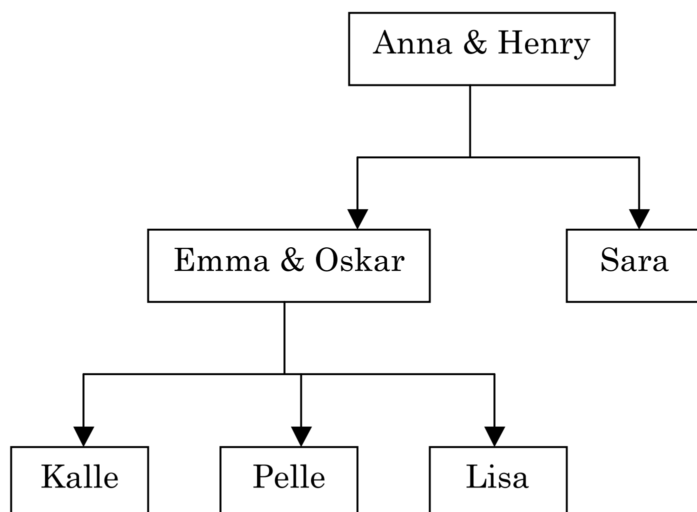
Prova på-laboration i Prolog

Peter Dalenius
petda@ida.liu.se
Institutionen för datavetenskap, Linköpings universitet
2006-09-12

1. Introduktion till Prolog

Programspråket Prolog konstruerades i början av 1970-talet och den första interpretatorn lanserades 1972. Namnet Prolog kommer av *programming logic* och antyder att det handlar om programmering med hjälp av logiska formler. Språket väckte inget större intresse förrän i början av 1980-talet när ett stort japanskt forskningsprojekt kring intelligenta maskiner bestämde sig för att använda Prolog. Då började forskare inom framför allt grenar av artificiell intelligens intressera sig för språket.

Prolog är ett *deklarativt* programspråk. Det innebär att man talar om vilka förutsättningar som gäller snarare än hur problemet ska lösas. Prolog tar sedan själv hand om att lösa problemet på lämpligt sätt. Ett Prolog-program består av *fakta* och *regler*. Fakta talar om vad som gäller i specifika fall och regler talar om vad som gäller allmänt. Som första exempel ska vi analysera nedanstående släktträd.



Fakta

Anna och Henry har två barn, Oskar och Sara. Oskar har tillsammans med Emma barnen Kalle, Pelle och Lisa. Denna information kan vi i Prolog representera med följande fakta:

```
woman(lisa).
woman(sara).
woman(emma).
woman(anna).
```

```
man(pelle).
man(kalle).
man(henry).
man(oskar).

father(henry, oskar).
father(henry, sara).
father(oskar, kalle).
father(oskar, pelle).
father(oskar, lisa).

mother(anna, oskar).
mother(anna, sara).
mother(emma, kalle).
mother(emma, pelle).
mother(emma, lisa).

married(henry, anna).
married(oskar, emma).
```

Alla fakta skrivs in med små bokstäver. Först på varje rad står vad det är för typ av fakta, därefter vilka personer som är inblandade och till sist en punkt. När vi skriver

```
father(henry, oskar).
```

menar vi att Henry är far till Oskar och inte tvärtom. Vi får själva komma ihåg att det är den ordningen på personer som gäller.

Namnen och relationerna (t.ex. `kalle`, `father`) kallas för *atomer* och skrivs som sagt alltid med små bokstäver. En hel rad som uttrycker ett faktapåstående eller en regel kallas för en *struktur* och atomen som står först kallas *funktor*. Atomerna inom parentes är *argument*.

Regler

Reglerna i ett Prolog-program uttrycker vad som gäller allmänt. Till exempel kan man säga att en person P är förälder till en annan person C om P är far till C eller P är mor till C. Detta kan vi i Prolog skriva så här:

```
parent(P, C) :- father(P, C); mother(P, C).
```

Först på raden skriver vi vad det är vi vill definiera, i det här fallet en regel för när P är förälder till C. Efter tecknen `:-` skriver vi vilka förutsättningar som gäller. Tecknet `;` (semikolon) betyder *eller* och regeln avslutas med punkt. I reglerna använder vi *variabler*, symboler som kan stå för vilken person som helst, och dessa skrivs med stora bokstäver.

Med hjälp av regeln för föräldrar kan vi definiera en regel för när en person är bror till en annan:

```
brother(X, S) :- parent(P, X), parent(P, S), man(X), (X \= S).
```

En person X är bror till S om det finns en person P som är förälder till både X och S och X är en man. Tecknet , (komma) betyder *och*, dvs alla fyra villkoren måste vara uppfyllda samtidigt. Den sista delen av reglen uttrycker att man inte kan vara bror till sig själv, dvs X är inte lika med S. Samtliga regler i exemplet med släktträdet ser ut så här:

```
parent(P, C) :- father(P, C); mother(P, C).

child(C, P) :- parent(P, C).

brother(X, S) :- parent(P, X), parent(P, S), man(X), (X \= S).
sister(X, S) :- parent(F, X), parent(F, S), woman(X), (X \= S).

son(X, P) :- man(X), parent(P, X).
daughter(X, P) :- woman(X), parent(P, X).

grandfather(G, C) :- father(G, P), parent(P, C).
grandmother(G, C) :- mother(G, P), parent(P, C).
```

2. Att använda Prolog

När man använder Prolog skriver man ett program bestående av fakta och regler. Därefter laddar man in programmet i Prolog och kan ställa frågor av typen *Vilka bröder har Lisa?* För den här laborationen kommer vi att använda oss av ett Prolog-system som heter SICStus[1].

Innan du startar SICStus Prolog bör du kopiera exempelfilen med släktträdet från kurskontot till din hemkatalog så att du kan utöka den. Filen heter

```
~TDDC66/pp/relatives
```

Starta sedan SICStus Prolog genom att i ett skalfönster skriva kommandot

```
~TDDC66/pp/prologdemo.sh
```

Detta kommer öppna ett Emacs-fönster som är delat i två delar. I den övre bufferten är det tänkt att du ska skriva in dina Prolog-program. I den undre bufferten interagerar du med Prolog och ställer dina frågor.

Se till att markören är i den övre bufferten. Öppna filen `relatives` så att du har möjlighet att ändra i den senare. Byt till den undre bufferten och ladda in filen relatives i Prolog genom att skriva `[relatives].` vid prompten. Observera punkten i slutet! Om du placerat filen i en underkatalog kan du ladda den genom att t.ex. skriva `['~/prolog/relatives'].`

[1] Du kan läsa mer om SICStus på http://www.sics.se/sicstus. Det är dock inte på något sätt nödvändigt för att klara denna laboration.

```
emacs@astmatix.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

% Prova på-laborationer i Prolog

% Peter Dalenius, petda@ida.liu.se
% 2004-06-09

% Fakta

woman(lisa).
woman(sara).
woman(emma).
woman(anna).
woman(ebba).  % Övning 1

man(pelle).
man(kalle).
-0:-- relatives        (Fundamental)--L1--Top-----------------------------------
SICStus 3.8.4 (sparc-solaris-5.7): Mon Jun 12 18:41:59 MET DST 2000
Licensed to ida.liu.se
| ?- ['~/prolog/relatives'].
{consulting /home/petda/prolog/relatives...}
{consulted /home/petda/prolog/relatives in module user, 10 msec 6480 bytes}

yes
| ?- father(X, lisa).

X = oskar ?

yes
| ?-

-0:** *prolog*         (Inferior Prolog:run)--L13--All----------------------------
```

Prolog visar upp en prompt `| ?-` där du kan skriva in dina frågor. Frågorna skrivs på samma sätt som fakta och regler. Om du undrar vem som är far till Lisa kan du skriva `father(X, lisa)`. En sådan fråga kallas i Prolog för ett *mål*. Prolog kommer då att söka igenom de fakta och regler som finns inladdade för att se om det finns någon person som kan matcha X. Om någon sådan person hittas skrivs detta ut. Därefter undrar Prolog som du vill söka vidare för att eventuellt hitta fler träffar. Tryck *Enter* för att avsluta sökningen. (Se körexempel nedan.)

På samma sätt kan du hitta Lisas föräldrar genom frågan `parent(X, lisa)`. Här borde det finnas flera svar, så skriv in `;` (semikolon) och tryck *Enter* efter varje svar. Till slut kommer Prolog att svara `no`, dvs det finns inga fler svar. (Se körexempel nedan.)

```
| ?- father(X, lisa).

X = oskar ?

yes
| ?- parent(X, lisa).

X = oskar ? ;

X = emma ? ;

no
```

```
| ?- brother(X, lisa).

X = kalle ? ;

X = pelle ? ;

X = kalle ? ;

X = pelle ? ;

no
| ?-
```

I exemplet ovan har vi också frågat efter Lisas bröder och får svaren Kalle och Pelle två gånger. Varför? Om vi tittar tillbaka på reglen för bröder så ser vi att X är bror till S om det finns en person P som är förälder till dem båda. När Prolog söker efter möjliga personer som kan vara X, S och P kommer den att märka att både Oskar och Emma kan vara personen P. Syskonen har ju två föräldrar och alltså får vi svaren två gånger.

Pröva själv att ställa några enkla frågor för att se att det fungerar. Du kan t.ex. fråga `child(X, emma).` eller `grandmother(X, pelle).`

För att avsluta Prolog ser du först till att markören är i den undre bufferten. Därefter trycker du Control-C två gånger för att få upp prompten `Prolog interruption (h for help)?` Där skriver du in `e` som i *Exit* och trycker *Enter*. Därefter kan du avsluta Emacs på vanligt sätt.

3. Övningar

Lös följande problem! Observera att varje gång du har ändrat något i filen `relatives` måste du spara den och ladda in den på nytt i Prolog.

Övning 1

Emmas föräldrar heter Peter och Ebba. Lägg till dessa fakta i databasen! Kontrollera att du sedan får två olika svar på frågan vem som är farfar/morfar till Kalle! (När man lägger till nya fakta bör man göra det i anslutning till de gamla så att alla fakta med samma funktor står tillsammans.)

Övning 2

Sara har gift sig med Jonas och de har fått en son som heter Edvin. Lägg till dessa fakta i databasen! Konstruera en regel för kusiner:

```
cousin(X, Y) :- ...
```

Kontrollera att du formulerat regeln rätt genom att fråga efter dels Edvins kusiner, dels Lisas kusiner.

Tips: Det kan underlätta om du skapar hjälpregler för till exempel syskon.

Övning 3

Konstruera en regel för farbror/morbror, dvs en man som är bror till en förälder eller är gift med en syster till en förälder:

```
uncle(X, Y) :- ...
```

Tips: Du kan uttrycka detta som två olika regler med samma vänsterled.

4. Problemlösning i Prolog: Ett exempel

Vi kan också använda Prolog för att lösa lite större logiska problem. Som exempel ska vi ta en gåta där vi har följande information given:

a) Tre män, Brown, Smith och Jones arbetar som läkare, advokat och lärare, men inte nödvändigtvis i den ordningen.
b) Läraren, som är sina föräldrars enda barn, tjänar minst.
c) Smith, som är gift med Browns syster, tjänar mer än advokaten.

Uppgiften är att lista ut vem av de tre männen som har vilket yrke. Detta borde inte vara något större problem för oss, med hjälp av uteslutningsmetoden och lite klurande, men hur skulle Prolog kunna göra? Vi börjar med att tala om för Prolog att vi har tre personer:

```
person(brown).
person(smith).
person(jones).
```

Därefter måste vi få Prolog att kunna lista ut vem som arbetar med vad. Vi hittar på en regel `works_as` och matar in grundförutsättningarna:

```
works_as(doctor, X) :- person(X).
works_as(lawyer, X) :- person(X).
works_as(teacher, X) :- person(X).
```

Dessa regler säger än så länge enbart att det måste vara en av de tre personerna i vår faktabas som har respektive yrke. Om vi ställer frågan `works_as(doctor,X)` får vi mycket riktigt alla tre personerna som möjliga svar. Nu måste vi alltså försöka koda resten av uppgifterna i Prolog för att få unika svar på alla tre frågorna. Vi börjar med att koda uppgifterna om att vissa har syskon och andra inte. Brown har åtminstone en syster och därför skriver vi:

```
has_sibling(brown).
```

Den person som arbetar som lärare har inga syskon och därför modifierar vi regeln för lärare till:

```
works_as(teacher, X) :- person(X), \+has_sibling(X).
```

Om vi nu frågar vem som är lärare utesluts i alla fall Brown. Nästa steg är att inkludera information om hur mycket pengar de olika personerna tjänar. Från ledtrådarna får vi veta att Smith tjänar mer än advokaten och att läraren tjänar minst. Smith borde alltså tjäna mest och vi lägger till följande faktum:

```
earns_most(smith).
```

Den som tjänar mest kan varken vara lärare eller advokat, alltså utökar vi regeln för doktor så här:

```
works_as(doctor, X) :- person(X), earns_most(X).
```

Svaret på frågan om vem som är doktor blir nu, föga förvånande, Smith. De andra frågorna får dock fortfarande flera svar. På frågan om vem som är advokat kan vi, med våra nuvarande formuleringar av reglerna, inte utesluta någon. Vi vet dock något som inte Prolog vet, nämligen att alla tre personera ska ha varsitt unikt yrke. Den som är advokat kan inte samtidigt vara läkare eller lärare. Vi lägger till denna information till reglerna, så att de ser ut så här:

```
works_as(doctor, X) :- person(X), earns_most(X).
works_as(lawyer, X) :- person(X), \+works_as(doctor, X),
    \+works_as(teacher, X).
works_as(teacher, X) :- person(X), \+has_sibling(X),
    \+works_as(doctor, X).
```

Med dessa regler kan vi ställa frågor om vem som är läkare, advokat och lärare och få unika svar på alla frågorna. För detta lilla exempel känns det kanske onödigt att blanda in Prolog, eftersom vi i princip löser hela problemet enbart genom att formulera reglerna, men det ger i alla fall en övning i att översätta ett problem från informell text till mer formell programkod.

Några kommentarer kring hur Prolog resonerar

När Prolog försöker svara på våra frågor och dra slutsatser ur de fakta och regler som vi definierat används ett antagande som brukar kallas *closed world assumption*. Det säger i princip att om man inte kan vare sig bevisa eller motbevisa ett påstående så gäller det inte. Detta är en nödvändig förenkling för att Prolog överhuvudtaget ska kunna komma fram till något. Antagandet säger egentligen att vår beskrivning av ”världen”, dvs våra fakta och regler, är fullständiga. Om vi inte kan vare sig bevisa eller motbevisa `works_as(doctor, jones)` så antar vi att det inte gäller.

Detta får till följd att regler som innehåller *inte* (dvs `\+`) behandlas lite speciellt. Om det i en regel står `\+has_sibling(X)` betyder det alltså inte att vi ska leta efter ett explicit faktum som säger att det inte gäller, utan att vi ska försöka bevisa det, men misslyckas. Om man inte tar hänsyn till att Prolog arbetar enligt den här metoden hamnar man lätt i problem när man ska formulera regler.

5. Fler övningar

Övning 4

Kopiera filen som innehåller reglerna från avsnittet ovan till ditt konto (eller ladda in den direkt i Prolog). Filen heter

```
~TDDC66/pp/works
```

Ställ i tur och ordning frågan vem som arbetar som läkare, advokat och lärare. Vad blir svaren? Kan man vända på frågorna och istället fråga efter vad Brown, Smith och Jones har för yrken?

Övning 5*

Följande lite svårare problem är en klassisk gåta där det gäller att ta fasta på alla fakta.

På ett tåg består personalen av Andersson, Petterson och Lundström. En av dem är eldare, en bromsare och en lokförare. Bland passagerarna på tåget finns tre affärsmän som har samma namn som de tre i personalen. För att skilja dem åt kallar vi affärsmännen herr Andersson, herr Petterson och herr Lundström. Följande fakta är kända:

a) Herr Pettersson bor i Göteborg.
b) Bromsaren bor exakt mitt emellan Göteborg och Stockholm.
c) Herr Lundström tjänar exakt 100 000 kr om året.
d) Bromsarens närmaste granne, en av de tre passagerarna, tjänar exakt tre gånger så mycket som bromsaren.
e) Andersson slår eldaren i biljard.
f) Passageraren med samma namn som bromsaren bor i Stockholm.

Formalisera ovanstående påståenden med hjälp av Prolog och svara på frågan vem som är lokförare.

6. Referenser

Framtida kurser

Det sätt som man formulerar regler på i Prolog liknar till stor *predikatlogik* och det kommer du läsa mer om i kursen *Logik* i årskurs 2. Där kommer du också att lära dig mer om *resolution*, den metod som Prolog använder för att ta reda på svaren på de frågor som ställs till systemet. Det finns också en särskild kurs *Logikprogrammering* som går in på djupet i Prolog. Den kursen kan du läsa i årskurs 3 eller 4. Att kunna hantera logiska formler och förstå hur de kan användas för olika syften inom datavetenskap är en viktig förkunskap till flera andra kurser, bl.a. *Artificiell intelligens*.

Litteratur

I den rekommenderade referensboken *Computer Science: An Overview* av J. Glenn Brookshear är särskilt avsnitt 6.7 intressant för denna laboration.

Du kan läsa mer om Prolog och logikprogrammering i t.ex. kapitel 14 i *Concepts of Programming Languages* av Robert W. Sebesta.

En bra resurssida på nätet som innehåller många hänvisningar till relevant information är http://vl.fmnet.info/logic-prog/

7. Frågor att fundera över

Som du kanske förstår är Prolog inte ett språk som man i första hand använder för att skriva större applikationer. Det kan dock fungera bra som en specialiserad del i ett större system. En av fördelarna med Prolog och deklarativa språk jämfört med många andra vanligare språk är att man tack vare att språket är baserat på logik mycket lättare kan verifiera och ibland kanske till och med bevisa att ett program verkligen är korrekt. Kan du komma på något sammanhang där man skulle kunna ha nytta av Prolog för att dra slutsatser och där kraven på att programmet verkligen drar korrekta slutsatser är höga? Vilka typer av uppgifter tror du att Prolog skulle kunna lösa i ett sådan sammanhang.

I de flesta vanliga programmeringsspråk beskriver man *hur* man löser ett problem. I Prolog beskriver man istället själva problemet och vilka förutsättningar som gäller. Prolog-systemet hittar lösningen och vi som använder det har inte så stor kontroll över exakt hur detta går till. Det enda vi vet är att Prolog söker genom fakta och regler uppifrån och ner och från vänster till höger i filen. Hur tror du att det påverkar programmens effektivitet? Har du några idéer kring hur man kan formulera fakta och regler för att Prolog lättare ska kunna hitta rätt svar?