Prolog, een andere manier van programmeren

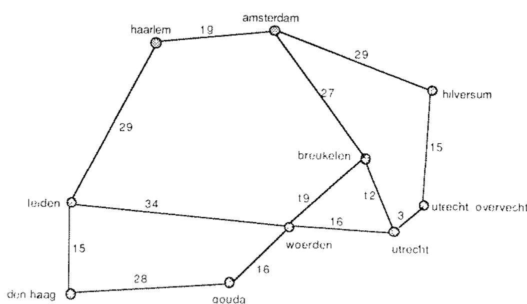
P.M.A. Bergervoet
OW & OC, RU Utrecht

Samenvatting

Prolog is een programmeertaal die sterk afwijkt van de veelal bekende programmeertalen. Het is één van de talen uit een nieuwe generatie programmeertalen, waarin niet meer expliciete instructies aan het apparaat gegeven worden. In Prolog omschrijft men een probleem in als-dan-regels en Prolog probeert met behulp van deze regels vragen over het probleem op te lossen. Deze methode leidt tot een volledig nieuwe stijl van programmeren.
Dit artikel bespreekt, aan de hand van een voorbeeld over grafen, de werking van Prolog. Het is de bedoeling dat in volgende artikelen verschillende toepassingen van Prolog bekeken zullen worden.

Wanneer men aan 'programmeren' denkt, is dat vooral in termen van reeksen zeer precies geformuleerde instructies in een of andere programmeertaal, zoals BASIC of Pascal. Die instructies vertellen het apparaat precies wat er moet gebeuren.
Inmiddels is echter ook een generatie algemeen toepasbare programmeertalen ontwikkeld, waarin de precieze instructies achterwege gelaten kunnen worden. Deze talen komen voort uit het onderzoek naar kunstmatige intelligentie en een van deze talen is *Prolog,* wat staat voor *programming in logic.*

Programmeren in Prolog verschilt sterk van programmeren in de bekende talen als BASIC en Pascal. Problemen worden op een geheel andere wijze benaderd. Daarom is dit artikel bedoeld om een indruk te geven van de manier waarop in Prolog geprogrammeerd wordt.



Een stukje spoorwegnet

fig. 1

Hoe ziet een Prolog-programma eruit?

Als voorbeeld zal ik Prolog-programma's voor het zoeken van wegen door een graaf geven. Daarbij wordt de graaf in figuur 1 gebruikt, die een deel van het Nederlandse spoorwegnet weergeeft.

Deze graaf kan op een eenvoudige manier in een Prolog-programma opgenomen worden, door alle directe verbindingen, de trajecten, uit te schrijven. Dit is gedaan in het programma in figuur 2. Dit programma dient als uitgangspunt voor het bespreken van Prolog.

```
% De trajecten in de graaf van het spoorwegnet.

traject(utrecht, utrecht_overvecht, 3).
traject(utrecht, woerden, 16).
traject(utrecht, breukelen, 12).
traject(utrecht_overvecht, hilversum, 15).
traject(woerden, leiden, 34).
traject(woerden, gouda, 16).
traject(breukelen, amsterdam, 27).
traject(gouda, den_haag, 28).
traject(den_haag, leiden, 15).
traject(breukelen, woerden, 19).
traject(hilversum, amsterdam, 29).
traject(amsterdam, haarlem, 19).
traject(haarlem, leiden, 29).
```

% een aantal regels voor het vinden van (korte) wegen door de graaf.

```
buur(X, Y, N):-
        traject(X, Y, N).

buur(X, Y, N):-
        traject(Y, X, N).

tweestap(X, Y, N):-
        buur(X, Z, N1),
        buur(Y, Z, N2),
        not X = Y,
        N is N1 + N2.
```

Een Prolog-programma voor grafen

fig. 2

Een Prolog-programma bestaat uit *feiten* en *regels*. Het programma in figuur 2 begint met een aantal feiten, de *trajecten*, die de graaf beschrijven. Deze feiten zijn van de vorm *traject(_,_,_)*, dit wordt een *predicaat* genoemd. In dit geval hebben we het predicaat *traject*, dat drie *argumenten* heeft. Voor Prolog betekent de lijst in figuur 2 niets anders dan dat er een aantal ware feiten met betrekking tot het predicaat *traject* gegeven wordt.
De aanduiding *predicaat* is afkomstig uit de logica, waarover later meer.

Prolog hecht geen enkele betekenis aan de predicaten, interpretatie is een zaak voor de schrijver of de gebruiker van het programma. In dit geval kan het predicaat *traject(X, Y, Z)* gelezen worden als *'er is een traject van plaats X naar plaats Y met een lengte van Z kilometer.'* Deze betekenis bestaat echter alleen in de ogen van de gebruiker; Prolog zou het ook goed vinden als er in de lijsten een feit *traject(wilhelmina, juliana, beatrix)* stond.

Het is van belang iets op te merken over de notatie. Er bestaan verschillende versies van Prolog, die ieder een eigen notatie hanteren. In dit artikel gebruik ik de notatie van Edinburgh-Prolog, een van de meest gebruikte versies van Prolog. Edinburgh-Prolog gebruikt de haakjes-notatie voor predicaten, feiten worden afgesloten met een punt. Verder zijn de plaatsnamen met kleine letters geschreven, omdat woorden die beginnen met hoofdletters door Prolog opgevat worden als variabelen. Het procent-teken, tenslotte, geeft commentaar aan.

Na de feiten met betrekking tot het predicaat *traject* volgen een aantal *regels*. De eerste twee regels leggen vast wanneer twee plaatsen *X* en *Y* buren zijn. Er zijn twee mogelijkheden: ofwel er is een traject van *X* naar *Y*, ofwel andersom. In beide gevallen kan de afstand worden overgenomen.
Dit is vastgelegd in twee regels. In de Prolog-notatie staat :- voor *als*, de hoofdletters geven variabelen aan. De eerste regel kan dus gelezen worden als 'X *en* Y *zijn buren met afstand* N *als er een traject van* X *naar* Y *is met afstand* N.'
De reikwijdte van de variabelen is zeer beperkt. Deze zijn alleen geldig binnen één Prolog-regel. Dit heeft tot gevolg dat we in de tweede regel voor *buur* gewoon weer *X*, *Y* en *N* als variabelen kunnen gebruiken; Prolog zal deze niet verwarren met de *X*, *Y* en *N* uit de eerste regel.

Als laatste is er in het programma een regel opgenomen voor het predicaat *tweestap(_,_,_)*, dat aangeeft wanneer twee plaatsen precies twee stappen (trajecten) van elkaar verwijderd liggen. In deze regel zijn er vier voorwaarden. Aan elk van deze voorwaarden moet voldaan worden; de komma tussen de voorwaarden staat dus voor *en*.
De regel kan als volgt gelezen worden: 'Twee plaatsen *X* en *Y* liggen twee stappen (trajecten) van elkaar verwijderd met afstand *N*, *als* er een plaats *Z* is zodat *X* een buur is van *Z* (afstand *N1*) *en* *Y* een buur is van *Z* (afstand *N2*) *en* *N* gelijk is aan *N1* + *N2*.' Daaraan is nog de voorwaarde toegevoegd dat *X* ongelijk is aan *Y*. Dit sluit het heen en weer reizen naar een buurstation uit.
De *is* in de laatste voorwaarde wordt niet op de standaard-manier opgeschreven, om aan te sluiten bij datgene dat men gewend is. Er had echter ook, meer consequent, *is(N, N1 + N2)* kunnen staan.

In de terminologie van Prolog wordt *tweestap(X, Y, N)* – het predicaat dat aan de :- vooraf gaat – aangeduid als de *kop* (head) van de regel, de vier voorwaarden vormen de *romp* (body) van de regel. Soms wordt ook wel van *doel* en *subdoelen* gesproken. Er is een belangrijk verschil tussen variabelen die in de kop van de regel voorkomen en variabelen die daarin niet voorkomen. De laatste spelen een andere rol, wat duidelijk wordt als we de regel iets formeler opschrijven:

∀ X, Y, N *geldt dat* tweestap(X, Y, N) *waar is* als

∃ Z, N1, N2 *zodat* buur(X, Z, N1) *waar is* en ...

Variabelen in de kop van de regel zijn voorzien van de kwantor *voor alle*, andere variabelen hebben de kwantor *er is*.

Hoe werk je met een prolog-programma?

We hebben in Prolog nu een aantal feiten en regels opgeschreven, maar waar staat nu wat het programma moet *doen*? Welnu, dit hoeft nergens expliciet aangegeven te worden. Hierin verschillen Prolog-programma's van programma's in bekende programmeertalen als BASIC en Pascal. Een Prolog-programma bevat geen rechtstreekse instructies aan een apparaat, het bestaat *alleen* uit een verzameling *feiten* en *regels*. Het is ook niet mogelijk een Prolog-programma uit te voeren, te *runnen*. In plaats daarvan kan de gebruiker het programma *bevragen*, Prolog probeert deze vragen te beantwoorden met behulp van de in het programma opgenomen regels en feiten.

Zo kunnen we Prolog vragen stellen over het predicaat *traject* door deze in te tikken, ook hier afgesloten met

een punt:

```
?-traject(woerden, leiden, 34).
yes

?-traject(leiden, woerden, 34).
no

?-traject(assen, groningen, 20)
no
```

De eerste query *slaagt*, omdat de vraag tussen de feiten gevonden wordt, de tweede en derde *falen*. Overigens is Prolog nogal uitgesproken in de antwoorden, omdat het uitgaat van de *closed world assumption*. Wanneer iets niet uit de verzameling feiten en regels afgeleid kan worden, is het volgens Prolog 'niet waar'; er komt geen antwoord als 'weet niet'. De laatste query levert dus 'no', terwijl het volgens het spoorboekje wel waar is!

De queries worden interessanter als we ook variabelen opnemen.

```
?-traject(woerden, leiden, N).
N = 34 ;

no (more solutions)

?-traject(woerden, X, N).
X = leiden
N = 34 ;

X = gouda
N = 16 ;

no (more solutions)
```

De betekenis van een vraag als *traject(woerden, leiden, N)* is 'Vind een waarde voor *N* zodanig dat *traject(woerden, leiden, N)* waar wordt.' Prolog vindt de waarden door de vraag 'tegen de feiten te leggen' en te onderzoeken of er overeenstemming bestaat tussen de argumenten in de vraag en in het feit. Deze procedure wordt *unificatie* genoemd. De vraag *traject(woerden, leiden, N)* stemt niet overeen met het eerste feit, omdat de eerste argumenten al verschillen, maar wel met het vijfde feit. Om de unificatie compleet te maken moet *N* de waarde 34 krijgen. Deze verschijnt in het antwoord.
Bij de tweede vraag zijn verschillende antwoorden mogelijk. In principe geeft Prolog er maar één, de eerste die gevonden wordt. Door een ; in te tikken kunnen we om meer oplossingen vragen. Er is geen beperking op het aantal variabelen in de vraag; wanneer we er drie opgeven, zal Prolog de complete lijst van trajecten produceren.

Met vragen over het predicaat *buur* krijgen we een vollediger beeld van de graaf:

```
?-buur(leiden, woerden, N).
N = 34 ;

no (more solutions)

?-buur(woerden, P, A).
P = leiden
A = 34 ;

P = gouda
A = 16 ;

P = breukelen
A = 19 ;

no (more solutions)
```

Prolog probeert nu de beide regels voor *buur* om tot antwoorden te komen. Bij de tweede query wordt eerst de eerste regel voor *buur* geprobeerd. Unificatie levert *X=woerden*, *Y=P* en *N=A*. De vraag wordt dan herleid tot *traject(woerden, P, A.)*. Dit geeft twee oplossingen. Daarna wordt de tweede regel geprobeerd, wat als nieuwe vraag *traject(P, woerden, A)* geeft. Dit leidt tot een derde oplossing.

Er is een essentieel verschil tussen de regels van Prolog en de procedures van een taal als Pascal. In zulke procedures ligt vast wat de invoer en wat de uitvoer is: Gegeven twee plaatsen, bepaal of het buren zijn en zo ja, bepaal de afstand. Dergelijke procedures kunnen maar voor één soort vragen gebruikt worden. In Prolog is dit niet het geval, de regels geven alleen relaties weer. We kunnen zelf bepalen wat we 'erin stoppen' en wat 'eruit moet komen'. Daardoor zijn ook vragen van een heel ander soort mogelijk:

```
?-buur(X, Y, 19).
X = breukelen
Y = woerden;

X = amsterdam
Y = haarlem;

X = woerden
Y = breukelen;

X = haarlem
Y = amsterdam;

no (more solutions)
```

In de vragen wordt bepaald wat invoer en wat uitvoer is. Voor het programma maakt dit in principe geen verschil, het is voor verschillende doeleinden bruikbaar.

Hoe vindt Prolog oplossingen?

Eigenlijk zou het zo moeten zijn, dat het er niet toe doet *hoe* Prolog oplossingen vindt. In het programma geven we de regels waar oplossingen aan moeten voldoen en Prolog dient zich daar een weg door te zoeken.
In de praktijk heeft Prolog echter een vaste methode van zoeken, die soms tot problemen aanleiding kan geven. Bij de methode gaat het eigenlijk om twee vragen:

1. In welke volgorde probeert Prolog de regels die op een (sub)doel van toepassing zijn?

2. In welke volgorde worden de voorwaarden van een regel nagegaan?

Prolog pakt het simpel aan: De regels worden geprobeerd in de volgorde waarin ze zijn ingevoerd. Als de toepassing van een regel niet tot een oplossing leidt, gaat Prolog terug en probeert de volgende regel. Deze procedure heet *backtracking*. In feite werkt het hetzelfde als de bekende procedures voor het wandelen door een doolhof: Probeer bij elk kruispunt eerst de meest linkse gang en als deze (uiteindelijk) doodloopt, ga dan terug en probeer de volgende.

De voorwaarden worden op dezelfde manier aangepakt, ze worden van boven naar beneden afgewerkt.

Backtracking treedt verscheidene keren op bij de query *tweestap(utrecht, amsterdam, N)*. Drie momentopnamen uit de afleiding staan in figuur 3, waarbij de nog te bewijzen voorwaarden steeds onder de streep staan.

Prolog begint met de eerste voorwaarde van *tweestap*, dat is *buur(utrecht, Z, N1)*. Deze slaagt, waarbij Z de waarde *utrecht_overvecht* krijgt en *N1* de waarde 3.

De tweede voorwaarde (momentopname 2) wordt nu *buur(amsterdam, utrecht_overvecht, N2)*. Beide regels voor *buur* worden geprobeerd, beide falen.

Prolog moet nu een stap verder terug (momentopname 3) en alternatieven voor *traject(utrecht, Z, N1)* zoeken. Dit slaagt met *Z=breukelen* en *N1=12*. Daarna slagen ook de andere voorwaarden, alleen in de tweede is nog één backtracking nodig.

Door deze zoekprocedure is de volgorde van regels voor één predicaat en de volgorde van voorwaarden in een regel soms van belang. Als we bijvoorbeeld de voorwaarde *N is N1+N2* vooraan in *tweestap* gezet hadden, zou Prolog foutmeldingen gaan geven:

(momentopname 1)

```
tweestap(utrecht, amsterdam, N)
① buur(utrecht, Z, N1)
    ① traject(utrecht, Z, N1)
        ① slaagt.          Z = utrecht_overvecht
                           N1 = 3
--------------------------------------------------
    buur(amsterdam, Z, N2)
    utrecht <> amsterdam
    N is N1 + N2
```

(momentopname 2)

```
tweestap(utrecht, amsterdam, N)
① buur(utrecht, Z, N1)
    ① traject(utrecht, Z, N1)
        ① slaagt.          Z = utrecht_overvecht
                           N1 = 3
  buur(amsterdam, utrecht_overvecht, N2)
    ① traject(amsterdam, utrecht_overvecht, N2)
      faalt.
    ② traject(utrecht_overvecht, amsterdam, N2)
      faalt.
  buur faalt.
--------------------------------------------------
    utrecht <> amsterdam
    N is 3 + N2
```

(momentopname 3)

```
tweestap(utrecht, amsterdam, N)
① buur(utrecht, Z, N1)
    ① traject(utrecht, Z, N1)
        ③ slaagt,          Z = breukelen
                           N1 = 12
buur(amsterdam, breukelen, N2)
① traject(amsterdam, breukelen, N2)
  faalt.
② traject(breukelen, amsterdam, N2)
    ⑦ slaagt,              N2 = 27
--------------------------------------------------
    utrecht <> amsterdam
    N is 12 + N2
```

fig. 3

N1 + N2 moet worden uitgerekend op een moment dat de waarden van beide variabelen nog onbekend zijn.

Recursie

Feiten, regels en queries zijn de enige taalconstructies die Prolog kent. Er bestaan geen if-then-else constructies of for-loops of while-loops. Een cyclus moet daarom op een andere manier weergegeven worden, namelijk door recursie.

In het voorbeeld van het spoorwegnet waren tot nog toe predicaten gedefiniëerd voor *buur* en *tweestap*. We kunnen ook regels geven voor een predicaat *weg* dat waar is als er een weg is tussen twee plaatsen, ongeacht de afstand.

```
weg(X, X, 0).

weg(X, Y, A):-
        buur(X, Z, A1),
        weg(Z, Y, A2),
        A is A1 + A2.
```

Wegen zoeken

fig. 4

De tweede regel bevat de recursie. De declaratieve lezing van deze regel is feitelijk '*Er is een weg van X naar Y als er een weg van een buur van X naar Y is.*' Veel verschil met de definitie van n! is er niet, in beide gevallen is er een recursieve regel en een 'stopregel':

$n! = 1$	als $n = 0$
$n! = n*(n-1)!$	als $n > 0$

Het gegeven programma werkt soms wel en soms niet. Hieronder staan twee vragen en de 'antwoorden' van Prolog:

?-*weg(utrecht, leiden, A).*	
A=95	*(via overvecht, hilversum, amsterdam en haarlem!)*

?-*weg(leiden, utrecht, A).*	
Error...	(te diepe recursie)

Recursie heeft een gevaar. Het is mogelijk dat voortdurend de tweede regel voor *weg* gebruikt wordt, waardoor het zoeken in principe oneindig lang doorgaat. Bij de tweede query gebeurde dit: Omdat Prolog steeds de eerste de beste buur neemt die beschikbaar is, raakte het zoekproces in een lus van leiden naar woerden en weer terug. Deze lus kon ontstaan omdat het programma niet onthoudt welke plaatsen al bezocht zijn. Deze plaatsen kunnen in een lijst worden onthouden.

Lijsten

Lijsten zijn niets anders dan een 'rijtje objecten'. Die objecten mogen in Prolog van alles zijn: constanten, variabelen, termen of weer lijsten. Voor lijsten zijn verschillende notaties in omloop. De meest populaire notatie werkt met blokhaken en komma's [1, 1, 2, 3, 5] en [hilversum, amsterdam, haarlem] zijn lijsten, maar [utrecht, 7, traject(haarlem, leiden, 29), [1, 2]] is ook een lijst.

We kunnen lijsten meer precies omschrijven door de volgende eigenschappen: Er is een *lege lijst*, aangeduid door [ ], en elke niet-lege lijst heeft een *kop*, het voorste element en een *staart*, dat is de lijst die overblijft als je de kop eraf haalt. Dit wordt meestal genoteerd als [Kop | Staart]. Een voorbeeld:

```
?-[1, 1, 2, 3, 5] = [Kop | Staart]
Kop = 1
Staart = [1, 2, 3, 5]
```

In het voorbeeld werd de constructie [Kop | Staart] gebruikt om het voorste element van een lijst 'af te knippen'. In Prolog kan dezelfde constructie gebruikt worden om een element aan een lijst te 'plakken': als Kop en Staart bekend zijn, levert [Kop | Staart] een nieuwe lijst die één element langer is.
Lijsten lenen zich uitstekend voor recursieve programma's, omdat de beschrijving van een lijst, met kop en staart, ook recursief is. Hieronder staan een paar recursieve programma's voor lijsten, steeds met een 'stopregel' en recursieve regel(s).

```
% zit_in(X, L) is waar als X een element van de lijst is.

zit_in(X, [X | Staart]).

zit_in(X, [Kop | Staart]):
        zit_in(X, Staart).

% aantal_e(L, N) is waar als N het aantal elementen van L is.

aantal_e([ ], 0).

aantal_e([Kop | Staart], A):-
        aantal_e(Staart, As),
        A is As+1.

% kleinste(L, X) is waar als X de kleinste uit de lijst (getallen) L is.

kleinste([X], X).

kleinste([E | F | Staart], X):-
        E >= F,
        kleinste([F | Staart], X).

kleinste([E | F | Staart], X):-
        E < F,
        kleinste([E | Staart], X).
```

De regels voor *zit_in* zeggen niets anders dan '*X zit in een lijst L* als *X* gelijk is aan de kop van *L* of als *X* in de staart van *L* zit.' Bij de vraag *zit_in(3, [1, 1, 2, 3, 5]* zal de tweede regel steeds elementen ongelijk aan 3 afpellen, totdat *zit_in(3, [3, 5])* resteert, wat slaagt door de eerste regel. In het laatste voorbeeld wordt de kop-staart-notatie dubbel op gebruikt. Het programma werkt volgens het principe dat de grootste van twee elementen voorop bij het zoeken naar de kleinste verder buiten beschouwing gelaten kan worden.

Wegen zoeken

Terug naar het programma dat een weg door de graaf zoekt. Het predicaat *weg* wordt zodanig herschreven dat het onthoudt welke plaatsen tijdens het zoeken al bezocht zijn. Bij elke stap wordt gecontroleerd of we niet in een al bezochte plaats uitkomen, waardoor lussen voorkomen worden. De bezochte plaatsen worden in een lijst onthouden, die we als extra argument aan *weg* meegeven. Deze lijst groeit bij elke stap door de graaf aan, doordat de nieuwe plaats aan de lijst van de bezochte plaatsen geplakt wordt. Verder wordt naast *weg(_,_,_,_)* een predicaat *weg(_,_,_)* gegeven, dat er voor zorgt dat het zoeken op de juiste wijze begint, met alleen het startpunt in de lijst van bezochte plaatsen.

```
weg(X, X, 0, Bezocht).

weg(X, Y, A, Bezocht):-
        buur(X, Z, A1),
        not zit_in(Z, Bezocht),
        weg(Z, Y, A2, [Z | Bezocht]),
        A is A1+A2.

weg(X, Y, A):-
        weg(X, Y, A, [X].
```

Backtracking zorgt ervoor dat eventuele wegen inderdaad gevonden worden. Wanneer Prolog naar een plaats *Z* reist die al bezocht is, zal *not zit_in(Z, Bezocht)* falen. Prolog keert dan op zijn schreden terug en probeert een andere buur te vinden. Wanneer nodig zal Prolog verschillende stappen terug doen om de weg te vinden. In figuur %F is de route aangegeven die Prolog volgt bij het zoeken van een weg van ... naar...

Lijsten kunnen ook gebruikt worden om de route van X naar Y vast te leggen. Ook dit kan gedaan worden door het toevoegen van een extra variabele. Als we (in de notatie van het bovenstaande programma) de route van *Z* naar *Y* weten, weten we ook de route van *X* naar *Y*; die krijgen we door er de plaats *X* aan toe te

voegen. Hieronder staat een programma dat de route bepaalt, de wijzigingen zijn daarin vet afgedrukt.

```
weg_r(X, X, 0, Bezocht, [X]).

weg_r(X, Y, A, Bezocht, [X | Route]):-
        buur(X, Z, A1),
        not zit_in(Z, Bezocht),
        weg_r(Z, Y, A2, [Z | Bezocht], Route),
        A is A1+A2.

weg_r(X, Y, A, R):-
        weg_r(X, Y, A, [X], R).

?-weg_r(utrecht, amsterdam, A, R).
A = 47
R = [utrecht, utrecht_overvecht, hilversum, amsterdam];

A = 98
R = [utrecht, woerden, leiden, haarlem, amsterdam];

(en nog een aantal toeristische routes)
```

Het vinden van de snelste weg door een graaf is een beroemd probleem. De eenvoudigste manier om dit in Prolog te programmeren is om Prolog een lijst van *alle* mogelijke routes te laten maken en daar de kortste uit te halen. Voor het maken van een lijst van alle mogelijke oplossingen heeft Prolog een speciaal predicaat: *bagof* (lees 'bag of'; een zak vol oplossingen). Omdat we zowel de weg als de afstand willen weten, laten we Prolog een lijst van termen maken: *route(R, A)*, waarin de weg en de afstand zijn opgenomen. Het programma wordt dan:

```
kortste_weg(X, Y, Rt, Af):-
        bagof(route(R, A), weg_r(X, Y, R, A),
          Oplossingen),
        kleinste_r(Oplossingen, route(Rt, Af)).
```

Het predicaat *kleinste_r* is een iets aangepaste versie van het eerder beschreven programma *kleinste* voor lijsten met getallen.
Het programma lijdt aan een zekere 'overkill': eerst alle routes vinden en dan pas de kleinste eruit lichten. Er zijn slimmere varianten mogelijk, maar deze vallen buiten het kader van dit artikel.

Toepassingen van Prolog

In het begin van dit artikel werd al opgemerkt dat de stijl van programmeren in Prolog sterk verschilt van talen als BASIC en Pascal. Bij de laatste twee moet je je vooral bezighouden met de vraag *hoe* een programma moet werken. Prolog is meer beschrijvend: Je geeft regels waaraan oplossingen moeten voldoen en daarna moet Prolog zelf maar uitzoeken hoe deze regels gebruikt moeten worden om daadwerkelijk tot oplossingen te komen. Daardoor is Prolog vooral geschikt voor problemen die zich in dergelijke regels laten opschrijven. Men kan hierbij vooral denken aan zaken die tot zoekproblemen te herleiden zijn, aan (natuurlijke) taal (grammatica-regels!) en aan het weergeven van (verschillende) strategieën om een probleem aan te pakken. Een voorbeeld hiervan is formulemanipulatie in de wiskunde. Het is de bedoeling dat dit artikel een vervolg krijgt dat daaraan gewijd is.

Tot slot een voorbeeld waarin het beschrijvende karakter van Prolog duidelijk tot uiting komt.
Het Nim-spel is een bekend luciferspelletje.
Twee spelers nemen om de beurt een aantal lucifers van een stapel, waarbij het aantal mogelijke 'zetten' beperkt is. Degene die niet meer kan zetten verliest. In de bekendste vorm mogen 1, 2 of 3 lucifers genomen worden, maar allerlei varianten ontstaan als je de zetten verandert.
In het voorbeeld zijn de toegestane zetten 2, 5 en 6, maar deze keuze is willekeurig. Ze kunnen zonder meer vervangen worden door andere. De gewonnen en verloren standen laten zich simpel in Prolog-regels omschrijven:

```
% de zetten:

toegestane_zet(2).
toegestane_zet(5).
toegestane_zet(6).

doe_zet(Stand, Zet, Nwe_stand):-
        toegestane_zet(Zet)'
        Zet = < Stand,
        Nwe_stand is Stand - Zet.

% gewonnen(Stand, Zet) is waar als je in de
% gegeven Stand kunt winnen
% door Zet lucifers te nemen.

gewonnen(Stand, Zet):-
        doe_zet(Stand, Zet, Nwe_stand),
        verloren(Nwe_stand).

verloren(Stand):-
        not gewonnen(Stand).
```

Twaalf tekstregels zijn voldoende om het probleem te programmeren. Het punt is dat Prolog zelf het afzoeken van alle mogelijkheden regelt. In de regel voor *gewonnen* kiest Prolog een (mogelijke) zet, via *doe-zet*. Als vervolgens niet bewezen kan worden dat de resulterende Nwe_stand verloren is (voor de tegenstander), zal Prolog 'backtracken' en een andere zet proberen! Pas als alle mogelijke zetten falen, zal *gewonnen* falen.

Het gegeven programma blinkt niet uit in efficiëntie. Van een bepaalde stand zal een groot aantal keren bepaald worden of deze gewonnen dan wel verloren is. Er zijn allerlei mogelijkheden om Prolog de eenmaal bepaalde standen te laten onthouden, maar deze vallen buiten het kader van dit artikel.

Beschikbaarheid van Prolog

Er zijn op het moment al allerlei versies van Prolog te verkrijgen die op gewone PC's werken. Veel versies geven een compleet beeld van de taal, maar zijn vaak

wat onvriendelijk in de werkomgeving voor de gebruiker en/of zeer traag. De meest populaire versie is ongetwijfeld Turbo Prolog. Deze is vlot en heeft een uitstekende werkomgeving, maar mist helaas een aantal essentiële elementen van Prolog. Zo is het onmogelijk om met algemene termen te werken, zoals de *route(R, A)* in het kortste-weg probleem. Daardoor vervallen juist de meest interessante mogelijkheden van Prolog op het gebied van taal en strategische regels. Het wachten is op een verbeterde versie van Turbo Prolog.

Ook verschijnen er veel boeken over Prolog en met name over Turbo Prolog. Boeken uit deze laatste groep zijn niet aan te raden; door de beperkingen in Turbo Prolog geven deze ook een te beperkt beeld van de taal. Een zeer goed boek is het boek van Sterling en Shapiro. Dit geeft een massa voorbeelden en een gedegen behandeling van de taal, misschien iets tè gedegen voor de lezer die zich alleen wil oriënteren. De andere boeken zijn dan goede alternatieven.

Literatuur

[1] Sterling, J. en E. Shapiro: *The Art of Prolog*, MIT-Press, 1986.

[2] Bratko, I, *Prolog Programming for Artificial Intelligence*, Addison Wesley, 1986.