waarin een deelprobleem slechts één oplossing heeft. Stel dat de conclusie B alleen dan kan worden getrokken, als aan de conditie A_1 is voldaan. In

$$B :- A_1, !.$$

is vastgelegd dat de gevonden oplossing de enige is en dat het derhalve niet nodig is naar andere oplossingen te zoeken.

De cut wordt dus gebruikt om kennis betreffende de eenduidigheid van bepaalde situaties vast te leggen. In het voorgaande is er al op gewezen dat een PROLOG programmeur bij het opstellen van de clauses voor zijn probleem zich bewust dient te zijn van de werking van de PROLOG interpreter; omdat de cut een zo sterke invloed op backtracking uitoefent, is juist bij het toepassen van de cut voorzichtigheid geboden. Het volgende voorbeeld uit [CLO81] illustreert tot welke onwenselijkheden een onvoorzichtig gebruik van de cut kan leiden.

In de onderstaande drie clauses is het aantal ouders van iemand gedefinieerd: iedereen heeft twee ouders, behalve Adam en Eva die er nul hebben.

```
aantal_ouders(adam,0) :- !.
aantal_ouders(eva,0) :- !.
aantal_ouders(X,2).
```

Op de vraag

```
?- aantal_ouders(eva,2).
```

antwoordt de interpretator "yes". Hoewel dit antwoord misschien onverwacht is, zal een ieder snel kunnen inzien hoe het is afgeleid.

4.6. Beïnvloeding van de inhoud van de database

Een PROLOG systeem biedt de programmeur een aantal hulpmiddelen om de inhoud van de database, het programma, tijdens de executie te wijzigen. Aan de database kunnen clauses worden toegevoegd op een van de volgende manieren:

```
asserta(C).
assertz(C).
```

Als *C* met een clause geïnstantieerd is, zorgt **asserta** ervoor dat de clause vóór alle andere clauses die op hetzelfde predicaat betrekking hebben, wordt toegevoegd; **assertz** voegt de clause ná alle andere clauses met hetzelfde predicaat toe.

Met behulp van

```
retract(C).
```

wordt de clause waarmee de variabele *C* geïnstantieerd is, uit de database verwijderd.

5. PROBLEM SOLVING MET PROLOG: de kortste-pad-bepaling in een graaf

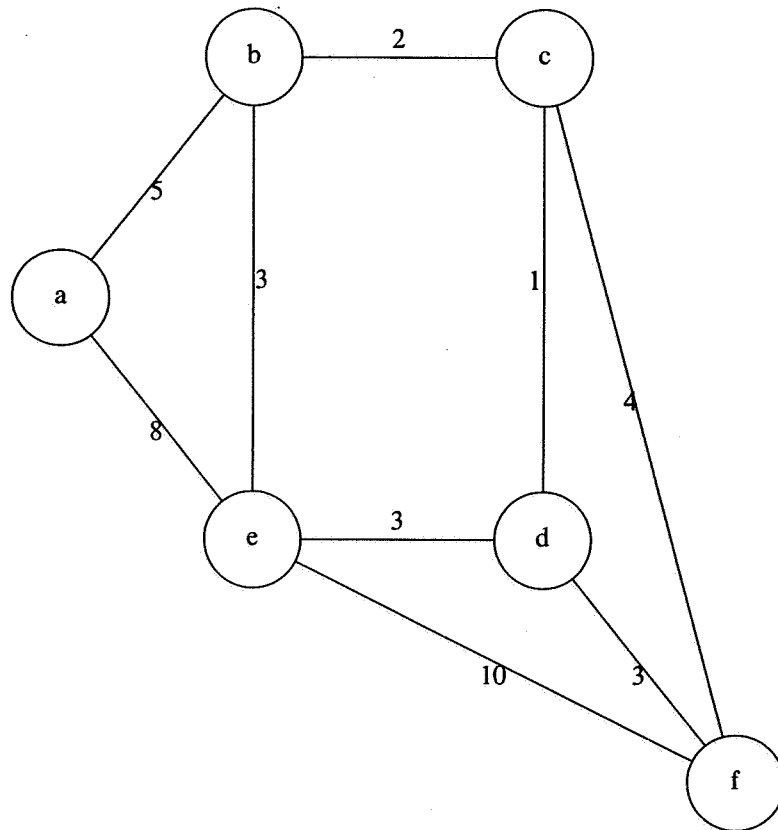
Zelfs als een aankomend PROLOG gebruiker het principe van de Horn clause kent en weet hoe de besturende component van een PROLOG systeem in theorie werkt, blijkt het in de praktijk niet mee te vallen voor een concreet probleem een PROLOG programma te schrijven. In deze paragraaf wordt, uitgaande van een concrete probleemstelling, in een aantal stappen een bijbehorend PROLOG programma ontwikkeld, om te laten zien hoe men te werk kan gaan.

5.1. De probleemstelling

Het probleem dat hier onder de loep genomen wordt, staat in de literatuur bekend als de *kortste-pad-bepaling* in een graaf, een probleem dat elegant met behulp van backtracking op te lossen is. Omdat van het backtracking algoritme in de PROLOG interpretator gebruik kan worden gemaakt, is met de specificatie van het probleem nagenoeg de implementatie verkregen.

Zij gegeven een ongerichte graaf $G = (V, E)$, met een verzameling knopen *V* en een verzameling takken $E \subset V \times V$, en een labelfunctie $l: E \rightarrow \mathbb{N}$, die aan elke tak $e \in E$ een integer toekent, die de lengte van de betreffende tak uitdrukt. Een *pad* is een rij takken $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$. De lengte van een pad is gelijk aan de som van de lengten van de samenstellende takken. Een pad wordt *acyclisch* genoemd als alle knopen op het pad verschillen. In het andere geval wordt het pad *cyclisch* genoemd. Als een graaf een cyclisch pad bevat, wordt van een cyclische graaf gesproken. Een acyclische graaf bevat geen enkel cyclisch pad. De vraagstelling is: bepaal voor twee gegeven knopen v_i en v_j een pad met minimale lengte.

Ter oriëntatie wordt de graaf in figuur 3 als voorbeeld genomen.



Figuur 3: Een cyclische graaf

5.2. De representatie van de graaf

De eerste stap om tot een oplossing van dit probleem in de vorm van een PROLOG programma te komen, is het onderscheiden van de voor dit probleem relevante objecten en relaties tussen objecten. Concreet aanwijsbare objecten in een graaf zijn de knopen en de labels. Tussen twee knopen en een label kan een relatie bestaan, die de lengte van de tak tussen deze twee knopen aangeeft. Deze relatie wordt de naam **tak** gegeven.

De kennis die betrekking heeft op het probleem moet vervolgens in feiten en regels over objecten en relaties worden uitgedrukt. Het feit dat er een tak met lengte 5 bestaat tussen de knopen a en b in de graaf van figuur 3, wordt gerepresenteerd als

tak(a,b,5).

De gehele graaf wordt gerepresenteerd door de volgende verzameling feiten

```

tak(a,b,5).
tak(a,e,8).
tak(b,e,3).
tak(b,c,2).
tak(d,e,3).
tak(c,d,1).
tak(e,f,10).
tak(d,f,3).
tak(c,f,4).

```

5.3. De specificatie van een pad

Voor het vastleggen van feiten en regels over paden in een graaf wordt behalve een relatie **tak**, nog een relatie gedefinieerd: een relatie tussen twee knopen en een geheel getal, die het bestaan van een pad met een bepaalde lengte, het gespecificeerde getal, tussen de twee knopen weergeeft. Deze relatie krijgt de toepasselijke naam **pad**. In figuur 3 bestaat er bijvoorbeeld tussen de knopen a en c een pad ter lengte 7. Dit kan met behulp van het volgende feit worden weergegeven

```
pad(a,c,7).
```

5.3.1. Een algemene regel

In plaats van alle mogelijke paden in een graaf op te sommen (in een cyclische graaf zijn dit er oneindig veel) en als feiten in de PROLOG database op te slaan, kan een algemene regel geformuleerd worden, waarin wordt uitgedrukt wanneer er een pad tussen twee knopen bestaat en welke lengte dat pad heeft:

tussen twee knopen X en Y bestaat een pad ter lengte N als

- (1) X en Y dezelfde knoop aanduiden. De lengte N van het pad is in dit geval gelijk aan 0.
- (2) er een knoop Z bestaat zodanig dat er een tak ter lengte K tussen X en Z bestaat en er een pad ter lengte M tussen Z en Y te vinden is. De lengte N van het gevonden pad is dan gelijk aan $K + M$.

In PROLOG wordt deze definitie in twee clauses vastgelegd met behulp van het eerder geïntroduceerde predicaat **pad**.

```

/*1*/ pad(X,X,0).
/*2*/ pad(X,Y,N) :-
    (tak(X,Z,K); tak(Z,X,K)),
    pad(Z,Y,M),
    N is K+M.

```

Clause 1 representeert het stopcriterium (1) van de recursie; in het feit zijn bij het predicaat **pad** de twee knopen gelijk, de variabele X komt zowel op de eerste als op de tweede parameterpositie voor, en is de padlengte, de derde parameter, gelijk aan 0. Als in de recursie op een gegeven moment een match met deze clause wordt gevonden, stopt het recursieproces, omdat de clause geen body heeft waarin een procedure-aanroep met dezelfde procedure-naam, **pad**, voorkomt.

In clause 2 is de daadwerkelijke recursie geformuleerd. Het zoeken naar een pad tussen de knopen X

en Y wordt door het bepalen van een knoop Z zodanig dat er een tak tussen X en Z bestaat, vereenvoudigd tot het zoeken naar een pad tussen Z en Y .

Het bepalen van zo'n knoop Z vindt plaats in de conditie

$$(\text{tak}(X,Z,K); \text{tak}(Z,X,K)).$$

Omdat de graaf in het probleem ongericht is en elke tak slechts eenmaal is opgeslagen, kan in clause 2 niet met de term $\text{tak}(X,Z,K)$ worden volstaan. Als er een tak tussen de twee knopen X en Z bestaat, kan deze tak immers als $\text{tak}(X,Z,K)$, maar ook als $\text{tak}(Z,X,K)$ gerepresenteerd zijn. In de conditie worden beide notatiemogelijkheden bekeken met behulp van de term $(\text{tak}(X,Z,K); \text{tak}(Z,X,K))$, waarin de $;$ de logische "or" weergeeft. Een andere oplossing voor het hier gesignaleerde probleem is de clause

$$\text{tak}(X,Y,K) :- \text{tak}(Y,X,K).$$

na de clauses die de graaf representeren, te specificeren.

Door de bepaling van de knoop Z is het probleem tot een iets eenvoudiger probleem van dezelfde aard gereduceerd. In de tweede conditie van clause 2 vindt de recursieve aanroep van de procedure **pad** plaats voor dit nieuwe probleem

$$\text{pad}(Z,Y,M).$$

In de derde conditie

$$N \text{ is } K+M.$$

vindt de berekening van de padlengte plaats.

5.3.2. Het uitzonderen van een cyclisch pad

In de voorgaande paragraaf zijn twee clauses opgesteld waarin een definitie van het begrip pad is vastgelegd. Met behulp van deze twee clauses kan in een graaf een pad gevonden worden; als het pad dat de interpretator kiest, een cyclisch pad is, stopt het zoekproces echter niet. Met behulp van clauses moet derhalve vastgelegd worden hoe de interpretator kan ontdekken dat het beschouwde pad cyclisch is en welke actie daarbij ondernomen moet worden.

Om te kunnen detecteren of het pad dat beschouwd wordt cyclisch is, moet bijgehouden worden welke knopen dat pad al heeft aangedaan. Zodra een knoop wordt aangedaan die al op het beschouwde pad ligt, kan het pad verlaten worden: een cyclisch pad kan aan een kortste-pad-bepaling geen zinvolle bijdrage leveren, omdat er in een dergelijk geval altijd een korter pad te vinden is. De reeds bezochte knopen kunnen bijvoorbeeld in een lijst bijgehouden worden; in de twee clauses van de vorige paragraaf wordt het predicaat **pad** uitgebreid met een extra parameter, deze lijst van bezochte knopen. **PROLOG** kent immers geen globale variabelen. Voorts moet telkens na het selecteren van een nieuwe knoop Z voor het pad, gecontroleerd worden of het pad deze knoop al bevat. Als dit het geval is, moet er een nieuwe knoop geselecteerd worden; als het pad de geselecteerde knoop nog niet bevat, moet deze knoop aan de lijst toegevoegd worden en moet de recursieve aanroep met deze nieuwe lijst plaatsvinden. Voor de besproken test wordt gebruik gemaakt van de relatie met de naam **element** uit paragraaf 2.2:

```

/*1*/ pad(X,X,0,Knopenlijst).
/*2*/ pad(X,Y,N,Knopenlijst) :-
    (tak(X,Z,K);tak(Z,X,K)),
    not(element(Z,Knopenlijst)),
    pad(Z,Y,M,[Z|Knopenlijst]),
    N is K+M.

/*3*/ element(X,[X|_]).
/*4*/ element(X,[_|Y]) :-
    element(X,Y).

```

5.3.3. Het verwerpen van een te lang pad

Met de clauses 1 tot en met 4 uit de voorgaande paragraaf kan in principe een programma gemaakt worden dat in een gegeven graaf alle cyclische paden tussen twee gegeven knopen bekijkt, en als ze allemaal bekeken zijn, het kortste pad kiest. Het is echter niet nodig om alle mogelijke paden tussen de twee knopen helemaal te bekijken.

Stel dat op een bepaald moment een pad ter lengte N gevonden is. Een pad waarvan tijdens het zoekproces de lengte deze waarde N overschrijdt, kan verlaten worden: er is immers al een korter pad gevonden. Voor een dergelijk stopcriterium moet telkens als er een tak aan een pad wordt toegevoegd, bekend zijn welke lengte het reeds bekeken deel van dat pad heeft en welke lengte het pad heeft dat op dat moment het kortste pad is.

In clause 2 vindt de berekening van de padlengte plaats als het gehele pad bekeken is: de optelling N is $K+M$ vindt immers ná de recursieve aanroep van de procedure `pad` plaats. Om telkens als er een tak aan een pad wordt toegevoegd de lengte van het reeds bekeken deel van dat pad te kennen, moet deze optelling vóór de recursieve aanroep plaatsvinden.

```

/*1*/ pad(X,X,Teller,Knopenlijst).
/*2*/ pad(X,Y,Teller,Knopenlijst) :-
    (tak(X,Z,K);tak(Z,X,K)),
    not(element(Z,Knopenlijst)),
    Nieuweteller is Teller+K,
    pad(Z,Y,Nieuweteller,[Z|Knopenlijst]).

```

De betekenis van de derde parameter van de procedure `pad` is in de clauses 1 en 2 veranderd: als er voor gezorgd wordt dat de variabele `Teller` bij de eerste aanroep van de procedure `pad` de waarde 0 heeft, bevat `Teller` bij elke aanroep van de procedure `pad` de lengte van het reeds bekeken deel van een pad. Pas op het moment dat clause 1 gebruikt wordt, bevat de variabele `Teller` de lengte van het hele pad.

Het gevolg van de bovenbeschreven constructie is echter dat de padlengte niet meer bekend is na de beeindiging van de executie van de eerste aanroep van de procedure `pad`. In die aanroep was `Teller` immers met de initiële waarde 0 geïnstantieerd; het instantiatiemechanisme heeft tot gevolg dat deze parameter niet meer als uitvoerparameter dienst kan doen. De padlengte moet, op het moment dat deze waarde bekend is, bewaard worden; dit is mogelijk door het predicaat `pad` met een extra parameter uit te breiden, of door een nieuw feit aan de database toe te voegen, waarin het pad

met de gevonden lengte wordt vastgelegd.

Hier wordt voor de tweede mogelijkheid gekozen. Met behulp van het standaardpredicaat `asserta` kunnen clauses aan de database toegevoegd worden. Voor het vastleggen van een pad met de bijbehorende lengte wordt een nieuw predicaat met de naam `kortste_pad`, met twee argumenten geïntroduceerd; het eerste argument bevat een pad gerepresenteerd in een lijst, en het tweede argument bevat de lengte van dat pad.

Als aan het stopcriterium, geformuleerd in clause 1, is voldaan, is de lengte van het beschouwde pad bekend. Voor het bewaren van deze waarde wordt clause 1 als volgt aangepast:

```
/*1*/ pad(X,X,Teller,Knopenlijst) :-
    asserta(kortste_pad(Knopenlijst,Teller)).
```

Als er voor gezorgd wordt dat het feit dat op deze manier aan de database wordt toegevoegd altijd het kortste pad bevat, kan op elk moment de lengte van het kortste pad in de database opgezocht worden.

De voorgestelde vergelijking van de lengte van het kortste pad met de lengte van het reeds bekeken deel van het pad onder beschouwing dient nu in clause 2 verwerkt te worden. De lengte van het pad dat op dat moment het kortste pad is, kan in de database opgezocht worden met behulp van `kortste_pad(_,Minimum)`; welk pad dat is, is niet van belang, zodat voor de eerste parameter een don't-care variabele is gebruikt. Bij de uitvoering van deze aanroep wordt de variabele `Minimum` met de gezochte padlengte geïnstantieerd. Deze waarde moet nu vergeleken worden met de som van de lengte van het reeds bekeken deel van het pad en de lengte van de tak die aan het pad wordt toegevoegd. Hiertoe wordt gebruik gemaakt van het standaardpredicaat `>=`. Clause 2 ziet er nu als volgt uit:

```
/*2*/ pad(X,Y,Teller,Knopenlijst) :-
    (tak(X,Z,K);tak(Z,X,K)),
    not(element(Z,Knopenlijst)),
    kortste_pad(_,Minimum),
    Minimum >= Teller + K,
    Nieuweteller is Teller + K,
    pad(Z,Y,Nieuweteller,[Z|Knopenlijst]).
```

Merk op dat de recursieve aanroep van de procedure `pad` alleen wordt uitgevoerd als het beschouwde pad niet langer is dan het op dat moment kortste pad. Als het bekeken pad te lang is, is niet aan de conditie `Minimum >= Teller + K` voldaan, zodat met backtracking een ander pad wordt geprobeerd.

De uitbreiding van clause 2 met de bovenbeschreven vergelijking heeft tot gevolg dat clause 1 alleen wordt gebruikt in het geval dat er een pad gevonden is, dat korter is dan het pad dat tot dat moment het kortste pad was. Dit achterhaalde feit moet in de database vervangen worden door het nieuwe kortste pad. Met behulp van het standaardpredicaat `retract` wordt het achterhaalde feit uit de database verwijderd:

```

/*1*/ pad(X,X,Teller,Knopenlijst) :-
    !, retract(kortste_pad(_, _)),
    asserta(kortste_pad(Knopenlijst, Teller)).

```

Merk op dat in clause 1 gebruik is gemaakt van het standaardpredicaat `!`.

Bij de aanvang van de uitvoering van de procedure `pad` dient er een feit in de database aanwezig te zijn waarin de lengte van het "kortste pad" met een maximum wordt geïnitieëerd. Door het programma kan afhankelijk van de beschouwde graaf een dergelijk maximum gegenereerd worden door de som van alle taklengten te nemen. Voor de eenvoud wordt in dit voorbeeld de initiële waarde 50 gekozen en als feit aan de database toegevoegd:

```

/*5*/ kortste_pad(_, 50).

```

5.3.4. Het bekijken van alle alternatieven

De kern van het programma is met de specificatie van de clauses 1 tot en met 5 af. Het programma kan bijvoorbeeld gestart worden met de vraag:

```

?- pad(a,f,0,[a]).

```

In de derde parameter wordt de waarde 0 als beginwaarde voor de telling van de padlengte meegegeven en in de vierde parameter wordt de lijst van gezochte knopen met de beginknoop `a` van het gezochte pad tussen `a` en `f` geïnitieëerd. Met behulp van een dergelijke vraag wordt slechts één pad tussen de opgegeven knopen bekeken. Voor een kortste-pad-bepaling moeten echter alle alternatieven bekeken worden.

In onderstaande clause

```

/*6*/ kortste_pad_bepaling(X,Y) :-
    pad(X,Y,0,[X]),
    fail.

```

worden, als `X` en `Y` geïnstantieerd zijn met knopen, alle alternatieve paden tussen `X` en `Y` bekeken. Na de uitvoering van `kortste_pad_bepaling(X,Y)` antwoordt de interpretator "no": omdat, als alle mogelijke paden tussen `X` en `Y` bekeken zijn, bij backtracking als gevolg van `fail`, ook niet meer aan de conditie `pad(X,Y,0,[X])` kan worden voldaan. Op dat moment is met het predicaat `kortste_pad` het gevonden kortste pad als feit in de database opgeslagen.

Met clause

```

/*7*/ kortste_pad_bepaling(X,Y) :-
    kortste_pad(P,L),
    uitvoer(X,Y,P,L).

```

wordt dit feit in de database opgezocht en wordt het resultaat met behulp van de aanroep van de procedure `uitvoer` aan de gebruiker kenbaar gemaakt. Het opstellen van de hiervoor benodigde

clauses wordt aan de creativiteit van de lezer overgelaten. Merk op dat clause 7 pas na de beschouwing van alle alternatieve paden met behulp van clause 6, met backtracking wordt bereikt.

5.4. Het programma

Het beschreven probleem is in het onderstaande programma gespecificeerd:

```

/*1*/ pad(X,X,Teller,Knopenlijst) :-
    !, retract(kortste_pad(_, _)),
    asserta(kortste_pad(Knopenlijst, Teller)).
/*2*/ pad(X,Y,Teller,Knopenlijst) :-
    (tak(X,Z,K); tak(Z,X,K)),
    not(element(Z,Knopenlijst)),
    kortste_pad(_, Minimum),
    Minimum >= Teller + K,
    Nieuweteller is Teller + K,
    pad(Z,Y,Nieuweteller,[Z|Knopenlijst]).

/*3*/ element(X,[X|_]).
/*4*/ element(X,[_|Y]) :-
    element(X,Y).

/*5*/ kortste_pad(_,50).

/*6*/ kortste_pad_bepaling(X,Y) :-
    pad(X,Y,0,[X]),
    fail.
/*7*/ kortste_pad_bepaling(X,Y) :-
    kortste_pad(P,L),
    uitvoer(X,Y,P,L).

```

Het kortste pad tussen de knopen a en f in de graaf van figuur 3, wordt nu verkregen door het stellen van de vraag

```
?- kortste_pad_bepaling(a,f).
```

6. BESCHOUWING

In dit artikel is een inleiding gegeven in de programmeertaal PROLOG. Ook is een overzicht gegeven van de terreinen waarop PROLOG met succes wordt toegepast. Het is hier echter op zijn plaats op te merken dat de programmeertaal PROLOG niet de oplossing is voor alle problemen in de informatica; hoewel het een onmiskenbaar voordeel is dat een PROLOG programmeur zijn aandacht primair op de beschrijving van een probleem kan richten, laat niet elk probleem zich eenvoudig in PROLOG formuleren. De problemen die zich efficiënt met behulp van PROLOG laten oplossen, hebben op de een of andere wijze de oplossingsmethodiek van de PROLOG interpretator nodig. Voorts blijkt PROLOG vooral op terreinen waar symboolverwerking van belang is, een geschikte taal.

7. LITERATUUR

- [CAM84] Campbell, J.A.
"Implementations of PROLOG"
Ellis Horwood, Chichester, 1984
- [CLO81] Clocksin, W.F. & C.S. Mellish
"Programming in PROLOG"
Springer-Verlag, Berlin Heidelberg, 1981
- [COE80] Coelho, H., J.C. Cotta & L.M. Pereira
"How to solve it with PROLOG"
Laboratorio Nacional de Engenharia Civil, Lisboa, 1980
- [GAL78] Gallaire, H. & J. Minker
"Logic and databases"
Plenum Press, New York, 1978
- [GEN85] Genesereth, M.R. & M.L. Ginsberg
"Logic programming"
Communications of the ACM, vol. 28, no. 9, pp. 933 - 941, 1985
- [KLU85] Kluzniak, F. & S. Szpakowicz
"PROLOG for programmers"
Academic Press, London, 1985
- [KOW79] Kowalski, R.A.
"Logic for problem solving"
North-Holland, New York, 1979
- [LLO83] Lloyd, J.W.
"An introduction to deductive database systems"
The Australian Computer Journal, vol. 15, no. 2, pp. 52 - 57, 1983
- [LUC86] Lucas, P.J.F. & L.C. van der Gaag
"PROLOG en expertsystemen"
Informatie, vol. 28, no. 1, pp. 16 - 25, 1986
- [ROB65] Robinson, J.A.
"A machine-oriented logic based on the resolution principle"
Journal of the ACM, vol. 12, no. 1, pp. 23 - 41, 1965
- [SUB85] Subrahmanyam, P.A.
"The software engineering of expert systems: is PROLOG appropriate ?"
IEEE Transactions on Software Engineering, vol. SE-11, no. 11, pp. 1391 - 1400, 1985
- [WAR80] Warren, D.H.D.
"Logic programming and compiler writing"
Software - Practice and experience, vol. 10, pp. 97 - 125, 1980

