Practicumhandleiding

Intelligente Systemen

2008–2009

Peter Lucas

iCIS, Radboud Universiteit Nijmegen

30 september 2008

Doelstelling van het practicum

Het practicum behorend bij het college *Intelligente Systemen* bestaat uit het maken van opgaven, waarvan de achterliggende theorie aan de orde komt in het college en beschreven is in de syllabus *Principles of Intelligent Systems*. De nadruk in dit practicum ligt op kennisrepresentatie en automatisch redeneren.

Opgave 1: Automatisch redeneren in de AI. Dit deel van practicum biedt u een overzicht van de belangrijkste redeneermethoden en redeneerstrategieën in op resolutie-gebaseerde theorem provers aan de hand van het OTTER-systeem. De achterliggende theorie wordt beschreven in Hoofdstuk 2 van de syllabus. De opgedane kennis wordt getoetst aan de hand van de OTTER-oefeningen; de oplossing van de **O.1**-opgave moet u ter beoordeling voorleggen aan de practicumassistentie. Van u wordt tenslotte gevraagd een wat omvangrijker probleem (**O.2**) met behulp van OTTER op te lossen. De uitwerking van deze opgave moet worden ingeleverd en wordt beoordeeld.

Opgave 2: Een AI-programmeertaal. In dit deel van het practicum leert u omgaan met de logische programmeertaal PROLOG. Net als bij OTTER is resolutie de basis van dit systeem. De opgedane kennis wordt getoetst aan de hand van de PROLOG-oefeningen; de oplossing van de **P.1**-opgave moet u ter beoordeling voorleggen aan de practicumassistentie. Als opdracht die uiteindelijk uw cijfer medebepaald moet u een PROLOG-programma ontwikkelen voor één van de **P.2**-opgaven.*Als alternatief voor deze programmeeropgave mag u ook een kennissysteem ontwikkelen met gebruikmaking van een eenvoudige, in* PROLOG *ontwikkelde, expert-system builder tool.* Zie hiervoor Opgave 3.

Opgave 3: De ontwikkeling van een kennissysteem voor een zelf uit te kiezen probleemgebied. Deze opgave is beschreven in **Hoofdstuk 3** van de practicumhandleiding en *fungeert als alternatief voor opgave P.2*. De opgave behelst: (1) de uitwerking van een probleemanalyse, (2) het opstellen van een ontwerp voor een kennissysteem, en (3) de implementatie van het systeem met behulp van een expert-system builder tool. *De onderdelen* **K.1** *tot en met* **K.3** *van deze opgave worden apart beoordeeld.*

De in te leveren oefeningen en opgaven zijn in de tekst door een '★' aangegeven. Alle andere opgaven, aangegeven met '►', zijn slechts bedoeld om u zo snel mogelijk met de desbetreffende systemen vertrouwd te maken. Deze opgaven behoeft u niet ter goedkeuring in te leveren.

Elk van de twee opgaven (O.2 en daarna P.2 of de K-opdracht) wordt beoordeeld.

Voor meer informatie over het practicum wordt men verwezen naar het World Wide Web:

`http://www.cs.ru.nl/~peterl/teaching/IS/`

Peter Lucas
Nijmegen, 30 september 2008

Inhoud

Hoofdstuk 1

Practicum-handleiding OTTER

In Hoofdstuk 2 van de syllabus 'Principles of Intelligent Systems' is rederen in de logica behandeld. Daarbij werd ingegaan op de mogelijkheid om volgens bepaalde inferentieregels uit een verzameling logische formules nieuwe formules af te leiden. Een inferentieregel is een voorschrift voor het manipuleren van logische formules; zo'n voorschrift heeft slechts betrekking op de vorm van de formules. Het herhaald toepassen van inferentieregels leidt tot *redeneergedrag* of *inferentie*. Een voorbeeld van een inferentieregel is resolutie.

Bij het herhaald toepassen van een stelsel inferentieregels is het vaak nodig een strategie te gebruiken die het aantal nieuw gegenereerde formules beperkt. De set-of-support strategie is zo'n inferentie-strategie. Deze wordt vooral toegepast bij refutatie-bewijzen; dit zijn bewijzen, waarbij wordt geprobeerd een stelling te bewijzen door de negatie ervan in de verzameling formules op te nemen, en vervolgens een inconsistentie aan te tonen.

Bij de set-of-support strategie wordt een verzameling formules in tweeën gesplitst: een *axiomaverzameling* en de *set-of-support*. De axiomaverzameling is gewoonlijk initieel logisch vervulbaar. Uit de set-of-support wordt telkens één formule geselecteerd en aan de axiomaverzameling toegevoegd. Uit de aldus ontstane verzameling worden nieuwe formules afgeleid, die aan de set-of-support worden toegevoegd. Hierbij moet elke nieuw gegenereerde formule in ieder geval de pas geselecteerde clause als ouder hebben. Inconsistentie is aangetoond zodra de lege clause[1] wordt afgeleid.

Het doel van het practicum OTTER is een kennismaking met automatisch redeneren, en het verschaffen van inzicht in enkele inferentieregels, met name binaire resolutie, hyperresolutie en demodulatie. In Paragraaf 1.1 worden de principes van OTTER besproken. In Paragraaf 1.2 is een aantal PROLOG-oefeningen opgenomen. Deze oefeningen hebben tot doel u in zeer korte tijd ervaring met de belangrijkste aspecten van OTTER bij te brengen. Op twee na zijn alle oefeningen aangegeven met het symbool '▶'. Eén oefening (**O.1**) dient u ter goedkeuring aan de practi-

[1] De lege clause is de clause, bestaande uit nul literals.

cumassistentie voor te leggen. Deze oefening is aangegeven met het symbool '★'. In Paragraaf 1.5 is een aantal opgaven (**O.2**) opgenomen, waaruit, *in overleg met de practicumassistentie*, één gekozen moet worden. Deze opgave heeft tot doel u ervaring met het formaliseren in logica en het gebruik van theorem provers te laten opdoen. De uitgewerkte opgave dient u ter beoordeling bij de practicumassistentie in te leveren.

Een handleiding van OTTER is gegeven in appendix A. Hier worden ook aanwijzingen voor het 'debuggen' van een invoerfile voor OTTER gegeven.

1.1 Inleiding tot OTTER

OTTER is een theorem prover voor eerste-orde predikatenlogica. In OTTER is een aantal inferentieregels geïmplementeerd, waaronder binaire resolutie (Paragraaf 1.2.1) en hyperresolutie (Paragraaf 1.2.2). Deze inferentieregels kunnen worden toegepast op een verzameling clauses, waarbij gebruik wordt gemaakt van de set-of-support strategie.

Verder biedt OTTER de mogelijkheid van demodulatie (Paragraaf 1.2.6). Hierbij wordt een clause herschreven met behulp van een verzameling gelijkheden (demodulatoren).

OTTER past zodoende de set-of-support strategie toe op een *drietal* verzamelingen van clauses, die in het systeem respectievelijk *usable*, *sos* en *demodulators* worden genoemd. Hierbij is *usable* een (vervulbare) axiomaverzameling, *sos* de set-of-support, en *demodulators* een verzameling gelijkheden.

Het door OTTER gebruikte algoritme ziet er in grote lijnen als volgt uit:

> Zolang de set-of-support niet leeg is en geen refutatie is gevonden, wordt:
>
> 1. een zogenaamde given-clause uit de set-of-support gekozen (in het algemeen zal dit de clause met het kleinste aantal literals zijn);
> 2. de given-clause uit de set-of-support verwijderd en toegevoegd aan de axiomaverzameling;
> 3. een aantal nieuwe clauses afgeleid met behulp van de geselecteerde inferentieregels en vervolgens verwerkt; iedere nieuwe clause is de resolvente van de given-clause en één of meer andere clauses uit de axiomaverzameling.

Stappen, die tijdens het verwerken van een nieuw afgeleide clause C altijd worden ondernomen, zijn:

1. Op C wordt demodulatie, inclusief evaluatie van de ingebouwde evalueerbare predikaten (Paragraaf 1.2.7), toegepast.
2. De 'dubbele' literals worden uit de clause C verwijderd.
3. Als de clause C een tautologie is, wordt C verworpen; einde verwerking.

4. Als de clause C wordt gesubsumeerd[2] door een clause C' uit de axiomaverzameling of de set-of-support, wordt C verworpen (forward subsumptie); einde verwerking.
5. De clause C wordt toegevoegd aan de set-of-support.
6. Als de clause C nul literals bevat, is een refutatie gevonden.
7. Als C één literal bevat, wordt in de axiomaverzameling en de set-of-support een clause gezocht, die met de clause C tot een unit conflict[3] leidt, om aldus een refutatie af te leiden.

Aan deze procedure kan een aantal stappen worden toegevoegd door bepaalde opties te selecteren (zie Paragraaf A.2).

1.2 OTTER-oefeningen

Aan de hand van enkele eenvoudige OTTER-specificaties, die in de directory `/vol/practica/IS/Otter` aanwezig zijn, kunnen de belangrijkste eigenschappen van OTTER worden bestudeerd. De volgende logische specificaties zijn in deze directory aanwezig:

binres.in	(*binaire resolutie in propositielogica*)
hyperres.in	(*hyperresolutie en UR-resolutie in propositielogica*)
backsub.in	(*terugwaardse subsumptie in propositielogica*)
vertaling.in	(*vertaling naar clausevorm*)
answer.in	(*gebruik van de answer literal*)
demod.in	(*gelijkheid en demodulatie*)
dollar.in	(*evalueerbare predikaten en functies*)
raa.in	(*causaal redeneren*)
fulladd.in	(*functioneel model – logisch circuit*)

Het OTTER-systeem kunt u activeren door het intikken van het commando:

```
otter
```

1.2.1 Binaire resolutie

Bij binaire resolutie mag ieder tweetal clauses met complementaire literals worden geresolveerd. Omdat OTTER de set-of-support strategie hanteert, moet steeds één van beide clauses deel uitmaken van de set-of-support, en de andere van de axiomaverzameling.

Als voorbeeld gaan we uit van de volgende verzameling clauses:

[2] Een clause C_1 wordt *gesubsumeerd* door een clause C_2, als deze een disjunctie is van C_2 en een (eventueel lege) clause C_3 (zie ook Paragraaf 1.2.3).

[3] Van een *unit conflict* wordt gesproken, als een clause C_1, bestaande uit één positieve literal, resolveert met een clause C_2, bestaande uit één negatieve literal.

```
% Opties voor OTTER:
set(binary_res).        % pas binaire resolutie toe.
clear(back_sub).        % pas geen backward-subsumptie toe.
clear(unit_deletion).   % pas geen unit clause verwijdering toe.
assign(stats_level,0).  % lever geen statistische uitvoer.

% Lijsten met clauses:
list(usable).           % lees axioma's in clause-vorm.
  -p | q.
  -p | -r | s.
  -q | r.
  -s.                   % probeer 's.' af te leiden.
end_of_list.

list(sos).              % lees sos in clause-vorm.
   p.                   % begin uit 'p.' nieuwe clauses af te leiden.
end_of_list.
```

Figuur 1.1: Invoerfile voor OTTER ter demonstratie van binaire resolutie.

$$S = \{\neg p \vee q, \neg p \vee \neg r \vee s, \neg q \vee r\}$$

We willen nu bewijzen dat, met behulp van deze clauses, uit p de clause s kan worden afgeleid. Hiertoe vullen we allereerst S aan met de clause p: $S' = S \cup \{p\}$. Bovendien voegen we $\neg s$ toe; immers, als uit S' de clause s kan worden afgeleid, kan uit $S' \cup \{\neg s\}$ de lege clause worden afgeleid, zodat een refutatie wordt gevonden. Hiermee is dan het bewijs geleverd.

De uiteindelijke verzameling clauses wordt dus:

$$S^* = \{\neg p \vee q, \neg p \vee \neg r \vee s, \neg q \vee r, p, \neg s\}.$$

Hieruit moet nog een set-of-support bepaald worden.

In het geval van binaire resolutie maakt het in principe niet uit welke clauses als set-of-support dienen. Aangezien we echter proberen te redeneren vanuit p, zal inzichtelijker zijn wat er tijdens de inferentie gebeurt, als we $\{p\}$ als set-of-support kiezen.

Figuur 1.1 laat zien hoe de invoerfile voor OTTER er voor dit probleem uit komt te zien. Wanneer we deze invoerfile 'binres.in' noemen, zorgt de aanroep

```
otter < binres.in > binres.uit
```

ervoor, dat de file 'binres.uit' de uitvoer met betrekking tot dit probleem bevat.

Allereerst wordt hierin de inhoud van de invoerfile herhaald; merk daarbij op dat het commentaar is weggelaten en dat de clauses zijn genummerd:


```
The process was started by peterl on pearl,
Wed Feb 25 15:13:45 2004
The command was "otter".  The process ID is 8684.

set(binary_res).
   dependent: set(factor).
   dependent: set(unit_deletion).
clear(unit_deletion).
clear(back_sub).
assign(stats_level,0).

list(usable).
1 [] -p|q.
2 [] -p| -r|s.
3 [] -q|r.
4 [] -s.
end_of_list.

list(sos).
5 [] p.
end_of_list.
```

Vervolgens begint het redeneren: de geselecteerde given-clause wordt steeds afgedrukt, tezamen met zijn 'gewicht' (zie Paragraaf A.4), omdat de optie `print_given` 'set' is (de default-waarde). De waarde van de optie `print_kept` is eveneens default 'set', zodat ook steeds de afgeleide clauses als output worden gegeven. Tussen vierkante haken staan steeds de afleidingsgegevens, bestaande uit de gebruikte inferentieregel, en de nummers van de clauses waarop deze is toegepast:

```
given clause #1: (wt=1) 5 [] p.
** KEPT (pick-wt=2): 6 [binary,5.1,2.1] -r|s.
** KEPT (pick-wt=1): 7 [binary,5.1,1.1] q.
```

Bijvoorbeeld, `2.1` wil zeggen dat literal 1 van clause 2 betrokken was bij een resolutiestap. De set-of-support bestaat nu uit twee nieuwe clauses: $(\neg r \vee s)$ en q; q is de 'lichtste' van de twee en wordt dus de nieuwe given-clause:

```
given clause #2: (wt=1) 7 [binary,5.1,1.1] q.
** KEPT (pick-wt=1): 8 [binary,7.1,3.1] r.
```

De nieuwe set-of-support bevat $(\neg r \vee s)$ en r; r is de 'lichtste':

```
given clause #3: (wt=1) 8 [binary,7.1,3.1] r.
** KEPT (pick-wt=2): 9 [binary,8.1,2.2] -p|s.
```

De set-of-support bestaat nu uit twee clauses met hetzelfde gewicht: $(\neg r \vee s)$ en $(\neg p \vee s)$; $(\neg r \vee s)$ is het eerst afgeleid:

```
given clause #4: (wt=2) 6 [binary,5.1,2.1] -r|s.
** KEPT (pick-wt=1): 10 [binary,6.1,8.1] s.
```

Bij het verwerken van de nieuw afgeleide clause s blijkt dat deze uit één literal bestaat, en dat in de axiomaverzameling een clause $\neg s$ staat, zodat door middel van een unit conflict de lege clause kan worden afgeleid:

```
----> UNIT CONFLICT at   0.00 sec ----> 11 [binary,10.1,4.1] $F.
```

Aangezien de waarde van `print_proofs` default 'set' is, wordt vervolgens het gevonden bewijs afgedrukt:

```
---------------- PROOF ----------------

1 [] -p|q.
2 [] -p| -r|s.
3 [] -q|r.
4 [] -s.
5 [] p.
6 [binary,5.1,2.1] -r|s.
7 [binary,5.1,1.1] q.
8 [binary,7.1,3.1] r.
10 [binary,6.1,8.1] s.
11 [binary,10.1,4.1] $F.

------------ end of proof -------------
```

De default-waarde van `max_proofs` is 1, dus OTTER stopt:

```
Search stopped by max_proofs option.

============ end of search ============
```

► *In de file* **binres.in** *is de in Figuur 1.1 weergegeven invoerfile voor* OTTER *opgenomen. De regels met respectievelijk de clauses* `-s.` *en* `p.` *zijn echter verwisseld. Probeer te voorspellen welke clauses* OTTER *nu zal genereren, en welk bewijs zal worden afgedrukt. Teken een met dit bewijs corresponderende refutatieboom (als in Figuur 2.2. van de syllabus). Controleer uw voorspelling met behulp van* OTTER.

Indien binaire resolutie toegepast wordt voor predikatenlogica, is het soms noodzakelijk factorisering toe te passen (optie `factor` in OTTER); deze wordt echter automatisch 'aangezet' zodra voor binaire resolutie gekozen wordt.

1.2.2 Hyperresolutie en UR-resolutie

Bij *positieve hyperresolutie* wordt een clause met tenminste één negatieve literal geresolveerd met een aantal positieve clauses[4], mits dit leidt tot een hyperresolvente,

[4]Een positieve clause is een clause, die uit alleen positieve literals bestaat.

die een positieve clause is. Bij *negatieve hyperresolutie* wordt een clause met tenminste één positieve literal geresolveerd met een aantal negatieve clauses[5], mits dit leidt tot een hyperresolvente, die een negatieve clause is. *UR-resolutie* is een combinatie van positieve en negatieve resolutie, met de beperking dat de resolvente een unit clause is. Aangezien OTTER de set-of-support strategie hanteert, moet één van al deze clauses deel uitmaken van de set-of-support, en de anderen van de axiomaverzameling. In OTTER is bij de toepassing van positieve of negatieve hyperresolutie altijd ook *geordende hyperresolutie* actief. Dit houdt in dat in de satellieten die in principe deelnemen in hyperresolutie alleen die complementaire literals mogen worden weggeresolveerd die maximale literals van de satelliet zijn volgens een lexicografische ordening van de literals. Dit is een vrij drastische beperking, vandaar dat altijd expliciet opgegeven moet worden:

```
clear(order_hyper).
```

In het volgende voorbeeld proberen we met behulp van positieve hyperresolutie nogmaals een refutatie te vinden voor de verzameling clauses, zoals die in de vorige paragraaf is beschreven (dus waarbij `p` in de set-of-support voorkomt, niet `-s`). De regel

```
set(binary_res).        % pas binaire resolutie toe.
```

in Figuur 1.1 is nu dus vervangen door:

```
set(hyper_res).         % pas positieve hyperresolutie toe.
```

OTTER doet nu de volgende inferentie-stappen:

```
given clause #1: (wt=1) 5 [] p.
** KEPT (pick-wt=1): 6 [hyper,5,1] q.
```

In tegenstelling tot bij binaire resolutie wordt de clause $(\neg r \vee s)$ hier niet afgeleid; deze bevat immers nog een negatieve literal, die niet kan worden weggeresolveerd met een positieve clause.

```
given clause #2: (wt=1) 6 [hyper,5,1] q.
** KEPT (pick-wt=1): 7 [hyper,6,3] r.
```

De volgende stap is een 'echte' positieve hyperresolutie-stap: p en r worden gebruikt om uit $(\neg p \vee \neg r \vee s)$ de negatieve literals weg te resolveren:

```
given clause #3: (wt=1) 7 [hyper,6,3] r.
** KEPT: 8 [hyper,7,2,5] s.
```

Wederom wordt bij het verwerken van de nieuw afgeleide clause s ten gevolge van een unit conflict met $\neg s$ de lege clause afgeleid. Merk op dat OTTER dit als een *binaire* resolutie-stap beschouwt:

[5]Een negatieve clause is een clause, die uit alleen negatieve literals bestaat.

```
----> UNIT CONFLICT at   0.00 sec ----> 9 [binary,8.1,4.1] $F.

--------------- PROOF ----------------

1 [] -p|q.
2 [] -p| -r|s.
3 [] -q|r.
4 [] -s.
5 [] p.
6 [hyper,5,1] q.
7 [hyper,6,3] r.
8 [hyper,7,2,5] s.
9 [binary,8.1,4.1] $F.

------------ end of proof -------------
```

Het voordeel van hyperresolutie boven binaire resolutie is dat minder tussenstappen worden genomen, zodat de inferentie sneller gaat en de bewijzen meestal korter zijn. Nadeel is dat hyperresolutie, gecombineerd met de set-of-support strategie niet altijd volledig is, zoals ook hieronder blijkt.

► *Bestudeer de file* **hyperres.in**. *Probeer te voorspellen welke clauses* OTTER *zal genereren, en welk bewijs zal worden afgedrukt. Onderscheid bij elke inferentie-stap nucleus en satellieten. Controleer uw voorspelling. Vervang de specificatie*

```
set(hyper_res).         % pas positieve hyperresolutie toe.
```

door

```
set(neg_hyper_res).     % pas negatieve hyperresolutie toe.
```

en ga het verschil tussen positieve en negatieve hyperresolutie na. Vervang vervolgens de laatste specificatie door

```
set(ur_res).            % pas UR-resolutie toe.
```

en vergelijk de resultaten met binaire resolutie, positieve en negatieve hyperresolutie.

► *In de file* **binres.in** *is de in Figuur 1.1 weergegeven invoerfile voor* OTTER *opgenomen. De regels met respectievelijk de clauses* `-s.` *en* `p.` *zijn echter verwisseld. Probeer nu respectieve positieve en negatieve hyperresolutie toe te passen, en bestudeer de verschillen in uitvoer. Probeer een verklaring te geven voor dit verschil.*

1.2.3 Backward-subsumptie

Wanneer de optie `back_sub` 'set' is (de default-waarde), past OTTER backward-subsumptie toe: alle clauses uit de axiomaverzameling en de set-of-support die gesubsumeerd worden door een nieuw gegenereerde clause, worden verwijderd. Een clause C_1 wordt gesubsumeerd door een clause C_2, als deze een disjunctie is van C_2 en een (eventueel lege) clause C_3. De clause $(p \vee q)$ wordt bijvoorbeeld gesubsumeerd door p. Merk op dat hierdoor geen informatie verloren gaat; immers, als p waar is, dan geldt zeker $(p \vee q)$.

Door uit Figuur 1.1 de regel

```
clear(back_sub).        % pas geen backward-subsumptie toe.
```

weg te laten, zal OTTER wèl backward-subsumptie toepassen. OTTER zal ook melden, op welke clauses backward-subsumptie wordt toegepast, aangezien de default-waarde van `print_back_sub` 'set' is:

```
given clause #1: (wt=1) 5 [] p.
** KEPT (pick-wt=2): 6 [binary,5.1,2.1] -r|s.
** KEPT (pick-wt=1): 7 [binary,5.1,1.1] q.
6 back subsumes 2.
7 back subsumes 1.
```

De clause $(\neg p \vee \neg r \vee s)$ wordt gesubsumeerd door $(\neg r \vee s)$, en q subsumeert $(\neg p \vee q)$.

```
given clause #2: (wt=1) 7 [binary,5.1,1.1] q.
** KEPT (pick-wt=1): 8 [binary,7.1,3.1] r.
8 back subsumes 3.
```

Het genereren van clause r heeft het verwijderen van $(\neg q \vee r)$ uit de axiomaverzameling tot gevolg.

```
given clause #3: (wt=1) 8 [binary,7.1,3.1] r.
```

De nieuwe given-clause r resolveert nu nergens mee, omdat $(\neg p \vee \neg r \vee s)$ verwijderd is, en $(\neg r \vee s)$ zich nog in de set-of-support bevindt:

```
given clause #4: (wt=2) 6 [binary,5.1,2.1] -r|s.
** KEPT (pick-wt=1): 9 [binary,6.1,8.1] s.

----> UNIT CONFLICT at   0.00 sec ----> 10 [binary,9.1,4.1] $F.

Length of proof is 4.  Level of proof is 3.

--------------- PROOF ----------------

1 [] -p|q.
2 [] -p| -r|s.
```

```
3 [] -q|r.
4 [] -s.
5 [] p.
6 [binary,5.1,2.1] -r|s.
7 [binary,5.1,1.1] q.
8 [binary,7.1,3.1] r.
9 [binary,6.1,8.1] s.
10 [binary,9.1,4.1] $F.

------------ end of proof -------------
```

► *Bestudeer de file* **backsub.in**. *Probeer te voorspellen welke clauses* OTTER *zal genereren, en welke clauses hierbij gesubsumeerd zullen worden. Controleer uw voorspelling.*

1.2.4 De omzetting van formules in clauses

Behalve clauses accepteert OTTER ook formules in eerste-orde predikatenlogica als invoer; deze worden in clauses vertaald en verder als zodanig behandeld.

Om een formule in een verzameling clauses te kunnen vertalen, moet deze allereerst in *conjunctieve normaalvorm* gebracht worden (zie Paragraaf 2.3 van de syllabus). Zo ontstaat een formule met een matrix van de vorm $(C_1 \wedge \cdots \wedge C_n)$. Deze levert vervolgens, eventueel na omzetting in Skolem normaalvorm, een verzameling clauses $\{C_1, \ldots, C_n\}$.

In de eerste-orde predikatenlogica worden variabelen expliciet gekwantificeerd. In clauses onderscheidt OTTER constanten en variabelen op grond van hun beginletter: variabelen dienen met een kleine letter u, v, w, x, y of z te beginnen. Tijdens het vertaalproces van formules naar clauses verandert OTTER daarom namen van gekwantificeerde variabelen, die nìet met u, v, w, x, y of z beginnen, in `x1`, `x2`, ... Namen van constanten, die met een kleine letter u, v, w, x, y of z beginnen, kunnen door de gebruiker worden verward met variabelen. OTTER geeft in dit geval een waarschuwing; het is beter de namen van de constanten anders te kiezen.

Beschouw nu het voorbeeld, zoals beschreven in Figuur 1.2. In de formules van de axiomaverzameling in deze invoerfile zijn de namen `plaats1`, `plaats2`, ..., gekwantificeerd met behulp van `all`; het zijn dus variabelen. Omdat namen die met een p beginnen in clauses niet als variabele herkend worden, worden ze vertaald in `x1`, `x2`, ...:

```
-------> usable clausifies to:

list(usable).
1 [] -Verbinding(x1,x2)
   | -Verbinding(x2,x3)
   | Verbinding(x1,x3).
2 [] -Verbinding(amsterdam,harlingen).
```



```
% Opties voor OTTER:
set(hyper_res).         % pas hyperresolutie toe.
clear(print_back_sub).  %
clear(print_kept).      % geef geen informatie over de genomen
clear(print_given).     %    inferentie-stappen.
assign(stats_level,0).  % lever geen statistische uitvoer.

formula_list(usable).   % lees axioma's in formule-vorm.
% Als er een verbinding is tussen plaats1 en plaats2, en er is een
% verbinding tussen plaats2 en plaats3, dan is er een verbinding
% tussen plaats1 en plaats3:
  (all plaats1  all plaats2  all plaats3
      (  (  Verbinding(plaats1,plaats2)
          & Verbinding(plaats2,plaats3) )
       -> Verbinding(plaats1,plaats3) )  ).
% AF TE LEIDEN: is er een verbinding tussen Amsterdam en Harlingen?
  -Verbinding(amsterdam,harlingen).
end_of_list.

formula_list(sos).      % lees sos in formule-vorm.
% GEGEVEN:
   Verbinding(amsterdam,utrecht).
   Verbinding(utrecht,zwolle).
   Verbinding(zwolle,leeuwarden).
   Verbinding(leeuwarden,harlingen).
end_of_list.
```

Figuur 1.2: Invoerfile voor OTTER waarbij de formules vertaald worden in clauses.

```
end_of_list.
```

Ook de formules van *sos* worden in clauses omgezet:

```
-------> sos clausifies to:

list(sos).
3 [] Verbinding(amsterdam,utrecht).
4 [] Verbinding(utrecht,zwolle).
5 [] Verbinding(zwolle,leeuwarden).
6 [] Verbinding(leeuwarden,harlingen).
end_of_list.
```

in dit geval is de vertaling dus gelijk aan de originele formules. De namen `utrecht` en `zwolle` staan voor constanten (want ze zijn niet gekwantificeerd), maar zouden

door de naamgeving verward kunnen worden met variabelen. Daarom wordt bij de vertaling van de formules 3, 4 en 5 een waarschuwing gegenereerd.

Beter is het daarom om de betreffende namen aan te passen, bijvoorbeeld door er een _ (underscore) voor te zetten (`_zwolle`), of door ze met een hoofdletter te laten beginnen:

```
formula_list(sos).
   Verbinding(Amsterdam,Utrecht).
   Verbinding(Utrecht,Zwolle).
   Verbinding(Zwolle,Leeuwarden).
   Verbinding(Leeuwarden,Harlingen).
end_of_list.
```

Voor zinvolle variabele-namen geldt analoog dat je ze kunt 'redden' door er `x_` voor te zetten.

We merken tenslotte op dat de set-of-support ook als volgt ingevoerd had kunnen worden:

```
formula_list(sos).
   (Verbinding(Amsterdam,Utrecht) &
    Verbinding(Utrecht,Zwolle) &
    Verbinding(Zwolle,Leeuwarden) &
    Verbinding(Leeuwarden,Harlingen)).
end_of_list.
```

► *De file* **vertaling.in** *bevat de laatst genoemde versie van de in Figuur 1.2 weergegeven invoerfile. Pas de variabele-namen in deze file zodanig aan dat ook ná vertaling duidelijk blijft dat ze plaatsen representeren. Controleer de aangepaste file vervolgens aan de hand van de vertaling die* OTTER *produceert. Controleer het bewijs.*

1.2.5 Het gebruik van answer literals

Een literal waarvan de predikaatnaam begint met `$ANS`, `$Ans`, of `$ans`, heet een answer literal, en wordt bij de inferentie genegeerd; een clause die slechts uit answer literals bestaat wordt dus beschouwd als de lege clause. Answer literals kunnen zodoende gebruikt worden om informatie over een gevonden bewijs op te slaan.

Uit het bewijs in de vorige paragraaf was bijvoorbeeld nauwelijks op te maken waarlangs de route van Amsterdam naar Harlingen loopt; door in enkele formules een answer literal toe te voegen wordt dit een stuk duidelijker:

```
formula_list(usable).   % lees axioma's in formule-vorm.
% Als er een verbinding is tussen plaats1 en plaats2, en er is een
% verbinding tussen plaats2 en plaats3, dan is er een verbinding
% tussen plaats1 en plaats3; onthoud dat deze via plaats2 loopt:
  (all plaats1  all plaats2  all plaats3
      (  (  Verbinding(plaats1,plaats2)
```



```
% Opties voor OTTER:
set(binary_res).        % pas binaire resolutie toe.
assign(stats_level,0).  % lever geen statistische uitvoer.

list(usable).           % lees axioma's in clause-vorm.
  -Evalueer(x) | $Ans(x).
end_of_list.

list(demodulators).     % lees demodulators in clause-vorm.
   (f(a,b) = d).        % of: Eq(f(a,b),d).
   (g(c,d) = e).        % of: Eq(g(c,d),e).
   (g(c,f(a,b)) = d).   % of: Eq(g(c,f(a,b)),d).
end_of_list.

list(sos).              % lees sos in clause-vorm.
   Evalueer( g(c,f(a,b)) ).
end_of_list.
```

Figuur 1.3: Invoerfile voor OTTER ter demonstratie van demodulatie.

```
         & Verbinding(plaats2,plaats3) )
       -> (  Verbinding(plaats1,plaats3)
           | $Ans_via(plaats2) ) )  ).
% Als er verbinding tussen A'dam en Harlingen is, onthoud dat dan:
  (   Verbinding(Amsterdam,Harlingen)
   -> $Ans_verbinding(Amsterdam,Harlingen) ).
end_of_list.
```

► *In de file* **answer.in** *is bovengenoemd voorbeeld van het gebruik van answer literals opgenomen. Voer deze file in* OTTER *in, en bestudeer het bewijs dat wordt afgeleid.*

1.2.6 Demodulatie

Bij demodulatie wordt een clause herschreven met behulp van een verzameling gelijkheden. Gelijkheid kan worden weergegeven door middel van het infix-predikaatsymbool `=`, of met behulp van een binair predikaatsymbool, beginnend met `EQ`, `Eq`, of `eq`. Demodulatie vindt plaats tijdens de verwerking van een nieuw gegenereerde clause.

OTTER hanteert bij demodulatie een wat andere vorm van gelijkheid dan in Hoofdstuk 2 van de syllabus is behandeld. Om oneindig 'heen en weer schrijven' te voorkomen vindt herschrijving namelijk slechts in één richting plaats. Het begrip 'gelijkheid' voldoet hier dus niet aan de symmetrie-eigenschap. Beschouw Figuur 1.3. Hierin wordt een term geëvalueerd met behulp van enkele herschrijfregels. Demodulatie vindt default van binnen naar buiten plaats, zodat het volgende resultaat wordt verkregen:

```
given clause #1: (wt=6) 5 [] Evalueer(g(c,f(a,b))).
** KEPT (pick-wt=0): 6 [binary,5.1,1.1,demod,2,3] $Ans(e).

-----> EMPTY CLAUSE at   0.00 sec ----> 6 [binary,5.1,1.1,demod,2,3]
 $Ans(e).

---------------- PROOF ----------------

1 [] -Evalueer(x)|$Ans(x).
2 [] f(a,b)=d.
3 [] g(c,d)=e.
5 [] Evalueer(g(c,f(a,b))).
6 [binary,5.1,1.1,demod,2,3] $Ans(e).

------------ end of proof -------------
```

Wanneer we het commando

```
set(demod_out_in).      % demoduleer van buiten naar binnen.
```

toevoegen, gebeurt het volgende:

```
given clause #1: (wt=6) 5 [] Evalueer(g(c,f(a,b))).
** KEPT (pick-wt=0): 6 [binary,5.1,1.1,demod,4] $Ans(d).

-----> EMPTY CLAUSE at   0.00 sec ----> 6 [binary,5.1,1.1,demod,4] $
Ans(d).

Length of proof is 0.  Level of proof is 0.

---------------- PROOF ----------------

1 [] -Evalueer(x)|$Ans(x).
4 [] g(c,f(a,b))=d.
5 [] Evalueer(g(c,f(a,b))).
6 [binary,5.1,1.1,demod,4] $Ans(d).

------------ end of proof -------------
```

In dit geval wordt dus een ander antwoord verkregen.

Een eindige automaat is een eindig wiskundig object dat wordt gebruikt om te bepalen of een gegeven invoer-symboolrij een bepaalde vorm heeft. Automaten worden bijvoorbeeld toegepast bij het bouwen van compilers voor formele talen. Een automaat verkeert telkens in een bepaalde toestand. Bij een eindige automaat is een verzameling toestanden gedefinieerd, waarvan één de *begintoestand* is en een aantal toestanden zogenaamde *eindtoestanden* zijn. Door telkens een symbool van de

invoer-symboolrij te bekijken, wordt de automaat in een nieuwe toestand gebracht. Hoe dat gebeurt, is vastgelegd in een bij de automaat behorende *transitiefunctie*, ook wel *overgangsfunctie* genoemd. Als na het verwerken van een gegeven invoer-symboolrij de eindige automaat in een eindtoestand verkeert, zegt men dat de invoer-symboolrij *geaccepteerd* is, anders noemt men de symboolrij *verworpen*.

Bij de automaat die hier beschouwd wordt, is de volgende verzameling toestanden gedefinieerd:

$$S = \{s_1, s_2, s_3, s_4\}$$

Hierin is s_1 de begintoestand en s_3 de eindtoestand. De transitiefunctie is gespecificeerd in de onderstaande verzameling toestandsovergangen of transities:

$s_1 \rightarrow \mathrm{a}s_1$
$s_1 \rightarrow \mathrm{a}s_2$
$s_1 \rightarrow \mathrm{b}s_1$
$s_2 \rightarrow \mathrm{b}s_3$
$s_3 \rightarrow \mathrm{b}s_4$
$s_4 \rightarrow s_3$

De laatste toestandsovergang is een zogenaamde stille overgang: er wordt geen gebruik gemaakt van een symbool van de invoer-symboolrij. De toestandsovergangen zijn eenvoudig in OTTER vast te leggen: een literal `transitie(s1,c) = s2` geeft de overgang aan van een toestand s_1 naar s_2 onder verwerking van het symbool 'c'. In `stil(s1) = s2` wordt een stille toestandsovergang gerepresenteerd. Dat s_3 de eindtoestand is, wordt aangegeven met het predikaatsymbool `Eind`.

▶ *De file* **demod.in** *bevat een implementatie in* OTTER *van een eindige automaat. Bestudeer deze file, en test de gegeven invoerstring. Waarom kan niet volstaan worden met slechts één functie-symbool* `transitie`*? Wat gebeurt er als een niet-passende string ingevoerd wordt?*

1.2.7 Evalueerbare predikaten en functies

Indien in een atoom een evalueerbaar predikaat wordt gebruikt, kan het atoom door een speciale interpretator naar true of false worden geëvalueerd. Deze vorm van evaluatie zullen we in het vervolg $-evaluatie noemen. $-evaluatie wordt toegepast tijdens het redeneren. In Paragraaf A.6 worden de belangrijkste evalueerbare predikaten beschreven.

Figuur 1.4 bevat een invoerfile met twee formules, die kunnen nagaan of de som van twee getallen groter, respectievelijk kleiner, is dan hun produkt. In plaats van `$GT(...)`, respectievelijk `$LT(...)`, zijn in de antecedenten van de axioma's de gelijkwaardige negatieve literals `-$LE(...)` en `-$GE(...)` gebruikt: bij de vertaling van formules in clauses worden deze negatieve literals positief ten gevolge van de herschrijfregels

```
% Opties voor OTTER
set(hyper_res).         % pas hyperresolutie toe.
clear(print_kept).      % geef geen informatie over de genomen
clear(print_given).     %    inferentie-stappen.
clear(print_proofs).    % druk de bewijzen niet af.
assign(max_proofs,2).   % zoek twee bewijzen.
assign(stats_level,0).  % lever geen statistische uitvoer.

formula_list(usable).   % lees axioma's in formule-vorm.
  (all x  all y
      ( (   Test_getallen(x,y)
         & -$LE($SUM(x,y), $PROD(x,y)) )
       -> (  $Ans_som(x,y,$SUM(x,y))
           | $Ans_prod(x,y,$PROD(x,y))
           | $Ans_som_gt_prod($GT($SUM(x,y), $PROD(x,y))) ) )  ).
  (all x  all y
      ( (  Test_getallen(x,y)
         & -$GE($SUM(x,y), $PROD(x,y)) )
       -> (  $Ans_som(x,y,$SUM(x,y))
           | $Ans_prod(x,y,$PROD(x,y))
           | $Ans_som_lt_prod($LT($SUM(x,y), $PROD(x,y))) ) )  ).
end_of_list.

list(sos).              % lees sos in clause-vorm.
  Test_getallen(1,2).
  Test_getallen(3,5).
end_of_list.
```

Figuur 1.4: Invoerfile voor OTTER ter demonstratie van $-evaluatie.

$(p \rightarrow q) \equiv (\neg p \vee q)$, en
$\neg(\neg p) \equiv p$,

zodat positieve hyperresolutie toegepast kan worden.

OTTER doet alle $-evaluatie-stappen in één keer. Om toch de afzonderlijke stappen te kunnen volgen zijn enkele answer literals toegevoegd.

De formules worden als volgt vertaald:

```
-------> usable clausifies to:

list(usable).
1 []
   -Test_getallen(x,y)
 | $LE($SUM(x,y),$PROD(x,y))
 | $Ans_som(x,y,$SUM(x,y))
 | $Ans_prod(x,y,$PROD(x,y))
 | $Ans_som_gt_prod($GT($SUM(x,y),$PROD(x,y))).
2 []
   -Test_getallen(x,y)
 | $GE($SUM(x,y),$PROD(x,y))
 | $Ans_som(x,y,$SUM(x,y))
 | $Ans_prod(x,y,$PROD(x,y))
 | $Ans_som_lt_prod($LT($SUM(x,y),$PROD(x,y))).
end_of_list.
```


Als 'test-invoer' voor deze formules worden de volgende clauses gebruikt:

```
list(sos).
3 [] Test_getallen(1,2).
4 [] Test_getallen(3,5).
end_of_list.
```

De eerste clause uit *sos* resolveert met de eerste clause uit *usable*, waarbij voor elke `x` een `1` wordt gesubstitueerd, en voor elke `y` een `2`. In dezelfde inferentie-stap evalueren achtereenvolgens de termen `$SUM(1,2)` en `$PROD(1,2)` tot `3` respectievelijk `2`, waarna de literal `$LE($SUM(x,y),$PROD(x,y))` wordt verwijderd, omdat immers geldt dat $3 \not\leq 2$, ofwel `-$LE(3,2)`:

```
-----> EMPTY CLAUSE at   0.00 sec ----> 5 [hyper,3,1,demod]
   $Ans_som(1,2,3)
 | $Ans_prod(1,2,2)
 | $Ans_som_gt_prod($T).
```

De clauses 2 en 4 resolveren op een analoge wijze:

```
-----> EMPTY CLAUSE at   0.00 sec ----> 6 [hyper,4,2,demod]
   $Ans_som(3,5,8)
 | $Ans_prod(3,5,15)
 | $Ans_som_lt_prod($T).

Search stopped by max_proofs option.

============ end of search ============
```

► *De in Figuur 1.4 weergegeven invoerfile voor* OTTER *is opgenomen in de file* **dollar.in**. *Voeg aan deze invoerfile een formule toe, die kan nagaan of de som van twee getallen* gelijk *is aan hun produkt. Test deze formule met de getallen* 2 *en* 2. *Druk de invoer- en een geformatteerde uitvoerfile af, en lever in met een korte verklaring.*

1.3 Causaal redeneren

Bij causaal redeneren wordt gebruik gemaakt van oorzaak-en-gevolg relaties. Causale kennis speelt een belangrijke rol in elk vakgebied. In de onderstaande opdracht wordt eigenlijk alleen

► *De file* **raa.in** *bevat een representatie in* OTTER *van de in Paragraaf 6.3 beschreven kennis over het RAA-systeem. Bestudeer nogmaals het gedrag van dit negatief teruggekoppelde regelsysteem door aan de set-of-support de twee clauses*

```
 (druk(bloed) = daling).
-(druk(bloed) = x) | $Ans(Bloeddruk(x)).
```

toe te voegen. (Het eerste bewijs zal een triviaal antwoord opleveren.) Verwijder daarna de toegevoegde clauses weer uit de set-of-support.

1.4 Functionele modellen

In Hoofdstuk 2 van de syllabus is de logische specificatie van het gedrag van logische circuits besproken. Als voorbeeld is de 'full adder' besproken, en het gebruik van een logische specificatie van een logisch circuit voor simulatie van gedrag en voor diagnostiek van defecte componenten.

★ (**O.1**) *De file* **fulladd.in** *bevat een logische specificatie van het gedrag van een full adder. Onderzoek de waarden van de uitgangssignalen o_1 en o_2 als de ingangssignalen de waarden $i_1 = i_2 = i_3 = 1$ hebben. Onderzoek dezelfde vraag vervolgens met gebruikmaking van negatieve hyperresolutie.*

Pas de file **fulladd.in** *nu aan, zodanig dat de logische specificatie gebruikt kan worden voor het vaststellen van de combinatie van defecte componenten die verantwoordelijk geacht kan worden voor een niet passende combinatie van ingangs- en uitgangssignalen. Stel dat bij ingangssignalen $i_1 = 1$, $i_2 = 0$, $i_3 = 1$ de uitgangssignalen $o_1 = 1$ en $o_2 = 0$ worden gemeten. Stel dan met behulp van de logische specificatie en* OTTER *vast welke componenten tenminste defect moeten zijn bij de gegeven ingangs- en uitgangssignalen. Bespreek uw resultaten met de assistentie.*

1.5 Opgave O.2: logische specificatie met OTTER

Deze opgave is bedoeld om u kennis te laten maken met het formaliseren van problemen in logica, en het gebruik van een theorem prover voor het oplossen van een probleem. In deze paragraaf wordt een aantal verschillende opgaven besproken. U dient, *in overleg met de practicumassistentie*, één opgave uit te kiezen.

1.5.1 Een kennissysteem voor diagnostiek

MYCIN is één van de eerste kennissystemen, waarin aan de hand van kennisregels van de vorm

Als
    patiënt A koorts heeft, en
    koude rillingen
dan
    heeft patiënt A griep

Het feit dat de patiënt koorts heeft, kan worden afgeleid met, bijvoorbeeld, de volgende regel:

Als
    de lichaamstemperatuur van patiënt A hoger is dan 37.5
dan
    heeft patiënt A koorts

Regels, zoals hierboven, worden vaak *classificatieregels* genoemd.

Deze regeks kunnen in logica worden gerepresenteerd door aan te nemen dat regels van de vorm 'Als P dan C' geïnterpreteerd kunnen worden als logische implicaties $P' \rightarrow C'$, waarbij P' and C' de resultaten zijn van de vertalingen van P en C. Om het mogelijk te maken ziekten van verschillende patiënten gelijktijdig te diagnosticeren, kan een implicatie de vorm:

$$\forall x((\text{Patiënt}(x) \wedge \cdots \wedge \cdots) \rightarrow \text{Diagnose}(x, \text{ziektenaam}))$$

hebben. Voor patiënt A zou bijvoorbeeld de conclusie 'Diagnose(A, ziektenaam)' getrokken kunnen worden.

★ *Opgave:*

- *Beschouw de ziekten acuut claucoom en glaucoma simplex, beide ziekten die het gevolg zijn van verminderde afvoer van glasvocht in het oog en die blindheid tot gevolg kunnen hebben (zie www.oogziekenhuis.nl voor details).*
- *Stel aan de hand van de informatie op www.oogziekenhuis.nl classificatieregels samen voor de diagnose van acuut glaucoom en glaucoma simplex.*
- *Vertaal deze regels in predicatenlogica.*
- *Stel met behulp van* OTTER *de diagnose acuut glaucoom of glaucoma simplex voor 4 verschillende patiënten. Motiveer in uw verslag de gekozen samenstelling van de set-of-support en de keuze van inferentieregels.*

1.5.2 Het waterbak-probleem

Beschouw een bak met een inhoud van 5 liter, die gevuld is met 4 liter water, en een tweede bak met een inhoud van 2 liter, die volledig gevuld is. Het is toegestaan de betreffende bakken leeg te gieten, of water van één bak in de andere bak over te gieten (zonder te morsen). Er is echter niet meer water beschikbaar dan de 6 liter die totaal in beide bakken aanwezig is.

★ *Opgave:*

- *Stel een algemene logische specificatie op voor dit probleem in eerste-orde predikatenlogica. Maak hierbij gebruik van het predikaatsymbool*

  $$Toestand(hoeveelheid(b_1, b_2), actie(a_1, y))$$

  voor de representatie van de toestandsaxioma's; hierin is b_1 de inhoud van de eerste bak en b_2 de inhoud van de tweede bak op een bepaald moment; a_1 is de uitgevoerde actie (leeggieten van de bak of overgieten van water van de kleine bak in de grote bak, of van de grote bak in de kleine bak), en de variabele y wordt gebruikt voor het verzamelen van eerder uitgevoerde acties.
- *Bewijs met behulp van* OTTER *dat het achtereenvolgens leeggieten van waterbakken en overgieten van water, tot een toestand kan leiden waarin 3 liter aanwezig is in de grote bak, en de kleine bak leeg is. Als neveneffect van het bewijs moeten de opeenvolgende stappen gepresenteerd worden die tot deze toestand geleid hebben. Deze opeenvolging van acties kan beschouwd worden als een planning. Motiveer in uw verslag de gekozen samenstelling van de set-of-support en de keuze van inferentieregels.*

1.5.3 Een besturingssysteem van een lift (of de A van Abeltje)

Beschouw een lift in een flat met drie verdiepingen. Bij de liftdeur op elke verdieping is een knop aangebracht, waarmee de lift naar de verdieping kan worden geroepen. De lift kan naar boven en naar beneden gaan. In de lift is een paneel met vier knoppen aangebracht, voor elke verdieping en de begane grond, één. Als de lift vanaf een bepaalde verdieping naar boven gaat, zal worden gestopt op elke verdieping waarvoor een knop in de lift of de betreffende verdieping is ingedrukt, in de volgorde waarin de lift de verdiepingen aandoet. Analoog zal elke verdieping waarvoor een knop is ingedrukt worden aangedaan als de lift naar beneden gaat. De lift zal slechts naar boven of naar beneden gaan, als hiervoor tenminste één opdracht is gegeven door indrukken van een knop op een verdieping of in de lift. Als de lift de bovenste verdieping heeft bereikt, zal deze automatisch naar beneden gaan, als hiervoor opdracht wordt gegeven; analoog zal de lift naar boven gaan als deze op de begane grond is aangekomen. Als geen opdracht wordt gegeven, zal de lift stoppen.

★ *Opgave:*

- *Vertaal de bovenstaande beschrijving in een logische specificatie in eerste-orde predikatenlogica. Maak hierbij gebruik van het predikaatsymbool*

  $$Toestand(verdieping, richting, v_0, v_1, v_2, v_3, l_0, l_1, l_2, l_3)$$

  waarin

  - verdieping *de verdieping (*0, 1, 2, *of* 3*) aangeeft waar de lift zich bevindt;*
  - richting *aangeeft of de lift stil staat, naar boven (*naar_boven*) of naar beneden (*naar_beneden*) gaat;*
  - *v_i aangeeft of de lift naar verdieping i geroepen wordt (v_i is gelijk aan aan of uit);*
  - *l_i geeft aan of de knop op het paneel in de lift voor verdieping i oplicht (l_i is gelijk aan aan of uit); iemand in de lift wil naar verdieping i als l_i = aan.*

  Beschouw de volgende clause

  ```
  Toestand(1, naar_boven, aan, uit, uit, aan, uit, uit, aan, aan).
  ```

  die tot uitdrukking brengt dat de lift zich op verdieping 1 bevindt en naar boven gaat, terwijl de lift gevraagd wordt naar verdieping 0 en 3 (ingedrukte liftknoppen op de verdiepingen) en 2 en 3 (ingedrukte knoppen in de lift) te gaan. De lift zal naar boven gaan en de verdieping 2 en 3 aandoen, en dan naar beneden gaan en stoppen op de begane grond. Maak gebruik van het predikaatsymbool

  $$Naar(verdieping, richting, volgende\ verdieping, verandering\ richting)$$

  *waarin '*volgende verdieping*' de verdieping is waar de lift naar toe zal gaan, en '*verandering richting*' (*wel_veranderd*/*niet_veranderd*) aangeeft of de lift in vergelijking met de vorige keer van richting is veranderd. Bijvoorbeeld,*

  ```
  Naar(1, naar_boven, 3, wel_veranderd).
  ```

  geeft aan dat de lift van de eerste verdieping naar boven, naar de derde verdieping, gaat, nadat de lift eerder naar beneden ging.
- *Maak nu gebruik van* OTTER *om vast te stellen in welke volgorde de lift de verdiepingen langs gaat als enkele personen op diverse verdiepingen op de liftknop hebben gedrukt, en bovendien enkele personen in de lift op bepaalde verdiepingen willen uitstappen.*

1.5.4 Een termherschrijfsysteem

Een *termherschrijfsysteem* (TRS) is een stelsel regels dat gebruikt kan worden voor de syntactische manipulatie van expressies. Beschouw bijvoorbeeld het volgende termherschrijfsysteem

$$\begin{array}{ll} add(0, x) & \rightarrow x \\ add(s(y), x) & \rightarrow s(add(y, x)) \\ mult(0, x) & \rightarrow 0 \\ mult(s(y), x) & \rightarrow add(mult(y, x), x) \end{array}$$

waarmee algebraïsche expressies waarin de prefix-operatoren *add* (optellen), *mult* (vermenigvuldigen), s (optellen met 1, de successorfunctie) of de constante 0 voorkomen, kunnen worden herschreven. De term $add(s(s(0)), s(0))$ kan, bijvoorbeeld, door toepassing van het bovenstaande TRS als volgt worden herschreven:

$$\begin{aligned} add(s(s(0)), s(0)) &\Rightarrow s(add(s(0), s(0))) \\ &\Rightarrow s(s(add(0, s(0)))) \\ &\Rightarrow s(s(s(0))) \end{aligned}$$

Hierin wordt met '$\Rightarrow$' de toepassing van een herschrijfregel aangeduid.

Formeel mag een term t herschreven worden tot de term s als er een regel $l \to r$ bestaat in de TRS en een substitutie σ, zodanig dat:

$$t \equiv l\sigma \Rightarrow r\sigma \equiv s.$$

We merken op dat een herschrijving ook mag plaatsvinden op een subterm van een term t. Bijvoorbeeld, de term $f(\ldots, t, \ldots)$ kan worden herschreven tot $f(\ldots, s, \ldots)$ door toepassing van de herschrijfregel $t \to s$.

Merk op dat de term $add(z, x)$ niet herschreven kan worden door de bovenstaande TRS. Immers, er bestaat geen substitutie σ zodanig dat $0\sigma \equiv z$ of $s(y)\sigma \equiv z$. Een term die niet herschreven kan worden, heet een *normaalvorm*.

★ *Opgave:*

- *Stel een logische specificatie op van bovenstaand termherschrijfsysteem, rekening houdend met de gewenste vorm van het antwoord (zie hieronder).*
- *Maak gebruik van* OTTER *om de waarde van de volgende expressie met behulp van de logische specificatie uit te rekenen:*

  $$mult(add(s(s(s(0))), mult(s(0), s(s(0)))), add(s(0), s(s(0))))$$

  De gewenste uitvoer van OTTER *is een slechts uit answer-literals bestaande clause, die toont* via welke tussenresultaten *tot het eindresultaat is gekomen. Het eindresultaat is een term van de vorm* $s(s(\cdots s(0)\cdots))$.

1.5.5 Maaltijd-adviseur

In steeds meer gezinnen hebben beide partners een baan. Een ongunstig gevolg van deze ontwikkeling is de soms matige kwaliteit van de avondmaaltijd in dit soort gezinnen. Dit is gedeeltelijk een gevolg van het feit dat onvoldoende tijd voor het samenstellen van de avondmaaltijd kan worden uitgetrokken. Deze prangende maatschappelijke kwestie vraagt om een vernuftige oplossing. Van u wordt een bijdrage in het oplossen van dit probleem gevraagd door de ontwikkeling van een geautomatiseerde maaltijd-adviseur.

De maaltijd-adviseur kan op basis van een logische specificatie van de ingrediënten die thuis in voorraad zijn, en informatie over de samenstelling van gerechten, een menu samenstellen. Hierbij moet rekening gehouden worden met de eis dat

de hoeveelheid Calorieën in een maaltijd niet lager is dan een gegeven minimum, en niet hoger dan een bepaald maximum. Deze informatie kan met behulp van de volgende unit-clauses worden gerepresenteerd:

- Representatie van de voorraad:

  Voorraad(*ingrediënt*,
  hoeveelheid beschikbaar in 100 gr,
  Calorieën/100 gr)

  Bijvoorbeeld:

  ```
  Voorraad(suiker, 10, 400).
  ```

  Dat wil zeggen, er is 1 kg suiker beschikbaar in de voorraadkast; 100 gr suiker bevat 400 Calorieën.

- Beschrijving van de gerechten in termen van ingrediënten, waarbij per ingrediënt opgegeven wordt hoeveel gram per persoon nodig is voor bereiding van het gerecht:

  Gerecht(*naam gerecht*,
  [[*ingrediënt*, *hoeveelheid in grammen*], ...],
  soort gerecht)

  Bijvoorbeeld:

  ```
  Gerecht(macaroni_met_ham_en_kaas,
          [[macaroni,100],
           [ham,100],
           [kaas,30],
           [tomatenpuree,50]],
          hoofdgerecht).
  ```

- Verder moet vastgelegd worden uit hoeveel personen het gezin bestaat (met behulp van het unaire predikaatsymbool `Gezin`), en wat de minimale en maximale hoeveelheid Calorieën is die een avondmaaltijd mag bevatten (met behulp van de predikaatsymbolen `MinCalorieen` en `MaxCalorieen`). Bijvoorbeeld:

  ```
  MinCalorieen(600).
  MaxCalorieen(1200).
  Gezin(4).
  ```

- De uitvoer van het OTTER-programma is een answer-literal van de vorm:

$\$Ans(Menu(voorgerecht,$
$\quad hoofdgerecht,$
$\quad nagerecht,$
$\quad totaal\ aantal\ Calorieën))$

waarbij '*totaal aantal Calorieën*' het totaal aantal Calorieën in de maaltijd per persoon is; dit aantal mag niet minder dan het gegeven minimum, en niet hoger dan het gegeven maximum zijn.

★ *Opgave:*

- *Ontwikkel een logische specificatie van het bovenbesproken probleem in de vorm van een kennisbank waarin zowel een beschrijving van enkele ingrediënten en gerechten is opgenomen, alsmede een logische specificatie met behulp waarvan een menu kan worden samengesteld.*
- *Produceer twee verschillende menu's door gebruik te maken van de kennisbank. Het resultaat van de afleiding met behulp van* OTTER *op basis van de kennisbank is een menu dat aan alle bovengenoemde eisen voldoet, dat wil zeggen een specifieke invulling is van het Menu-predikaat.*

Hoofdstuk 2

Practicum-handleiding PROLOG

Het PROLOG-practicum stelt u in de gelegenheid in korte tijd ervaring op te doen met de programmeertaal PROLOG. Er wordt bij het practicum gebruik gemaakt van de SWI-PROLOG interpretator. SWI-PROLOG is een van C-PROLOG afgeleide interpretator die vooral in de onderzoekswereld gebruikt wordt. De C-PROLOG-versie van PROLOG wordt momenteel min of meer als de standaard taaldefinitie beschouwd.

In Paragraaf 2.1 wordt de positie van PROLOG binnen de kunstmatige intelligentie en programmeertalen in algemene termen beschreven. In Paragraaf 2.2 is een aantal PROLOG-oefeningen opgenomen. Deze oefeningen hebben tot doel u in zeer korte tijd ervaring met de belangrijkste aspecten van PROLOG bij te brengen. De oefeningen geven u ook een overzicht van diverse eenvoudige toepassingen van PROLOG. Op twee na zijn alle oefeningen aangegeven met het symbool '▶'. Eén oefening (**P.1**) dient u ter goedkeuring aan de practicumassistentie voor te leggen. Deze oefening saangegeven met het symbool '★'. Als boek voor het leren programmeren in PROLOG wordt gebruik gemaakt van *I. Bratko, PROLOG Programming for Artificial Intelligence, 3rd ed., Addison-Wesley, 2000.*

In Paragraaf 2.3 is een aantal opgaven (**P.2**) opgenomen, waaruit, *in overleg met de practicumassistentie*, één gekozen mag worden ter vervanging van de K-opgaven. Deze opgave heeft tot doel u programmeerervaring met PROLOG te laten opdoen. De uitgewerkte opgave dient u ter beoordeling bij de practicumassistentie in te leveren.

2.1 Intelligente systemen en programmeertalen

2.1.1 Symboolverwerking

Bij het ontwikkelen van een AI-systeem, of van programmatuur die hierbij toegepast kan worden, kan in principe gebruik gemaakt worden van elke programmeertaal, ook talen zoals C of Pascal. Toch wordt in het vakgebied der kennissystemen, en in

de kunstmatige intelligentie in het algemeen, meestal gebruik gemaakt van andere programmeertalen, namelijk van de talen PROLOG en LISP. Beide talen hebben een grondslag die niet, in tegenstelling tot C of Pascal, voortkomt uit bestaande computerarchitectuur, maar voortkomt uit wiskundige noties van berekening. Deze talen zijn bij uitstek geschikt voor het manipuleren van complexe datastructuren, en dat zijn de datastructuren in programmatuur voor kennissystemen inderdaad vaak. Aan LISP zal in deze cursus verder geen aandacht worden besteed.

PROLOG wordt in het bijzonder toegepast voor de ontwikkeling van interpretatoren voor bepaalde (formele) talen, bijvoorbeeld programmeertalen, algebraïsche talen, kennisrepresentatie-formalismen, etc. In de syntaxis van veel formele talen spelen onder andere *sleutelwoorden* een belangrijke rol, dat wil zeggen symbolische namen met een vaste betekenis. In Pascal hebben we bijvoorbeeld de sleutelwoorden `program`, `begin`, `end`. Het is relatief eenvoudig PROLOG-programma's te ontwikkelen om dit soort sleutelwoorden in een tekstbestand te herkennen, en er de juiste betekenis aan toe te kennen. PROLOG wordt om deze reden wel een programmeertaal voor *symboolverwerking* genoemd. Met deze naam wordt overigens, ten onrechte, impliciet de indruk gewekt dat een imperatieve programmeertaal, zoals Pascal, niet geschikt zou zijn voor symboolverwerking. Immers, veel van de constructen die in Pascal of C beschikbaar zijn, zijn juist aanwezig om symboolverwerking mogelijk te maken. Imperatieve programmeertalen, zoals Pascal of C, zijn voortgekomen uit het Von Neumann machinemodel (stored program machine). Het programmeren in termen van toestandsveranderingen van registers en geheugenplaatsen is duidelijk in het ontwerp van deze klasse van talen te herkennen (denk aan het assignment statement). Imperatieve programmeertalen zijn minder geschikt voor symboolverwerking omdat het manipuleren van symbolen slechts indirect in deze talen mogelijk gemaakt wordt, namelijk door middel van toestandsverandering, en geen wezenlijk kenmerk van deze talen is.

Anderzijds wordt soms de indruk gewekt dat PROLOG een taal is die slechts geschikt is voor onderzoek op het gebied van de kunstmatige intelligentie. Ook dit is onjuist. Bij PROLOG is in het geheel geen rekening gehouden met een specifieke computerarchitectuur, zodat deze taal wezenlijk algemener is dan de imperatieve programmeertalen. In principe kunnen in PROLOG net zo goed andere dan kunstmatige-intelligentie toepassingen ontwikkeld worden. De toepassingsmogelijkheden van PROLOG nemen, door verbeteringen in de taal en de interpretatoren en compilers, nog steeds toe. Er zijn enkele redenen waarom toch soms de voorkeur gegeven wordt aan een imperatieve programmeertaal; zo levert de implementatie van een algoritme in een imperatieve programmeertaal vaak een programma op dat sneller is, en waarvan de executable minder groot is. Uiteraard wordt dit voordeel van imperatieve programmeertalen veroorzaakt door het feit dat deze qua berekeningsmodel zo nauw aansluiten bij de conventionele Von Neumann machine. Voor het ontwikkelen van prototypen van systemen zullen de laatstgenoemde aspecten echter niet erg belangrijk zijn.

2.1.2 Algemene kenmerken van PROLOG

PROLOG is geschikt voor de ontwikkeling van programma's voor geavanceerde toepassingen om de volgende redenen:

- *Eén enkele datastructuur als uitgangspunt van de taal.* In PROLOG is de *term* de datastructuur die gebruikt kan worden voor de implementatie van elke andere datastructuur (arrays, lijsten, bomen, records, queue's, etc.). Termen in PROLOG kunnen op eenvoudige en elegante wijze worden gemanipuleerd. Slechts pointer-variabelen in imperatieve programmeertalen hebben een vergelijkbare flexibiliteit. Pointers zijn echter berucht omdat zij aanleiding kunnen geven tot moeilijk te begrijpen programma's en bugs. Bovendien is de manipulatie van pointers niet erg eenvoudig.

- *Eenvoudige syntaxis.* De syntaxis van PROLOG is aanzienlijk eenvoudiger dan die van de imperatieve programmeertalen. Een PROLOG-programma is syntactisch gezien een rij termen. Zo correspondeert het PROLOG-programma

  ```
  p(X) :- q(X).
  ```

  met de term `:-(p(X),q(X))`. Terwijl men voor de beschrijving van de syntaxis van C of Pascal vele pagina's nodig heeft, kan de syntaxis van PROLOG in één regel beschreven worden. Merk tevens op dat de syntaxis van een PROLOG-programma gelijk is aan de syntaxis van gegevens!

- *Programma – data equivalentie.* Omdat in PROLOG programma's en gegevens voldoen aan dezelfde syntaxis, is het mogelijk gegevens als programma's te laten interpreteren en programma's als gegevens te beschouwen. Dit is een eigenschap die bijzonder handig is bij de ontwikkeling van interpretatoren voor formele talen.

- *Zwakke typering.* Variabelen in PROLOG hoeven niet expliciet gedeclareerd te worden. Terwijl van elk object in een C- of Pascal-programma bekend moet zijn van welk type het is (bijvoorbeeld `integer`, `real`, `char`), wordt het type van een variabele in PROLOG pas relevant bij het uitvoeren van een operatie op de betreffende variabele (zo mogen in PROLOG twee karakters niet vermenigvuldigd worden, hetgeen men pas merkt als de bewerking wordt uitgevoerd). Deze zogenaamde zwakke typering van programma-objecten heeft als voordeel dat allerlei beslissingen ten aanzien van objecten in een programma kunnen worden uitgesteld, hetgeen de taak van de programmeur verlicht. Men moet zich echter realiseren dat hierdoor bepaalde programma-bugs niet meer vóór de uitvoering van het programma kunnen worden gedetecteerd. Ervaren programmeurs hebben echter niet veel behoefte aan dit soort eenvoudige middelen voor het opsporen van bugs.

- *Interactieve programmeeromgeving.* Gewoonlijk is het noodzakelijk in C of Pascal een vrijwel volledig programma te ontwikkelen, alvorens men enige indruk

heeft of de implementatie van het programma aan de specificatie voldoet. Bij de ontwikkeling van een programma in PROLOG worden meestal slechts kleine delen van een programma ontwikkeld en daarna direct uitgeprobeerd met behulp van de interpretator. Telkens behoeven slechts die delen van het volledige programma opnieuw te worden geïnterpreteerd die zijn veranderd. Het is hierbij niet noodzakelijk om een nieuwe executable te maken, zoals bij imperatieve talen met de mogelijkheid van separate compilatie wel noodzakelijk is. Ook kan een expressie direct door de programmeur worden ingetikt, om bijvoorbeeld inzicht te verkrijgen in een bepaalde operatie. In C of Pascal moet eerst een volledig programma worden ontwikkeld – waarin de betreffende expressie opgenomen is – het programma moet vervolgens worden gecompileerd, waarna het pas kan worden uitgevoerd.

- *Uitbreidbaarheid.* Een PROLOG-systeem biedt de mogelijkheid de taal PROLOG zelf uit te breiden, of te wijzigen. In principe is het mogelijk de syntaxis en semantiek van deze programmertaal zodanig aan te passen dat een volkomen andere programmeertaal verkregen wordt. Men zou bijvoorbeeld een PROLOG-interpretator kunnen veranderen in een Pascal-interpretator door toevoeging van een programma. Het is niet mogelijk een Pascal-compiler of interpretator aan te passen (tenzij men de source code van de compiler wijzigt). Uiteraard moet men met dit soort mogelijkheden zeer voorzichtig zijn. Het geeft echter wel aan hoe flexibel PROLOG is.

2.2 PROLOG-oefeningen

Aan de hand van enkele eenvoudige PROLOG-programma's, die in de directory `/vol/practica/IS/Prolog` aanwezig zijn, kunnen de belangrijkste eigenschappen van PROLOG worden bestudeerd. De programma's zijn in de volgende files in deze directory aanwezig:

cursus1	(*lijsten en recursie*)
cursus2	(*backtracking*)
cursus3	(*parameters*)
cursus4	(*ordening van clauses*)
cursus5	(*fail en cut*)
cursus6	(*eindige automaat*)
cursus7	(*sorteren*)
cursus8	(*relationele database*)
cursus9	(*natuurlijke taal verwerking*)

Het PROLOG-systeem kunt u activeren door het intikken van:

```
pl
```

U kunt het PROLOG-systeem weer verlaten door middel van het commando:


```
halt.
```

U kunt een programma in een file, bijvoorbeeld de file **cursus1**, na het opstarten van het PROLOG-systeem inlezen door het volgende in te typen:

```
?- [cursus1].
```

Dit noemt men het *consulteren* van een file; het symbool `?-` is de systeemprompt. Indien een filenaam de extensie `.pl` heeft, kan men deze voor het inlezen van de file weglaten. Deze extensie wordt vaak gebruikt om aan te geven dat de betreffende file een PROLOG-programma bevat. Indien men een file waarvan de naam een andere extensie dan `.pl` heeft wil inlezen, moet de volledige naam tussen apostrofs worden geplaatst. Bijvoorbeeld, door

```
?- ['prog.pro'].
```

in te tikken, wordt de file `prog.pro` ingelezen. Het is ook mogelijk om een programma in het PROLOG-systeem zelf in te typen. Dit kan gebeuren door de file **user** te consulteren.

```
?- [user].
|
```

De file **user** representeert de terminal. De invoer kan worden beëindigd door <*cntrl*>d in te voeren. U kunt echter in de meeste gevallen beter van de vi- of emacs-editor gebruik maken, aangezien deze u aanzienlijk betere edit-mogelijkheden levert. (Voor de emacs-gebruikers is er een PROLOG-mode beschikbaar voor het doelmatig wijzigen en consulteren van een programma.) Bovendien gaat het door u met user ingetikte programma verloren bij het verlaten van het PROLOG-systeem.

In het navolgende zal aangenomen worden dat de betekenissen van de PROLOG standaardpredikaten bekend zijn. Een uitgebreide bespreking kunt u vinden in het Hoofdstuk 'Inleiding tot PROLOG' in de syllabus en het boek van Bratko.

2.2.1 Lijsten – feiten, regels en recursie

De lijst is een veel gebruikte datastructuur in PROLOG. Een *lijst* is een rij elementen

[e_1, e_2,..., e_n]

waarin e_i, $1 \leq i \leq n$, $n \geq 0$, een willekeurige PROLOG-term is (die dus ook opnieuw een lijst mag zijn).

Beschouw de volgende lijst:

```
[a,b,c,d]
```

Deze lijst bestaat uit vier elementen, waarbij elk element een zogenaamd *atoom* is, dat wil zeggen een constante die geen getal is.

De volgende lijst, bestaande uit drie elementen,

`[`e_1`,`e_2`,`e_3`]`

is eigenlijk gedefinieerd als

`[`e_1`|[`e_2`|[`e_3`|[]]]]`

waarin e_1 de *kop* van de lijst is, en `[`e_2`|[`e_3`|[]]]` de *staart* van de lijst is. Dus een lijst is eigenlijk een term, waarvan de functor twee argumenten (kop en staart) heeft. Een lijst met één element `[`e`]` is dus eigenlijk de volgende term: `[`e`|[]]`. De notatie `[`e_1`,`e_2`,`e_3`]` noemt men wel *syntactische suiker*, omdat deze notatie voor de programmeur voor het invoeren van lange lijsten eenvoudiger is. (Syntactische suiker maakt een programma als het ware 'smakelijker' voor de programmeur.) Merk nu op dat de volgende vijf lijsten

```
[a,b,c]
[a|[b,c]]
[a|[b|[c]]]
[a|[b|[c|[]]]]
[a,b|[c]]
```

alle dezelfde lijst voorstellen.

In het hoofdstuk 'Inleiding tot PROLOG' wordt een voorbeeld besproken van een eenvoudig PROLOG-programma dat onderzoekt of een element tot een verzameling van elementen, gerepresenteerd in een lijst, behoort. De volgende clauses zijn gegeven:

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

In het genoemde hoofdstuk is uiteengezet dat een PROLOG-programma bestaat uit:

- *feiten*, zoals clause 1,
- *regels*, zoals clause 2, en
- *vragen*, zoals bijvoorbeeld de vraag

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

Merk op dat het predikaat `element` voorkomt in de conditie (body) en in de conclusie (kop) van regel 2. Dit is daarom een voorbeeld van een *recursieve regel*.

► *De twee bovenbeschreven Horn-clauses zijn opgenomen in de file* **cursus1**. *Consulteer deze file en stel vragen aan het systeem. (Als het* PROLOG*-systeem antwoord heeft gegeven, dient u een* `<return>` *te geven).*

2.2.2 Unificatie en matching

Een belangrijk, aan de PROLOG-interpretator ten grondslag liggend, mechanisme is het *unificeren* van termen. Unificatie is een techniek waarmee twee termen syntactisch gelijk worden gemaakt door het substitueren van termen voor de variabelen die in de twee oorspronkelijke termen voorkomen. In alle gevallen wordt de meest algemene substitutie (most general unifier) genomen. Een gebruiker kan het unificeren van twee termen forceren met behulp van het infix-predikaat '`=`'. Als bijvoorbeeld

`p(X,a,f(Y,g(b)))` en
`p(f(b),a,f(c,g(b)))`

twee termen zijn, dan leidt de vraag

```
?- p(X,a,f(Y,g(b))) = p(f(b),a,f(c,g(b))).
```

tot de volgende substituties:

```
X = f(b)
Y = c
```

Immers,

- de predikaten in de termen zijn gelijk,
- voor het gelijk maken van de respectievelijke eerste argumenten, moet de term `f(b)` voor de variabele `X` gesubstitueerd worden,
- de respectievelijke tweede argumenten zijn gelijk,
- om de respectievelijke derde argumenten gelijk te maken, moeten de termen `f(Y,g(b))` en `f(c,g(b))` gelijk gemaakt worden. In deze twee termen zijn de functoren en de respectievelijk tweede argumenten gelijk. Als de term `c` voor de variabele `Y` wordt gesubstitueerd, zijn daarmee de eerste argumenten ook gelijk gemaakt.

Als het gelukt is twee termen te unificeren, zegt men dat er een *match* gevonden is.

► *Probeer nu bij de volgende paren termen na te gaan voor welke paren een match kan worden gevonden, en welke substituties bij unificatie van de twee gegeven termen worden uitgevoerd; verifieer het antwoord met behulp van de* PROLOG*-interpretator.*

`p(f(X,g(b)),Y)` en `p(f(h(q),Y),g(b))`
`p(X,a,g(X))` en `p(f(b),a,g(b))`
`[[a]|[b]]` en `[X,Y]`
`p(X,f(X))` en `p(Y,Y)`

Voor het laatste paar termen kan geen match gevonden worden, maar toch wordt dit niet door de PROLOG-interpretator herkend. Eerst wordt door het matching-algoritme voor de eerste parameterpositie een substitutie voor de variabele `X` uitgevoerd:

```
X = Y
```

De substitutie die de respectievelijke tweede argumenten van het predikaat `p` gelijk zouden kunnen maken is:

```
Y = f(X)
```

Hiermee is een cyclische binding tussen de variabelen `X` en `Y` tot stand gebracht: de twee termen zijn derhalve niet unificeerbaar. PROLOG detecteert dit echter niet en raakt in een oneindige lus. Het deel van een unificatie-algoritme dat onderzoekt of bij substitutie een cyclische binding kan ontstaan, heet de *occur-check*. Deze occur-check is niet in de PROLOG-interpretator opgenomen. De variant van het unificatie-algoritme die in PROLOG gebruikt wordt, noemt men, ter onderscheiding van het algoritme dat wel gebruik maakt van de occur-check, *matching*.

2.2.3 Backtracking

Backtracking is het mechanisme dat in PROLOG geboden wordt voor het systematisch zoeken naar mogelijke oplossingen van een in Horn-clauses gespecificeerd probleem. Bepaalde declaratieve specificaties hebben meer dan één oplossing. PROLOG vindt echter telkens slechts één van de mogelijke oplossingen. Indien het, uitgaande van een partiële oplossing, niet mogelijk is een volgend subdoel te bewijzen, dan worden de direct aan dit subdoel voorafgegane substituties ongedaan gemaakt, en worden alternatieve substituties geprobeerd.

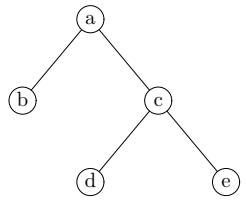
De zoekruimte die met de PROLOG benadering overeenkomt heeft de vorm van een boom; men spreekt wel van een *refutatieboom*. Het onderstaande programma dat in 'Inleiding tot PROLOG' is beschreven, doorloopt een boom.

```
/*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).
```

► *Lees het programma, dat in de file* **cursus2** *is opgenomen, in, en stel de onderstaande vraag aan de* PROLOG*-database. Stel eventueel nog een aantal vragen en verifieer telkens het antwoord.*

```
?- pad(a,d).
```

In Figuur 2.1 is de boom aanschouwelijk gemaakt. Aan de hand van deze figuur kan de executie van het programma gevolgd worden. PROLOG bevat een aantal faciliteiten om de door het PROLOG-systeem uitgevoerde stappen zichtbaar te maken. In de volgende paragraaf worden deze faciliteiten besproken.



Figuur 2.1: Een binaire boom.

2.2.4 Tracing en spying

PROLOG-systemen bieden een aantal faciliteiten voor het 'debuggen' van programma's. Een tweetal technieken zal hier behandeld worden. Bij de volgende oefeningen kunt u telkens van deze technieken gebruik maken om de werking van de PROLOG interpretator op de voet te volgen. Het 'debuggen' van een programma kan plaatsvinden door

- het zichtbaar maken van de executie van een programma, met behulp van het standaard-predikaat `trace`.
- het selectief traceren van clauses met behulp van zogenaamde *spy-points*. Voor dit doel is het predikaat `spy` beschikbaar.

Aan de hand van een voorbeeld zal zichtbaar gemaakt worden hoe het 'debuggen' met de predikaten `trace` en `spy` in de praktijk in zijn werk gaat.

De gebruiker consulteert een programma en zet vervolgens de 'trace' aan door de vraag

```
?- trace.
```

aan het systeem te stellen. Als nu het programma uit de geconsulteerde file wordt uitgevoerd, worden de door het systeem uitgevoerde stappen zichtbaar gemaakt.

Als na het consulteren van de file **cursus2**, waarin het in Paragraaf 2.2.3 beschreven programma is opgenomen, onderstaande vragen worden gesteld

```
| ?- trace.

Yes
| ?- pad(a,e).
```

antwoordt het PROLOG-systeem met

```
   (6) Call: pad(a,e) ?
```

Het vraagteken geeft aan dat een gebruiker een optie kan invoeren; de mogelijke opties kunt u opvragen door '*h*', *help*, in te tikken. Door telkens *<return>* in te tikken, volgt:

```
| ?- pad(a,e).
   Call:  (6) pad(a, e) ? creep
   Call:  (7) tak(a, L83) ? creep
   Exit:  (7) tak(a, b) ? creep
   Call:  (7) pad(b, e) ? creep
   Call:  (8) tak(b, L106) ? creep
   Fail:  (8) tak(b, L106) ? creep
   Redo:  (7) pad(b, e) ? creep
   Fail:  (7) pad(b, e) ? creep
   Redo:  (7) tak(a, L83) ? creep
   Exit:  (7) tak(a, c) ? creep
   Call:  (7) pad(c, e) ? creep
   Call:  (8) tak(c, L95) ? creep
   Exit:  (8) tak(c, d) ? creep
   Call:  (8) pad(d, e) ? creep
   Call:  (9) tak(d, L118) ? creep
   Fail:  (9) tak(d, L118) ? creep
   Redo:  (8) pad(d, e) ? creep
   Fail:  (8) pad(d, e) ? creep
   Redo:  (8) tak(c, L95) ? creep
   Exit:  (8) tak(c, e) ? creep
   Call:  (8) pad(e, e) ? creep
   Exit:  (8) pad(e, e) ? creep
   Exit:  (7) pad(c, e) ? creep
   Exit:  (6) pad(a, e) ? creep

Yes
| ?-
```

Deze informatie is als volgt opgebouwd: het getal tussen haakjes geeft het recursieniveau van de aanroep (met 6 als startniveau); `Call` geeft een te bewijzen subdoel aan; `Exit` geeft aan dat het gespecificeerde subdoel bewezen is en `Fail` geeft aan dat het subdoel niet bewezen kan worden. Indien in de executie van het programma via backtracking naar een al eerder beschouwd subdoel gekeken wordt, dan wordt dit met `Redo` aangegeven. Merk op dat de door de gebruiker gespecificeerde variabelen intern een andere naam krijgen: een `L` gevolgd door het interne volgnummer van de betreffende variabele.

► *Het bovenvermelde programma is aanwezig in de file* **cursus2**. *Probeer de trace-faciliteit van* PROLOG *uit.*

Selectieve informatie van de executie kan worden verkregen door gebruik te maken van spy-points. Een spy-point kan als volgt worden geplaatst:

`spy(<`*`specificatie`*`>)`

Een `<`*`specificatie`*`>` is een naam van een predikaat, of een naam voorzien van een ariteit-specificatie (het aantal argumenten van de predikaatnaam). Bijvoorbeeld:

```
spy(pad).
spy(tak/2).
```

Een spy-point kan worden verwijderd met het predikaat **nospy**:

```
nospy(pad).
nospy(tak/2).
```

De structuur van de informatie die door de PROLOG-interpretator wordt gegenereerd in het geval van spy-points is vrijwel gelijk aan de toepassing van het predikaat **trace**.

► *Probeer het plaatsen, gebruiken en verwijderen van spy-points aan de hand van het boom-algoritme.*

2.2.5 Variatie van input-outputparameters

Bij een Horn-clause kan een argument fungeren als een input of een output parameter afhankelijk van de context waarin de clause gebruikt wordt. Een programma kan hierdoor voor verschillende doeleinden gebruikt worden. Beschouw het onderstaande programma dat in de file **cursus3** is opgenomen:

```
append([],L,L).
append([X|L1],L2,[X|L3]) :-
     append(L1,L2,L3).
```

Dit programma kan gebruikt worden om twee lijsten te concateneren (achter elkaar te plaatsen), maar ook om verschillende mogelijke opsplitsingen van een lijst te genereren.

► *Ga dit na door na het consulteren van de file* **cursus3**, *de volgende vragen te stellen aan de* PROLOG*-interpretator (bij de eerste vraag forceert u met '*`;`*' backtracking en genereert u alternatieve oplossingen).*

```
?- append(X,Y,[a,b,c,d,e]).

?- append([a,b],[c,d],X).
```

Het feit dat eenzelfde programma voor verschillende toepassingen kan worden gebruikt, is een belangrijke eigenschap van veel PROLOG-programma's. Een ander voorbeeld is een programma dat zowel in staat is symbolisch te differentiëren als te integreren (zie het hoofdstuk 'Inleiding tot PROLOG' in de syllabus).

2.2.6 De ordening van clauses en condities

Volgens de principes van programmeren in de logica resulteert de beschrijving van een probleem in een verzameling Horn clauses, waarin de onderlinge volgorde dus niet van belang is. Als gevolg van de zoekstrategie die door de PROLOG interpretator gebruikt wordt, blijkt de volgorde waarin de clauses en de condities binnen de clauses in de PROLOG database worden gespecificeerd van essentieel belang te zijn. Het is zelfs mogelijk dat een in Horn clauses gespecificeerd probleem wel een oplossing heeft, maar dat die oplossing door de volgorde van de clauses, of door de volgorde van de condities, toch niet door de PROLOG interpretator wordt gevonden. In de volgende programma's, die opgenomen zijn in de file **cursus4**, wordt een en ander verduidelijkt.

De volgende feiten betreffende de relatie `ouder(X,Y)`, waarmee aangeven wordt dat X een ouder is van Y, zijn in de PROLOG-database opgeslagen na het consulteren van de file **cursus4**.

```
ouder(pam,bob).
ouder(tom,bob).
ouder(tom,liz).
ouder(bob,ann).
ouder(bob,pat).
ouder(pat,jim).
```

► *Stel een aantal vragen aan het programma. Hoe vraagt u bijvoorbeeld hoe de kinderen van Tom heten? En wie de ouders van Bob zijn?*

In de volgende twee clauses is recursief gedefinieerd dat `X` een voorouder is van `Z`, `voorouder1(X,Z)`, als `X` een ouder is van `Z` òf als er een persoon `Y` is, zodanig dat `X` een ouder is van `Y` en `Y` een voorouder is van `Z`.

```
/*1*/ voorouder1(X,Z) :-
          ouder(X,Z).
/*2*/ voorouder1(X,Z) :-
          ouder(X,Y),
          voorouder1(Y,Z).
```

► *Stel een aantal vragen aan het programma en onderzoek met behulp van de 'debug'-faciliteiten hoe de PROLOG database doorzocht wordt.*

Er zijn nog drie andere declaratief correcte specificaties van het probleem mogelijk, die alleen in de ordening van de clauses en condities verschillen. In de eerste plaats kan de volgorde van de twee clauses worden omgedraaid. Het volgende programma is hiervan het gevolg:

```
/*3*/ voorouder2(X,Z) :-
          ouder(X,Y),
          voorouder2(Y,Z).
/*4*/ voorouder2(X,Z) :-
          ouder(X,Z).
```


► *Stel vragen aan het programma en vergelijk het effect hiervan met het eerste programma.*

Ook is het mogelijk om de twee condities in de tweede clause van het oorspronkelijke programma te verwisselen. Dan wordt het volgende programma verkregen.

```
/*5*/ voorouder3(X,Z) :-
          ouder(X,Z).
/*6*/ voorouder3(X,Z) :-
          voorouder3(X,Y),
          ouder(Y,Z).
```

► *Onder welke omstandigheid zal dit programma niet in staat zijn een antwoord te vinden op een vraag?*

Tenslotte kunnen nog zowel clause 1 en 2 als de beide condities in clause 2 worden omgewisseld. Dan wordt het volgende resultaat verkregen.

```
/*7*/ voorouder4(X,Z) :-
          voorouder4(X,Y),
          ouder(Y,Z).
/*8*/ voorouder4(X,Z) :-
          ouder(X,Z).
```

► *Ga na welke vragen niet door dit programma beantwoord kunnen worden en probeer te verklaren waardoor dit veroorzaakt wordt.*

2.2.7 Fail en cut

Er bestaat in PROLOG een aantal predikaten die u in staat stellen de besturing die door de PROLOG interpretator geboden wordt, expliciet te beïnvloeden. Twee veel gebruikte methoden zijn de *fail* en de *cut*. Als het predikaat `fail` als conditie wordt gespecificeerd, is dat een conditie waaraan nooit voldaan kan worden. `fail` wordt meestal gebruikt om backtracking te forceren. Beschouw het onderstaande programma dat in de file **cursus5** is opgenomen:

```
element1(X,[X|_]).
element1(X,[_|Y]) :-
     element1(X,Y).

alle_elementen1(L) :-
```

```
     element1(X,L),
     write(X),
     nl,
     fail.
alle_elementen1(_).
```

Met behulp van de clauses `alle_elementen1` worden de elementen van een lijst `L` één voor één afgedrukt. Hierbij wordt gebruik gemaakt van de clauses `element1` die u al eerder heeft bekeken.

► *Experimenteert u eens met* `alle_elementen1`*. Waarom is de tweede clause van* `alle_elementen1` *aanwezig ?*

Aan de hand van dit programma zal tevens de tweede methode voor de beïnvloeding van de besturing van de PROLOG interpretator worden besproken: de *cut*. Met behulp van de cut wordt backtracking juist voorkomen. In Paragraaf 7.3.11 van 'Inleiding tot PROLOG' wordt de cut uitvoerig besproken. Beschouw het gebruik van de cut in

```
element2(X,[X|_]) :- !.
element2(X,[_|Y]) :-
     element2(X,Y).

alle_elementen2(L) :-
     element2(X,L),
     write(X),
     nl,
     fail.
alle_elementen2(_).
```

► *Onderzoek wat het verschil is met* `alle_elementen1` *en probeer te verklaren waardoor dit verschil veroorzaakt wordt.*

2.2.8 Een eindige automaat

Een eindige automaat is een eindig wiskundig object dat wordt gebruikt om te bepalen of een gegeven invoer-symboolrij een bepaalde vorm heeft. Automaten worden bijvoorbeeld toegepast bij het bouwen van compilers. Een automaat verkeert telkens in een bepaalde toestand. Bij een eindige automaat is een verzameling toestanden gedefinieerd, waarvan één de *begintoestand* is en een aantal toestanden zogenaamde *eindtoestanden* zijn. Door telkens een symbool van de invoer-symboolrij te bekijken, wordt de automaat in een nieuwe toestand gebracht. Hoe dat gebeurt, is vastgelegd in een bij de automaat behorende *transitiefunctie*, ook wel *overgangsfunctie* genoemd. Als na het verwerken van een gegeven invoer-symboolrij de eindige automaat in een eindtoestand verkeert, zegt men dat de invoer-symboolrij *geaccepteerd* is, anders noemt men de symboolrij *verworpen*.

Bij de automaat die hier beschouwd wordt, is de volgende verzameling toestanden gedefinieerd:

$$S = \{s_1, s_2, s_3, s_4\}$$

Hierin is s_1 de begintoestand en s_3 de eindtoestand. De transitiefunctie is gespecificeerd in de onderstaande verzameling toestandsovergangen of transities:

$s_1 \rightarrow \mathrm{a}s_1$
$s_1 \rightarrow \mathrm{a}s_2$
$s_1 \rightarrow \mathrm{b}s_1$
$s_2 \rightarrow \mathrm{b}s_3$
$s_3 \rightarrow \mathrm{b}s_4$
$s_4 \rightarrow s_3$

De laatste toestandsovergang is een zogenaamde stille overgang: er wordt geen gebruik gemaakt van een symbool van de invoer-symboolrij. De toestandsovergangen zijn eenvoudig in PROLOG vast te leggen: een predikaat `transitie(s1,c,s2)` geeft de overgang aan van een toestand s_1 naar s_2 onder verwerking van het symbool 'c'. In `stil(s1,s2)` wordt een stille toestandsovergang gerepresenteerd. Dat s_3 de eindtoestand is, wordt aangegeven met het predikaat `eind`. De bovenstaande transitiefunctie kan dus als volgt worden weergegeven:

```
eind(s3).

transitie(s1,a,s1).
transitie(s1,a,s2).
transitie(s1,b,s1).
transitie(s2,b,s3).
transitie(s3,b,s4).

stil(s4,s3).
```

Onderstaande regels onderzoeken of een string al dan niet moet worden geaccepteerd.

```
accepteert(S,[]) :-
    eind(S).
accepteert(S,[X|Rest]) :-
    transitie(S,X,S1),
    accepteert(S1,Rest).
accepteert(S,String) :-
    stil(S,S1),
    accepteert(S1,String).
```

► *Ga met behulp van dit programma na welke invoer-symboolrijen van 3 en 4 symbolen door de automaat worden geaccepteerd. Het programma is opgenomen in de file* **cursus6**. *U dient* `accepteert` *als eerste argument de begintoestand en als tweede*

argument een invoer-symboolrij, gerepresenteerd als een lijst van symbolen, mee te geven. Geef een korte beschrijving van de strings die volgens u worden geaccepteerd door het programma, en geef een verklaring in termen van de executie van de clauses in het programma. Lever het antwoord op deze vraag ter beoordeling bij de practicumassistentie in.

2.2.9 Sorteren

Voor het sorteren van een eindige rij getallen bestaan verschillende technieken. Eén van die technieken is de *bubble sort*. Met de bubble sort worden telkens twee opeenvolgende elementen uit de rij getallen bekeken. Als de twee elementen in de juiste volgorde staan, wordt het volgende paar bekeken; in het andere geval worden de twee elementen van plaats verwisseld, alvorens het volgende paar elementen te bekijken. De rij wordt net zolang doorlopen totdat er geen elementen meer zijn verwisseld. Deze bubble sort laat zich eenvoudig in PROLOG formuleren. Met de onderstaande clauses wordt een rij elementen, gerepresenteerd in een lijst, in oplopende volgorde gesorteerd.

```
bubsort(L,S) :-
     append(X,[A,B|Y],L),
     B < A,
     append(X,[B,A|Y],M),
     !,
     bubsort(M,S).
bubsort(L,L).

append([],L,L).
append([H|T],L,[H|V]) :-
     append(T,L,V).
```

► *Onderzoek hoe het programma, dat in de file* **cursus7** *is opgenomen, werkt.*

2.2.10 Een relationele database

Clauses in PROLOG kunnen gebruikt worden om *relaties*, zoals die gedefinieerd zijn in het relationele datamodel, vast te leggen (b.v. `salaris(schaal,bedrag)`). Als in een relatie de argumenten zijn ingevuld met constanten (atomen of getallen) spreekt men van *tupels*. De argumenten worden *attributen* van de relatie genoemd. Vragen aan de database worden *queries* genoemd.

In de file **cursus8** is een voorbeeld van een database opgenomen met de volgende relaties:

- werknemer(naam,afdelingsnummer,schaal).
- afdeling(afdelingsnummer,naam_afdeling).
- salaris(schaal,bedrag).

De database bevat de onderstaande tupels:

De file **cursus8** bevat de volgende relaties:

```
werknemer(jansen,1,5).
werknemer(pietersen,2,3).
werknemer(dirksen,1,2).
werknemer(klaassen,4,1).
werknemer(petersen,5,8).
werknemer(evertsen,1,7).
werknemer(nelissen,1,9).

afdeling(1,directie).
afdeling(2,personeels_dienst).
afdeling(3,productie).
afdeling(4,technische_dienst).
afdeling(5,administratie).

salaris(1,1000).
salaris(2,1500).
salaris(3,2000).
salaris(4,2500).
salaris(5,3000).
salaris(6,3500).
salaris(7,4000).
salaris(8,4500).
salaris(9,5000).
```

Gegeven enige relaties, dan kan men hierop een aantal bewerkingen definiëren die nieuwe relaties produceren. Veel gebruikte bewerkingen zijn de *selectie*, de *projectie* en de *join*, ontleend aan de relationele algebra. De selectie, projectie en join worden in het onderstaande in PROLOG uitgewerkt. Als aanknopingspunt voor degenen die bekend zijn met query talen, worden de queries in de voorbeelden in SQL-achtige notatie weergegeven; SQL is een bekende query taal voor relationele database systemen. Gewoonlijk specificeert u de queries in bijvoorbeeld een SQL-achtige taal, waarna deze door het database systeem worden omgezet, in dit geval in PROLOG. Deze fase is hier weggelaten om educatieve doeleinden: de queries worden direct in PROLOG beschreven.

Bij *selectie* vraagt men alle attributen van die tupels in een relatie die aan bepaalde condities voldoen. Bijvoorbeeld de query: 'Geef alle werknemers uit afdeling nummer 1 die in een loonschaal hoger dan 2 zijn ingeschaald.' In een SQL-achtige notatie zou deze query als volgt gedefinieerd worden:

```
SELECT * FROM werknemer WHERE afdelingsnummer = 1 AND
         schaal > 2.
```

Als de onderstaande PROLOG vraag wordt gesteld, dan wordt één tupel geselecteerd dat aan de gestelde condities voldoet:

```
?- werknemer(Naam,Afdelings_N,Schaal),
   Afdelings_N = 1,Schaal > 2.
```

► *Consulteer de file cursus8 en tik bovenstaande* PROLOG *vraag in.*

Als antwoord zult u krijgen:

```
Naam = jansen
Afdelings_N = 1
Schaal = 5.
```

Met behulp van de ';' (het $\vee$ connectief) kunt u backtracking forceren en de andere werknemers die aan de twee gestelde condities voldoen, genereren. Merk op dat hiermee de selectie nog niet is geïmplementeerd; er wordt immers telkens maar één tupel uit de database bekeken. In onderstaande clauses is de selectie gedefinieerd, waarin met behulp van `fail` alle tupels die aan de gestelde condities voldoen, worden geselecteerd:

```
selectie(X,Y) :-
     call(X),
     call(Y),
     write(X),
     nl,
     fail.
selectie(_,_).
```

Het predikaat `call` geeft aanleiding tot het genereren van een doel dat overeenkomt met de binding van het argument. In het eerste argument van `selectie` dient men aan te geven uit welke relatie tupels moeten worden geselecteerd; in het tweede argument worden de condities waaraan de tupels dienen te voldoen, gespecificeerd. Alle werknemers uit afdeling nummer 1 die in een loonschaal hoger dan 2 zijn ingeschaald, worden verkregen met behulp van:

```
?- selectie(werknemer(Naam, Afdelings_N, Schaal),
   (Afdelings_N = 1,Schaal > 2)).
```

► *Hoe verkrijgt u alle salarisschalen waarbinnen men tussen de 3.000 en 4.500 gulden per maand verdient ? Hoe verkrijgt u alle werknemers die of in salarisschaal nummer 1 ingeschaald zijn of op afdeling nummer 1 werken ?*

Bij een *projectie* selecteert men een beperkt aantal attributen uit de tupels in een relatie. Bijvoorbeeld de query: 'Geef van alle werknemers de naam en de schaal'. In SQL-achtige notatie:

```
SELECT naam,schaal FROM werknemer.
```


De onderstaande PROLOG vraag heeft weer de selectie van één tupel tot gevolg:

```
?- werknemer(Naam,_,Schaal).
```

Met ';' kunnen weer de resterende tupels bekeken worden.

► *Tik bovenstaande* PROLOG *vraag in. Waarom is hier gebruik gemaakt van de anonieme variabele _?*

In onderstaande clauses worden alle tupels uit de database bekeken. Opgemerkt zij, dat hiermee nog niet de projectie is geïmplementeerd: bij projectie mogen dubbele tupels van de nieuwe relatie slechts eenmaal opgenomen worden.

```
projectie(X,Y) :-
     call(X),
     write(Y),
     nl,
     fail.
projectie(_,_).
```

In het eerste argument geeft men aan welke relatie beschouwd dient te worden; in het tweede argument worden de te projecteren attributen gespecificeerd. De naam en de schaal van alle werknemers worden verkregen met

```
?- projectie(werknemer(Naam, Afdelings_N, Schaal),
   (Naam, Schaal)).
```

► *Experimenteert u eens met deze projectie.*

Bij een gecombineerde selectie en projectie specificeert men zowel de zoekcondities als de attributen die men wil hebben. Bijvoorbeeld de query: 'Geef naam en schaal van alle werknemers uit afdeling 1 in een salarisschaal hoger dan schaal 2'. In SQL-achtige notatie:

```
SELECT naam,schaal FROM werknemer WHERE afdelingsnummer = 1
       AND schaal > 2.
```

► *Definieert u nu zelf een predikaat* `sel_pro` *voor deze gecombineerde selectie en projectie.*

Met behulp van de *join* worden nieuwe tupels gevormd door tupels uit verschillende relaties aan elkaar te koppelen, op grond van een *join conditie*. Veronderstel dat men een lijst van werknemers wil, waarin bij elke werknemer het salaris dat hij verdient, is gespecificeerd. Hiervoor is informatie uit de relaties `werknemer` en `salaris` nodig: bij elke werknemer moet op grond van de salarisschaal waarin hij is ingedeeld, in de relatie `salaris` het met die schaal corresponderende salaris opgezocht worden. De join conditie is hierin derhalve de gelijkheid van het attribuut `schaal` in de relaties `werknemer` en `salaris`. In SQL-achtige notatie:

```
JOIN werknemer WITH salaris WHERE
     werknemer.schaal = salaris.schaal
```

Met behulp van de onderstaande clauses worden de juiste tupels bij elkaar gezocht:

```
join(X,Y,Z) :-
     call(X),
     call(Y),
     call(Z),
     write(X),
     write(Y),
     nl,
     fail.
join(_,_,_).
```

Hierin specificeren het eerste en het tweede argument de relaties waarover de join plaatsvindt, en geeft het derde argument de join conditie aan, zodat

```
?- join(werknemer(Naam,Afdelings_N,Schaal1),
   salaris(Schaal2,Bedrag),
   (Schaal1 = Schaal2)).
```

het gewenste resultaat oplevert.

★ (**P.1**) *Formuleer in* PROLOG *een join tussen de relaties* `werknemer` *en* `afdeling`. *Lever uw antwoord ter beoordeling in.*

2.2.11 Natuurlijke taal verwerking

PROLOG heeft in de natuurlijke taal verwerking reeds vanaf het ontstaan een belangrijke rol gespeeld. Dit geldt vooral voor de syntactische aspecten van natuurlijke taal: herschrijfregels die bij de syntactische analyse van taal gebruikt worden, zijn op elegante wijze in Horn-clauses om te zetten. Het in 'Inleiding tot PROLOG' in Paragraaf 7.3.1 gegeven programma, is in de file **cursus9** opgeslagen.

```
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).
```


► *Consulteer dit programma en maak zelf een eenvoudig woordenboek met behulp van een editor. Ga na welke zinnen door het programma geaccepteerd worden.*

► *Wat is de functie van het tweede argument van de hulppredikaten* `naamwoordelijk_gezegde` *en* `werkwoordelijk_gezegde`*? Is dit programma ook in staat om zinnen te genereren die voldoen aan de grammatica die correspondeert met het bovengegeven* PROLOG*-programma? Zie ook 'Inleiding tot* PROLOG *' voor de herschrijfregels van de grammatica. Wijzig nu het programma zodanig dat het gebruik maakt van de append-clause (file:* **cursus3***) in plaats van de lijst-notatie in de head van de clauses om een zin op te splitsen. (Opmerking: deze wijziging maakt het tweede argument van bovengenoemde predikaten overbodig.)*

2.3 Opgave P.2: programmeren in PROLOG

Deze opgave is bedoeld om u kennis te laten maken met de implementatie van probleem-oplossende programma's in PROLOG. Voor de implementatietechnieken die u bij het maken van deze opgave kunt gebruiken, wordt u verwezen naar het boek *I. Bratko, PROLOG Programming for Artificial Intelligence, 3rd ed., Addison-Wesley, 2000.* In deze paragraaf wordt een aantal verschillende opgaven besproken. U dient, *in overleg met de practicumassistentie*, één opgave uit te kiezen.

2.3.1 Obstakel-vermijdende robot

Beschouw een vierkant vlak dat verdeeld is in $n \times n$ velden, $n \geq 1$. Ergens op dit vlak bevindt zich een robot, die zich met het geringst mogelijk aantal bewegingen wil verplaatsen naar een ander veld op het vlak. Op sommige velden van het vlak bevinden zich echter obstakels, waardoor de robot deze velden niet kan betreden.

Beschreven in termen van probleemoplossen probeert de robot van een begintoestand (het startveld van de robot) in een doeltoestand (het eindveld van de robot) te komen, zodanig dat het aantal velden dat betreden wordt minimaal is. Een elementaire beweging van de robot is een verplaatsing naar een aangrenzend veld, in één van de volgende richtingen:

noord, oost, zuid of west

Indien de robot zich aan de rand van het vlak bevindt, zijn sommige bewegingen niet mogelijk.

★ *Ontwikkel een* PROLOG*-programma voor het beschreven probleem, dat aan de volgende eisen van invoer en uitvoer voldoet:*

- *De invoer van het programma is een lijst met als eerste element het aantal rijen of kolommen in het vlak. Het tweede element is een lijst met coördinaten van de obstakels. Het derde element geeft de coördinaten van het veld waar de robot zich bevindt, en het vierde element geeft de coördinaten van het eindveld. Een voorbeeld van invoer is:*

O	O			O						O
O										
				O	O			O		
O		O		O				O	O	
		O	R	P	O			O		
	O			P	P	O	O	G		
	O	O			P	P	P	P		
	O	O			O					
	O	O		O	O					
				O	O				O	
				O						

Figuur 2.2: Een 11×11 matrix met een route.

`[11, [(2,2), (2,3), (3,3)], (3,2), (1,4)]`

waarmee een 11×11 vlak met drie obstakels wordt gespecificeerd, waarbij de robot zich bevindt op het veld op de derde rij en tweede kolom; het eindveld bevindt zich op de eerste rij en vierde kolom.

- *De uitvoer is een matrix, waarop de startpositie van de robot wordt aangegeven door middel van R, elk obstakel door middel van O, het eindveld door middel van G, en elk veld op de route van startveld naar eindveld met behulp van P. In Figuur 2.2 is een voorbeeld van mogelijke uitvoer weergegeven.*

2.3.2 Kwartjes-en-dubbeltjes puzzel

Beschouw de volgende rangschikking van kwartjes en dubbeltjes:

KDKDK

waarin 'K' staat voor een kwartje en 'D' staat voor een dubbeltje. Deze rangschikking kan in een aantal stappen worden getransformeerd in de volgende rangschikking:

KKKDD

waarbij de toegepaste transformatiestappen het verplaatsen van een KD- of DK-paar zijn. Beschreven in termen van probleemoplossen is een begintoestand KDKDK hierdoor getransformeerd in de eindtoestand KKKDD. Een KD- of DK-paar mag slechts verplaatst worden naar een positie waarbij tenminste één van de munten van het paar links of rechts aansluit bij één van de andere munten. Een paar mag dus ook verplaatst worden naar een ruimte tussen de munten, mits deze ruimte tenminste twee posities groot is. Aangenomen wordt dat een dubbeltje even groot is als een kwartje.

Beschouw het volgende voorbeeld van mogelijke opeenvolgende rangschikkingen:

$$\begin{aligned} \underline{\text{KD}}\text{KDK} &\Rightarrow \text{K}\underline{\text{DK}}\textit{KD} \\ &\Rightarrow \underline{\textit{DK}}\text{K..KD} \\ &\Rightarrow \text{K..}\underline{\text{KD}}\textit{DK} \\ &\Rightarrow \text{K}\textit{KD}\text{..DK} \end{aligned}$$

de onderstreepte paren zijn paren die verplaatst worden; cursief is aangegeven waar het betreffende paar in de rij munten terecht is gekomen. Een '.' geeft een lege positie in de rij munten weer. De volgende verplaatsing

K....KD$\underline{\text{KD}}$ $\Rightarrow$ K.KD.KD

is niet toegestaan; immers, geen van de munten van het paar sluit na verplaatsing aan bij een van de andere munten in de rij.

★ *Ontwikkel een* PROLOG*-programma voor het beschreven probleem. In het* PROLOG*-programma dient gebruik te worden gemaakt van een evaluatiefunctie $f(v) = g(v) + h(v)$ die het probleemoplossen heuristisch stuurt (zie Paragraaf 12.1 in het boek van Bratko en Hoofdstuk 2 in de syllabus Kennistechnologie). Geef zelf een invulling voor de functies $g(v)$ en $h(v)$.*

Het PROLOG*-programma dient aan de volgende eisen van invoer en uitvoer te voldoen:*

- *Het programma heeft als invoer een bepaalde rangschikking van drie kwartjes en 2 dubbeltjes als begintoestand en doeltoestand.*
- *Het programma drukt als uitvoer het aantal gegenereerde rangschikkingen en de oplossing, dat wil zeggen, de transformatie van begin- naar doeltoestand, af. Test uw programma met als invoer de begintoestand KDKDK en doeltoestand KKKDD.*

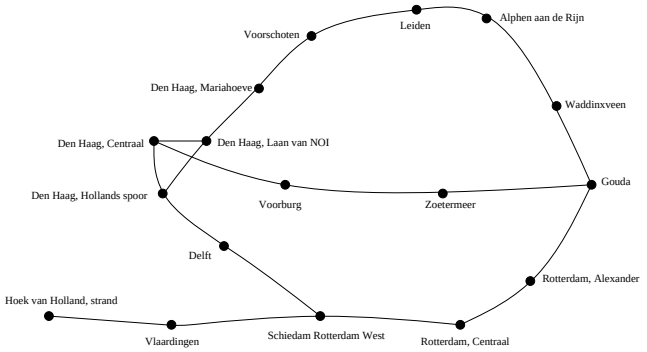
2.3.3 Reisplanner

In Figuur 2.3 is een klein deel van het Nederlandse spoorwegennet weergegeven. De bijbehorende tijden- en afstandentabel is weergegeven in Tabel 2.1. In deze tabel zijn de namen van de stations, verbindingen, vertrektijden en het aantal te rijden kilometers opgenomen. Deze zijn slechts ten dele op de werkelijkheid gebaseerd. De eerste kolom in de tabel geeft de naam van het station van vertrek. In de tweede kolom is het station van bestemming opgenomen, en of de trein naar dit station een intercity (I) of een stoptrein (S) is. De vertrektijden zijn gegeven in minuten na *ieder* heel uur. De aankomsttijden kunnen berekend worden aan de hand van de vertrektijd, de gemiddelde snelheid en de afstand tussen de plaatsen. Aangenomen wordt dat:

1. Voor alle reizen geldt dat de vertrektijd en de aankomsttijd op dezelfde dag vallen;
2. De gemiddelde snelheid van intercitytreinen 60 kilometer per uur is;

Station	Verbindingen	Vertrektijd	Afstand (km.)
Den Haag Centraal	I Den Haag Hollands spoor	45, 59,15,29	3
	I Gouda	4,20	28
	S Voorburg	9,39	4
	S Den Haag Laan van NOI	2,12,52,57,22	3
	S Den Haag Hollands spoor	10,46	3
Den Haag Hollands spoor	I Rotterdam Centraal	3,12,27,57	23
	I Den Haag Centraal	45,29	3
	I Leiden	22,52	15
	S Delft	14,44,59	9
	S Den Haag Laan van NOI	9,19,39,49	3
Delft	S Den Haag Hollands spoor	23,53	9
	S Schiedam Rotterdam West	52,22	10
Schiedam Rotterdam West	S Rotterdam Centraal	2,16,32,46	4
	S Delft	0,30	10
	S Vlaardingen	14,26,45,56	6
Rotterdam Centraal	I Den Haag Hollands spoor	3,16,33,46	24
	I Gouda	10,24,54	24
	S Schiedam Rotterdam West	10,24,40,54	4
	S Rotterdam Alexander	10,24,40,54	10
Rotterdam Alexander	S Rotterdam Centraal	11,25,41,54	10
	S Gouda	3,19,33,49	14
Vlaardingen	S Schiedam Rotterdam West	8,23,38,53	6
	S Hoek van Holland strand	21,51	18
Hoek van Holland strand	S Vlaardingen	19,49	18
Gouda	I Rotterdam Centraal	13,29,42,59	24
	I Leiden	11,42	33
	I Den Haag Centraal	9,52	28
	S Rotterdam Alexander	13,30,59	14
	S Zoetermeer	0,30	16
	S Waddinxveen	11,41	8
Zoetermeer	S Gouda	18,48	16
	S Voorburg	11,41	9
Voorburg	S Zoetermeer	18,48	9
	S Den Haag Centraal	22,52	4
Den Haag Laan van NOI	S Den Haag Centraal	9,26,44,56	3
	S Den Haag Mariahoeve	0,30	4
Den Haag Mariahoeve	S Den Haag Laan van NOI	27,57	4
	S Voorschoten	4,34	6
Voorschoten	S Den Haag Mariahoeve	22,52	6
	S Leiden	9,39	5
Leiden	I Den Haag Hollands spoor	0,15,30,45	15
	I Gouda	21,51	33
	S Aphen aan de Rijn	23,53	15
	S Voorschoten	8,47	5
Alphen aan de Rijn	S Leiden	10,40	15
	S Waddinxveen	7,37	10
Waddinxveen	S Gouda	18,48	8
	S Aphen aan de Rijn	0,30	10

Tabel 2.1: Tijden- en afstandentabel van de Nederlandse Spoorwegen.



Figuur 2.3: Spoorwegennet in het westelijke deel van de randstad.

3. De gemiddelde snelheid van stoptreinen 40 kilometer per uur is;
4. Slechts het overstappen van een stoptrein op een sneltrein, of andersom, wordt geteld als één overstap;
5. Bij een overstap (zoals boven aangegeven) moet een standaard overstaptijd van 4 minuten in acht worden genomen.

★ *Ontwikkel een* PROLOG*-programma dat de volgende vragen kan beantwoorden:*

- *Geef de snelste route van plaats A naar plaats B.*
- *Geef de route met het minste aantal malen overstappen van plaats A naar plaats B.*

Het PROLOG*-programma dient aan de volgende eisen van invoer en uitvoer te voldoen:*

- *Het programma vraagt om twee plaatsen en een vertrektijd. De invoer wordt als lijst ingevoerd en is van de vorm:* [plaats_A, plaats_B, vertrektijd]. *De vertrektijd wordt ingevoerd als 'hh:mm'. Bijvoorbeeld:*

  ```
  ['Den Haag Centraal', 'Leiden', 10:00]
  ```

- *De uitvoer bestaat uit twee delen; het eerste deel geeft de snelste verbinding aan tussen de plaatsen* plaats_A *en* plaats_B. *Het tweede deel geeft de verbinding aan met het minste aantal overstappen tussen de plaatsen* plaats_A *en* plaats_B. *In beide gevallen worden alle tussenliggende stations afgedrukt. Wanneer er een overstap plaatsvindt, moet dit gemeld worden. Een voorbeeld van mogelijke uitvoer is:*

```
Snelste verbinding:
Van Hoek van Holland, strand naar Vlaardingen
Van Vlaardingen naar Schiedam Rotterdam West
Van Schiedam Rotterdam West naar Rotterdam Centraal
Overstappen
Van Rotterdam Centraal naar Gouda
```

- *Nadat de uitvoer gegeven is, moet het programma om invoer voor een nieuwe zoekactie vragen.*

2.3.4 Studie-adviseur

Studenten informatica van de Universiteit Utrecht kunnen bij de samenstelling van het vakkenpakket voor het derde en het vierde studiejaar advies vragen aan een studie-adviseur. De studie-adviseur heeft voor het verlenen van advies de volgende studie-informatie nodig:

- De ingangseisen voor elk vak.
- De dagen en tijden waarop de vakken gegeven worden, om te kunnen vaststellen welke vakken niet gelijktijdig gevolgd kunnen worden.
- De zwaarte van een vak in aantal studiepunten; de afstudeeropdracht heeft een zwaarte van 21 studiepunten.
- De aard van het vak: toepassingsgericht of fundamenteel.
- De richting waartoe het vak behoort: de informatietechnologie (IT), programmatuurkunde (PK) of algemeen (A).
- Het totaal aantal benodigde studiepunten (tenminste 84).

Daarnaast moet de studie-adviseur het volgende van de student weten:

- Heeft de student een voorkeur voor toepassingsgerichte of fundamentele vakken?
- Heeft de student een voorkeur voor vakken behorende tot de informatietechnologie of voor programmatuurkunde? Als een student kiest voor één van deze richtingen, moeten tenminste 20 studiepunten met vakken uit de betreffende richting gevuld worden.

We nemen aan dat de student alle vakken van de eerste twee jaar van de studie gehaald heeft, en eventueel al enkele vakken in het derde of vierde jaar gevolgd en gehaald heeft.

★ *Ontwikkel een* PROLOG*-programma dat dienst kan doen als studie-adviseur voor informatica-studenten. Het programma stelt een vakkenpakket samen voor de student na invoer van een collegekaartnummer, de gewenste richting (informatietechnologie*

– IT – of programmatuurkunde – PK) en de belangstelling van de student (toepassingsgericht of fundamenteel).

Het PROLOG*-programma dient daartoe aan de onderstaande eisen van invoer en uitvoer te voldoen:*

- *Het programma verwerkt invoergegevens interactief.*
- *Als uitvoer geeft het* PROLOG*-programma een volgorde waarin geselecteerde vakken gevolgd moeten worden per semester (in een semester is de volgorde irrelevant). Het derde en het vierde jaar worden ingedeeld in de semesters 1, 2, 3 en 4. Een voorbeeld van uitvoer is:*

```
Semester 1: Vakken
            Expertsystemen (IT),
            Object-georienteerde Databases (IT),
            Reason Maintenance Systems (IT),
            Kennislogica (PK)
Semester 2: Vakken
            Probabilistisch Redeneren (IT),
            Architectuur van Gedistribueerde Databases (IT),
            Termherschrijfsystemen (PK)
Semester 3: Vakken
            Informatica en Samenleving (A),
            Niet-monotoon Redeneren (PK),
            Afstudeeropdracht (A).
Semester 4: Afstudeeropdracht (A).
```

- *Nadat uitvoer gegeven is, moet het programma weer om invoer voor een nieuw advies vragen.*

Stel zelf een kleine database met vakgegeven en een database met studentengegevens samen. U kunt zich hierbij laten inspireren door het studieprogramma van het derde en vierde jaar van de opleiding Informatica aan de Universiteit Utrecht (maar kies het aantal studiepunten per vak iets hoger dan in de studiegids).

2.3.5 Roosterprogramma

Elk jaar wordt ten bate van de opleiding informatica een onderwijsrooster samengesteld. Het opstellen van zo'n rooster is niet eenvoudig omdat met verschillende eisen en wensen rekening gehouden moet worden:

- Het onderwijsprogramma, waarin de verschillende vakken en onderwijsvormen (hoorcollege, werkcollege, practicum, seminarium, bibliotheekopdracht) beschreven zijn, alsmede de zwaarte van de vakken in aantallen studiepunten.
- Het om didactische redenen op elkaar laten aansluiten van bepaalde vakken of onderwijsvormen.
- De beschikbaarheid van de docenten – een docent kan niet op hetzelfde tijdstip meer dan één vak verzorgen, of zal op bepaalde dagen of tijden om andere

redenen niet beschikbaar zijn; bovendien hebben docenten een voorkeur voor de spreiding van te geven onderwijs over verschillende dagen.

- Studenten hebben een voorkeur voor onderwijs waarin zo min mogelijk tussenuren voorkomen; bovendien zal tenminste twee uur (bij voorkeur vier uur), en niet meer dan acht uur onderwijs (hoorcolleges, werkcolleges en practicum) gevolgd moeten worden op een bepaalde dag.
- Er moet rekening gehouden worden met verplichte onderwijsvrije perioden.

★ *Ontwikkel een* PROLOG*-programma waarmee een rooster kan worden opgesteld voor het onderwijs in het eerste en tweede jaar van de opleiding informatica. Ga hierbij wat betreft de gegeven vakken uit van de studiegids.*

Het PROLOG*-programma dient daartoe aan de onderstaande eisen van invoer en uitvoer te voldoen:*

- *De invoer van het programma bestaat uit een specificatie van het onderwijsprogramma, eisen en wensen van de docenten (individueel) en de studenten (als groep) en onderwijsvrije dagen.*
- *Als uitvoer geeft het* PROLOG*-programma een rooster voor het volledige collegejaar, met een precieze specificatie van welke vakken (inclusief onderwijsvorm) op welke dag en tijd worden gegeven.*

Probeer het programma te voorzien van zodanige heuristieken dat het rooster dat geproduceerd wordt globaal genomen tenminste zo goed is als het huidige rooster van het eerste en tweede studiejaar. Probeer zo mogelijk met het programma een beter rooster te produceren dan het huidige rooster, en verklaar aan de hand van de gebruikte heuristieken waarom het rooster beter is.

2.3.6 Een interpretator voor een imperatieve programmeertaal

PROLOG wordt in het wetenschappelijk onderzoek veelvuldig gebruikt voor de ontwikkeling van interpretatoren voor andere programmeertalen. Beschouw, bijvoorbeeld, de C-achtige programmeertaal C`--` waarin alle variabelen locale variabelen zijn, die slechts van het type `bool`, `real`, `int` of `char` kunnen zijn. Variabelen behoeven niet gedeclareerd te worden (dus typen van variabelen worden bepaald door de toegekende waarde). In een assignment statement van de vorm '$x = e$', met x een variabele en e een expressie, kan de waarde van een arithmetische, Boolse of `char` expressie worden toegekend aan een variabele. Neem aan dat de meest gebruikelijke operatoren, zoals `+`, `-`, `*` voor `int` en `real` constanten en variabelen en `not`, `and`, `or` voor Boolse expressies, en de relationele operatoren `=`, `<`, etcetera, beschikbaar zijn. Verder zijn naast assignment de volgende statements beschikbaar: het `while`, `for`, `if`, `return`, `read` en het `print` statement. Tevens kan een programma, net als in C, van commentaar worden voorzien. Met behulp van het `read`-statement kan één waarde voor een variabele worden ingelezen van standaard-input; met behulp van het

$x + 0 = x$	$x - 0 = x$
$x - x = 0$	$- - x = x$
$x \cdot 0 = 0$	$x \cdot 1 = x$
$x \cdot x = x^2$	$x \cdot x^{-1} = 1$
$\frac{x}{1} = x$	$\frac{x}{0} = undefined$
$x^1 = x$	$x^0 = 1$

Tabel 2.2: Regels voor vereenvoudiging.

`print`-statement kunnen waarden van een willekeurig aantal constanten of variabelen naar standaard-output worden gezonden. De Boolse functies `eoln` en `eof` geven aan of in de input stream het einde van een regel of het einde van de file bereikt is. Een C`--` programma bestaat uit een aantal functiedefinities; parameteroverdracht verloopt slechts via call by value.

Beschouw, bijvoorbeeld, het volgende C`--` programma:

```
main()        /* copieer input naar output */
{
  read(c);
  while not eof do
  { print(c);
    read(c)
  }
}
```

dat invoer onveranderd naar de uitvoer copieert. Net zoals in C, kunnen statements met behulp van accoladen worden gegroepeerd.

★ *Ontwikkel een* PROLOG*-programma dat C`--` programma's kan executeren, en dat aan de volgende eisen van invoer en uitvoer voldoet:*

- *Het programma heeft als invoer een file met het C`--` programma.*
- *De uitvoer van het programma wordt volledig bepaald door het invoer-uitvoer gedrag van het gegeven C`--` programma.*

2.3.7 Symbolische formule-manipulatie

Het differentiëren en primitiveren (integreren) van functies in de wiskunde kan worden opgevat als een vorm van formule-manipulatie. De mogelijkheden van PROLOG voor het manipuleren van symbolen kan goed gebruikt worden voor de algebraïsche manipulatie van formules. We illustreren het karakter van formule-manipulatie van differentiëren en primitiveren aan de hand van de volgende voorbeelden:

- De *afgeleide* van de functie $f(x) = x^3$ is $f'(x) = \frac{df}{dx} = 3x^2$; hierbij is het cijfer drie van plaats veranderd, en op de plaats van het cijfer 3 is het cijfer 2 terecht gekomen.

$\frac{dc}{dx}$	=	0	(c een constante)
$\frac{dx}{dx}$	=	1	(x een variabele)
$\frac{d(u+v)}{dx}$	=	$\frac{du}{dx} + \frac{dv}{dx}$	(somregel)
$\frac{d(u-v)}{dx}$	=	$\frac{du}{dx} - \frac{dv}{dx}$	(verschilregel)
$\frac{d(u \cdot v)}{dx}$	=	$u(x) \cdot \frac{dv}{dx} + v(x) \cdot \frac{du}{dx}$	(produktregel)
$\frac{d(-f)}{dx}$	=	$-\frac{df}{dx}$	(minregel)
$\frac{d(\frac{f}{g})}{dx}$	=	$\frac{f(x) \cdot \frac{dg}{dx} - g(x) \cdot \frac{df}{dx}}{g^2(x)}$	(quotiëntregel)
$\frac{d(f^n)}{dx}$	=	$n \cdot f^{n-1}(x) \cdot \frac{df}{dx}$	(machtsregel)
$\frac{d(\ln f)}{dx}$	=	$\frac{\frac{df}{dx}}{f(x)}$	
$\frac{d(\sin f)}{dx}$	=	$\cos f(x) \cdot \frac{df}{dx}$	
$\frac{d(\cos f)}{dx}$	=	$-\sin f(x) \cdot \frac{df}{dx}$	

Tabel 2.3: Enkele regels voor het differentiëren.

- De *primitieve* van de functie $g(x) = \frac{1}{x}$ $(x \neq 0)$ is de functie $G(x) = pr(g(x)) = \ln|x|$; in dit geval is een volledig nieuwe functie geïntroduceerd ($\ln x$), die de absolute waarde van de noemer van $g(x)$ als argument heeft.

Na het differentiëren en primitiveren is het vaak noodzakelijk het resultaat te *vereenvoudigen*, ten einde een echt bevredigende algebraïsche expressie te verkrijgen. Differentiëren van de functie $f(x) = x^2 + 4$ levert bijvoorbeeld $f'(x) = \frac{df}{dx} = 2x^1 + 0$; de expressie kan worden vereenvoudigd tot $2x$.

★ *Ontwikkel een* PROLOG*-programma dat in staat is functies van één variabele te differentiëren of primitiveren, waarbij gebruik gemaakt kan worden van een lijst van bekende standaard-afgeleiden en standaard-primitieven, en bepaalde regels die bij het differentiëren en primitiveren toegepast kunnen worden. De functionaliteit van het programma wordt beperkt door de gegeven regels. De regels zijn in de tabel 2.2, 2.3 en 2.4 opgenomen; als additionele regel bij vereenvoudiging kan gebruik worden gemaakt van de commutativiteit van de optelling en vermenigvuldiging.*

Het PROLOG*-programma dient aan de volgende eisen te voldoen:*

- *Er moet een predikaat* `d` *worden gedefinieerd dat als eerste parameter een te differentiëren functie heeft, en als tweede parameter de afgeleide na vereenvoudiging geeft (indien mogelijk). Bijvoorbeeld:*

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


$pr(f(x))$	=	$pr(f(u) \cdot \frac{du}{dx})$	(substitutieregel)
$pr(\frac{df}{dx} \cdot g)$	=	$f(x) \cdot g(x) - pr(f(x) \cdot \frac{dg}{dx})$	(partieel primitiveren)
$pr(x^n)$	=	$\frac{x^{n+1}}{n+1}$	
$pr(a^x)$	=	$\frac{a^x}{\ln a}$	
$pr(\frac{1}{x})$	=	$\ln\|x\|$	
$pr(\sin x)$	=	$-\cos x$	
$pr(\cos x)$	=	$\sin x$	

Tabel 2.4: Enkele regels voor het primitiveren (de integratieconstante is weggelaten).

- *Tevens moet een predikaat* `pr` *worden gedefinieerd dat als eerste parameter een functie heeft waarvan de primitieve moet worden bepaald; de tweede parameter levert de primitieve van de betreffende functie na vereenvoudiging op (indien mogelijk). Bijvoorbeeld:*

```
?- pr(sin(x)+x,Primitieve).
   Primitieve = -cos(x)+1/2*x^2
```

Hoofdstuk 3

Knowledge Engineering

Deskundigen in een bepaald vakgebied zijn gewoonlijk in staat problemen op te lossen die niet goed gedefinieerd zijn, met gebruikmaking van methoden die niet volledig begrepen zijn. Deze expertise is verkregen door langdurige ervaring in het oplossen van problemen. Expertise bestaat uit vele bruikbare feiten en vuistregels. Vuistregels – met spreekt ook wel van *heuristieken* – betreffen kennis omtrent voorwaarden of acties die de beste resultaten opleveren bij het oplossen van een probleem. Deze vorm van ervaringskennis wordt in de literatuur wel *heuristische kennis* genoemd. Expertise is niet eenvoudig te expliciteren: mensen maken gewoonlijk gebruik van kennis van een bepaald vakgebied, zonder er zich rekenschap van te geven hoe zij een gegeven probleem oplossen. Gelukkig is het vaak toch mogelijk met behulp van speciale technieken (een deel van) die kennis in kaart te brengen.

Knowledge engineering is het proces van het verzamelen en vastleggen van kennis uit een probleemgebied, met het uiteindelijke doel een kennissysteem te verkrijgen dat gebruikt kan worden ter ondersteuning bij het oplossen van problemen in dat vakgebied. De benodigde kennis kan afkomstig zijn van één of meer deskundigen in het vakgebied, maar ook verkregen worden uit de literatuur of gegevensverzamelingen. De persoon die een kennissysteem ontwikkelt wordt de *knowledge engineer* genoemd. Knowledge engineering betreft niet slechts het vastleggen van kennis met behulp van een kennisrepresentatieformalisme, maar ook het analyseren van de kennis in een probleemdomein, en het ontwikkelen van *modellen* van deze kennis. Analyse en modellering gaan aan de implementatie van een kennissysteem vooraf, en zijn belangrijke aspecten van knowledge engineering. De kwaliteit van de verzamelde kennis, en het inzicht in het probleemdomein dat door analyse en modellering verkregen wordt, bepalen mede de kwaliteit van het uiteindelijke kennissysteem.

3.1 Leerdoelen

Dit deel van het practicum heeft tot doel u enigszins met de begrippen uit knowledge engineering vertrouwd te maken en u enige vaardigheid in het ontwikkelen van een kennissysteem bij te brengen.

3.2 Aspecten van knowledge engineering

3.2.1 Problemen bij knowledge engineering

Bij de ontwikkeling van een kennissysteem wordt men geconfronteerd met vele onzekerheden. Onzeker is in het begin bijvoorbeeld de gewenste afbakening van het probleemdomein, de wijze waarop de kennis gerepresenteerd zal worden, de keuze van de te gebruiken programmatuur, etc. Door een bepaalde *methodiek* van knowledge engineering toe te passen kan men proberen greep te krijgen op het ontwikkelingsproces en zo de onzekerheden de baas worden.

Elke ontwikkelingsmethodiek houdt een *fasering* of *structurering van activiteiten* in. Het grote voordeel van zo'n fasering is dat al in een vroeg stadium mogelijke problemen gedetecteerd worden en zo nodig adequate acties ondernomen kunnen worden. Elke fase kent zijn eigen niveau van abstractie. Zo zal in de beginfase van de ontwikkeling van een kennissysteem niet direct op het niveau van een programmeertaal of kennisrepresentatieformalisme worden gewerkt, maar veelal met natuurlijke taal, een gestyleerde vorm van natuurlijke taal of een grafische beschrijving van het probleemdomein. Men bespreekt in dit verband wel van *kennismodellen*. Door in het begin van een project met (informele) kennismodellen te werken, die dicht staan bij de aard van een domein, kan voorkomen worden dat na een langdurige periode van implementatie vastgesteld wordt dat het probleemdomein niet volledig gerepresenteerd kan worden in een kennissysteem.

Merk op dat er duidelijke analogieën zijn tussen software en knowledge engineering. In beide gevallen vindt de ontwikkeling van programmatuur plaats in elkaar opeenvolgende fasen, waarbij in elke fase een ander niveau van abstractie wordt gehanteerd om het probleem te representeren. De ontwikkeling van de programmatuur (meer specifiek het kennissysteem) laat zich beschrijven aan de hand van transformaties tussen de verschillende fasen. Er zijn echter ook verschillen tussen software en knowledge engineering. Het belangrijkste verschil is dat het modelleren van domeinkennis bij knowledge engineering een veel centralere plaats inneemt dan bij software engineering. Tevens heeft de wens kennis op een bepaalde wijze in programmatuur vast te leggen, en ermee automatisch te kunnen redeneren, binnen het vakgebied van kennissystemen een groot aantal speciale technieken opgeleverd, zoals uitgebreid in het boek *Principles of Intelligent Systems* beschreven wordt.

Het laatste decennium is veel onderzoek verricht naar methodieken voor knowledge engineering. Het onderzoek heeft echter nog geen methodiek opgeleverd die algemeen ingang gevonden heeft. Vaak wordt de CommonKADS-methodiek als mogelijke kandidaat genoemd. Van deze methodiek, die zich vooral richt op domeinmodellering, is de bruikbaarheid tot op heden echter niet overtuigend aangetoond. In dit practicum zullen we daarom gebruik maken van een methodiek, waarin een aantal elementen van andere methodieken is terug te vinden.

3.2.2 Keuze van het probleemdomein

De eerste beslissing waar men bij het ontwikkelen van een kennissysteem voor komt te staan, betreft het vaststellen of een gegeven probleemdomein al dan niet geschikt is om er een kennissysteem voor te ontwikkelen. Vooral in de beginjaren van kennissystemen werd vaak besloten een kennissysteem te ontwikkelen voor een probleemdomein, waarvoor tot dan toe andere informaticatechnieken gefaald hadden. Een voorbeeld van zo'n probleemdomein is het configureren van computerapparatuur, dat later in XCON opgelost werd. In het algemeen zal dit geen goede uitgangspositie zijn voor de ontwikkeling van een kennissysteem, want de ontwikkeling zal dan zeker niet eenvoudig zijn. Het is vooral aantrekkelijk een kennissysteem voor een bepaald probleemdomein te ontwikkelen als:

- de potentiële voordelen van de toepassing van zo'n systeem groot zijn,
- de moeilijkheidsgraad van het ontwikkelen beperkt is,
- het probleemdomein goed gedefinieerd en eenvoudig af te grenzen is.

XCON bespaart Digital Equipment bijvoorbeeld enkele miljoenen dollars per jaar; de voordelen van dit systeem zijn dus onmiskenbaar.

Het bepalen van de moeilijkheidsgraad van de ontwikkeling van een kennissysteem is in het algemeen niet mogelijk; vooral ervaring met soortgelijke probleemgebieden zal hierbij van belang zijn. Als het probleemgebied niet eenvoudig af te grenzen is, loopt men het risico dat onduidelijk is wanneer de ontwikkeling van het systeem afgerond is. Indien het probleem niet goed gedefinieerd is, zal het veel tijd kunnen vergen voldoende inzicht te krijgen in het probleemdomein. Omdat problemen die goed gedefinieerd zijn meestal ook oplosbaar zijn met behulp van technieken uit de software engineering, zullen de meeste probleemdomeinen waarvoor een kennissysteem wordt ontwikkeld echter niet goed gedefinieerd zijn. Veel van de technieken van knowledge engineering proberen dan ook de onzekerheid van de ontwikkeling, veroorzaakt door het slecht gedefinieerde karakter van het domein, te bestrijden.

3.2.3 Kennisacquisitie

De activiteit van het verzamelen van kennis door het raadplegen van één of meer *kennisbronnen* voor de ontwikkeling van een kennissysteem wordt *kennisacquisitie* genoemd. Voorbeelden van kennisbronnen zijn menselijke deskundigen, literatuur en gegevensverzamelingen. In het geval dat de kennis die verzameld wordt afkomstig is van een menselijke deskundige, wordt soms gesproken van *kenniselicitatie*.

Kennisacquisitie is een vak apart. Het vereist speciale vaardigheden van uiteenlopende aard, zoals communicatieve vaardigheden, psychologisch inzicht, kennis van informatica en van kunstmatige intelligentie. Menselijke kennis is vaak onvolledig, vaag, intuïtief en emotioneel getint, en daarom niet eenvoudigweg in kaart te brengen. Daarom wordt kennisacquisitie vaak een bottle-neck voor het ontwikkelen

van kennissystemen genoemd. Dit is echter maar ten dele het geval. Immers, een deel van de kennis kan niet verzameld worden, juist omdat die kennis te vaag en te intuïtief van aard is. De echte uitdaging van kennisacquisitie is de kennis, die wel geëxpliciteerd kan worden, te verzamelen, en dit zo systematisch mogelijk te doen. Uiteraard zal op deze manier niet een systeem kunnen resulteren, dat een goed psychologisch model van de vakkennis van de deskundige is. Maar dat is ook niet de doelstelling van de ontwikkeling van een kennissysteem. Immers, de doelstelling is een systeem te ontwikkelen, dat in staat is ondersteuning te bieden bij het oplossen van het soort problemen dat gewoonlijk door menselijke deskundigen wordt opgelost. De wijze waarop dit in het systeem gebeurt, mag best anders zijn dan de wijze waarop de deskundige dit doet. Van belang is dus primair de correctheid van de oplossingen die door het uiteindelijke systeem gegenereerd worden, gegeven een bepaald probleem.

De vakkennis van de ervaren deskundige wordt echter wel vaak als uitgangspunt genomen voor de ontwikkeling van een kennissysteem, zodat het verkrijgen van inzicht in de wijze waarop een deskundige problemen oplost, een belangrijk onderdeel is van de ontwikkeling. Hiervoor kan men gebruik maken van diverse technieken; twee hiervan zullen hier kort besproken worden.

Protocolanalyse

Bij protocolanalyse gaat men uit van geregistreerde interviews met menselijke deskundigen. Een in tekstvorm vastgelegd interview wordt wel een *transcript* genoemd.

Er zijn diverse methoden in gebruik voor de inventarisatie van de kennis die de deskundige gebruikt bij het oplossen van problemen in zijn vakgebied. Veel mensen zijn niet gewend over de wijze waarop zij bij het oplossen van problemen te werk gaan te spreken. Het is echter toch mogelijk om een deel van de gebruikte kennis op te sporen door de deskundige concrete problemen ter oplossing voor te leggen. De deskundige wordt nu gevraagd het probleem op te lossen en elke overweging die aldoende bij hem opkomt kenbaar te maken. Het protocol dat resulteert wordt met een lelijke naam wel een 'hardop-denk' protocol genoemd.

De op deze manier verkregen protocollen moeten nauwkeurig worden geanalyseerd. Het is de bedoeling dat de knowledge engineer uiteindelijk een duidelijk beeld krijgt van de wijze waarop de deskundige problemen in zijn vakgebied oplost.

De matrixmethode

Kenniselicitatie wordt soms bemoeilijkt, doordat de deskundige niet in staat is aan te geven welke factoren echt van belang zijn bij het oplossen van een probleem. Sommige van de gebruikte factoren kunnen, bijvoorbeeld, wel van belang zijn, maar zodanig met elkaar verwant zijn dat men die factoren het beste als één geheel kan beschouwen. De *matrixmethode* (*repertory grid method*) maakt het mogelijk dit soort verwantschappen op te sporen.

		E_1	E_2	E_3	E_4	E_5	E_6	E_7		
C_1	Onduidelijke variabelenamen	4	2	3	5	3	1	1	Duidelijke namen	C_1
C_2	Geen commentaar	5	2	5	4	4	3	1	Commentaar	C_2
C_3	Slechte structuur	5	2	1	4	5	2	2	Goede structuur	C_3
C_4	Slechte opzet	5	2	4	3	4	1	2	Goede opzet	C_4
C_5	Wel goto's	5	2	1	5	5	1	1	Geen goto's	C_5
C_6	Slecht leesbaar	5	1	3	4	4	1	2	Goed leesbaar	C_6
C_7	Geen procedures	5	1	4	5	5	5	1	Modulair	C_7

E_1 = Anna (goed)
E_2 = Tim (slecht)
E_3 = Fred (acceptabel)
E_4 = Jan (zeer goed)
E_5 = Karel (goed)
E_6 = Marc (matig)
E_7 = Eline (zeer slecht)

Tabel 3.1: Matrix met eigenschappen van computerprogramma's.

Een *matrix* (repertory grid) is een tabel waarin in vertikale richting een aantal karakteristieke factoren wordt aangegeven en in horizontale richting een aantal typische voorbeelden. De karakteristieke factoren worden *constructen* genoemd, en aangeduid door C_i; de voorbeelden worden *elementen* genoemd, en aangegeven met E_j. Voor elk element E_j wordt in de matrix een score gegeven voor elk construct C_i. De score kan *true* of *false* zijn, of een geheel getal uit een bepaald bereik, bijvoorbeeld 1–5. De mate waarin de constructen verwant zijn met elkaar, kan nu worden bepaald door de correlatiecoëfficiënt $\rho(C_i, C_j)$ van twee constructen C_i, C_j te berekenen. Indien twee constructen sterk met elkaar gecorreleerd zijn, dat wil zeggen $\rho^2(C_i, C_j) = 1 - \epsilon$, $0 \leq \epsilon \ll 1$, kan het zijn dat ze met elkaar verwant zijn. Als $\rho(C_i, C_j) = \pm 1$, dan zijn de constructen C_i en C_j lineair afhankelijk van elkaar.

Voorbeeld. Een projectleider wilde de programmeervaardigheid van de programmeurs in zijn groep bepalen. Hij was echter slechts in staat om in relatief vage termen een beschrijving te geven van wat onder een 'goed' programma verstaan moest worden. Er werd daarom aan hem gevraagd de programma's van specifieke programmeurs te karakteriseren. De constructen die de projectleider opnoemde waren: variabelenamen, commentaar, structuur, opzet, goto statements, leesbaarheid en modulariteit. Voor elke programmeur werd een score van 1 tot 5 voor elk van deze constructen gevraagd. Een score van 1 werd toekend indien de projectleider van mening was dat de programmeur onduidelijke variabelenamen gebruikte, de programma's niet van commentaar voorzag, de structuur slecht was, veel gebruik werd gemaakt van goto's, de programma's slecht leesbaar waren en niet modulair opgezet. Een score van 5 voor elk van deze constructen werd beschouwd als een positieve

uitspraak over de programmeur. Een goede programmeur zou dus een hoge totale score moeten behalen en een slechte programmeur een lage totale score. In Tabel 3.1 is de matrix die de projectleider opgesteld heeft, weergegeven. De eerste en laatste kolom in deze tabel geven de constructen weer met de interpretatie van de minimale en maximale score in het bereik 1–5. De interpretatie van het construct 'variabelenamen' met score 1 is dus 'onduidelijke variabelenamen' en de interpretatie van hetzelfde construct met score 5 is 'duidelijke variabelenamen'. Bovendien is onder de tabel voor elk van de programmeurs de globale (niet op de scores gebaseerde) beoordeling door de projectleider weergegeven. Voor elk paar constructen kan nu de correlatiecoëfficiënt worden berekend. Bijvoorbeeld $\rho(C_4, C_6) = 0{,}898$. Er bestaat derhalve een duidelijke correlatie tussen de constructen C_4 en C_6. In dit geval zijn goede leesbaarheid en goede opzet van een programma met elkaar verwant. Nadat men de correlatiecoëfficiënt van elk tweetal constructen berekend heeft, kunnen gecorreleerde constructen gegroepeerd worden. Merk op dat de projectleider ten onrechte meende dat Jan een betere programmeur was dan Anna op grond van de factoren die hij zelf opgenoemd heeft. ■

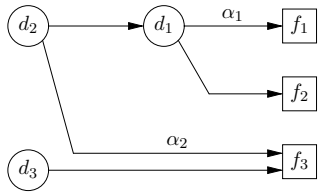
3.2.4 Kennismodellen

Na de analyse van een domein is het meestal duidelijk welke begrippen (ook wel factoren of variabelen genoemd) van belang zijn. Tevens is meestal duidelijk geworden hoe concrete problemen met behulp van deze begrippen opgelost worden. Tegenwoordig wordt het van toenemend belang gevonden om de kennis, en de wijze waarop de kennis toegepast wordt voor het oplossen van problemen, door middel van (informele) modellen expliciet te maken. De volgende drie modeltypen worden wel onderscheiden:

- Het *expertisemodel*: een modelmatige beschrijving van domeinkennis, min of meer los van de wijze waarop de kennis gebruikt wordt.
- Het *taakmodel*: een modelmatige beschrijving van de wijze waarop de kennis wordt gebruikt bij het oplossen van problemen, bijvoorbeeld bij het stellen van een (medische of technische) diagnose of bij constructie.
- Het *organisatiemodel*: een beschrijving van de structuur van een organisatie, vooral met het oog op de toekomstige gebruikers van het kennissysteem.

In dit practicum zal geen aandacht besteed worden aan het organisatiemodel.

Bij de ontwikkeling van een *expertisemodel* wordt frequent gebruik gemaakt van het begrip 'causaliteit' als modelleerprincipe. Hierbij probeert men mogelijke oorzaak-gevolg verbanden tussen de verschillende begrippen in het domein in kaart te brengen. Door de verzamelde causale verbanden weer te geven als een geëtiketteerde, gerichte graaf, wordt een overzichtelijk informeel model van domeinkennis verkregen. Er zijn echter ook andere manieren om de verbanden in de domeinkennis in kaart te brengen. Soms is het beter op zoek te gaan naar empirische verbanden (ervaringsverbanden), in plaats van naar causale verbanden; soms volstaat het de



Figuur 3.1: Causaal expertisemodel.

afhankelijkheden tussen de begrippen in het domein te inventariseren (afhankelijkheidsnetwerk, zie Paragraaf 3.2.5).

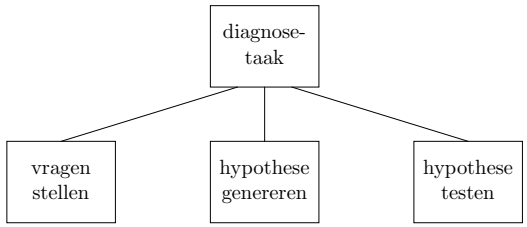
Voorbeeld. In medische handboeken worden ziektebeelden gewoonlijk beschreven in termen van verschijnselen, die geobserveerd kunnen worden bij een patiënt met de betreffende aandoening. Bijvoorbeeld, bij een patiënt met influenza zal men vrijwel altijd koorts kunnen observeren, maar ook dorst, omdat koorts de patiënt dorstig maakt. Dit soort kennis laat zich vrij goed uitdrukken in de vorm van oorzaak-gevolg verbanden. In Figuur 3.1 is door middel van een gerichte graaf, een zeer eenvoudig causaal expertisemodel weergegeven, met de volgende betekenis:

d_1:	koorts	f_1:	rillingen
d_2:	influenza	f_2:	dorst
d_3:	sport	f_3:	spierpijn

waarbij d_i staat voor een 'aandoening' (sport is dus een aandoening of ziekte), en f_j staat voor een 'observeerbaar verschijnsel'. Bijvoorbeeld, $d_2 \xrightarrow{\alpha_2} f_3$ betekent dat influenza *soms* spierpijn veroorzaakt; $d_2 \longrightarrow d_1$ betekent dat influenza *altijd* koorts veroorzaakt. Uiteraard is het niet noodzakelijk een expertisemodel met behulp van een graaf weer te geven; het is ook mogelijk hiervoor een eenvoudige taal te ontwikkelen, zoals hierboven feitelijk gedaan is door gebruik te maken van de symbolen '$\xrightarrow{\alpha}$' en '$\longrightarrow$'.

In dit geval kan dezelfde kennis ook in de vorm van empirische verbanden tot uitdrukking worden gebracht. Zo zou bijvoorbeeld gezegd kunnen worden dat iemand met koorts in de meeste gevallen influenza heeft. Ook deze kennis zou met behulp van een gerichte graaf weergegeven kunnen worden, maar de pijlen zouden in dit geval een andere betekenis hebben dan voor de causale kennis die hierboven gegeven is. ■

Ook voor de ontwikkeling van een *taakmodel* staan de knowledge engineer diverse mogelijkheden ter beschikking. Een taakmodel kan, bijvoorbeeld, weergegeven worden met behulp van een geëtiketteerde graaf, maar ook is het goed mogelijk gebruik te maken van pseudocode. Meestal is het mogelijk een taakmodel op te delen in verschillende deelmodellen. Men spreekt in dit verband van een *taakdecompositie*. In sommige gevallen kan gebruik gemaakt worden van een taakmodel, dat onafhan-



Figuur 3.2: Taakdecompositie voor diagnose.

kelijk is van een specifiek probleemdomein, bijvoorbeeld, een algemeen taakmodel voor diagnose. In andere gevallen is een taakmodel volkomen domeinafhankelijk.

We illustreren een en ander aan de hand van een taakmodel voor diagnose.

Voorbeeld. De meeste vormen van diagnose, zowel in de techniek als in de geneeskunde, kunnen beschreven worden in termen van de volgende deeltaken:

- Het *genereren van (diagnostische) hypothesen*;
- Het *stellen van vragen* aan de gebruiker om invoergegevens (meestal voor het al dan niet bevestigen van hypothesen);
- Het *testen van hypothesen*.

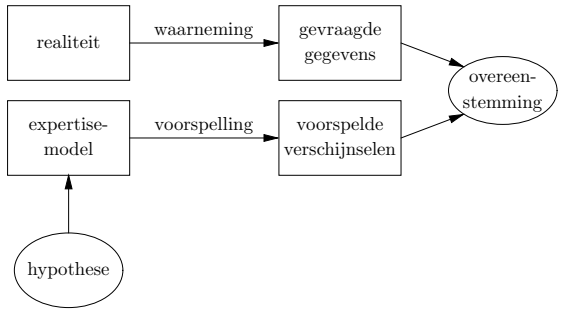
Als een hypothese de test doorstaat, en geaccepteerd wordt, spreken we van een *diagnose*. De betreffende taakdecompositie is weergegeven in Figuur 3.2. Het taakmodel geeft slechts een indruk van de opsplitsing van de verschillende deeltaken. De pseudocode in Figuur 3.3 vult dit verder in door duidelijk te maken hoe de deeltaken onderling samenhangen; in het bijzonder wordt de volgorde, waarin bepaalde taken uitgevoerd worden, vastgelegd. Men spreekt in dit verband wel van het vastleggen van een *strategie*. In Figuur 3.3 staat EM voor het expertisemodel (ook op te vatten als de kennisbank), en staan *Generate*, *Ask* en *Test* voor de drie deeltaken. Deze deeltaken kunnen weer verder worden uitgewerkt.

In Figuur 3.4 is de testtaak in de vorm van een graaf uitgewerkt. De testtaak wordt volgens deze figuur opgevat als het, met behulp van het expertisemodel, vergelijken van voorspelde en gevraagde verschijnselen. Deze vergelijking kan, uiteraard, op verschillende manieren worden ingevuld. Het causale model dat in het vorige voorbeeld is besproken, zou heel goed gebruikt kunnen worden voor het genereren van voorspelde verschijnselen f_i aan de hand van hypothesen, die uit veronderstelde aandoeningen d_j bestaan. De invulling van de testtaak in Figuur 3.4 is echter niet geschikt voor een expertisemodel, waarin gebruik wordt gemaakt van empirische verbanden. In dat geval worden gevraagde gegevens direct geclassificeerd aan de hand van hypothesen; de idee van 'voorspelling' ontbreekt in dat geval.

Tenslotte merken we op dat het top-down inferentie-algoritme vrij sterk lijkt op de bovengegeven beschrijving van diagnose in pseudocode. Dit verklaart dan


```
task Diagnosis(EM)
    diagnosis ← ∅;
    ready ← false;
    while not ready do
        hypothesis ← Generate(EM);
        if hypothesis = ∅ then
            ready ← true
        else
            findings ← Ask(EM, hypothesis);
            accept ← Test(EM, hypothesis, findings);
            if accept then
                diagnosis ← diagnosis ∪ {hypothesis}
            fi
        fi
    od
end
```

Figuur 3.3: Taakmodel met strategie voor diagnose.



Figuur 3.4: Deeltaak voor diagnose.

ook waarom top-down inferentie redelijk geschikt is als redeneermethode voor een diagnostisch systeem, waarin de gebruiker om gegevens wordt gevraagd. ■

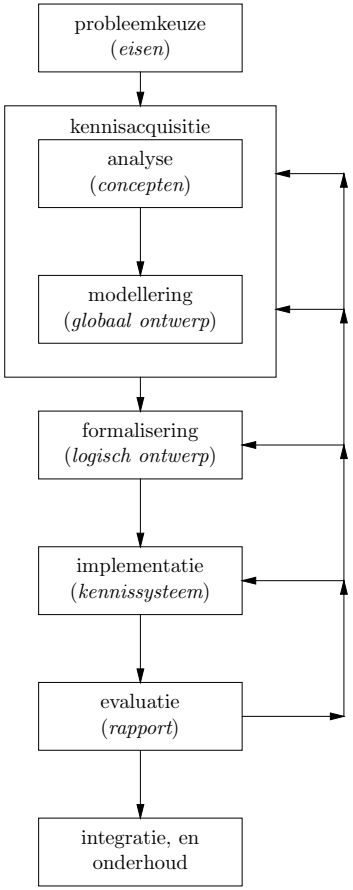
3.2.5 Een methodiek voor knowledge engineering

Een methodiek voor de ontwikkeling van een kennissysteem geeft een systematische en gedetailleerde beschrijving van de achtereenvolgende fasen die tijdens de ontwikkeling doorlopen moeten worden. Tegenwoordig wordt vaak de software life-cycle als uitgangspunt genomen voor de diverse stadia in de ontwikkeling van een kennissysteem. Bij de invulling van de diverse fasen wordt echter wel rekening gehouden met de speciale eigenschappen van kennissystemen. Het is bijvoorbeeld goed bekend dat de geschiktheid van de gekozen benadering moeilijker in te schatten is dan voor de traditionele software life-cycle. Het is meestal ook praktisch niet mogelijk de relatie tussen verwachte invoer en uitvoer van een kennissysteem uitputtend te beschrijven. Bij traditionele programma's is dat vaak wel mogelijk. Bovendien gaan analyse en ontwerp bij knowledge engineering hand in hand: exploratief programmeren, prototyping en experimenteren zijn veel gebruikte technieken bij de ontwikkeling van kennissystemen. De ontwikkeling van een kennissysteem wordt dan ook vaak als een cyclisch proces beschreven, waarin sommige fasen meer dan één keer doorlopen kunnen worden tijdens een proces van verfijning van een kennisbank (knowledge base).

In Figuur 3.5 is een veel gebruikte ontwikkelingsmethodiek voor kennissystemen schematisch weergegeven. De software industrie staat echter nogal huiverig tegenover deze benadering van knowledge engineering aangezien men daar gewend is aan een meer lineaire wijze van het ontwikkelen van programmatuur. Als reactie hierop hebben enkele onderzoeksgroepen relatief statische methodieken voor knowledge engineering ontwikkeld, waarin de ontwikkeling van een kennissysteem als een lineair in plaats van als een cyclisch proces beschreven wordt. Inmiddels is ook binnen deze methodieken meer aandacht voor het exploratieve karakter van knowledge engineering gekomen. In het algemeen moet echter wel getracht worden ten minste een deel van de ontwikkeling van een kennissysteem op lineaire wijze te laten plaatsvinden, immers bij een cyclische methodiek is het verleidelijk de verfijning van het kennissysteem te blijven herhalen. De volgende fasen worden onderscheiden:

1. *Probleemkeuze*. In deze fase wordt besloten of het type probleem geschikt is voor de ontwikkeling van een kennissysteem. Tevens worden de eisen geformuleerd waaraan het systeem moet voldoen. Tenslotte worden de deelnemers in het project gekozen en de benodigde faciliteiten geïnventariseerd, zoals computerfaciliteiten en beschikbare tijd voor de ontwikkeling.

2. *Analyse*. In deze fase wordt de taak die de deskundige uitvoert zorgvuldig geanalyseerd om inzicht te verwerven in de aard van de initieel gegeven data, de conclusies die getrokken worden en de strategie die bij het trekken van de conclusies gevolgd wordt. Het resultaat van deze fase is een verzameling



Figuur 3.5: Cyclische ontwikkeling van een kennissysteem.

belangrijke begrippen (vocabulaire van concepten of termen) in het probleemdomein.

3. *Modellering*. De begrippen worden onderling met elkaar in verband gebracht, hetgeen een *expertisemodel* oplevert. Tevens wordt een beschijving gegeven van de wijze waarop een probleem met behulp van de kennis uit het expertisemodel kan worden opgelost, hetgeen een *taakmodel* oplevert. Men zou het resultaat van deze fase het globale ontwerp van het kennissysteem kunnen noemen.

4. *Formalisering*. De kennismodellen worden vertaald naar geschikte kennisrepresentatieformalismen (produktieregels, logica, frameformalisme, etc.), en inferentiemethoden. De keuze van geschikte formalismen gaat uiteraard aan het formaliseren vooraf. Het resultaat van de fase is een gedetailleerd logisch ontwerp van het kennissysteem.

5. *Implementatie*. Dit betreft de vertaling van de formele specificatie van de kennis in een implementatie-georiënteerd formalisme dat ondersteund wordt door een expert-system shell of andere programmatuur. Bij sommige systemen vallen de formalisering en de implementatie samen (bijvoorbeeld, indien gebruik wordt gemaakt van een resolutie-gebaseerde theorem prover).

6. *Evaluatie*. Hierbij wordt onderzocht of het systeem aan de oorspronkelijk geformuleerde eisen voldoet.

7. *Integratie en onderhoud*. Het ontwikkelde kennissysteem wordt in de organisatie ondergebracht; speciaal opgeleid personeel wordt aangesteld om het kennissysteem te onderhouden en aan te passen indien er zich nieuwe ontwikkelingen voordoen in het domein.

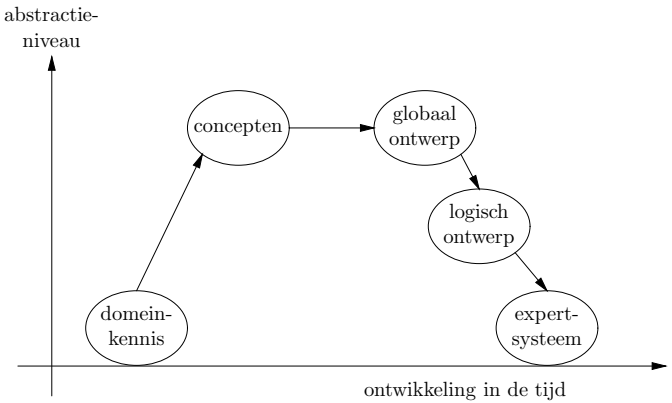
In Figuur 3.6 is grafisch zichtbaar gemaakt hoe het ontwikkelen van een kennissysteem op verschillende abstractieniveaus in de tijd plaatsvindt.

De *probleemkeuze* is al in Paragraaf 3.2.2 geïntroduceerd. Wij zullen nog enkele additionele opmerkingen hierover maken. Bij de beslissing of een probleemdomein geschikt is voor de ontwikkeling van een kennissysteem speelt een groot aantal criteria een rol. Drie belangrijke criteria zijn:

- Het probleem moet slechts door een deskundige opgelost kunnen worden, en er is een deskundige beschikbaar voor deelneming in het project.

- Een traditionele software engineering benadering, waarbij uiteindelijk gebruik gemaakt wordt van een imperatieve programmeertaal voor het representeren van domeinkennis, lijkt niet geschikt.

- Bij het oplossen van het probleem wordt vooral gebruik gemaakt van kwalitatieve of onzekere kennis; numerieke (met uitzondering van probabilistische) methoden spelen bij het oplossen een ondergeschikte of geheel geen rol.



Figuur 3.6: Abstractieniveaus in het ontwerp van een kennissysteem.

Na de keuze van het probleemdomein wordt begonnen met het verzamelen en analyseren van de domeinkennis, met andere woorden, de kennisacquisitie.

De kennisacquisitie is hier in twee fasen verdeeld: in de eerste fase (*analyse*) wordt de kennis verzameld en geanalyseerd. Een fout die in deze fase van de kennisacquisitie vaak gemaakt wordt, is dat bij het verzamelen en analyseren van de domeinkennis al rekening gehouden wordt, of zelfs gebruik gemaakt wordt, van de kennisrepresentatievorm van de expert-system shell die bij de implementatie toegepast wordt. Dit kan er toe leiden dat essentiële aspecten van het probleemdomein over het hoofd gezien worden. Een heldere, en uitgebreide, beschrijving en analyse van het probleemdomein in gewoon Nederlands mag dus nooit ontbreken. Het meest concrete resultaat van de analyse is een *vocabulaire* met concepten of termen die van belang zijn in het domein, waarin elke belangrijke term in het domein wordt beschreven.

Bij het opstellen van het *globale ontwerp*, de tweede fase van de kennisacquisitie, probeert men inzicht te verkrijgen in de volgende aspecten van het probleemdomein:

- Welke soort kennis domineert in het probleemdomein: kennis over *tijd*, *ruimte*, *causaliteit* (oorzaak–gevolg relaties), of *heuristieken* (ervaringsrelaties tussen verschijnselen van onduidelijke aard), etc.

- Het soort probleem dat het systeem moet oplossen: *diagnose*, *constructie*, *registratie van toestandsveranderingen van een object*, etc.

Voorbeeld. Medische diagnostiek, het vaststellen van de naam van een aandoening van een patiënt op grond van bepaalde verschijnselen die bij de patiënt aanwezig zijn, is de meest bekende vorm van diagnostiek. Onder diagnostiek wordt echter ook

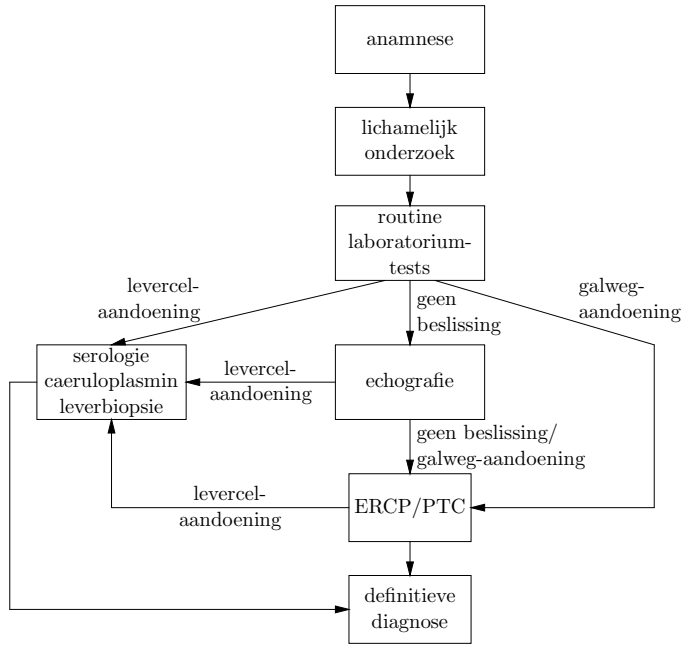
verstaan het vaststellen van de oorzaak van het niet-functioneren van een apparaat. Constructie is het samenstellen van een bepaald object, meestal een bepaald apparaat, met gebruikmaking van gegeven componenten. Bij diagnostiek is vrijwel altijd een vaste verzameling mogelijkheden (bijvoorbeeld aandoeningen, zoals infectieziekten) vooraf bekend; bij constructie is meestal slechts een aantal voorwaarden waaraan het te construeren object moet voldoen, bekend. Het zal duidelijk zijn dat het belangrijk is in de analyse van het probleemdomein dit soort aspecten zo precies mogelijk te beschrijven. ■

Zoals in Paragraaf 3.2.4, waar een algemeen taakmodel voor diagnose werd besproken, is beweerd, kan een taakmodel ook domeinspecifiek zijn. In het volgende voorbeeld wordt de lezer een indruk gegeven van hoe zo'n domeinspecifiek taakmodel er in de praktijk uit kan zien.

Voorbeeld. De diagnostiek en behandeling van aandoeningen van de lever en galwegen is een specialistisch deel van de interne geneeskunde, aangeduid met de naam hepatologie. In de hepatologie staat de arts een groot aantal diagnostische hulpmiddelen ter beschikking, die echter niet allemaal vroeg in het diagnostisch proces gebruikt worden. Sommige van die diagnostische technieken zijn invasief van aard (bijvoorbeeld een leverbiopsie, het nemen van een stukje leverweefsel uit de patiënt); dit soort technieken wordt bij voorkeur toegepast als er voldoende duidelijkheid is dat zo'n onderzoek nuttige informatie kan opleveren, en het risico op complicaties acceptabel is voor de betreffende patiënt gegeven de mogelijke informatie die men op deze wijze kan verkrijgen.

In de hepatologie wordt derhalve eerst eenvoudig onderzoek gedaan, zoals het afnemen van de anamnese, het verrichten van een lichamelijk onderzoek en eenvoudig laboratoriumonderzoek. Op grond van deze informatie kan soms al vastgesteld worden of de patiënt een aandoening heeft van de lever òf van de galwegen. Afhankelijk van de uitkomst van dit onderzoek wordt besloten welk aanvullend onderzoek geïndiceerd is bij de betreffende patiënt.

Bij de ontwikkeling van het HEPAR kennissysteem, een medisch kennissysteem dat de internist behulpzaam kan zijn in de diagnostiek van aandoeningen van de lever en galwegen, is de bovenstaande diagnostische strategie als uitgangspunt genomen voor de ontwikkeling. In Figuur 3.7 is deze strategie, met de diverse taken, schematisch weergegeven. In het geval dat op grond van anamnese, lichamelijk onderzoek en eenvoudig laboratoriumonderzoek blijkt dat de patiënt vermoedelijk een aandoening van de levercellen heeft, wordt onderzoek naar het voorkomen van antilichamen en antigenen in het bloed verricht (serologie), wordt het eiwit caeruloplasmine in het bloed bepaald, dat afwijkend is bij twee chronische aandoeningen van de lever, en wordt eventueel leverweefsel onderzocht na het verrichten van een biopsie. In het geval dat nog niet voldoende duidelijk is wat de patiënt voor aandoening heeft, kunnen met behulp van ultrageluid (echografie) de galwegen worden afgebeeld, en eventuele afwijkingen worden opgespoord. ERCP (Endoscopische Retrograde Cholangio-Pancreaticografie) is een onderzoek waarbij visueel gestuurd met



Figuur 3.7: Taakmodel voor de diagnose van aandoeningen van lever of galwegen.

behulp van glasvezeltechnologie röntgencontrastmiddelen in de galwegen worden gespoten; PTC (Percutane Transhepatische Cholangiografie) is een onderzoek waarbij röntgencontrastmiddelen direct in de galwegen worden gespoten door een canule, via de huid (percutaan), in de lever te steken. Deze onderzoekingen mogen slechts gedaan worden bij patiënten waarvan vrij zeker is dat er sprake is van een aandoening van de galwegen. Uit bovenstaande beschrijving moge blijken dat er een goede reden is waarom diagnostiek in de hepatologie volgens de gegeven strategie plaatsvindt.

Merk op dat bij het opstellen van de strategie nog geen gebruik gemaakt wordt van de specifieke gegevens (zoals klachten van de patiënt – koorts, geelzucht – of specifieke aandoeningen – hepatitis A, alcoholcirrose), maar slechts melding gemaakt wordt van categorieën gegevens. Er is slechts één uitzondering: de vermelding van caeruloplasmine, dat de betreffende hepatoloog blijkbaar zo belangrijk vond dat het expliciet opgenomen werd. ■

Bij het opstellen van het globale ontwerp, worden de gegevens (variabelen) die verkregen zijn uit de analyse als uitgangspunt genomen. Soms is het verstandig bij het

opstellen van de kennismodellen geen aandacht te besteden aan de waarden die de variabelen kunnen aannemen. Dat maakt het mogelijk de kennismodellen abstracter te houden.

Hoewel voor het opstellen van het globale ontwerp van een kennissysteem nog geen algemeen aanvaarde methode beschikbaar is, is de volgende eenvoudige methode redelijk bruikbaar. In deze methode wordt:

1. Een onderscheid in *gegevensgroepen* gemaakt: de termen in het vocabulaire worden gegroepeerd op grond van onderlinge verwantschap (die verwantschappen moeten beschreven zijn in de analyse).

2. Er wordt een *expertisemodel* opgesteld; in Paragraaf 3.2.4 is een eenvoudig voorbeeld gegeven.

3. Er wordt een *taakmodel* opgesteld; in Paragraaf 3.2.4 is een taakmodel voor diagnose gegeven.

Soms is het erg moeilijk de semantische verbanden tussen de gegevens in een domein op te sporen. In dat geval is het vrijwel altijd mogelijk het expertisemodel en taakmodel te laten samenvallen, door slechts de *afleidingsafhankelijkheden* tussen de gegevens op te sporen. Hiermee kan worden vastgelegd welke gegevens als invoergegeven (*input*), welke als uitvoergegeven (*output*) en welke gegevens fungeren als tussenresultaat (*intermediate*). Bij de beschrijving is het handig van een graafnotatie gebruik te maken. Elke knoop in de graaf doet dienst als gegeven; de pijlen in de graaf representeren de afleidingsafhankelijkheid. We noemen zo'n graaf een *afhankelijkheidsnetwerk*. De precieze logische structuur (in termen van conjuncties, disjuncties en negaties) wordt nog niet vastgelegd. De gegevens worden met een bepaalde mate van abstractie gerepresenteerd: specifieke waarden van variabelen (of attributen) worden niet in de graaf weergegeven.

We illustreren de toepassing van een afhankelijkheidsnetwerk als onderdeel van het ontwerp aan de hand van een voorbeeld.

Voorbeeld. Beschouw het onderstaande voorbeeld van een vocabulaire en globaal ontwerp van een kennisbank voor een kennissysteem ten bate van advies bij de behandeling van hartafwijkingen bij jonge kinderen.

- *Vocabulaire*:

```
TERM                    OMSCHRIJVING
name                    naam van het kind
age                     leeftijd van het kind in weken
blood pressure          bloeddruk
collateral circulation  sterke bloedstroom buiten de grote circulatie om
complications           complicaties van de behandeling
dyspnoea                ademnood
failure to thrive       groeiachterstand
feed volume             aantal centiliters vloeibaar voedsel dat
                        het kind binnen krijgt
heart failure           onvermogen van het hart tegemoet te komen aan
                        de eisen van het lichaam ten aanzien van
                        bloedtoevoer
heart rate              hartfrequentie (aantal pulsaties/minuut)
hepatomegaly            vergrote lever
hypertension            hoge bloeddruk
length                  lengte van het kind in cm.
peripheral perfusion    bloeddoorstroming in armen, benen en huid
pulmonary crepitations  bij auscultatie van longen wordt een
                        krakend geluid gehoord
respiratory rate        ademhalingsfrequentie (ademhalingen/minuut)
sex                     geslacht van de patient
therapy                 behandeling van de patient: operatie
                        of medicamenteus
weight                  gewicht van het kind in kg.
```

- *Gobaal ontwerp*:

  1. *Gegevensgroepen*:

```
* PERSONAL DATA
  - name   : (input) SINGEVALUED TEXT ANY
  - age    : (input) SINGEVALUES INT
  - sex    : (input) SINGEVALUED TEXT {male,female}
  - length : (input) SINGEVALUED INT
  - weight : (input) SINGEVALUED REAL

* PHYSICAL EXAMINATION
  - blood pressure      : (input) SINGEVALUED INT
  - dyspnoe             : (input) BOOLEAN
  - qheart rate         : (intermediate) SINGLEVALUED
                          TEXT {nomal,decreased,increased}
  - heart rate          : (input) SINGLEVALUED INT
  - hepatomegaly        : (input) BOOLEAN
  - peripheral
    perfusion           : (input) BOOLEAN
  - pumonary
    crepitations        : (input) BOOLEAN
  - qrespiratory rate   : (intermediate) SINGLEVALUED
                          TEXT {nomal,decreased,increased}
  - respiratory rate    : (input) SINGLEVALUED INT

* PATIENT STATUS
  - collateral
    circulation         : (input) BOOLEAN
  - failure to thrive   : (input) BOOLEAN
  - qfeed volume        : (intermediate) SINGLEVALUED
                          TEXT {normal,
                                strongly_decreased,
```

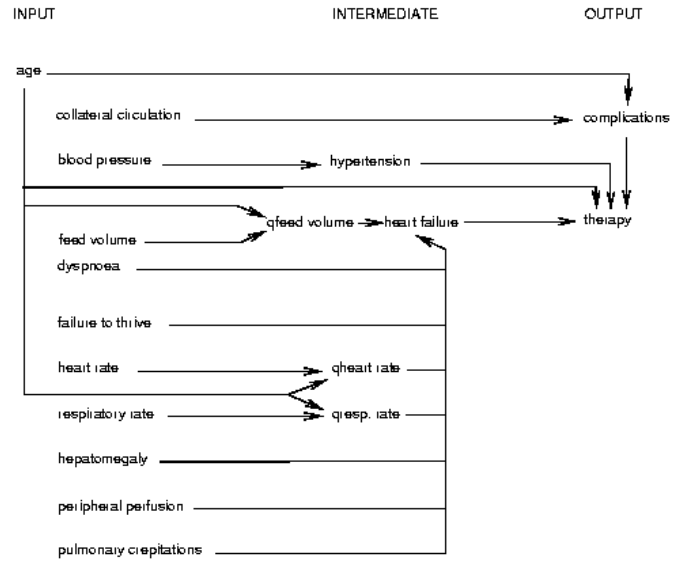
```
                                decreased}
  - feed volume          : (input) SINGLEVALUED INT
  - heart failure        : (intermediate) SINGLEVALUED
                           TEXT {none,severe,moderate}
  - hypertension         : (intermediate) SINGLEVALUED
                           TEXT {none,severe,moderate}

* DECISIONS
  - complications : (output) MULTIVALUED
                    TEXT {aortic aneurysm,
                          paradoxical hypertension,
                          persistent hypertension,
                          recoarctation/residual coarctation,
                          postcoarctectomy syndrome,
                          aortic dissection}
  - therapy       : (output) MULTIVALUED
                    TEXT {prophylaxis bacterial endarteritis/
                          endocarditis,
                          positive inotropic medication,
                          diuretics,
                          balloon angioplasty,
                          operate at 4 yrs unless untreatable
                          heart failure,
                          operate as soon as possible}
```

Met behulp van de prefix `q` wordt een kwalitatieve versie van een gegeven weergegeven. Bijvoorbeeld, met behulp van `qheart rate` wordt vastgelegd of de hartfrequentie normaal, verlaagd of verhoogd is. Het gegeven `heart rate` bevat de hartfrequentie in termen van het aantal pulsaties per minuut. Dus `qheart rate` bevat informatie over de hartfrequentie na interpretatie.

2. *Afleidingsafhankelijkheid*: de bovenbeschreven gegevens zijn aan elkaar gerelateerd zoals weergegeven in Figuur 3.8. In deze graaf is, bijvoorbeeld, vastgelegd dat de bepaling van de kwalitatieve hartfrequentie gebaseerd is op de hartfrequentie èn de leeftijd van de patiënt.

Merk op dat in de graaf niets weergegeven is ten aanzien van de waarden die corresponderende attributen mogen aannemen. Bijvoorbeeld, in de graaf is niet aangegeven welke waarden het met het gegeven `age` corresponderende attribuut `age` zal hebben voor de conclusie `complications`. Merk tevens op dat niet alle gegevens die beschreven worden ook daadwerkelijk een plaats krijgen in de graaf. De gegevens die niet zijn opgenomen, zijn wel van belang voor het verzamelen van patiëntgegevens, maar spelen geen rol in het nemen van de beslissing. ■

In het *gedetailleerde logische ontwerp* worden alle verbanden tussen de specifieke gegevens beschreven in termen van een aantal kennisrepresentatieformalismen. De specifieke waarden van de attributen, en de relaties tussen de attribuutwaarden



Figuur 3.8: Combinatie van expertise- en taakmodel: afhankelijkheidsnetwerk.

van verschillende attributen (indien van een object-attribuut-waarde representatie gebruik wordt gemaakt), worden altijd ingevuld.

Indien het niet voldoende zeker is of het mogelijk is het gegeven probleem met behulp van technieken uit het vakgebied van kennissystemen op te lossen, kan soms, nadat een deel van het domein geïnventariseerd is, een zogenaamd *demonstratieprototype* ontwikkeld worden. De doelstellingen van de ontwikkeling van een demonstratieprototype zijn:

- Het verkrijgen van meer inzicht in de aard en de reikwijdte van het probleem en de door de deskundige gebruikte wijze van probleemoplossen.
- Onderzoek te doen naar de gewenste functionaliteit van het kennissysteem.
- Het kritisch onderzoeken van de voorlopige ontwerpbeslissingen.

In feite wordt met het demonstratieprototype de cyclus formalisering – implementatie – evaluatie versneld doorlopen. Indien het probleemdomein voldoende aangrijpingspunten biedt voor de knowledge engineer, zal men terughoudend moeten zijn met de ontwikkeling van een demonstratieprototype.

Bij de implementatie van een kennissysteem zal veelal gebruik gemaakt worden van een expert-system shell of een expert-system builder tool (bijvoorbeeld CLIPS). Deze systemen bieden de ontwikkelaar al een deel van de benodigde faciliteiten, die de knowledge engineer dus niet zelf hoeft te ontwikkelen. Merk op dat pas aan het einde van de kennisacquisitie duidelijk geworden is welke programmatuur geschikt is voor de ontwikkeling van het kennissysteem. Indien aan het begin van een project al voor bepaalde programmatuur wordt gekozen is het niet uitgesloten dat deze uiteindelijk niet geschikt is.

Bij de evaluatie van het kennissysteem zullen enkele aspecten van het systeem aan een toetsing onderworpen worden. Wij noemen:

- Het vermogen correcte adviezen te geven.
- De begrijpelijkheid van het gegeven advies voor verschillende gebruikers.
- De functionaliteit van het systeem.
- De mogelijkheid het kennissysteem aan te passen.

In praktijk wordt vaak onvoldoende aandacht besteed aan de evaluatiefase, waarschijnlijk omdat de projectdeelnemers het onterechte gevoel hebben het project al te hebben afgerond, voordat met de evaluatie begonnen is.

3.3 Opgaven

3.3.1 Opzet

De doelstelling van deze opgaven is u ervaring te laten opdoen met knowledge engineering en het gebruik van expert-system shells en builder tools door de ontwikkeling van een klein kennissysteem. Voor het practicum kunt u gebruikmaken van het PROLOG-programma `expert.pl` dat u in de directory


```
/vol/practica/IS/Prolog
```

aantreft. (U mag ook eventueel van de expert-builder tools CLIPS, JESS of van DELFI-2 gebruikmaken. Vraag in dat geval om ondersteuning bij het team van het vak Intelligente Systemen.)

Van de practicant wordt verwacht dat, *maximaal in groepen van twee*, een kennissysteem ontwikkeld wordt, betreffende een zelf uit te kiezen probleemgebied. Het resulterende kennissysteem dient ondanks de beperkte opzet toch een serieuze indruk te maken. Het aantal produktieregels en attributen speelt geen overwegende rol bij de beoordeling, de mate waarin het systeem een serieuze poging is om het probleem op te lossen, speelt wel een rol. Uiteraard zal het voor u veelal niet mogelijk zijn om in het kader van het practicum samen met een domeindeskundige een kennissysteem te ontwikkelen. De noodzakelijke domeinkennis kunt u daarom het beste uit een boek of artikel halen.

Geschikte voorbeelden van probleemgebieden voor verwerking in een kennissysteem zijn:

- Medische diagnostiek (longziekten, kinderziekten, leverziekten, hoofdpijn, epilepsie, etc.).
- Keuze van behandeling in de geneeskunde (medicamenteuze therapie van infectieziekten, hartziekten, etc.).
- Opsporen van defecten in apparatuur (motoren of printers, bijvoorbeeld).
- Advies bij syntaxfouten in een Java programma voor een C++ programmeurs.
- Classificatie van chemische verbindingen, etc.
- Het samenstellen van een voedzame maaltijd met ingrediënten die voorradig zijn.
- Het configureren van apparatuur.
- Besturing van een gesimuleerde machine, met gebruikmaking van ervaringskennis.

Het dierherkenningsprobleem is in zoveel boeken beschreven, dat dit in ieder geval niet in aanmerking komt. Uiteraard mag u ook niet gebruik maken van de gegeven voorbeeldkennisbanken.

3.3.2 Verslaglegging

Men dient een verslag in te leveren waarin het volgende tenminste aan bod komt:

★ (**K.1**) *Een nauwkeurige* beschrijving en analyse *van het gekozen probleemdomein. Bij deze beschrijving en analyse dient men volledig te abstraheren van de kennisrepresentatieformalismen die door de expert-system shell die gekozen is voor de implementatie geboden wordt. Tevens moet de beschrijving* volledig *zijn, d.w.z. het mag niet voorkomen dat er in het uiteindelijke kennissysteem bepaalde begrippen voorkomen die in de beschrijving en analyse niet voorkomen. Bij de analyse van het probleemdomein dient men in het bijzonder zo precies mogelijk aan te geven op welke wijze problemen in het kennisdomein opgelost worden (door de deskundige, of zoals beschreven in de literatuur). Tenslotte moet het vocabulaire voor het domein worden gegeven.*

★ (**K.2**) *Een systematische beschrijving van de gekozen oplossing en de structuur die men in het probleem heeft onderkend aan de hand van een* ontwerp. *Hierbij dient men onderscheid te maken tussen een globaal en gedetailleerd ontwerp:*

- *In het* globaal ontwerp *wordt het probleemdomein verder beschreven aan de hand van gegevensgroepen. Tevens word een expertisemodel (bijvoorbeeld, een causaal netwerk) en een taakmodel gegeven (inclusief de* strategie *die gebruikt wordt voor het oplossen van het probleem); in sommige gevallen kan volstaan worden met een afhankelijkheidsnetwerk.*
- *In het* gedetailleerd ontwerp *wordt een afbeelding naar een kennisrepresentatieformalisme, bijvoorbeeld: produktieregels, frames, procedures en logica gegeven.*

★ (**K.3**) *Verder moet worden ingeleverd:*

- *De documentatie van de* implementatie *van het kennissysteem bestaat uit een beschrijving van de wijze waarop de knowledge base die men geïmplementeerd heeft, opgebouwd is.*
- *Enkele consultaties van het kennissysteem aan de hand van een aantal testgevallen. Deze testgevallen dienen kort beschreven te worden.*
- *Een listing van de volledige knowledge base.*
- *De kennisbank dient u bovendien per e-mail naar de practicumleiders te sturen.*

Appendix A

Handleiding OTTER

OTTER is een op resolutie gebaseerde theorem prover voor eerste-orde predikatenlogica met gelijkheid en ordeningspredikaten. De naam OTTER is een afkorting van 'Other Techniques for Theorem-proving and Effective Research'.

In OTTER is een aantal inferentieregels geïmplementeerd, waaronder binaire resolutie en hyperresolutie. Deze inferentieregels kunnen worden toegepast op een verzameling clauses, waarbij gebruik wordt gemaakt van de set-of-support strategie.

Behalve clauses accepteert OTTER ook formules in eerste-orde predikatenlogica als invoer; deze worden in clauses vertaald en verder als zodanig behandeld.

Verder biedt OTTER de mogelijkheid van demodulatie. Hierbij wordt een clause herschreven met behulp van een verzameling gelijkheden (demodulatoren).

OTTER is half-automatisch: de gebruiker moet zèlf de te gebruiken inferentieregel(s) selecteren, beslissen welke formules of clauses deel uitmaken van de initiële set-of-support en welke gelijkheden als demodulator moeten optreden.

A.1 Het gebruik van OTTER

OTTER is niet interactief, zodat eerst met een editor een file aangemaakt moet worden, die vervolgens als invoer dient voor het programma. Om de uitvoer te kunnen bestuderen kan deze het beste ook naar een file gestuurd worden. Een aanroep van OTTER ziet er dus als volgt uit:

```
otter < input_file
```

òf

```
otter < input_file > output_file
```

Ondanks het feit dat in het laatste geval de uitvoer naar *output_file* wordt geschreven, verschijnt dan op het beeldscherm nog wel een aantal meldingen, bijvoorbeeld wanneer de invoerfile fouten bevat, wanneer een refutatie is gevonden, of wanneer de set of support leeg is (zie Paragraaf A.8.1).

A.1.1 Commando's

Een invoerfile voor OTTER bestaat uit een serie commando's. Allereerst moeten de commando's gegeven worden, waarmee de gewenste opties, bijvoorbeeld met betrekking tot de toe te passen inferentieregel(s), of de uitvoer die gegenereerd moet worden, geselecteerd worden. De belangrijkste commando's hiervoor zijn:

```
set(flag).                  % zet een flag 'aan'
clear(flag).                % zet een flag 'uit'
assign(parameter,integer).  % geef een parameter een waarde
```

Flags en parameters zijn soorten opties; in Paragraaf A.2 worden de belangrijkste opties beschreven.

Na de commando's voor de diverse opties volgen de lijsten met clauses c.q. formules. De volgende commando's dienen om deze lijsten aan te kondigen:

```
list(usable).           % lees axioma's in clause-vorm.
list(sos).              % lees sos in clause-vorm.
list(demodulators).     % lees demodulators in clause-vorm.
formula_list(usable).   % lees axioma's in formule-vorm.
formula_list(sos).      % lees sos in formule-vorm.
```

Elke lijst moet worden afgesloten met `end_of_list.` . Iedere clause of formule in een lijst moet worden afgesloten met een `.` (punt).

De syntax waaraan een clause respectievelijk een formule moet voldoen, staat beschreven in Paragraaf A.3. Merk op dat demodulatoren alleen kunnen worden geschreven als clauses.

A.1.2 Commentaar

Vaak is het nodig om (delen van) een kennissysteem van commentaar te voorzien, bijvoorbeeld over de betekenis van formules of van gebruikte namen. In OTTER maakt het karakter `%` dit mogelijk: alle tekens tussen de eerste `%` op een regel en het einde van die regel worden door OTTER genegeerd. Dit betekent echter óók dat ze niet naar de uitvoerfile worden geschreven.

A.2 Opties

OTTER kent twee soorten opties: flags en parameters.

A.2.1 Flags

Flags zijn opties die 'aan' of 'uit' kunnen staan. Hun waarde wordt geregeld met behulp van de commando's `set` en `clear` (Paragraaf A.1.1), die de naam van de flag als argument meekrijgen. Wanneer geen waarde gespecificeerd is, levert OTTER zelf een waarde; dit heet de *default*-waarde.

Hieronder volgt een beschrijving van de belangrijkste flags, met hun default-waarde.

▶ Flags die de inferentie regelen

- `binary_res`: default 'clear'.
  Als deze flag 'set' is, wordt binaire resolutie toegepast om nieuwe clauses te genereren (samen met eventueel andere geselecteerde inferentieregels).
- `hyper_res`: default 'clear'.
  Als deze flag 'set' is, wordt *positieve* hyperresolutie toegepast om nieuwe clauses te genereren (samen met eventueel andere geselecteerde inferentieregels).
- `neg_hyper_res`: default 'clear'.
  Als deze flag 'set' is, wordt *negatieve* hyperresolutie toegepast om nieuwe clauses te genereren (samen met eventueel andere geselecteerde inferentieregels.
- `order_hyper`: default 'set' als hyper_res of neg_hyper_res aangezet zijn. Zet deze altijd op 'clear'.
- `ur_res`: default 'clear'.
  Als deze flag 'set' is, wordt UR-resolutie toegepast om nieuwe clauses te genereren (samen met eventueel andere geselecteerde inferentieregels).

▶ Flags die de uitvoer regelen

- `print_kept`: default 'set'.
  Als de waarde van deze flag 'set' is, worden de nieuw gegenereerde clauses afgedrukt, nadat ze verwerkt zijn.
- `print_back_sub`: default 'set'.
  Als de waarde van deze flag 'set' is, wordt er melding van gemaakt welke clauses door middel van backward-subsumptie worden verwijderd (zie ook de flag `back_sub` en het voorbeeld in Paragraaf 1.2.3).
- `print_given`: default 'set'.
  Als deze flag 'set' is, worden clauses als output opgeleverd, zodra ze 'given-clause' worden.
- `demod_history`: default 'set'.
  Als deze flag 'set' is, worden de nummers van de demodulatoren die bij het afleiden van een clause zijn gebruikt, toegevoegd aan de lijst met derivatiegegevens van de betreffende clause.
- `print_proofs`: default 'set'.
  Als de waarde van deze flag 'set' is, wordt elk gevonden bewijs als output opgeleverd.

► **Overige flags**

- `back_sub`: default 'set'.
  Als deze flag 'set' is, wordt tijdens het verwerken van een nieuw gegenereerde clause backward-subsumptie toegepast; dat wil zeggen dat alle clauses uit *usable* (de axiomaverzameling) en *sos* die gesubsumeerd worden door de nieuw gegenereerde clause, worden verwijderd (zie het voorbeeld in Paragraaf 1.2.3).

  Een clause C_1 wordt gesubsumeerd door een clause C_2, als C_1 een disjunctie is van C_2 en een (eventueel lege) clause C_3, dus als C_1 gelijk is aan $(C_2 \vee C_3)$.

- `demod_out_in`: default 'clear'.
  Als deze flag 'set' is, vindt demodulatie (Paragraaf A.5) plaats van buiten naar binnen en van links naar rechts. Als de waarde van deze flag 'clear' is, worden termen van binnen naar buiten gedemoduleerd (zie het voorbeeld in Paragraaf 1.2.6).

A.2.2 Parameters

Parameters zijn opties die een integer-waarde hebben. Hun waarde wordt geregeld met behulp van het commando `assign` (Paragraaf A.1.1), dat als eerste argument de naam van de parameter vereist, en als tweede argument de integer-waarde.

Hieronder volgt een beschrijving van de belangrijkste parameters. Hierbij stelt n de waarde van de parameter voor; MAX_INT is een grote integer.

► **Parameters die het zoekproces aan banden leggen**

- `max_proofs`: default 1, range $[0, \ldots,$ MAX_INT$]$.
  Als $n = -1$, dan zal OTTER zoveel mogelijk bewijzen proberen te vinden, anders zal OTTER net zolang door proberen te zoeken totdat n bewijzen zijn gevonden.

- `max_seconds`: default 0, range $[0, \ldots,$ MAX_INT$]$.
  Als $n \neq 0$, dan wordt het zoekproces na ca. n seconden beëindigd.

- `max_gen`: default 0, range $[0, \ldots,$ MAX_INT$]$.
  Als $n \neq 0$, dan wordt het zoekproces beëindigd nadat ca. n clauses zijn afgeleid.

► **Overige parameters**

- `stats_level`: default 2, range $[0, \ldots, 3]$.
  Deze parameter regelt de hoeveelheid statistische uitvoer, die aan het einde van het zoekproces wordt afgedrukt. Als $n = 0$, wordt geen statistische uitvoer geproduceerd.
  Aangezien deze statistieken in het algemeen weinig relevante informatie bevatten, wordt geadviseerd deze parameter de waarde 0 toe te kennen.

A.3 Syntaxis

A.3.1 Namen

Namen zijn opgebouwd uit hoofdletters, kleine letters, en eventueel de karakters `$` en _; ze bestaan uit maximaal 50 karakters.

Namen kunnen gebruikt worden als constanten, functie- en predikaat-symbolen en variabelen. Het type wordt in het algemeen bepaald door de context. In clauses geldt echter dat variabelen met een kleine letter u, v, w, x, y of z moeten beginnen. Dit is nodig om onderscheid te kunnen maken tussen constanten en variabelen. In formules kan elke naam als variabele gebruikt worden, omdat variabelen daar met behulp van de kwantoren `all` en `exists` aangeduid worden.

Om aan de syntactische eisen voor clauses te voldoen, verandert OTTER sommige namen bij de vertaling van formules in clauses:

- Gekwantificeerde namen (variabelen), die nìet met een kleine letter u, v, w, x, y of z beginnen, worden vervangen door `x1`, `x2`, ...

- Constanten in een formule die met een kleine letter u, v, w, x, y of z beginnen, worden niet vertaald; wel wordt een waarschuwing gegenereerd.

De overige namen blijven onveranderd. In Paragraaf 1.2.4 wordt hier een voorbeeld van gegeven.

Een naam kan slechts voor één doel gebruikt worden: het vóórkomen van verschillende aantallen argumenten[1] bij één naam resulteert in een foutmelding (zie Paragraaf A.8.2). Deze eigenschap van OTTER is een handig hulpmiddel om fouten in het programma tijdig op te sporen.

OTTER kent een aantal *speciale* namen:

- Namen die beginnen met `EQ`, `Eq`, of `eq`, worden gezien als gelijkheidspredikaten in prefixpositie voor demodulatie; `=` is het gelijkheidspredikaat in infixpositie (Paragraaf A.5).

- Predikaat-symbolen die beginnen met `$ANS`, `$Ans`, of `$ans` worden aangemerkt als 'antwoord'-predikaat (Paragraaf A.7).

- Andere symbolen die met een `$` beginnen zijn evalueerbare functies of predikaten (Paragraaf A.6).

A.3.2 Termen en lijsten

1. Een constante is een term.

2. Een variabele is een term.

[1] Het aantal argumenten bij een functie- of predikaat-symbool wordt ook wel *ariteit* genoemd. Constanten zijn in feite functies met ariteit 0; proposities zijn predikaten met ariteit 0.

3. Als $t_1, \ldots, t_n$ termen zijn, en f is een functiesymbool, dan is f`(`t_1`,...,`t_n`)` een term.

Verder kent OTTER een PROLOG-achtige lijstnotatie; het symbool `[]` is een afkorting voor `$nil`, `[`t_1`|`t_2`]` een afkorting voor `$cons(`t_1`,`t_2`)`, en `[`t_1`,`t_2`,`t_3`,`t_4`]` is een afkorting voor `$cons(`t_1`,$cons(`t_2`,$cons(`t_3`,$cons(`t_4`,$nil))))`. De notatie `[`t_1`,`t_2`|`t_3`]`, echter, is *niet* toegestaan; zo'n term moet worden geschreven als `[`t_1`|[`t_2`|`t_3`]]`.

A.3.3 Atomen

1. Als $t_1, \ldots, t_n$ termen zijn, en P is een predikaatsymbool, dan is P`(`t_1`,...,`t_n`)` een atoom.
2. Als t_1 en t_2 termen zijn dan zijn `(`t_1 `=` t_2`)` en `(`t_1 `!=` t_2`)` atomen.

De symbolen `=` en `!=` zijn *infix*-predikaatsymbolen; ze geven respectievelijk gelijkheid en ongelijkheid weer. De spaties rondom `=` en `!=`, en de haakjes zijn verplicht.

A.3.4 Syntaxis van clauses

1. Als A een atoom is, dan is A een *positieve* en `-`A een *negatieve literal*.
2. Een *clause* bestaat uit een of meer literals gescheiden door `|`.

Binnen een negatieve literal mag geen spatie staan tussen het negatie-teken en het atoom. Tussen de literals mogen wel spaties voorkomen.

A.3.5 Syntaxis van formules

1. Een atoom is een formule.
2. Als F en G formules zijn, dan zijn `(`F `<->` G`)` en `(`F `->` G`)` formules.
3. Als $F_1, \ldots, F_n$ formules zijn, dan zijn `(`F_1 `|` $\cdots$ `|` F_n`)` en `(`F_1 `&` $\cdots$ `&` F_n`)` formules.
4. `all` en `exists` zijn kwantoren. Als $Q_1, \ldots, Q_n$ kwantoren zijn, $x_1, \ldots, x_n$ zijn namen, en F is een formule, dan is `(`Q_1 x_1 $\cdots$ Q_n x_n F`)` een formule.
5. Als F een formule zonder negatie-teken is, dan is `-`F een formule.

De gebruikte symbolen hebben hun gebruikelijke logische betekenis:

`<->` : 'dan en slechts dan als'
`->` : 'impliceert'
`|` : 'of', $\vee$
`&` : 'en', $\wedge$
`all` : 'voor alle', de alkwantor $\forall$
`exists` : 'er bestaat', de existentiële kwantor $\exists$

Alle haakjes zijn verplicht. Voor en na `<->`, `->`, `|` en `&` moeten spaties staan. `all`, `exists`, en hun bijbehorende variabelen moeten gevolgd worden door een spatie.

A.4 Het 'gewicht' van een clause

Bij het kiezen van een zogenaamde given-clause uit *sos* wordt geselecteerd op het 'gewicht' van de in *sos* aanwezige clauses: de 'lichtste' clause wordt gekozen.

- Het gewicht van een *clause* is gelijk aan de som van de gewichten van zijn literals.
- Het gewicht van een *literal*, positief of negatief, is gelijk aan het gewicht van zijn atoom.
- Het gewicht van een *atoom* is gelijk aan het aantal variabele-, constante-, functie- en predikaat-symbolen, waaruit het betreffende atoom bestaat.

Wanneer de flag `print_given` 'set' is (de default-waarde, zie Paragraaf A.2.1), worden de geselecteerde given-clauses afgedrukt. De toegekende gewichten worden daarbij weergegeven als `(wt=`n`)`.

A.5 Demodulatie

De verwerking van een nieuw gegenereerde clause bestaat onder andere uit het toepassen van demodulatie op die clause. Bij demodulatie wordt een clause herschreven met behulp van de verzameling gelijkheden, zoals die in *demodulators* staat beschreven. Gelijkheid kan worden weergegeven door middel van het infix-predikaatsymbool `=`, of met behulp van een binair predikaatsymbool, beginnend met `EQ`, `Eq`, of `eq`.

Demodulatie vindt default van binnen naar buiten plaats. Met behulp van het commando `set(demod_out_in).` (Paragraaf A.2.1) kan men dit desgewenst veranderen.

A.6 Evalueerbare predikaten en functies

OTTER kan een aantal vergelijkingen, rekenkundige, Boolse en conditionele expressies evalueren. Deze evaluatie wordt toegepast tijdens demodulatie en hyperresolutie.

Wanneer een clause bijvoorbeeld de term `$SUM(`i_1`,`i_2`)` bevat, zal deze bij demodulatie vervangen worden door i_3, de som van de twee integers i_1 en i_2, alsof de demodulator `($SUM(`i_1`,`i_2`) =` i_3`)` deel uitmaakt van de lijst van demodulatoren.

Bij hyperresolutie zal een negatieve literal `-$LT(`i_1`,`i_2`)` verwijderd worden wanneer voor de integers i_1 en i_2 geldt dat $i_1 < i_2$, alsof de unit clause `$LT(`i_1`,`i_2`)` aanwezig is (zie ook het voorbeeld in Paragraaf 1.2.7).

Het gedrag van de in Tabel A.1 genoemde functies en predikaten is als volgt:

$int \times int \rightarrow int$	`$SUM, $PROD, $DIFF, $DIV, $MOD`
$int \times int \rightarrow bool$	`$EQ, $NE, $LT, $LE, $GT, $GE`
$term \times term \rightarrow bool$	`$ID, $LNE, $LLT, $LLE, $LGT, $LGE`
$bool \times bool \rightarrow bool$	`$AND, $OR`
$bool \rightarrow bool$	`$NOT`
$\rightarrow bool$	`$T, $F`

Tabel A.1: Evalueerbare functies en predikaten.

1. $int \times int \rightarrow int$: `$DIV` berekent het integer-deel van een deling, `$MOD` berekent de rest.
2. $int \times int \rightarrow bool$: `$EQ, $NE, $LT, $LE, $GT` en `$GE` staan voor respectievelijk $=, \neq,$ $<, \leq, >$ en $\geq$.
3. $term \times term \rightarrow bool$: deze functies zijn vergelijkbaar met de functies voor $int \times int \rightarrow bool$, behalve dat lexicografische vergelijking wordt toegepast in plaats van rekenkundige.

De overige functie- en predikaat-symbolen spreken voor zich.

A.7 Answer literals

Elke literal waarvan de predikaatnaam met `$ANS`, `$Ans`, of `$ans` begint, is een 'antwoord'-predikaat. Dergelijke literals worden bij de inferentie genegeerd. Wanneer een clause wordt gegenereerd, die slechts uit answer literals bestaat, wordt deze dus beschouwd als de lege clause.

Answer literals kunnen zodoende gebruikt worden om informatie over een gevonden bewijs op te slaan, zonder dat dit de inferentie beïnvloedt (zie het voorbeeld in Paragraaf 1.2.5).

A.8 Meldingen en fouten

A.8.1 Meldingen naar het beeldscherm

Wanneer OTTER wordt aangeroepen met

`otter` < *input_file* > *output_file*

verschijnt, ondanks het feit dat de uitvoer naar *output_file* wordt geschreven, op het beeldscherm nog wel een aantal meldingen:

- Telkens wanneer een refutatie wordt gevonden, verschijnt:

  ```
  ---------- PROOF ----------
  ```


  mits de optie `print_proofs` 'set' is.

- Wanneer de set of support leeg is, wordt gemeld:

  ```
  sos empty.
  ```

  en verschijnt de prompt weer op het scherm.

- Wanneer het zoekproces door de waarde van een bepaalde parameter wordt afgebroken, geeft OTTER bijvoorbeeld één van de volgende meldingen:

  ```
  search stopped by max_seconds option.  of
  search stopped by max_gen option.
  ```

  en verschijnt eveneens de prompt weer.

- Wanneer de invoerfile grammaticale fouten bevat, wordt bijvoorbeeld gemeld:

  ```
  5 input errors were found.
  ```

  waarna de prompt op het scherm verschijnt.

A.8.2 Foutmeldingen in de uitvoer

Wanneer de invoerfile grammaticale fouten bevat, wordt niet alleen een boodschap naar het scherm gestuurd, maar staan op de uitvoerfile ook aanwijzingen op welk punt een fout is gedetecteerd, en wat voor type fout gemaakt is.

Het punt waar de fout is gedetecteerd, wordt aangegeven met

```
----------------------^
```

Meestal is de werkelijke fout echter al eerder gemaakt.

Het is mogelijk dat één fout een hele reeks van foutmeldingen tot gevolg heeft. Bij het 'debuggen' van een programma kan daarom in het algemeen het beste van voor naar achter worden gewerkt.

Er zijn verschillende typen foutmeldingen, waarvan een aantal hieronder wordt genoemd. Sommige spreken voor zich, andere zijn wat cryptischer.

▶ `ERROR, multiple arities :`*naam*`:`

In OTTER kan een naam slechts voor één doel gebruikt worden: het vóórkomen van verschillende aantallen argumenten bij één naam resulteert in een foutmelding. Deze eigenschap van OTTER is een handig hulpmiddel om fouten in een programma tijdig op te sporen.

Als voorbeeld nemen we een invoerfile, waarop onder andere de volgende formule staat:

```
  (all plaats1  all plaats2  all plaats3
      (  (  Verbinding(plaats1,plaats2)
           & Verbinding(plaats2,plaats3) )
       -> (  Verbinding(plaats1,plaats3)
           | $Ans( Verbinding(plaats1,plaats3,via(plaats2)) ) ) )  ).
```

Het symbool `Verbinding` heeft hier een dubbele functie: in de eerste drie voorkomens is het een predikaat-symbool met ariteit 2; bij het laatste staan echter drie argumenten. Wanneer de betreffende file in OTTER wordt ingevoerd, treffen we in de uitvoer de volgende toelichting aan:

```
ERROR, multiple arities :Verbinding:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) | $Ans(Verbinding(plaats1,plaat
s3,via(plaats2))) ) )  ).
------------------------------------------------------------------
------------------------------------------------------------------
-----------------------------------------------------^
```

Na enig puzzelen blijkt het pijltje van de foutmelding inderdaad naar het laatste voorkomen van `Verbinding`, op de derde regel, te wijzen.

▶ `ERROR, too few arguments:`

Deze foutmelding kan wijzen op een teveel aan openingshaakjes (of haakjesparen), zoals in de volgende clause/formule:

```
ERROR, too few arguments:
    ( Verbinding(Amsterdam,Utrecht) ).
------^
```

▶ `ERROR, too many arguments:`

Dit duidt op een tekort aan haakjes.

```
ERROR, too many arguments:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   ->   Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
------------------------------------------------------------------
------------------------------------------------------------------
-----------------------------------^
```

Hier is na `->` een openingshaakje vergeten, zodat een niet-welgevormde formule is ontstaan, waarin iets van de vorm (*P* `->` *Q* `|` *R*) voorkomt.

▶ `ERROR, bad quantified formula:`

Deze foutmelding duidt erop dat het gekwantificeerde deel van de formule niet met haakjes is omgeven.

```
ERROR, bad quantified formula:
(all plaats1  all plaats2  all plaats3
      ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
------------------------------------------------------------------
------------------------------------------------------------------
---^
```

Hier is na de kwantoren een openingshaakje vergeten. Op het moment dat OTTER `->` tegenkomt, wordt iets van de vorm (*Q*$_1$ *x*$_1$ ⋯ *Q*$_n$ *x*$_n$ *P* `->` ⋯ gelezen, zodat dàn de fout wordt opgemerkt.

▶ `ERROR, operators switched:`

Deze foutmelding wijst eveneens op het vergeten van een haakje.

```
ERROR, operators switched:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3)
   -> ( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
------------------------------------------------------------------
------------------------------------------------------------------
---^
```

Hier is het sluithaakje voor `->` vergeten. Deze formule bevat dus iets van de vorm (*P* `&` *Q* `->` *R*).

▶ `ERROR, bad word:`

Dit is een vrij aspecifieke foutmelding, die slechts aanduidt dat er iets anders staat dan OTTER op grond van zijn grammatica had verwacht. We laten enkele voorbeelden zien:

```
ERROR, bad word:
( Verbinding(Amsterdam,Utrecht) | Verbinding(Utrecht,Zwolle) ).
--------------------------------^
```

In deze clause is sprake van een teveel aan haakjesparen: ten gevolge van het openingshaakje past hier volgens OTTER's grammatica voor clauses slechts `=` of `!=`.

```
ERROR, bad word:
(all plaats1  all plaats2  all plaats3
```

```
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) )   ).
-------------------------------------------------------------------
-------------------------------------------------------------------
-----------------------------------------------------------^
```

In deze formule is een van de laatste sluithaakjes vergeten.

```
ERROR, bad word:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) |$Ans_via(plaats2) ) )  ).
-------------------------------------------------------------------
-------------------------------------------------------------------
-----------------------------------^
```

En in deze formule is de spatie na `|` vergeten.

▶ `ERROR, text after term:`

Deze foutmelding wijst op het vergeten van de punt na een commando.

▶ `ERROR, text after formula:`

Deze foutmelding kan eveneens wijzen op het vergeten van een punt, maar wijst ook vaak op een teveel aan sluithaakjes in een formule:

```
ERROR, text after formula:
     Verbinding(Amsterdam,Utrecht)).
-----------------------------------^
```

of juist een tekort aan openingshaakjes:

```
ERROR, text after formula:
 all plaats1  all plaats2  all plaats3
  (   (  Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   -> (  Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
----^
```

In de laatste formule is het openingshaakje voor `all` vergeten, zodat volgens OTTER's grammatica voor formules `all` als een volledige formule wordt gezien en daar dus een `.` werd verwacht.

Wanneer deze foutmelding meerdere malen voorkomt, kan de oorzaak zijn dat een `formula_list(...).` is gedeclareerd in plaats van `list(...).`.

Ook kan het zijn dat een `formula_list(...).` niet met `end_of_list.` is afgesloten. In dat geval wordt een volgend commando `list(...).` niet gezien als commando, maar als formule, en de daarbij behorende clauses zullen, met name wat betreft de haakjes, in het algemeen niet aan de verwachte grammatica voor formule's voldoen.

▶ `ERROR, | or .  expected in clause:`

Deze foutmelding duidt op het vergeten van een punt aan het einde van een clause, of op het gebruik van een onbekende operator:

```
list(sos).
ERROR, | or . expected in clause:
     Verbinding(Amsterdam,Utrecht) & Verbinding(Utrecht,Zwolle).
-----------------------------------^
end_of_list.
```

De grammatica voor clauses kent de `&`-operator niet.

▶ `ERROR, comma or ) expected:`

Deze foutmelding spreekt meestal voor zich:

```
ERROR, comma or ) expected:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2  & Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
-------------------------------------------------------------------
-----------------------------------^
```

Hier is het sluithaakje na `plaats2` vergeten.

In onderstaande formule is echter nauwelijks te achterhalen wat de oorzaak van de foutmelding is:

```
ERROR, comma or ) expected:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   ->( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
-------------------------------------------------------------------
-------------------------------------------------------------------
-----------------------------------^
```

De spatie na `->` is vergeten.

▶ `ERROR, '=' or '!=' expected:`

In clauses kan de oorzaak een verkeerde of verkeerd gebruikte operator zijn:

```
ERROR, '=' or '!=' expected:
    ( p(b,p(a,b)) =! e ).
---------------------^
end_of_list.
```

OTTER kent geen operator `=!`.

```
ERROR, '=' or '!=' expected:
         ( p(b,p(a,b))=e ).
-------------------------^
```

De spaties rondom = zijn vergeten, zodat deze operator niet werd herkend.

Wanneer deze foutmelding in formules voorkomt, kan de oorzaak zijn dat een `list(...).` is gedeclareerd in plaats van `formula_list(...).` .

```
ERROR, '=' or '!=' expected:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) & Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
--------------^
```

Hier werd tussen `all` en `plaats1` één van de betreffende symbolen verwacht.

Ook kan het zijn dat `end_of_list.` vergeten is als afsluiting van `list(...).` . In dat geval wordt een volgend commando `formula_list(...).` niet gezien als commando, maar als clause en de daarbij behorende formules zullen in het algemeen niet aan de verwachte grammatica voor clauses voldoen.

► `ERROR, logical operator, '=', or '!=' expected:`

Deze foutmelding is het equivalent van de vorige voor formules.

```
ERROR, logical operator, '=', or '!=' expected:
(all plaats1  all plaats2  all plaats3
  (   ( Verbinding(plaats1,plaats2) &Verbinding(plaats2,plaats3) )
   -> ( Verbinding(plaats1,plaats3) | $Ans_via(plaats2) ) )  ).
------------------------------------------------------------------
-----------------------------------^
```

De spatie na & is vergeten, zodat deze niet als logische operator werd herkend.

► `ERROR, command not found:` *commando*

Het vergeten van `formula_list(...).` of `list(...).` leidt in het algemeen tot een reeks van foutmeldingen, eindigend met:

```
ERROR, command not found: end_of_list.
```

mits natuurlijk `end_of_list.` niet óók vergeten is.

A.8.3 Invoerfouten, die geen foutmeldingen geven

Ook als OTTER geen grammaticale fouten kan ontdekken, is het mogelijk dat uit een verzameling formules c.q. clauses de lege clause nìet afgeleid wordt, ondanks het feit dat dit op het oog wel wordt verwacht.

In dergelijke gevallen is het bij het gebruik van de clause-vorm verstandig de namen van de variabelen en constanten eens na te lopen op hun beginletters (zie Paragraaf A.3.1).

Bij het gebruik van formules moet met name op de namen van de constanten gelet worden; hierbij kan de vertaling naar clauses gemakkelijk gebruikt worden (zie Paragraaf 1.2.4). Bovendien moet erop gelet worden dat alle variabelen daadwerkelijk gekwantificeerd zijn Ook dit kan in eerste instantie het gemakkelijkst in de vertaling gecontroleerd worden: niet-gekwantificeerde namen kunnen onbedoeld nìet naar `x1`, `x2`, ... vertaald worden.

A.9 De belangrijkste opties en hun defaults

```
clear(binary_res).
clear(hyper_res).
clear(neg_hyper_res).
clear(ur_res).
clear(demod_out_in).
clear(factor).

set(print_kept).
set(print_back_sub).
set(print_given).
set(demod_history).
set(print_proofs).
set(back_sub).
set(order_hyper).

assign(max_proofs,1).
assign(max_seconds,0).
assign(max_gen,0).
assign(max_literals,0).
assign(stats_level,2).
```