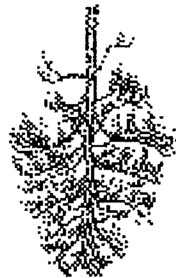
En introduktion till programmering i

PROLOG



Robert Eklund

Denna sida avsiktligen blank

En introduktion till programmering i

PROLOG

Robert Eklund

Femte, något ännu mer omarbetade och (ytterligare en gång till igen!) långt ifrån slutgiltiga, upplagan

© Robert Eklund 1996

Denna sida avsiktligen blank

INNEHÅLL

Förord 5

1 Introduktion 9

2 Databaser 19

3 Cut, True och Fail 27

4 Rekursion och listor 29

5 Avlusning ("Debugging") 39

6 Hur man skriver och kommenterar program 45

7 Input/output – Läsning från skärmen 49

8 IF-THEN-ELSE-strukturer 51

9 Aritmetik 57

10 Unifiering 61

11 Retract/assert – dynamisk ändring av databaser 63

12 Subrutiner 65

13 Generativ grammatik 73

14 Parsning 79

15 Definite Clause Grammar (DCG) 93

16 Differenslistor 97

17 Filläsning och filskrivning 103

Terminologi 107

Litteratur 113

Index 115

FÖRORD

Detta kompendium är ämnat som en grundläggande introduktion till programmeringsspråket PROLOG.[1] Eftersom det operativa ordet här är "grundläggande" så förstås att kompendiet inte har några anspråk på att tillfredsställa en professionell *hackers*[2] alla lustar. Särskild hänsyn har i stället tagits till de personer vilka inte har någon som helst tidigare programmeringserfarenhet. Detta innebär att personer som redan är förtrogna med andra programmeringsspråk kan komma att tycka att framställningen till viss del och i någon mening är trivial (och måhända på gränsen till felaktig, en fara vid alla försök att förenkla). Det innebär också att mycket är utelämnat, och att således personer som redan kan prolog kan komma att utbrista "Men varför tog du inte med det här?!". Jag har försökt att ta med sådant som oundgängligen utgör ett slags bas för att gå vidare. Det som har utelämnats är givetvis inte oviktigt, utan sådant som inte krävs för att kunna leka och ha kul med prolog som första bekantskap. Om man berättar allt på första träffen så finns ju inga hemligheter kvar att upptäcka!

Syntaxen i prolog skiljer sig en smula från andra programmeringsspråk som t ex PASCAL och LISP, och man skulle kunna säga att medan man talar i *imperativ* i dessa båda språk, så talar man i *indikativ* i prolog. Detta innebär att snarare än att säga "gör det här", så säger man "så här vill jag att det skall se ut" och så tar prolog självt hand om vad som behöver göras för att nå dithän. Detta kanske inte säger så mycket, men en poäng här är att det på grund därav kan vara en fördel att *inte* känna till något annat programmeringsspråk när man börjar med prolog, eftersom man då inte har någon förutfattad mening om hur det skall fungera. Man brukar säga att prolog är ett *deklarativt* språk, medan språk som PASCAL, LISP, C med flera kallas *procedurella*. Just denna skillnad brukar vara det som bjuder den erfarne PASCAL-programmeraren på problem, eftersom det kräver ett annat slags sätt att tänka när man skriver på program. För att ge en lite svepande förklaring till vad skillnaden mellan deklarativ och procedurell innebär i detta sammanhang kan man säga att medan man i språk som pascal för att lösa ett problem måste specificera hur problemet skall lösas i detalj och steg-för-steg, så räcker det i prolog med att beskriva hur slutresultatet kommer att se ut, varvid prolog självt kommer att klara av detaljerna som krävs för att nå dithän.

Ovanstående förklaring kanske inte säger så mycket, och för att förstå skillnaden krävs det egentligen att man själv arbetar med prolog och på så sätt får se hur man arbetar och tänker vid prolog-programmering.

Programmeringsspråk, liksom naturliga språk, existerar i en mängd dialekter, och det finns ingen 'COMMON PROLOG' på samma sätt som det finns en COMMON LISP. Detta leder till att vissa så kallade *predikat* (dvs, de "ord" man arbetar med i prolog) kan ha olika betydelser och/eller effekter i olika prologer. Vidare skillnader är att vissa prologer accepterar att *å*, *ä* och *ö* används i koden – såsom MACPROLOG och AAIS PROLOG (även den för Macintosh) – medan andra, t ex ARITY, inte gör detta.[3]

Vad är då prolog? Prolog är ett *programmeringsspråk*, och tillhör således samma familj som andra programmeringsspråk som PASCAL, BASIC, C, LISP, FORTRAN med flera. Ett programmeringsspråk skiljer sig från ett *program* (som t ex WORD, SUPERPAINT, EXCEL osv) på så sätt att inget är "färdigt" från början utom "verktygslådan". I själva verket är program (som ovan nämnda) skrivna i programmeringsspråk (som ovan nämnda).

---

[1]Jag kommer hädanefter att skriva "prolog" med gemener.

[2]Folk med fyrkantiga ögon och fingrar som är utnötta upp till åtminstone andra fingerleden!

[3]En liten varning kan emellertid utfärdas här. Vissa prologer accepterar *å*, *ä*, *ö* men inte i alla situationer. Således har det rapporterats för mig att MACPROLOG inte accepterar *å*, *ä*, *ö*, som första bokstav i predikatnamn. Om något inte fungerar kan det vara värt att kolla om det är våra svenska bokstäver som ställer till problem.

Ett programmeringsspråk använder man för att få datorn att göra en mängd saker man vill att den skall göra åt en. Ett program som man köper har en redan väldefinierad funktion, det kan t ex vara ett kalkyl- eller ordbehandlingsprogram, medan ett programmeringsspråk tillåter en att be datorn om vilka tjänster som helst. Detta kräver dock, i någon mån, mer kunskaper än vid användandet av ett program (även om de kan vara nog så komplicerade understundom!).

Språk – naturliga som artificiella – innehåller en mängd ord som används för att uttrycka sig, men också möjligheter att skapa nya ord.[4] På detta sätt skiljer sig prolog inte från andra programmeringsspråk.

De "ord" – i prolog benämnda *predikat* – som prolog redan tillhandahåller kallas *predefinierade*, dvs de finns redan färdiga att använda och behöver inte specificeras av användaren. (Mera därom senare.) Vilka dessa är kan variera från prolog till prolog, och man får kontrollera vilka predikat ens egen prolog innehåller. Dessutom har man – som i andra programmeringspråk – gränslösa möjligheter att skapa nya, egna, predikat som gör det man vill.

Ett exempelprogram kan åskådliggöra hur detta fungerar (predikaten – och andra detaljer – kommer att nämnas senare i kompendiet igen). Programmet som sådant behöver naturligtvis inte förstås nu, dit kommer vi senare!

Vi *definierar* (skapar) predikatet **hej** som skriver ut en hälsning på skärmen:

```
hej :-
  nl,
  write('Hej och välkommen till prolog!'),
  nl.
```

Här har vi själva skapat ett eget, nytt, predikat **hej**, och i detta använt oss av de bägge predefinerade predikaten **nl**, som skriver ut en radmatning på skärmen, och **write**, som skriver ut det man själv skriver in i *argumentparentesen.*

I kompendiet kommer ett antal användbara predefinierade predikat, såväl som egendefinierade predikat, att användas. Vad som är vad kommer att nämnas, även om vissa predikat inte alltid är predefinierade i alla prologer, utan måste skrivas in.

Tack till...

Jag vill passa på att tacka den hela hoper personer som kommit med (all!) slags kritik och kommentarer till detta ständigt växande kompendium genom åren: Lärare, kurskamrater, studenter och (datalogiska) vänner har på många sätt bidragit. En mycket flitig vidräknare den senaste tiden har varit Nikolaj 'Mr Parser King' Lindberg, som här får ett speciellt tack.

```
tack_till :-
  write('Tack, Niko!!'), nl.
```

---

[4]Termer och begrepp kommer att repeteras i slutet av varje kapitel. I övrigt hänvisas till avsnittet **Terminologi** i slutet av kompendiet, där ett antal termer och begrepp som förekommer i samband med prolog förklaras.

Nya termer och begrepp

programmeringsspråk
hacker
syntax
imperativ
indikativ
predikat
PASCAL
BASIC
C
LISP
FORTRAN
program
WORD
SUPERPAINT
EXCEL
predefinierad
nl
write
argumentparentes
deklarativ
procedurell

Några kommentarer rörande typsnitt

För att underlätta läsandet av detta kompendium har olika typsnitt valts för olika ändamål. Dessa presenteras nedan.

Löptext anges (som denna mening) i Times mager rak.

Programkod och **programpredikat** anges i **Monaco fetstil.**

Programkommentarer ges i Monaco mager rak.

Nya **termer** (som kan vara bra att kunna) anges i *kursiv stil.*

Kommentarer i programkod (dvs, saker som skall "fyllas i") anges i *Times 10 punkt kursiv stil.*

Beskrivningar av argument i strukturer anges inom <vinkelparenteser>.

1 INTRODUKTION

Här skall prologs grundläggande strukturer och terminologi förklaras. Medan förståelse av strukturerna är nödvändig för att kunna programmera i prolog så är terminologin som sådan naturligtvis inte avgörande för ens förmåga att hantera prolog. Om man däremot också vill kunna *tala* om sina program (och förstå vad en lärare säger!) så är det en fördel att känna till terminologin. Därvidlag skiljer sig ju inte prolog från vilket annat ämne som helst.

Vi börjar med ett enkelt exempel.

```
man(bertil).
```

Det första att lägga märke till är *punkten*. Alla prologuttryck avslutas med en punkt. Det andra att lägga märke till är att det inte finns något mellanslag mellan **man** och den inledande vänsterparentesen. Detta säger helt enkelt att det finns en man som heter Bertil, och ser ut exakt som det skulle göra i predikatlogik: *man(bertil)*. **man** i detta sammanhang benämnes (som i predikatlogik) *predikat*, och skrives med liten initialbokstav.[1] Att det liknar predikatlogik är ingen slump. I själva verket står prolog för *PROgramming in LOGic*,[2] och är baserat på första ordningens predikatlogik.[3] Kutym är att sammansatta predikatord skrivs med understrykningslinje:

```
en_man(hilding).
```

Predikat tar – igen som i predikatlogik – *argument*. I fallet:

```
man(bertil).
```

... är **bertil** argument till predikatet **man**. Man säger, i paritet med predikatlogik, att predikatet **man** är *enställigt*, vilket betyder att predikatet har *ett* argument. *Ställigheten* – eller *ariteten*, som det brukar kallas i prolog – anges ofta medelst ett snedstreck och en siffra. Sålunda hänvisar vi till predikatet **man** på följande sätt: **man/1**. Lägg märke till att siffran aldrig skrivs in i själva koden, utan bara används vid namngivandet av ett predikat för att underlätta identifieringen av detta.

I det följande kommer alla predikat som nämns att skrivas på detta sätt. En orsak till att vara noga med detta är att det är möjligt i prolog att använda samma predikatnamn flera gånger, och ofta är det just ställigheten som skiljer olika 'versioner' av predikat åt, varför ett predikat **foo/1** kan vara ett helt annat predikat än **foo/2**.[4] Generellt kan strukturen alltså sägas vara:

predikat($argument_1$, $argument_2$, ... , $argument_n$).

Vad gäller argumenten **bertil** och **hilding** som sådana, så benämnes de här *atomer*. Atomer, även kallade *konstanter*, är sådant som har ett givet och "odelbart" värde, och initialbokstaven skrivs med liten bokstav (vilket leder till att personnamn också måste

---

[1] I själva verket är det hela uttrycket som betecknas predikat.

[2] Eller, på franska *PROgrammation et LOGique*.

[3] Att känna till grunderna i logik (sats- och predikat-) underlättar förståelsen av prolog betydligt. Jag kommer i detta kompendium inte att presentera logik. För en introduktion till satslogik och predikatlogik se t ex Allwood, Andersson och Dahl: *Logic in linguistics*, Cambridge University Press, Cambridge 1977.

[4] I alla programmeringsböcker brukar man skriva *foo* som namn på allt som det inte spelar någon roll vad det heter. Etymologiska förklaringar till detta ord råder det ingen brist på, men de är sällan överens. Jag dristar mig inte till att ange någon här.

stavas med gemener). Om man nödvändigtvis vill stava Bertil med stor bokstav måste det "citeras":

```
man('Bertil').
```

Lägg märke till att citationstecknen är enkla, enligt engelskt mönster. Sådana små detaljer är av yttersta betydelse för att prolog skall förstå vad man säger, helt enkelt! Hela *uttrycket*:

```
man(bertil).
```

... benämnes ett *faktum* (*fakta* i plural).

Vi vill nu använda prolog. För att göra det så definierar vi predikatet **pappa/2**. Vi skapar ett faktum:

```
pappa(hilding,robert).
```

Var skriver man då detta?

Hur man arbetar med prolog varierar från system till system och prolog till prolog, men även om olika prologer tillhandahåller olika verktyg för körningar, så är grundprincipen ändå densamma: man arbetar dels med en *regelfil*, dels i en *interpretator* som tolkar reglerna man skrivit. Vissa prologer som AAIS och MacProlog (båda för Macintosh) och Arity Prolog (för PC) tillhandahåller egna 'ordbehandlare' så kallade *editorer*, medan andra, som SICStus (för minidatorer under UNIX) kräver att reglerna skrivs in i separata textbehandlare. Reglerna läses sedan från prologen.

En interpretator tolkar helt enkelt de regler man skrivit in, och tillåter användaren att testa dem. Innan reglerna tolkats har de samma värde som vilken text som helst i en ordbehandlingsfil.

Vissa prologer tillhandahåller även en *kompilator*, som inte bara tolkar reglerna, utan också skriver om dem till maskinkod (datorns eget programmeringsspråk), vilket gör att programkörningar går mycket fortare, eftersom de arbetar på en mer "grundläggande" nivå i datorn. Kompilerade filer finns oftast inte tillgängliga för läsning (dvs, man kan inte öppna dem i ett ordbehandlingsprogram), och även om så vore fallet skulle de inte säga så mycket.

När man arbetar med prolog skriver man alltså först in ett antal regler i en regelfil (editor), interpreterar sedan dessa (olika prologer har olika handgrepp för att göra detta) och testar – eller *evaluerar* – reglerna i interpretatorn. Således arbetar man så att säga på två ställen enligt följande figur:

Regelfil	Interpretator
pappa(hilding,robert).	?- pappa(X,robert). X = hilding yes ?- pappa(hilding,robert). yes ?- pappa(robert,X). no

Vi skall här visa hur det går till, steg-för-steg.

Vi börjar med att bestämma oss för att skriva in en *släktdatabas*. Alltså öppnar vi editorn, och skriver in de regler som vi vill använda. Vi skriver in en enda regel, den ovan nämnda:

pappa(hilding,robert).

Detta läses som:

Hilding är pappa till Robert

... eller:

Relationen 'pappa' råder mellan Hilding och Robert

Vi skriver in detta i regelfilen.

Regelfil

```
% SLÄKT.RUL

pappa(hilding,robert).
```

Vi sparar filen som (dvs, döper filen till) **släkt.rul**,[5] och skriver – med *kommentartecken* – namnet på filen högst upp i regelfilen, för att lätt kunna se vilken fil vi tittar på. Kommentartecknet % gör att raden som följer inte läses av prolog, dvs inte tolkas som en del av programmet (mera om kommentering i kapitel 6). När detta är gjort går vi över till interpretatorn. (Detta kan ske på olika sätt i olika prologer.) Väl där – i det så kallade *Query Mode* (≈ 'frågeläge') – möts vi av en *frågepromt* som har utseendet:

?-

Vi arbetar nu således med två fönster: ett regelfönster och interpretatorn, där frågeprompten väntar på att vi skall göra något.

---

[5]Regelnamnen är givetvis avhängiga vilken dator och vilken prolog man arbetar med. Sålunda kräver ju PC att namnet har en extension, vilket Macintosh eller UNIX inte gör. Observera även att man inte kan döpa filen som **släkt.rul** om ens prolog inte accepterar *å, ä, ö*.

Regelfil | Interpretator

Regelfil	Interpretator
% SLÄKT.RUL % Regelfil för min släkt. pappa(hilding,robert).	?-

Väl i interpretatorn läser man in filen med hjälp av predikaten **consult/1** eller **reconsult/1**. (Vissa prologer tillhandahåller mekanismer för automatisk *reconsultning* (dvs inläsning) varje gång man flyttar sig från regelfilen till interpretatorn.) Dessa predikat tar som argument namnet på en regelfil, läser in denna, och interpretatorn är därefter beredd att svara på frågor rörande innehållet.

Skillnaden mellan de bägge predikaten är att **consult** läser in en fil och **reconsult** läser in förändringar av en redan inläst fil. Det innebär att en fil man arbetar med skall *consultas* första gången, men *reconsultas* efter varje förändring.

I detta fall säger vi att regelfilen ovan heter **släkt.rul**. Anropet ser då ut på följande sätt:

```
?-  consult('släkt.rul').
  yes
```

I många prologer läser man in (egentligen reconsultar) filer med följande anrop:

```
?-  ['släkt.rul'].
```

Prolog svarar här med ett **yes**, vilket betecknar att prolog lyckats med uppgiften. Prolog svarar alltid med **yes** eller **no** på alla frågor man ställer. Ibland kan svaren verka lite ointuitiva, vilket kommer att belysas i det följande.

Regelfil	Interpretator
% SLÄKT.RUL % Regelfil för min släkt. pappa(hilding,robert).	?- consult('släkt.rul'). yes

Vi kan nu ställa frågor till prolog, och evaluera reglerna i databasen **släkt.rul**.

Om vi t ex frågar:

```
?- pappa(X,robert).
```

... så svarar prolog:

```
X = hilding
  yes
```

Man säger även att prolog *returnerar* **`hilding`** som svar på frågan vad **`X`** kan vara.

Notera att prolog här svarar **`yes`** på två olika sätt. Genom att visa att ett **`X`** kan hittas så har prolog redan svarat "ja" på uppgiften.

Man kan nu ställa flera olika frågor, vilka illustreras av frågorna och svaren i det högra fönstret i följande figur:

Regelfil

```
% SLÄKT.RUL
% Regelfil för min släkt.

pappa(hilding,robert).
```

Interpretator

```
?- pappa(X,robert).
X = hilding
  yes
?- pappa(hilding,robert).
  yes
?- pappa(robert,X).
  no
?- pappa(hilding,X).
X = robert
  yes
```

Lägg märke till att alla prologuttryck, även frågor, måste avslutas med ovannämnda punkt. Om man skulle glömma punkten och ändå försöka *evaluera* uttrycket (med radmatningstangenten) så radmatas bara, men inget händer. Man kan då lägga till punkten och evaluera igen.

```
?- pappa(X,robert)
 .
```

Prolog svarar då:

```
X = hilding
  yes
```

`X` i frågorna är en så kallad *variabel*, och heter så för att den har ett varierande värde. När vi frågar prolog **`pappa(X,robert).`** så letar prolog igenom de fakta som finns tillgängliga i programfilen för att hitta ett **`X`** som matchar strukturen **`pappa(X,robert)`**. I vårt fall hittas **`pappa(hilding,robert)`**. **`X`** *binds* då till **`hilding`**, eller **`X`** *får värdet* **`hilding`**. Detta benämnes också att **`X`** *instantieras* som **`hilding`**.

Alla variabler börjar med stor bokstav eller understrykningslinje, och allt som börjar med stor bokstav eller understrykningslinje tolkas som variabler, vilket är orsaken till att Bertil tidigare måste skrivas med gemener. I annat fall hade Bertil inte haft något eget värde alls. Sammansatta variabler skrivs enligt kutymen samman, men med varje enskilt ord påbörjat med versal:

```
barn(ÄldsteSon,YngsteSon,ÄldstaDotter,YngstaDotter).

skriv_ut(DetSomSkallSkrivasUt).
```

Om man vill kan man göra ett mellanslag efter varje kommatecken, om man tycker att det blir lättare att läsa variabelnamnen så. Således är:

```
pappa(ÄldsteSon, YngsteSon, ÄldstaDotter, YngstaDotter).
```

... också godkänd *syntax*. Syntax kallas de regler som definierar godkända uttryck i språk, såväl naturliga som programmeringsspråk. Alla de regler som nämns i detta kompendium utgör en del av prologs syntax. Om man skriver något som inte följer prologs syntax så betyder det man skrivit ingenting för prolog, och medan man kan uttrycka sig med bristfällig syntax i naturliga språk och ändå bli förstådd, så är programmeringsspråk notoriskt "dumma", och måste bli tilltalade på exakt rätt sätt för att förstå vad man vill säga.

Variabler kan ha vilka namn som helst,[6] men det är lämpligt att ge dem mnemotekniskt lämpliga namn, dvs namn som betecknar det som de faktiskt är menade att beteckna.

För att visa att variabelnamn är oviktiga kan vi fråga:

```
?- man(Hejsan).
```

Prolog svarar då:

```
Hejsan = bertil
  yes
```

Vi frågar:

```
?- man(_88).
```

... och prolog svarar:

```
_88 = bertil
  yes
```

Om vi frågar:

```
?- man(ADOHIFG).
```

... så svarar prolog:

```
ADOHIFG = bertil
  yes
```

Vi går nu tillbaka till regelfilen och lägger till ett faktum:

```
man(benny).
```

Sedan flyttar vi oss tillbaka till interpretatorn och *reconsultar* den uppdaterade regelfilen. Observera att detta är nödvändigt för att prolog skall kunna se att något hänt i regelfilen.

---

[6]OBS! Det är dock livsviktigt att de behåller sina namn (ett per variabel) *i samma regel*!! Annars kan inte prolog se att det rör sig om samma objekt!! Däremot kan variabler ha olika namn om de ligger i olika regler.

Den regelfil vi "läst in" är fortfarande den förra, och ändringar i regelfilen syns bara efter att reglerna lästs in igen. Om vi nu frågar:

```
?- man(X).
```

... så svarar prolog:

```
X = bertil
  yes
```

Om vi nu vill veta om det finns flera **X** sådana att **X** är män i databasen så skriver vi ett semikolon:

```
;
```

... vilket i prolog har innebörden av logiskt *eller*, dvs ett så kallat *inklusivt eller*, vilket har den ungefärliga betydelsen *och/eller*, eller *A eller B eller båda* (i logisk notation $\vee$).[7] Semikolonet benämnes *operator*. En operator är ett tecken som i prolog har en särskild – ofta logisk – betydelse. Vi kommer i det följande att stöta på flera operatorer, och de kommer att förklaras allteftersom de dyker upp. Vi frågar nu **man(X)**, och skriver ett semikolon efter svaret, och därefter vagnretur:

```
?- man(X).
X = bertil ;
X = benny ;
  no
```

Som tidigare nämnts svarar prolog **yes** eller **no**, vilket säger om programmet lyckats eller inte lyckats. Att prolog svarar **no** här betecknar att prolog inte hittat ytterligare predikat **man** än **bertil** och **benny**. Annars kan ett **no** här verka lite ointutivt, eftersom vi ju just sett att det visst gick att hitta lösningar!

Fakta och predikat kan i princip ha hur många argument som helst, eller inga alls. Vi lägger till ytterligare några tvåställiga:

```
pappa(bertil,sune).
```

... vilket kan utläsas, med ett språkbruk hämtat från logiken:

Det finns en pappa Bertil sådan att denne Bertil är pappa till Sune

Man kan också säga att relationen "pappa" råder mellan Bertil och Sune.

Vi lägger till några fler fakta:

```
pappa(jonas,isak).
pappa(jonas,peter).
pappa(benny,anna).
```

Om vi nu frågar med olika argument givna (dvs instantierade) så kan vi ställa olika frågor:

?- pappa(X,isak).	betyder ungefär:	*Finns det någon pappa till Isak?*
?- pappa(X,Y).	betyder ungefär:	*Vilka pappa-barn-par finns det?*

[7]Det finns även ett "rent", *exklusivt eller* i logik, som betyder *A eller B men inte båda*. Detta skrivs O. Exempel på skillnaden mellan de båda i naturligt språk kan vara *Vill du ha té eller kaffe?* (Exklusivt eller) och *Vill du ha socker eller grädde?* (inklusivt eller).

```
?- pappa(benny, X).
```
betyder ungefär: *Har Benny några barn?*

Vi kan också ställa *sammansatta frågor*. Som tidigare nämnts så betecknar semikolon *och/eller*, och detta kan användas i frågor:

```
?- pappa(X,isak) ; pappa(benny,Y).
```

Denna fråga betyder således någonting i stil med:

Finns det någon pappa till Isak och/eller har Benny några barn?

En viktig sak att känna till är att prolog endast kan uttala sig om sådant som finns i databasen. Vi frågar t ex om jag har en bror:

```
?- bror(X,robert).
  no
```

Detta är nu inte sant, eftersom jag har en bror. Saken är den att detta finns det ingen information om i prologs databas/regelfil. Detta förhållande att bara utgå från en given, specificerad mängd av information benämnes *closed world assumption*, och anger att man bara handhar avgränsade delar av världen och inte kan uttala sig om vad som finns utanför denna delmängd av universum.

Nya termer och begrepp

predikatlogik
predikat
argument
enställigt
ställighet
aritet
predikatnamn
foo
atomer
uttryck
faktum
fakta
regelfil
interpretator
editor
AAIS
MACPROLOG
ARITY
SICSTUS
UNIX
textbehandlare
kompilator
maskinkod
evaluering
kommentartecken
%
frågeprompt
Query Mode
consult/1
reconsult/1
variabel
bindas till
få värde
instantiering
syntax
;
operator
inklusivt eller
exklusivt eller
yes
no
sammansatt fråga
closed world assumption
returnering

Övningar:

- Skriv in olika enkla relationer av typen ovan och ställ olika frågor i frågefönstret. Ändra på fakta, *reconsulta* och fråga på nytt och se att prolog nu lämnar andra svar.

2 DATABASER

Som påpekades i kapitel 1 så lämpar sig prolog alldeles utsökt för *databaser*. Vi kan nu bygga en fylligare släktrelationsdatabas. Faktum är att man kan definiera samtliga släktrelationer utifrån predikaten:

```
man
kvinna
pappa
mamma
```

Det vill säga att alla andra släkttermer kan beskrivas utifrån enbart dessa fyra. I själva verket räcker det med enbart tre predikat, t ex **man**, **pappa**, och **mamma**, om man dessutom använder det predefinierade predikatet **not**, ofta med namnet **\+**, som negerar predikat. Om man negerar **man**, så får man ju **kvinna** (vilket inte skall tolkas som ett inlägg i jämlikhetsdebatten!). Negering av predikat medför emellertid ofta problem av annan natur, och för att undvika de komplikationer som lätt uppstår när man använder **not**, så väntar vi med det till en början.

Först kan vi börja med att lägga in rena fakta i databasen, genom att använda ovannämnda predikat.

```
pappa(hilding,robert).
pappa(hilding,roger).
mamma(ingabritt,robert).
mamma(ingabritt,roger).
man(hilding).
man(robert).
man(roger).
kvinna(ingabritt).
```

Detta säges utgöra den *extensionella* delen av databasen, dvs den samling av relationer som finns i världen man specificerar.

Nåväl, det sades i inledningen av detta kapitel att man kunde definera samtliga släktskapstermer utifrån dessa fyra fakta. Hur gör man då det? I prolog kan man, förutom rena fakta, även definiera *regler* som beskriver hur fakta förhåller sig till varandra. Dessa regler utnyttjar specificerade fakta för att definiera nya relationer. Dessa regler säges utgöra den *intensionella* delen av en databas. Som vi kommer att se formulerar vi reglerna som en mängd *deduktiva* lagar, och använder dem till just att dra slutsatser, eller etablera kunskap, om relationer som inte explicit finns inskriven i databasen.

För att exemplifiera hur detta går till definierar vi predikatet **son**.

```
son(X,Y) :-
   man(X),
   pappa(Y,X).

son(X,Y) :-
   man(X),
   mamma(Y,X).
```

Det första att lägga märke till är två nya operatorer.

Den första operatorn :- utläses som *implikationspil* i logik och betyder ungefär *om ...så* och läses från höger till vänster. Själva utseendet är tänkt att vara en representation av en (omvänd) logisk implikationspil: $\leftarrow$

Således kan strukturen beskrivas som:

detta gäller :-
... om detta kan bevisas eller styrkas

... eller mer kortfattat:

... så :-
om ...

Det till vänster om pilen kallas *huvud*, det till höger *kropp*, och själva operatorn :- kallas *hals*. Själva uttrycket benämnes *regel*. Rent formaliserat har en regel således strukturen:

<huvud> :-
<kropp>

En sak att lägga märke till här är att prolog saknar globala variabler. Detta innebär att man inte kan definiera en variabel som sedan "ses" av alla predikat eller klausuler i programmet. Om man vill använda en variabel i ett predikats kropp så måste man därför "ta med sig" variabeln genom huvudet. Man skulle kunna säga att huvudet är ett slags "dörr" in i kroppen.

Det är (vanligtvis) valfritt om man vill göra mellanslag eller inte före implikationspilen. Man brukar vanligtvis *indentera* (dvs, göra radindrag för att få snyggare layout) varje ny rad av kroppen för att underlätta läsning av koden. (Hur man skriver prologprogram rent layoutmässigt beskrivs i mer detalj i kapitel 6.)

Den andra operatorn är *kommatecknet* som här inte har riktigt samma funktion som inuti argumentstrukturer. Inuti parentesuttryck används kommatecken för att skilja argument åt. I regler som **son/2** har kommatecknet betydelsen *och* (i logik tecknat $\wedge$).

Således kan regeln ovan läsas ut som:

X är man och X är son till Y om Y är pappa till X eller Y är mamma till X

För att göra det ännu tydligare upprepar vi reglerna, med en "översättning" till höger.

`son(X,Y) :-`	*X är son till Y om ...*
`  man(X),`	*... X är man och...*
`  pappa(Y,X).`	*... Y är pappa till X*
	(... eller ...)
`son(X,Y) :-`	*X är son till Y om ...*
`  man(X),`	*... X är man och...*
`  mamma(Y,X).`	*... Y är mamma till X*

Vi måste ha med båda predikaten eftersom en son ju både har en mamma och en pappa. En mycket intressant och betydelsefull detalj är det "*(... eller ...)*" som skrivits in mellan reglerna. Det visar på den för prolog karakteristiska egenskapen att alltid försöka hitta en (eller alla!) lösning(ar) på en fråga. (Mera därom senare.) Om prolog inte lyckas besvara frågan om **X** är en son till **Y** medelst den första regeln, så fortsätter prolog vidare nedåt bland reglerna och försöker hitta en lösning. Prolog läser alltid regler uppifrån och ned – som vi läser böcker (t ex detta kompendium) – vilket ibland kan ha betydelse för reglers betydelse, vilket vi skall se senare. Därför kan man säga att det råder ett slags 'inklusivt-eller-förhållande' mellan olika regler med samma namn.

Således (som den vakne läsaren redan insett!) skulle man kunna skriva om de båda reglerna ovan till en enda regel genom att använda semikolon, som ju som bekant betyder *eller*. Detta, mer "ekonomiska", sätt att skriva regeln, skulle då få utseendet:

```
son(X,Y) :-
  mamma(Y,X),
  man(X) ;
  pappa(Y,X),
  man(X).
```

Detta har exakt samma innebörd och betydelse som de två reglerna ovan. Detta sätt att göra "en regel av två" kan emellertid göra läsning av program svårare och undviks ofta för att underlätta förståelsen. Vissa författare rekommenderar dessutom att man skriver semikolon på egen rad för att underlätta läsning. Således skulle man skriva regeln ovan som:

```
son(X,Y) :-
  man(X),
  mamma(Y,X)
  ;
  man(X),
  pappa(Y,X).
```

Uttrycket **man(X)** är som synes gemensamt för de bägge delarna av eller-uttrycket, och genom att använda parenteser kan man flytta ut **man(X)**, sålunda:

```
son(X,Y) :-
  man(X),
  (mamma(Y,X)
  ;
  pappa(Y,X)).
```

Detta skulle utläsas som:

X är man och X är son till Y om Y är pappa till X eller Y är mamma till X.

Predikatet **man** här kallas för en *restriktion*. Det räcker inte med att **X** har en far eller mor (det har döttrar också) utan ett vidare, avgränsade, krav är att **X** dessutom är av manligt kön. Ett predikat som **barn** kräver inte denna restriktion:

```
barn(X,Y) :-
  pappa(Y,X).

barn(X,Y) :-
  mamma(Y,X).
```

Om vi vill kan vi givetvis skriva om dessa båda regler som en enda:

```
barn(X,Y) :-
  pappa(Y,X)
  ;
  mamma(Y,X).
```

Ytterligare en restriktion som man kan behöva lägga in i en släktdatabas dyker upp när man försöker definiera "bror". Vi börjar med att lägga in ett par söner:

```
pappa(albert,sune).
pappa(albert,ansgar).
man(albert).
man(sune).
man(ansgar).
```

Av dessa fakta framgår att Sune är bror till Ansgar. Vi definierar ett predikat som uttrycker denna relation. Först definerar vi emellertid relationen **förälder**, för att undvika "dubbla" regler, en för **pappa** och en för **mamma**:

```
förälder(X,Y) :-
  pappa(X,Y)
  ;
  mamma(X,Y).
```

Vid det här laget skall väl inte denna regel bjuda på några problem.

Vi kan nu definiera regeln **bror**:

```
bror(X,Y) :-
  man(X),
  förälder(Z,X),
  förälder(Z,Y).
```

Detta program utläses:

X är man och X är bror till Y om det finns ett Z sådant att detta Z är förälder till X och samma Z är förälder till Y.

Den sista restriktionen behövs för annars skulle X kunna vara en kvinna, vilket inte brukar känneteckna någon som är bror. Lägg märke till att Y mycket väl kan vara kvinna så som regeln är formulerad. Om vi däremot hade skrivit en regel **bröder(X,Y)** så hade vi givetvis varit tvungna att lägga restriktionen **man** både på **X** och **Y**.

Om vi nu kör detta program kan vi dock bli förvånade:

```
?- bror(X,Y).
X = sune Y = sune
  yes
```

Vad hände nu då? Jo, när prolog kör detta program så lämnas **X** och **Y** oinstantierade tills vidare, medan **Z** binds till **albert**. Prolog letar sedan efter ett **X** som kan bindas och finner **sune**. Bra så, nu skall prolog bara hitta ett **Y** som skall bindas och letar i databasen och finner **sune**. Som tidigare nämnts börjar prolog varje sökning uppifrån bland reglerna, och **sune** är ju det första som dyker upp. En restriktion som här krävs är alltså att **X** och **Y** inte får vara samma person (om man nu inte anser att man är sin egen bror!). Detta skrivs i prolog:

```
bror(X,Y) :-
  man(X),
  förälder(Z,X),
  förälder(Z,Y),
  X \= Y.
```

... vilket betyder att **X** inte får vara lika med **Y**, dvs, de kan inte bindas till samma person. Operatorn **\=** betyder alltså *går inte att göra lika med.* Regeln ovan utläses alltså:

X är man och X inte är samma person som Y och X är bror till Y om det finns ett Z sådant att detta Z är förälder till X och och förälder till Y

Vi har nu sett att man på olika sätt måste lägga restriktioner, eller så kallade *test*, i regler för att de skall bete sig på önskat sätt.

Ett annat problem är att man i vissa fall känner att ett program inte säger allt det borde. För att exemplifiera detta definierar vi ett predikat **`gifta`**. Vi börjar med att lägga in ett faktum:

```
gifta(albert,hanna).
```

Vi kan nu ställa frågan

```
?- gifta(albert,hanna).
   yes
```

... men inte frågan

```
?- gifta(hanna,albert).
   no
```

Detta beror på att **`albert`** ligger som första argument i databasen, och prolog kan inte uttala sig om mer än vad som finns i databasen. Vad som är väsentligt här är själva ordningen på argumenten. Argumentens inbördes ordning är, som tidigare nämnts, fixerad. Om detta inte vore fallet så skulle ju ett predikat som **`pappa(albert,sune)`** lika väl betyda att **`sune`** var pappa till **`albert`**!

Hur skall vi då uttrycka att Hanna är gift med Albert? Det finns nu två sätt att lösa detta problem. Det ena är att lägga in ett predikat till:

```
gifta(hanna,albert).
```

Detta löser problemet alldeles utmärkt, men om man nu vill lägga in många gifta par i databasen så leder denna metod till att man får skriva exakt dubbelt så många predikat som man intuitivt känner vore nödvändigt. Vi vet ju att om **`X`** är gift med **`Y`**, så är **`Y`** gift med **`X`**. Varför då inte skriva en regel som uttrycker denna ömsesidighet i programmet! Vi skriver in regeln:

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

Detta verkar väl naturligt, men tyvärr leder detta till ett nytt problem! Prolog skulle som svar på denna fråga helt enkelt råka in i en oändlig *loop*, där Albert är gift med Hanna som är gift med Albert som är gift med Hanna, osv.[1] Prologs inbyggda egenskap att alltid försöka hitta *alla* lösningar innan den ger upp (dvs, svarar **`no`**) leder till att en fråga nu skulle leda till att prolog helt enkelt *loopar* på grund av regelns *(vänster)rekursiva* utseende. Vi går händelserna lite i förväg, men ordningen på regler och mål har betydelse i programmen. För att undvika att prolog snurrar fram och tillbaka mellan Albert som är gift med Hanna som är gift med Albert som är gift med Hanna och så vidare, så måste vi definiera ett nytt predikat, **`gift_med/2`**, som i sin tur anropar **`gifta/2`**:

```
gift_med(X,Y) :-
   gifta(X,Y)
   ;
   gifta(Y,X).
```

[1]Detta gäller som principiellt riktigt. I själva verket kan en sådan regel fungera i somliga fall, t ex om fakta ligger före regeln i databasen.

Detta program utläses:

X är gift med Y om antingen X och Y är gifta, eller Y och X är gifta.

Den smått tautologiska ('tårta-på-tårta'-iga) formuleringen förklaras som sagt av att argumentordningen är av avgörande betydelse i prologregler. Givet enbart faktumet **`gifta(albert,hanna)`** kan vi nu ställa frågan:

```
?- gift_med(albert,hanna).
  yes
```

... och dessutom frågan

```
?- gift_med(hanna,albert).
  yes
```

Att detta program inte loopar beror på att **`gift_med/2`** inte är en rekursiv regel. Vi har således på detta vis undvikit risken för loop.

Notera att detta gäller även predikat där argumentordningen är symmetrisk. Om vi definierar predikatet **`bröder`** som:

```
bröder(albert,sune).
```

... så är ju själva predikatsnamnet symmetriskt såtillvida att *betydelsen* är oavhängig ordningen på argumenten. För prolog spelar emellertid ordningen fortfarande roll. Således erhåller vi svaren:

```
?- bröder(albert,sune).
  yes

?- bröder(sune,albert).
  no
```

Alltså får vi på samma sätt som med **`gift_med/2`** definiera ett predikat **`bror_till/2`**:

```
bror_till(X,Y) :-
  bröder(X,Y)
  ;
  bröder(Y,X).
```

Om man undantar alla problem som kan uppstå med skilsmässor, halvsyskon och så vidare, så är detta ett utmärkt sätt att beskriva släkter.

Självfallet kan man upprätta databaser över vad som helst, t ex sin CD-samling, enligt mönstret:

```
cd(Artist,Titel).
```

Med som exempel på fakta:

```
cd(genesis,selling_england_by_the_pound).
cd(genesis,duke).
cd(mozart,kröningsmässan).
cd(mozart,requiem).
```

Frågar man då:

```
cd(genesis,X).
```

... så får man ut alla sina cd:s med Genesis.

```
X = selling_england_by_the_pound ;
X = duke
```

Frågar man "åt andra hållet" så får man en artist:

```
?- cd(X,kröningsmässan).
X = mozart
```

Nya termer och begrepp

databaser
not
\+
extensionell databas
relationer
regler
intensionell databas
deduktion
:-
implikationspil
huvud
kropp
hals
global variabel
,
restriktion
\==
loop
rekursion
test

Övningar:

- Skriv en databas över din egen familj och testa den med olika frågor. Försök att med regler lägga in predikat som *farfar*, *farmor*, *morfar*, *mormor*, *sonson* osv. Notera vilka problem som dyker upp med restriktioner och dylikt. Om du klarar av relationer som *brylling*, *syssling* och dylikt kan du vara mer än nöjd!

- Skriv en databas över ditt hem som tar två argument, ett rum och vad som finns i rummet. Således:

  ```
  hem(Rum,Innehåll).
  ```

  Exempel kan vara:

  ```
  hem(kök,spis).
  hem(kök,mat).
  hem(sovrum,säng).
  ```

  ... osv. På detta sätt kan du hålla reda på var du har dina prylar!

- Skriv ett predikat, **`urmoder/1`**, som kollar vilken kvinna som är äldst i (den kända) släkten. För att göra detta kan man använda sig av negationspredikatet **`not/1`** eller **`\+`**. Försök använda negation för att lyckas med detta.

3 CUT, TRUE OCH FAIL

Som vi har sett tidigare söker prolog alltid igenom hela regelfilerna efter en lösning innan prolog ger upp och ger svaret **no**. Ibland vill man dock förhindra prolog att söka vidare, eftersom man vid ett visst svar inte längre är intresserad av alternativa lösningar. Det finns i prolog ett sätt att förhindra vidare sökning, vilket benämnes *cut*.[1]

Cut skrives med ett utropstecken, **!**, och läggs in i koden efter den lösning som man nöjer sig med. Vad cut gör är att begränsa prologs inherenta *icke-deterministiska* sökning, som man säger. Icke-deterministisk innebär att det kan finnas fler än ett svar på en fråga, och som vi minns från exemplet med **;** kan ju prolog hitta flera olika svar om de kan styrkas.

Vi säger att vi vill skriva ett program som helt enkelt kontrollerar om två givna variabler är olika. Vi kallar det **olika/2**.

```
%  Olika/2 tar två variabler och kontrollerar om de är olika.

olika(X,Y) :-           % Är två variabler X och Y olika?
  X == Y, !, fail       % Om X = Y, så bryt sökning och misslyckas.
  ;                     % Annars...
  true.                 % ...lyckas predikatet.
```

Här stöter vi även på de bägge predikaten **true/0** och **fail/0**. Ibland vill man ”tvinga” prolog att lyckas, när prolog av sig självt skulle misslyckas, eller vice versa, tvinga prolog att misslyckas när prolog egentligen skulle ha lyckats. Detta gör man helt enkelt genom att skriva in **fail** respektive **true** i koden.

Vad programmet ovan gör är att ta två variabler, **X** och **Y**, och kontrollera via *ekvivalenssprediktatet* == om de är identiska. Om **X == Y** så lägger vi in ett cut, vilket betyder att vi inte behöver (för)söka vidare, vi har redan svaret. Genom att skriva **fail** tvingar vi prolog att misslyckas. Om **X** *inte* är identiskt med **Y**, så skriver vi helt enkelt in i koden att predikatet lyckats genom att lägga in ett **true**. Man kan säga att **true** är ett predikat som alltid lyckas.

Användandet av cut kan ställa till problem med programs betydelse om man inte vet exakt var man skall lägga in det. Man brukar skilja mellan *gröna* och *röda* cut, där gröna mer eller mindre enbart ökar programmens effektivitet, medan röda ändrar programmens *deklarativa* innebörd, dvs faktiskt ändrar betydelsen på programmen. Jag skall här inte närmare gå in på skillnaderna, utan hänvisar till litteraturlistan för närmare diskussioner rörande problem med användandet av cut (vilket beskrivs i samtlig prologlitteratur!).

---

[1] På svenska säger man ofta *snitt*. Jag skall dock i fortsättningen använda den engelska termen.

Nya termer och begrepp

cut
`!`
determinism
`true/1`
`fail/1`
unifieringspredikat
`==`
gröna cut
röda cut

Övningar!

- *Inga*!

4 REKURSION OCH LISTOR

Innan vi fortsätter med databashantering skall vi beskriva en egenskap som är av yttersta betydelse för all programmering: *rekursion*. Ett ganska bra sätt att beskriva vad rekursion går ut på är att säga att det är ett predikat som biter sig självt i svansen, eller mer formellt ”anropar sig självt”.

Inom lingvistiken kan man säga att en rekursiv *omskrivningsregel* är en regel som kan expanderas som sig själv, t ex $S \rightarrow S$ *och* S.

Vi börjar med ett enkelt program:

```
skriv_hej :-
  write('Hej!').
```

Det första att lägga märke till här är att programmet är nollställigt, dvs saknar argument. Det andra är att vi nu använder oss av det predefinierade predikatet **write/1**, som skriver ut sitt argument.

Om vi kör detta program så händer följande:

```
?- skriv_hej.
Hej! yes
```

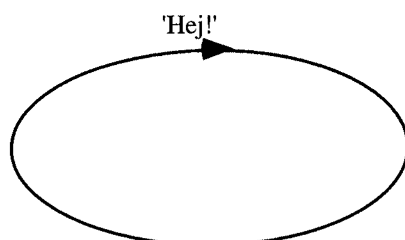
Detta har ni redan räknat ut. Nu skall vi göra programmet rekursivt.

```
skriv_hej :-
  write('Hej!'),
  skriv_hej.
```

Det första att lägga märke till här är att kroppen innehåller två stycken uttryck, dels **write**-uttrycket, dels ett anrop till **skriv_hej** självt.

Vad har vi då gjort? Vi har helt enkelt låtit programmet ”bita sig i svansen”. Det anropas, skriver ut ”Hej!”, anropar sig självt igen, skriver ut ”Hej!”, anropar återigen sig självt, skriver ut ”Hej!”, anropar.... och så vidare. Lägg märke till att reglerna läses uppifrån och ned i funktionskroppen. En överskådlig bild kan vara att föreställa sig program som en ”promenad”, där man går uppifrån och nedåt i programkroppen. Således skulle man kunna ”avbilda” det första **skriv_hej/0** på detta sätt:

I och med att vi skriver till predikatnamnet självt i slutet av programkroppen så låter vi promenadstigen bli en ”cirkel”, dvs, man kommer tillbaks till början, ungefär som historierna om ökenvandrare som hittar fotspår att följa, utan att förstå att de går i sina egna, eftersom de gått i cirkel. Således kan man avbilda den rekursiva versionen av **skriv_hej/0** på detta sätt:



Som framgår av denna senare bild blir ett anrop nu farligt:

```
?- skriv_hej.
Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!
Hej!Hej!Hej!Hej!Hej!HejHej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!HejHe
j!Hej!Hej!Hej!Hej!Hej!Hej!Hej!HejHej!Hej!Hej!Hej!Hej!Hej!Hej
!Hej!HejHej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!HejHej!Hej!Hej!Hej!H
ej!Hej!Hej!Hej!HejHej!Hej!Hej!Hej!Hej!Hej!Hej!Hej!Hej
```

... och så vidare. Som synes kräver rekursion uppenbarligen viss övervakning.[1]

Rekursion är väldigt användbart och kraftfullt, och därutöver väldigt lätt att implementera i prolog. Särskilt användbart är rekursion när man arbetar med strukturer vilkas längd man inte känner till på förhand, och därför används rekursion ofta när man vill arbeta med *listor* av okänd längd. Vi definierar nu predikatet **skriv_ut_lista/1**, som tar en lista av obestämd längd som argument och skriver ut dess medlemmar, en efter en, tills listan är slut.

```
skriv_ut_lista([]).

skriv_ut_lista([FörstaElementet|RestenAvListan]) :-
    write(FörstaElementet), nl,
    skriv_ut_lista(RestenAvListan).
```

Här finns det lite att förklara. Vi börjar med listan. En lista i prolog skrivs inom hakparenteser med ett kommatecken mellan de olika elementen i listan . Ett par exempel:

Sifferlista: **[1,2,3,4,5,6].**

Namnlista: **[robert,carin,gunnar,gunnel,benny].**

Allt som står inom hakparentser räknas som en lista. Listor kan inkludera andra listor. Här följer en lista med fyra element varav det andra också är en lista:

[1,[2,3],4,5].

Alltså:

Element ett:	**1**	= en atom
Element två:	**[2,3]**	= en lista
Element tre:	**4**	= en atom
Element fyra:	**5**	= en atom

Av *elementen* i en lista kan prolog bara "se" de översta eller första elementen. En vanlig liknelse som brukar användas är tallrikstravarna som finns på vissa restauranter. Man kan enbart ta den översta tallriken, och när den försvinner åker nästa tallrik upp, och blir den översta. I sifferlistan ovan ses alltså enbart siffran **1**, i namnlistan så "syns" bara namnet **robert**. Listor delas således upp i två delar: dels det första elementet och dels resten av listan. Dessa avskiljes av en lodlinje (eller lodstreck) på följande sätt:

[Första | Resten]

I själva verket har listor ett mer komplext utseende, men prolog tillhandahåller denna överskådliga struktur för att förenkla användandet.

---

[1]Parentetiskt kan nämnas att om program *loopar* på detta sätt finns det tangentkommandon för att avbryta körningen. Dessa varierar mellan prologer, ofta avhängigt vilken typ av dator man kör på. På Macintosh gör man KOMMANDO-PUNKT, på PC CTRL-ALT-DEL, och i SICSTUS KONTROLL-D.

På engelska (nästan alla böcker ni kommer att läsa om prolog är på engelska) heter detta oftast **[First|Rest]**, men även **[X|Xs]** förekommer, där **Xs** kan ses som en pluralform av **X** och läsas ut *flera X*, t ex.

Vi kan nu ta namnlistan som argument och provköra programmet:

```
?-  skriv_ut_lista([robert,carin,gunnar,gunnel,benny]).
robert
carin
gunnar
gunnel
benny
  yes
```

Hur gick nu detta till? När programmet anropas så instantieras

[FörstaElementet|RestenAvListan]

... med den lista vi tillhandahåller, alltså:

[robert,carin,gunnar,gunnel,benny].

En liten skillnad är det dock, i det att programmet har "delat" listan i två delar: **FörstaElementet** och **RestenAvListan**. Alltså instantieras listan som

[robert|[carin,gunnar,gunnel,benny]]

... med en lodlinje mellan **robert** och resten av listan. Observera att detta enbart sker teoretiskt. Det som finns efter lodlinjen syns ju inte, som påpekades inledningsvis! I själva verket instantieras listan egentligen som:

[robert|RestenAvListan]

... eller egentligen:

[robert|_107]

... där **_107** (eller någon annan siffra!) är en intern *pekare* till var i datorn resten av listan befinner sig. Således instantieras argumenten sålunda:

FörstaElementet som **robert**

RestenAvListan som **[carin,gunnar,gunnel,benny].**

... dvs listan minus det första elementet. Programmet skriver nu ut **robert** på skärmen och radmatar (med det predefinierade predikatet **nl/0**, för *new line*), samt anropar sig självt med **RestenAvListan** som argument. Alltså så "körs" programmet igen, fast denna gång med ursprungslistan minus det första namnet som argument, dvs:

[carin,gunnar,gunnel,benny].

Denna lista delas nu upp i:

FörstaElementet, som är **carin**

... och:

RestenAvListan som är **[gunnar,gunnel,benny]**

Listan har nu alltså utseendet :

`[carin|[gunnar,gunnel,benny]].`

Det första elementet skrivs ut (dvs, `carin`), en radmatning görs och programmet körs nu på den nya `RestenAvListan`, dvs `[gunnar,gunnel,benny]`. Denna delas nu i sin tur upp i ett `FörstaElementet` och en `RestenAvListan`, som nu har blivit

`[gunnar|[gunnel,benny]]`

... och så vidare till alla namn har skrivits ut. Detta sätt att arbeta har många fördelar, varav den främsta är att man inte behöver veta i förväg hur långa listorna är, programmet fungerar ändå.

Nåväl, den första lilla programsnutten då? Vad betyder:

```
skriv_ut_lista([]).
```

Jo, när `benny` är utskrivet så är listan tom, dvs den näst sista listan har utseendet:

`[benny|[]].`

... och anropet efter det att `benny` har skrivits ut får som argument:

`[]`

`RestenAvListan` är alltså tom, och för att prolog skall lyckas med ett anrop med en tom lista krävs det att programmet klarar av detta fall, det så kallade *basfallet.* Programmet

```
skriv_ut_lista([]).
```

... ”känner igen” situationen med en tom lista, och lyckas alltså. I program bör man lägga avgörande test, basfall, så tidigt som möjligt, dvs sådana test som kan leda till att programmet avbryts (eller *terminerar*, med ett annat ord). Programmet `skriv_ut_lista` terminerar när listan är tom, varför regeln som känner igen den tomma listan läggs först. Basfall *kan* även läggas sist, men i flertalet fall så blir programmen effektivare om basfallen läggs först.

Predikatet `skriv_ut_lista` visar på en av prologs mer udda egenskaper: att kunna använda samma predikatnamn för flera olika fall. När man anropar `skriv_ut_lista` letar prolog efter ett predikat med det namnet i regelfilen med rätt antal argument (man kan nämligen definiera flera predikat med samma namn men med olika antal argument!). När ett sådant predikat hittas så körs det, och om det lyckas uppfylls programmets så kallade *mål*, dvs det man vill att programmet ska uppnå. Om programmet inte lyckas, så letar prolog efter ett annat (eller flera) predikat med namnet `skriv_ut_lista` i regelfilen. I fallet ovan innebär det att vid varje rekursivt anrop provas `skriv_ut_lista([])` först, men misslyckas eftersom listan inte är tom. Prolog letar då efter ett annat `skriv_ut_lista`, hittar ett och kör detta. Först när listan är tom lyckas det första `skriv_ut_lista`. Man säger att varje ”version” av ett sådant predikat eller regel är en *klausul.*[2] Ett predikat som `skriv_ut_lista` består alltså av ett predikat med två klausuler.

En sak som måste nämnas är att man mycket väl kan titta på fler än enbart det första elementet i en lista. Om vi definierar predikatet `skriv_ut_två/1` så blir detta klart:

---

[2]*Klausul* används ofta som synonymt med regel.

```
skriv_ut_två([]).

skriv_ut_två([Första,Andra|RestenAvListan]) :-
  write(Första), nl,
  write(Andra), nl,
  skriv_ut_två(RestenAvListan).
```

Detta program borde inte bjuda på några problem att förstå vid det här laget! Vad programmet gör är helt enkelt att skriva ut två element åt gången, och sedan anropa sig självt rekursivt. För att just detta program skall fungera – som det nu ser ut – kräver dock att listan innehåller ett jämnt antal medlemmar.

Som vi skall se finns det även ett sätt att göra `Rest` direkt åtkomligt – *differenslistor* – vilket kan vara mycket användbart.

Vi skall även visa (i ett senare kapitel) att denna möjlighet att specificera olika "varianter" av samma regel/predikat vilka känner igen olika situationer är ett idiomatiskt sätt att i prolog uttrycka IF-THEN-ELSE-uttryck.

Vi definierar ytterligare ett program för att åskådliggöra rekursion. Detta program blir aritmetiskt, och vi kallar det `plus_ett/1`.

```
plus_ett(Siffra) :-
  NySiffra is Siffra + 1,
  write(NySiffra), nl,
  plus_ett(NySiffra).
```

Här har vi använt oss av ytterligare två predefinierade operatorer: `is` ger variabler värden. I programmet ovan sätts värdet på `NySiffra` till värdet av `Siffra` + 1. Det skrives med mellanslag till höger och vänster, vilket kan se "konstigt" ut, eftersom vi hittills är vana vid kommatecken och parenteser.

Operatorn `+` har samma funktion som i vanlig matematik, och adderar helt enkelt sina argument.

Ett anrop kommer att få följande förlopp:

```
?- plus_ett(2).
3
4
5
6
7
8
9
10
11
```

... osv i all oändlighet. Den oundvikliga frågan: hur går nu detta till? Jo, `plus_ett` läser in `Siffra` som ges som argument (man kan alltså börja med vilken siffra som helst). I kroppen skapar man variabeln `NySiffra` vars värde sätts till `Siffra` plus 1. `NySiffra` skrivs sedan ut på skärmen med predikatet `write`. En radmatning görs med `nl`. Därefter anropar man programmet `plus_ett` med `NySiffra` som argument. Vad som är viktigt att förstå är att nu är `NySiffra` plötsligt `Siffra`, eftersom den *nya* siffran hamnat i huvudets argumentparentes. När vi kommer ned i kroppen är därför `NySiffra` oinstantierad, och dess värde bestäms igen när vi tar `Siffra` (dvs, den före detta `NySiffra`!) `+ 1`.

Kutym är att när man på detta sätt transformerar argument (på ett eller annat sätt), ge de "omgjorda" variablerna namnet **Ny** (eller **Nytt**, om man är genuspurist!) plus det gamla namnet, till exempel **Namn** blir **NyttNamn**, **Ord** blir **NyttOrd** osv.

En mycket viktig sak att känna till rörande basfall är deras *placering*! Om vi till exempel vill att programmet **plus_ett/1** skall sluta addera när input-siffran är 10, och i så fall skriva ut något annat på skärmen, t ex "Hej!", så kan vi lätt skriva detta program:

```
plus_ett(Siffra) :-
  Siffra >= 10,
  write('Hej!').
```

Vi har här använt operatorn **>=**, som i detta fall betyder "kontrollera om **Siffra** är större än eller lika med 10". Detta innebär att vi vill att programmet skall bete sig på följande sätt:

```
?- plus_ett(5).
6
7
8
9
10
Hej!
  yes
```

... dvs, ta siffran 5, göra "plus 1" på den tills man nått 10, och då, i stället för att fortsätta additionen, skriva ut "Hej!" på skärmen. Vi kallar **plus_ett** som skriver ut "Hej!" för *basfallet* eftersom det inte innehåller något rekursivt anrop. Man kan också säga att basfallet "bottnar".

Ett problem är att avgörande för huruvida detta kommer att fungera är om vi lägger basfallet *före* eller *efter* den rekursiva klausulen **plus_ett**! Eftersom prolog alltid läser regler uppifrån och ned så kan det tänkas att ett en viss klausul aldrig blir läst.

Om vi lägger den rekursiva delen före basfallet så kommer i själva verket aldrig basfallet att läsas! Programmet kommer fortsätta att addera i all oändlighet! Om vi däremot lägger basfallet före den rekursiva klausulen så kommer programmet att terminera.

Detta är så väsentligt för programskrivning i prolog att vi går igenom det i detalj. Först skriver vi programmet "fel", sedan "rätt".

Vi antar att programmet är skrivet så här i vår regelfil:

```
% En version av plus_ett/1 som INTE FUNGERAR!

% Rekursivt adderande del
plus_ett(Siffra) :-
  NySiffra is Siffra + 1,
  write(NySiffra), nl,
  plus_ett(NySiffra).

% Basfall
plus_ett(Siffra) :-
  Siffra >= 10,
  write('Hej!').
```

Ett anrop med detta program skulle leda till evig addering, utan att basfallet någonsin exekverades. Detta beror på att det rekursiva programmet hela tiden kommer att lyckas. Om vi anropar med, t ex, siffran 3, så adderas den och binds till **NySiffra** som skrivs ut, och **plus_ett** anropas med **NySiffra**. Som tidigare nämnts läser prolog alltid

uppifrån i regelfilen, så därför läses nu det *första* **plus_ett** som prolog finner, dvs, det rekursiva anropet. Detta är även fallet när **NySiffra** är 10, då den övre klausulen anropas. Den övre klausulen innehåller ingen särskild information rörande siffran 10, varför denna glatt adderas, skrivs ut, och programmet fortsätter addera med 11, 12 och så vidare.

Vi sätter därför reglerna i motsatt ordning i regelfilen, på detta sätt:

```
% En version av plus_ett/1 som FUNGERAR!

% Basfall
plus_ett(Siffra) :-
  Siffra >= 10,
  write('Hej!').

% Rekursivt adderande del
plus_ett(Siffra) :-
  NySiffra is Siffra + 1,
  write(NySiffra), nl,
  plus_ett(NySiffra).
```

Om vi nu gör ett anrop så kommer programmet att skriva ut siffror tills 10 är nått, då det skriver ut "Hej!" och avbryter. Detta beror på att reglerna nu ligger i en sådan ordning att basfallet *alltid* kollas innan den rekursiva klausulen anropas. Om vi t ex anropar med siffran 7, hittas basfallet först, men misslyckas, eftersom 7 inte är lika med 10, varför *nästa* **plus_ett** anropas, som lyckas (och adderar i vanlig ordning!). När det rekursiva anropet görs (med 8) kollas basfallet först igen (eftersom basfallet ligger först!), och misslyckas igen, 8 är ju inte heller lika med 10. Alltså provas nästa **plus_ett**, som lyckas. Ett nytt rekursivt anrop görs (med 9). Återigen provas basfallet först och misslyckas. Vid nästa rekursiva anrop *är* siffran 10. Basfallet provas, och lyckas. *Hej!* skrivs ut på skärmen, och eftersom programmet lyckats, behöver inga andra versioner av **plus_ett** provas, eftersom prolog bara letar vidare om det behövs!

Detta visar återigen att det spelar roll var basfallen ligger i regelfilen. Oftast är det bäst (om inte nödvändigt, som i ovannämnda fall!) att lägga dem först, vilket också de flesta författare rekommenderar.

För att återvända till rekursion, kan vi nämna predikatet **medlem/2**, som använder sig av rekursion, och kontrollerar huruvida ett objekt finns med i en lista av arbiträr längd.[3]

```
% Basfall
medlem(X, [X|Resten]).

% Rekursivt anrop
medlem(X, [Y|Resten]) :-
  medlem(X,Resten).
```

Detta predikat kan beskrivas så att det först kollar om ett element **X** är likadant som det första elementet i en lista, dvs X = X (basfallet). Om detta inte är fallet (det rekursiva anropet), dvs $X \neq Y$, körs **medlem** på resten av listan.

Predikatet **medlem** kan användas på olika sätt:

Vi kollar om ett visst element – **gunnel** – är med i namnlistan:

```
?- medlem(gunnel, [robert,carin,gunnar,gunnel,benny]).
  yes
```

---

[3]**Medlem/2** finns ofta predefinerat i prolog under namnet **member/2**.

Vi kan också fråga vilka namn som är med i listan:

```
?-  medlem(X,[robert,carin,gunnar,gunnel,benny]).
X = robert
```

Om vi skriver semikolon får vi resten av listan:

```
X = robert;
X = carin;
X = gunnar;
X = gunnel;
X = benny;
  no
```

Om det synes värdelöst att kolla om något är medlem i en lista som man måste skriva ut *in extenso* (eftersom man då ju ser om något är en medlem eller inte!), så kan det påpekas att det är givetvis inte är på *exakt* detta sätt man använder sig av **medlem** i praktisk programmering, då man i stället hämtar sina listor från annat håll, utan att veta vad som finns i dem!

Man kan även *bygga* listor med **medlem** genom att som andra element ge en variabel:

```
?-  medlem(janne,X).
```

Detta anrop lägger **janne** i en lista. Detta går till på så sätt att eftersom **medlem** är definierad såsom havande en lista som andra argument, där **X** är det första elementet, så läggs helt enkelt **janne** in som första element i den lista man ger som andra element. Detta sätt att använda regler "i två riktningar" är också något som är typiskt för prolog.

Om vi sålunda anropar:

```
?-  medlem(janne,X).
```

... så svarar prolog med:

```
X = [janne|_107]
```

... dvs lägger **janne** främst i en lista.

En sista sak att påpeka är att listor även kan ha variabler som element.

Nya termer och begrepp
rekursion omskrivningsregel **write/1** lista lodlinje pekare **nl/0** basfall terminering mål klausul **is** **+**

Övningar:

- Skriv ett program som kollar om en lista med bokstäver är medlem i en annan lista. Ett anrop kan alltså komma att se ut på detta sätt:

  ```
  ?- medlem([a,b],<en lista med bokstäver eller bokstavslistor>).
   yes
  ```

- Dryga ut programmet **`skriv_ut_två/1`** så att det även klarar att skriva ut ett udda antal element. Du kan kalla predikatet **`skriv_ut_alla/1`**.

- Dryga ut släktdatabasen med predikatet **`förfader/2`**. Detta måste göras rekursivt och kan använda sig av ett predikat **`förälder/2`**. Tänk på att det skall gälla ett godtyckligt antal generationer mellan **`X`** och **`Y`**! Eftersom vissa prologer inte klarar av *å*, *ä* och *ö* kan du skriva på engelska (*forefather* eller *ancestor*). Andra lösningar är att skriva **`foerfader`** eller **`forfader`**.

5 AVLUSNING ("DEBUGGING")

Prolog erbjuder väldigt trevliga *programkontroller*. Dessa visar hur programmet arbetar steg för steg. På grund därav kan man se exakt var någonstans det går snett i program som inte gör som man vill. Det finns två huvudpredikat, **step** och **trace**.[1] De gör samma sak, men medan **trace** låter programmet rusa fram över skärmen, måste man "radmata fram" med **step**. Därför kan man använda **trace** när programmet håller på med delar som fungerar, och slå över till **step** när man vill hinna förstå vad som händer (eller inte händer).

Ett utmärkt sätt att lära sig programmera är just att skriva program, och när de inte fungerar (för det gör de inte!) så läser man igenom med **trace** eller **step** och ser var man tänkt eller skrivit fel.

Debugfunktionerna **step** och **trace** väljs från en av menyerna i programmet (om prologen är menyorienterad!!), eller anropas som nollställiga predikat efter frågeprompten.

För att exemplifiera ett par enkla trace skriver vi in en enkel släktdatabas. Databasen innehåller tre faktapredikat **man/1**, **mamma/2** och **pappa/2**, samt reglerna **farfar/2** och **bröder/2**:

```
% En liten databas över några män i min släkt.

man(robert).
man(hilding).
man(roger).
man(hjalmar).

pappa(hilding,roger).
pappa(hilding,robert).
pappa(hjalmar,hilding).

farfar(X,Y) :-
  pappa(X,Z),
  pappa(Z,Y).

bröder(X,Y) :-
  pappa(Z,X),
  pappa(Z,Y),
  man(X),
  man(Y).

bröder(X,Y) :-
  mamma(Z,X),
  mamma(Z,Y),
  man(X),
  man(Y).
```

Vi går nu ut till frågepromten, *consultar* släktbasfilen ovan och aktiverar **trace**, antingen genom att välja det från en meny eller genom att skriva **trace** efter promten, på följande sätt:

```
?- trace.
  yes
```

[1]Trace kallas ibland på svenska *spår*. Jag skall dock använda den engelska termen i detta kompendium.

För att avaktivera tracet så skriver man **untrace** (eller **notrace** eller dylikt – se din prologs manual) vid prompten på följande sätt:

```
?- notrace.
   yes
```

Nåväl, vi har nu aktiverat **trace**. När vi nu anropar med något av predikaten rusar prologs sökning fram över skärmen på följande sätt:

```
?- farfar(X,robert).
  (1:0) Call: farfar(_0,robert)
  |(2:1) Call: pappa(_0,_4)
  |(2:1) Exit: pappa(hilding,roger)
  |(2:2) Call: pappa(roger,robert)
  |(2:2) Fail: pappa(roger,robert)
  |(2:1) Redo: pappa(_0,_4)
  |(2:1) Exit: pappa(hilding,robert)
  |(2:2) Call: pappa(robert,robert)
  |(2:2) Fail: pappa(robert,robert)
  |(2:1) Redo: pappa(_0,_4)
  |(2:1) Exit: pappa(hjalmar,hilding)
  |(2:1) Call: pappa(hilding,robert)
  |(2:1) Exit: pappa(hilding,robert)
  (1:0) Exit: farfar(hjalmar,robert)
     X = hjalmar
?-
```

Det finns först och främst två saker att lägga märke till. Det första är att prolog använder sig av fyra anrop, **Call**, **Fail**, **Redo** och **Exit** som förklaras i det följande. Det andra är lodlinjerna som föregår varje rad. Dessa förklaras i slutet av kapitlet.

För att förklara *tracet* ovan upprepar vi det med insprängda kommentarer (i Times 10 punkter) som belyser rad för rad i anropet. För varje led visas hur regeln **farfar** "ser ut" i körningen, med en liten pil, ←, som visar exakt var prolog "arbetar" för tillfället. Detta är givetvis fel, eftersom prolog inte arbetar med *hela* regeln på en gång, utan med de enstaka delarna var för sig, men det kan vara ett bra sätt att åskådliggöra just detta genom att visa hur hela regeln ser ut för tillfället.

```
?- farfar(X,robert).
```

Vi anropar med predikatet **farfar/2**, med andra argumentet instantierat som **robert**.

```
(1:0) Call: farfar(_0,robert)
```

Funktionen **Call** anropar med **farfar/2**. Som synes är det andra argumentet instantierat, medan det första är en obunden variabel, markerad **_0**. Regeln ser just nu ut sålunda:

```
farfar(_0,robert) :-    ←
   pappa(X,Z),
   pappa(Z,robert).
```

Notera att **Y** instantierats som **robert** i funktionskroppen.

```
|(2:1) Call: pappa(_0,_4)
```

Prolog går ned i regelns funktionskropp och anropar med det första predikatet i kroppen, **pappa/2**. Lägg märke till att här är båda argumenten oinstantierade. Regeln ser nu ut:

```
farfar(_0,robert) :-
   pappa(_0,_4),        ←
   pappa(Z,robert).
```

Observera att `_0` står på två ställen, vilket visar att prolog försöker binda de bägge variablerna till samma värde.

|(2:1) Exit: pappa(hilding,roger)

Prolog hittar en instantiering av `pappa/2` i databasen, nämligen `pappa(hilding,roger)`. Detta anges med `Exit`, vilket betyder att prolog "går vidare" med detta faktum. Regeln ser nu ut:

```
farfar(hilding,robert) :-
   pappa(hilding,roger),      ←
   pappa(roger,robert).
```

Prolog försöker gå vidare i kroppen med det instantierade faktumet. Notera att `hilding` som första argument till `farfar` ännu inte är bevisat, utan bara är ett sätt att visa (felaktigt, egentligen!) vad `X` instantierats som. Lägg också märke till att `roger` är instantierat i bägge predikaten **pappa/2** eftersom det motsvaras av samma variabel, `Z`.

|(2:2) Call: pappa(roger,robert)

Prolog går nu vidare i `farfar`:s funktionskropp till det andra predikatet `pappa/2`. Nu är det första argumentet instantierat som `roger`. Prolog skall alltså försöka styrka faktumet `pappa(roger,robert)` i databasen. Regeln ser ut:

```
farfar(hilding,robert) :-
   pappa(hilding,roger),
   pappa(roger,robert).       ←
```

|(2:2) Fail: pappa(roger,robert)

Prolog misslyckas med att hitta något faktum `pappa(roger,robert)` i databasen, vilket anges med `Fail`. Regeln ser ut:

```
farfar(hilding,robert) :-
   pappa(hilding,roger),
   pappa(roger,robert).       ←
```

|(2:1) Redo: pappa(_0,_4)

Alltså *backtrackar* prolog upp till den första punkt där det gick fel, och börjar alltså om med ett nytt faktum `pappa/2`. Detta anges med `Redo`, som betyder att prolog är villigt att prova något nytt. Prolog ger inte tappt så lätt, inte!

```
farfar(_0,robert) :-
   pappa(_0,_4),          ←
   pappa(_0,robert).
```

Prolog försöker alltså om med det första predikatet pappa för att hitta en alternativ lösning.

|(2:1) Exit: pappa(hilding,robert)

Prolog hittar ett nytt faktum i databasen, i själva verket *nästa* faktum, eftersom prolog alltid läser uppifrån och ned bland reglerna. (Hört det förut!?)

```
farfar(hilding,robert) :-
   pappa(hilding,robert),      ←
   pappa(robert,robert).
```

Prolog hittar faktumet `pappa(hilding,robert)` och går vidare med detta.

|(2:2) Call: pappa(robert,robert)

Prolog går vidare i `farfar`:s funktionskropp och försöker matcha `pappa(robert,robert)` mot något faktum i databasen.

```
farfar(hilding,robert) :-
   pappa(hilding,robert),
   pappa(robert,robert).     ←
```

Prolog försöker nu styrka faktumet `pappa(robert,robert)` i databasen.

```
|(2:2)  Fail:  pappa(robert,robert)
```

Detta misslyckas.

```
farfar(robert,robert) :-
   pappa(hilding,robert),
   pappa(robert,robert).     ←
```

```
|(2:1)  Redo:  pappa(_0,_4)
```

Prolog backar upp igen och försöker återigen med en ny instantiering av `pappa/2`.

```
farfar(_0,robert) :-
   pappa(_0,_4),        ←
   pappa(_O,robert).
```

```
|(2:1)  Exit:  pappa(hjalmar,hilding)
```

Prolog hittar nästa faktum i databasen: `pappa(hjalmar,hilding)`.

```
farfar(hjalmar,robert) :-
   pappa(hjalmar,hilding),      ←
   pappa(hilding,robert).
```

Prolog hittar faktumet `pappa(hjalmar,hilding)` i databasen.

```
|(2:1)  Call:  pappa(hilding,robert)
```

Prolog går återigen ned till nästa predikat i `farfar`:s funktionskropp och försöker matcha `pappa(hilding,robert)` mot något faktum i databasen.

```
farfar(hjalmar,robert) :-
   pappa(hjalmar,hilding),
   pappa(hilding,robert).       ←
```

Prolog försöker nu hitta faktumet `pappa(hilding,robert)` i databasen.

```
|(2:1)  Exit:  pappa(hilding,robert)
```

Detta lyckas, så prolog gör `Exit` med detta faktum. `Z` i regeln `pappa/2` har instantierats som `hilding`, och prolog har lyckats med att finna en matchning för detta i databasen.

```
farfar(hjalmar,robert) :-
   pappa(hjalmar,hilding),
   pappa(hilding,robert).       ←
```

```
(1:0)  Exit:  farfar(hjalmar,robert)
```

Genom att binda `Z` i regeln `pappa/2` till `hilding` har prolog lyckats instantiera `X` i anropet som `hjalmar`, dvs `X` i anropet. Prolog backar alltså ur anropet med `X` instantierat som `hjalmar`, och meddelar detta på skärmen:

```
X = hjalmar
```

Vi visar ytterligare ett trace – denna gång utan kommentarer – för att visa hur prolog söker lösningar i databasen. Studera noga hur prolog försöker predikat efter predikat, och antingen lyckas eller misslyckas med sina anrop. Vi lägger till faktumet:

```
pappa(albert,hjalmar).
```

... i databasen.

Vi anropar med predikatet `bröder/2`:

```
?- bröder(robert,roger).
  (1:0) Call: bröder(robert,roger)
  |(2:1) Call: pappa(_2,robert)
  |(2:1) Exit: pappa(hilding,robert)
  |(2:2) Call: pappa(hilding,roger)
  |(2:2) Exit: pappa(hilding,roger)
  |(2:3) Call: man(robert)
  |(2:3) Exit: man(robert)
  |(2:3) Call: man(roger)
  |(2:3) Exit: man(roger)
  (1:0) Exit: bröder(robert,roger)
     yes
?-
```

Vi visar även ett exempel på ett anrop av `farfar/2` med bägge argumenten oinstantierade. Observera att de olika lösningarna presenteras på grund av att vi skriver in ett semikolon efter varje presenterad lösning, och att prolog svarar `no` när det inte finns några flera lösningar.

```
?- farfar(X,Y).
  (1:0) Call: farfar(_0,_1)
  |(2:1) Call: pappa(_0,_5)
  |(2:1) Exit: pappa(hilding,roger)
  |(2:2) Call: pappa(roger,_1)
  |(2:2) Fail: pappa(roger,_1)
  |(2:1) Redo: pappa(_0,_5)
  |(2:1) Exit: pappa(hilding,robert)
  |(2:2) Call: pappa(robert,_1)
  |(2:2) Fail: pappa(robert,_1)
  |(2:1) Redo: pappa(_0,_5)
  |(2:1) Exit: pappa(hjalmar,hilding)
  |(2:2) Call: pappa(hilding,_1)
  |(2:2) Exit: pappa(hilding,roger)
  (1:0) Exit: farfar(hjalmar,roger)
     X = hjalmar, Y = roger ;        Vi skriver ett semikolon för flera lösningar!
  |(2:2) Redo: pappa(hilding,_1)
  |(2:2) Exit: pappa(hilding,robert)
  (1:0) Exit: farfar(hjalmar,robert)
     X = hjalmar, Y = robert ;       Vi skriver ett semikolon för flera lösningar!
  |(2:1) Redo: pappa(_0,_5)
  |(2:1) Exit: pappa(albert,hjalmar)
  |(2:1) Call: pappa(hjalmar,_1)
  |(2:1) Exit: pappa(hjalmar,hilding)
  (1:0) Exit: farfar(albert,hilding)
     X = albert, Y = hilding ;       Vi skriver ett semikolon för flera lösningar!
     no
```

Vad betydde nu lodlinjerna? Jo, lodlinjerna visar *inbäddning*, dvs hur ”djupt ned” i ett program vi befinner oss. I ovanstående exempel görs bara *direkta* anrop, dvs varje predikat anropar *direkt* i databasen. I större program kan det emellertid vara så att ett predikat inte söker direkt i databasen, utan söker via ett eller flera andra predikat. För varje sådan ”nivå”, så läggs en lodlinje till i tracet för att visa hur ”djupt” vi befinner oss. Tracet i kapitel 13 ger exempel på sådan inbäddning.

Om man vill studera endast ett specifikt predikat i ett program som innehåller flera olika predikat kan man göra detta med det predefinierade predikatet **`spy/1`**. Detta går till så att man anropar det predikat man vill ”spionera” på med **`spy/1`**, på följande sätt:

```
?-  spy(farfar(X,robert)).
  yes
```

Nya termer och begrepp

programkontroll
`step`
`trace`
debugfunktion
meny
`untrace`
`notrace`
`Call`
`Fail`
`Redo`
`Exit`
backtracking
inbäddning
`spy/1`

Övningar:

- *Trace:a* alla program vi hittills gått igenom. Fortsätt sedan framgent att tracea allt!

6 HUR MAN SKRIVER OCH KOMMENTERAR PROGRAM

För att program skall bli lättlästa finns det vissa oskrivna regler rörande hur de skall skrivas. Således skall huvudet (med eller utan ett mellanslag) åtföljas av implikationspilen, varefter man radmatar och sedan *indenterar* (gör indrag) kroppen, antingen med mellanslag eller med ett tabbstopp. Alltså:

<huvud> :-
    <kropp>
    <mera kropp>
    <ännu mera kropp>
    <rena kroppkakan>.

Om regeln är mycket kort kan den skrivas på en rad:

<huvud> :- <kort kropp>.

Vad gäller själva programlösningarna är det en fördel om varje regel/program klarar av så lite som möjligt, och sedan sätts samman på en högre nivå (se kapitlet om *subrutiner*). Ju mer specialiserade delprogram är, desto överskådligare. Om dessutom någon regel innehåller en *lus* (eller *bug*), dvs är felaktig, är det lättare både att upptäcka och rätta till det hela om man bara behöver mixtra med en separat regel (med **spy/1**).

I övrigt gäller samma allmänna layouthänsynstaganden som för vilket dokument som helst! Om detta kompendium hade varit skrivet utan radmatningar, rubriker etc hade det varit betydligt mindre överskådligt, eller hur!

När man skriver program bör man förutom själva koden också lämna en *programbeskrivning*. Detta sker medelst kommentarer som beskriver vad programmet gör, och hur det gör det. För att kunna lägga in kommentarer i en programfil finns vissa tecken som berättar för prolog att vad som följer inte är programkod och således inte skall interpreteras när man gör **consult** eller **reconsult**.

Kommentarer har vi redan stiftat bekantskap med. De kan i prolog kan skrivas på två olika sätt:

Enradskommentarer skrivs med `%`-tecken:

```
% Detta program dubblerar argumentet som är en siffra.
dubblera(Siffra) :-
    NySiffra is Siffra + Siffra,
    write(NySiffra), nl.
```

Om kommentaren tar flera rader i anspråk så kan man inleda varje rad med ett `%`-tecken:

```
% Detta är ett program som tar en siffra som argument, instantierar
% siffran, adderar siffran till sig själv och skriver ut resultatet
% på skärmen samt radmatar.
```

Ett annat sätt att skriva kommentarer som tar mer än en rad är att använda `/*` och `*/` mellan vilka allt ses som en kommentar:

```
/* Detta är ett program som tar en siffra som argument, instantierar
siffran, adderar siffran med sig själv och skriver ut resultatet på
skärmen samt radmatar. */
dubblera(Siffra) :-
    NySiffra is Siffra + Siffra,
    write(NySiffra), nl.
```

Om man vill ha enhetliga kommentarer kan man även använda `/* Kommentartext */` på en rad:

```
/* Detta är en också en korrekt kommentar.*/
```

Kommentarer skall beskriva vad programmet gör på ett adekvat sätt. Vad ”ett adekvat sätt” är kan diskuteras, men en god regel är att det skall förstås av en person som aldrig har sett det tidigare. En bra regel kan vara att *man själv* skall förstå vad ett program handlar om när man kommer tillbaks till det om ett par år (vilket inte alltid är fallet, har jag märkt!!).

Som tidigare nämnts brukar man också ange hur många argument ett predikat tar, eftersom en av prologs egenheter är att predikat med samma namn kan ta olika många argument. Ett predikat delas upp i *funktor* och *aritet*. Själva namnet är funktorn och ariteten är antalet argument. Detta skrives:

<funktorsnamn>/<antal argument>

Ett exempel på en sådan kommentar:

```
/* byggnad/1 tar som argument ett exempel på en byggnad. byggnad/2 tar
som första argument ett exempel på en byggnad, och som andra argument
denna byggnads engelska översättning. */

byggnad(hus).
byggnad(hus,house).
```

Ytterligare en konvention är att man kan tillhandahålla argumentstrukturen som sådan i kommentaren:

```
% byggnad(SvenskByggnad,EngelskÖversättning)
```

I detta fall brukar man ange i kommentaren huruvida en variabel skall tillhandahållas/instantieras av användaren, instantieras av prolog eller accepterar bägge fallen. Om en variabel skall ges av användaren prefigeras den med ett *plustecken*, om prolog självt skall instantiera en variabel prefigeras den med ett *minustecken*, och om det inte spelar någon roll/bägge fallen är möjliga prefigeras variabeln med ett *frågetecken*.

Alltså skulle programmet ovan kunna beskrivas sålunda:

```
/* byggnad(+SvenskByggnad, -EngelskByggnad) tar som första argument
namnet på en svensk byggnad, och skickar tillbaka byggnadens engelska
översättning. */
```

Om programmet är tänkt att fungera som en tvåvägsöversättare där endera argumentet kan instantieras så vore en korrektare beskrivning:

```
/* byggnad(?SvenskByggnad, ?EngelskByggnad) tar som första argument
namnet på en svensk byggnad, och skickar tillbaka byggnadens engelska
översättning, eller vice versa. */
```

Man kan även lägga in kommentarer till höger om programkoden:

```
% Detta är en databas över en släkt
pappa(hilding).    % NB! Det är min pappa!
```

Nya termer och begrepp

indentering
subrutin
lus
bug
programbeskrivning
`%`
`/*    */`
funktor
`+Kommentar`
`-Kommentar`
`?Kommentar`

Övningar:

- Kommentera alla era program! (Ganska svår övning!)
- Skriv ett översättningsprogram, som tar ett svenskt ord som argument och skriver ut ordets engelska översättning på skärmen. Ett anrop – med svar – kan till exempel se ut så här:

```
?- översätt(hund).
hund på engelska heter dog!
```

7 INPUT OCH OUTPUT - LÄSNING FRÅN SKÄRMEN

Dialog mellan datorer och människor förekommer överallt i samhället. Varje gång man använder en bankomat, t ex, råkar man ut för ett sådant "samtal" med en dator som kollar vad man säger och vill. Ett programs *input/output* (eller på svenska: *indata/utdata*) förkortas ofta *I/O*. Man kan med prolog också agera interaktivt med antingen en människa (skärmen) eller textfiler.

Vi börjar med ett litet hälsningsprogram:

```
hej :-
  write('Hej, vad heter du?'),
  nl,
  read(Namn),
  nl,
  write('Hej på dig '),
  write(Namn),
  write('!!'),
  nl.
```

Och så provar vi:

```
?- hej.
Hej, vad heter du?
```

Vi skriver något, t ex vad vi heter! (Glöm inte punkten!)

```
robert.
```

Prolog läser nu, med predikatet **read/1**, in vad som skrivs på skärmen. I programmet ovan har vi kallat detta **Namn**. Prolog svarar:

```
Hej på dig robert!!
  yes
```

Om man inte vill bli hälsad på ett gement sätt (ha, ha!), så får man svara med sitt namn inom enkla citationstecken.

```
?- hej.
Hej, vad heter du?
'Robert'.
Hej på dig Robert!!
  yes
```

Om vi hade skrivit **Robert** i stället för **'Robert'** så hade det tolkats som en variabel, och dialogen hade sett ut sålunda:

```
?- hej.
Hej, vad heter du?
Robert.
Hej på dig _0!!
  yes
```

... där _0 representerar en oinstantierad variabel.

Nya termer och begrepp

input
output
indata
utdata
I/O
`read/1`

Övningar:

- Skriv ett program som ser ut som följer vid körning:

`?- hejsan.`	(Ditt anrop)
`Hej, vad heter du?`	(Prologs fråga)
`robert.`	(Ditt svar)
`Vad jobbar du med, då?`	(Prologs andra fråga)
`undervisning.`	(Ditt andra svar)
`Är det kul med undervisning, robert?`	(Prologs kommentar)

Alltså, ett program som både frågar efter namn och yrke, och svarar med bådadera!

8 IF-THEN-ELSE-STRUKTURER

I detta kapitel skall jag beskriva hur man klarar av *alternativ* i prologprogrammering. Tidigare har vi sett hur vi har skrivit olika klausuler för basfall och rekursiva anrop och hur en klausul misslyckas på grund av att ett (eller flera) villkor inte är uppfyllt.

I all programmering förekommer IF-THEN-ELSE-strukturer ymnigt. IF-THEN-ELSE ställer villkor på och ger alternativ till hur programmet skall bete sig i en given situation enligt det formaliserade mönstret:

IF	ett villkor man ställer
THEN	vad programmet skall göra om det ställda villkoret ovan är uppfyllt
ELSE	vad programmet skall göra om det ställda villkoret ovan inte är uppfyllt

IF-THEN-ELSE kan skrivas på många sätt i prolog, bland annat med det predefinierade predikatet **`ifthenelse/3`**, som emellertid inte finns i alla prologer.

Det uppenbara sättet att skriva IF-THEN-ELSE i prolog är helt enkelt att använda de båda operatorerna **,** och **;**:

ett villkor **,** *vad som skall göras om villkoret är uppfyllt* **;** *vad som skall göras annars*

Eller formellt:

X **,** Y **;** Z

...vilket utläses:

om X så Y, annars Z

Det kan vara lite ointuitivt att utläsa kommatecknet som ”så”, men eftersom prolog inte fortsätter till Y om inte X är uppfyllt blir kommatecknets betydelse i stort sett *så* här. Man kan, om man vill, lika gärna läsa ut det som:

X och Y, annars Z

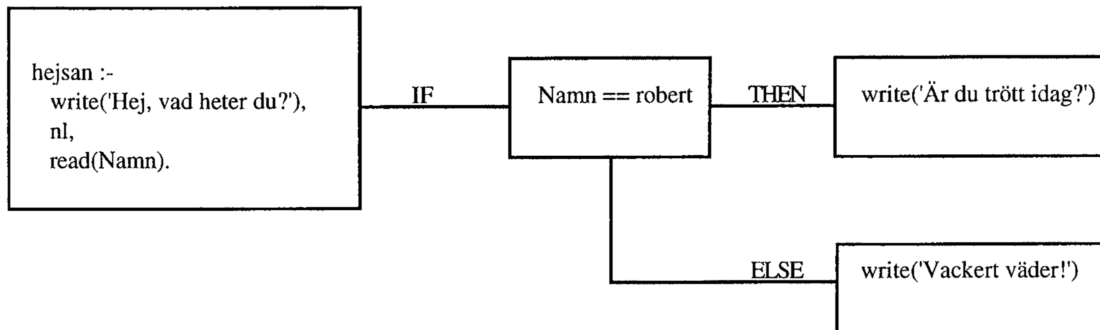
..vilket ju i princip betyder samma sak!

Vi skriver nu ett program som använder denna syntax:

```
/* hejsan/0 frågar efter ett namn. Om namnet är "robert" så skrivs "Är
du trött idag?" ut på skärmen,i annat fall skrivs "Vackert väder!". */

hejsan :-
  write('Hej, vad heter du?'),
  nl,
  read(Namn),
    Namn == robert,                          % IF
    write('Är du trött idag?')               % THEN
      ;                                      % ELSE
      write('Vackert väder!').
```

Lägg märke till att vi indenterat programmet! Detta är inte nödvändigt, men det gör det lättare att läsa.

Vi kan åskådliggöra hur programmet arbetar i följande *flödesdiagram*:



Om de olika delarna innehåller flera uttryck så läggs de inom parentes:

```
/* hejsan/0 frågar efter ett namn. Om namnet är "robert" så skrivs
"Var du trött i morse, robert" ut på skärmen, i annat fall skrivs "Går
det bra att lära sig prolog, NAMN" ut. */

hejsan :-
  write('Hej, vad heter du?'),
  nl,
  read(Namn),
  nl,
  (Namn == robert,                                       % IF
    write('Var du trött i morse, '),                     % THEN
    write(Namn)
      ;                                                  % ELSE
      write('Går det bra att lära sig Prolog, '),
      write(Namn)).
```

Lägg märke till de dubbla parenteserna som avslutar programmet. Den första parentesen avslutar **(Namn)** och den andra avslutar IF-THEN-ELSE-uttrycket. Lägg också märke till att vi lagt in ett mellanslag mellan orden och citationstecknen i **morse,**<här>**'** och **Prolog,**<här>**'** för att inte namnet skall följa direkt på kommatecknet.

Vissa prologer erbjuder den predifinierade operatorn **->**, som kan utläsas som en implikationspil. Att använda den innebär ingen skillnad, men gör program lättare att läsa. Programmet enligt ovan skulle då få utseendet:

```
hejsan :-
  write('Hej, vad heter du?'),
  nl,
  read(Namn),
  nl,
  (Namn == robert ->                                     % IF
    write('Var du trött i morse, '),                     % THEN
    write(Namn)
      ;                                                  % ELSE
      write('Går det bra att lära sig Prolog, '),
      write(Namn)).
```

Ett annat sätt att skriva IF-THEN-ELSE är, som redan nämnt, att använda det predefinierade predikatet **ifthenelse/3**, som har syntaxen:

```
ifthenelse(IF,THEN,ELSE)
```

Ovan nämnda program får då utseendet:

```
hejsan :-
  write('Hej, vad heter du?'),
  nl,
  read(Namn),
  nl,
  ifthenelse(Namn == robert,                  % IF
    (write('Var du trött i morse, '),        % THEN
    write(Namn)),
      (write('Är Prolog roligt, '),          % ELSE
      write(Namn))).
```

Lägg märke parentesbruket! Eftersom `ifthenelse/3` är treställigt så måste vi med parenteser se till att antalet argument är just tre.

Vi provkör nu (det spelar ingen roll vilken version av programmet ovan vi kör – alla fungerar exakt likadant!):

```
?- hejsan.
Hej, vad heter du?
robert.
Var du trött i morse, robert
  yes

?- hejsan.
Hej, vad heter du?
bertil.
Är Prolog roligt, bertil
  yes
```

Precis som var fallet med listor kan man även lägga IF-THEN-ELSE-uttryck inuti andra IF-THEN-ELSE-uttryck, enligt mönstret:

IF	ett villkor (1)	
THEN	IF	ett villkor (2)
	THEN	vad som skall göras om villkor 1 och 2 är uppfyllda
	ELSE	vad som skall göras om villkor 1 men inte villkor 2 är uppfyllt
ELSE	vad som skall göras om varken villkor 1 eller 2 är uppfyllda	

Medan exemplet ovan visar hur man lagrar *villkor* på varandra (IF-uttrycken ackumuleras, så att säga), kan man också lägga in olika *fall* genom att inbädda på ett annat sätt:

IF	ett villkor (1)		
THEN	vad som skall göras om villkor 1 är uppfyllt		
ELSE	IF	ett villkor (2) om inte villkor 1 är uppfyllt	
	THEN	vad som skall göras om villkor 2 är uppfyllt	
	ELSE	IF	ett villkor (3) om inte villkor 1 och 2 är uppfyllda
		THEN	vad som skall göras om villkor 3 är uppfyllt
		ELSE	vad som skall göras om varken villkor 1, 2 eller 3 är uppfyllt

På detta sätt sätt arbetar många program, med så kallade *inbäddningar*, och vi skall se hur inbäddade strukturer som exemplet ovan kan användas när man har fler än ett alternativ.

Om vi vill skriva ett program med flera olika alternativ, i stället för bara ett, kan vi använda oss av inbäddade IF-THEN-ELSE-uttryck enligt exemplen ovan. Vi säger att vi vill ha ett program som frågar efter ett namn, och skriver ut "skräddarsydda" meddelanden avhängiga de namn som skrivs in.

```
/* hejsan/0 skriver ut skräddarsydda meddelanden till ett antal
personer, och ger annars ett standardsvar. */

hejsan :-
  write('Hej, vad heter du?'), nl,                % HÄLSA!
  read(Namn),                                     % LYSSNA PÅ SVAR!
  (Namn == gunnel,                                  % IF
  write('Sågat någon uppsats idag?'), nl, nl        % THEN
    ;                                                 % ELSE...
    (Namn == robert,                                  % ...IF
    write('Är du hungrig idag?'), nl, nl              % THEN
      ;                                                 % ELSE...
      (Namn == janne,                                   % ...IF
      write('Är du glad?'), nl, nl                      % THEN
        ;                                                 % ELSE...
        (Namn == eva,                                     % ...IF
        write('Lagt schemat än?'), nl, nl                 % THEN
          ;                                                 % ELSE
          (write('Vilket trevligt namn!'), nl, nl))))).
```

Lägg märke till att varje inbäddning innebär en vänsterparentes, så att det krävs en hel del högerparenteser för att "stänga" uttrycket på slutet!

Vi provar med några anrop:

```
?- hejsan.
Hej, vad heter du?
robert.
Är du hungrig idag?

?- hejsan.
Hej, vad heter du?
börje.
Vilket trevligt namn!

?- hejsan.
Hej, vad heter du?
gunnel.
Sågat någon uppsats idag?
```

Det här sättet att använda sig av *inbäddade* uttryck kan lätt bli svårläst, och är väldigt otypiskt för prolog-programmering. I kapitlet om subrutiner skall vi se hur man i stället brukar göra.

Nya termer och begrepp

`ifthenelse/3`
`->`
flödesdiagram
inbäddning

Övningar:

- Skriv ett program som frågar hur man mår och ger vissa alternativ, t ex ”bra” ”uselt” ”javars” och skriver ut olika uppmuntrande svar beroende på hur man svarar.
- Skriv en egen IF-THEN-ELSE. (Ett utsökt exempel på prologs deklarativa karaktär.) Utseendet (efter logik):

```
my_if_then_else(P,Q,R) :-
    <fyll i här>
```

... osv.

9 ARITMETIK I PROLOG

Prolog – som alla andra programmeringsspråk – tillhandahåller möjligheter för aritmetik. Innan vi går igenom några av prologs aritmetiska predikat exemplifierar vi ett sådant. Vi definierar **summa/2**, ett program som adderar sina båda argument:

```
summa(Tal1, Tal2) :-
   Summa is Tal1 + Tal2,
   write('Summan är '),
   write(Summa),
   nl.
```

Vi anropar detta program:

```
?- summa(8,9).
Summan är 17
   yes
```

Aritmetiska predefinierade predikat (dvs predikat som redan finns inbyggda i prolog) är bland andra:

\+ Adderar sina argument, som kan vara fler än två.

EX: `2 + 2`

\- Subtraherar sina argument.

EX: `4 - 2`

/ Dividerar sina argument.

EX: `9 / 3`

is Gör det man skulle kunna tro att unifieringspredikatet = gör (se kapitel 10), dvs räknar ut summor, produkter och dylikt. Formellt kan man beskriva det:

<Resultat> **is** <Aritmetiskt uttryck>.

EX: `X is 3 + 3` evalueras som `6 is 3 + 3`

EX: `X is 3 * 3` evalueras som `9 is 3 * 3`

* Multiplicerar sina argument.

EX: `3 * 4`

< Kollar om det första argumentet är mindre än det andra.

Exempel på anrop:

```
?- 3 < 4.
  yes

?- 4 < 3.
  no
```

> Kollar om det första argumentet är större än det andra.

Exempel på anrop:

```
?- 4 > 3.
  yes

?- 3 > 4.
  no
```

>= Kollar om det första argumentet är större än eller lika med det andra.

Exempel på anrop:

```
?- 4 >= 4.
  yes

?- 5 >= 4.
  yes
```

=< Kollar om det första argumentet är mindre än eller lika med det andra.

Exempel på anrop:

```
?- 4 =< 4
  yes

?- 5 =< 4.
  no
```

Utöver dessa artimetiska predikat finns det också en mängd så kallade *systempredikat*. Till dessa hör de redan tidigare nämnda `,` (konjunktion) och `;` (disjunktion), `not/1`, `fail/0`, `true/0`, `!` (cut) med flera.

Vi kan nu använda några av ovannämnda predikat till att skriva programmet `dubblera/1`, göra det rekursivt och lägga till ett *stoppvillkor*, så att det inte fortsätter i evighet.

```
/* dubblera/1 tar en siffra (heltal) som argument, dubblerar denna
rekursivt och skriver ut resultatet på skärmen. När den dubblerade
siffran blir lika med eller överstiger 3000 så bryts programmet. */

dubblera(Siffra) :-
  NySiffra is Siffra * 2,
  ifthenelse(NySiffra < 3000,                          % IF
    (write(NySiffra), nl, dubblera(NySiffra)),         % THEN
    fail).                                             % ELSE
```

Här ser vi också ett bra exempel på användandet av **fail**. Vi vill att programmet skall misslyckas om **NySiffra** är större än 3000, alltså skriver vi in det i koden. Detta leder till att prolog svarar **no** när **NySiffra** överstiger 3000. Om vi i stället hade skrivit **true**, så hade samma sak skett: prolog hade avbrutit dubblera, men genom att skriva **yes**.

Nya termer och begrepp

+
-
/
is
*
<
>
>=
=<
systempredikat

Övningar:

- Skriv ett predikat **dubblera(+Siffra)** som inte använder IF-THEN-ELSE-struktur, utan i stället använder sig av flera predikat, på ett mer prologigt sätt. Ett av predikaten skall leda till ett rekursivt anrop, ett annat måste innehålla stoppvillkoret.

- Medelavståndet mellan jorden och månen är 384 400 km. Om man viker ett papper dubbelt så dubbleras dess tjocklek vid varje vikning. Ett hundragramspapper är ungefär 1/10 mm tjockt. Skriv ett program som räknar ut hur många gånger ett papper skall vikas dubbelt för att dess tjocklek skall nå till månen. Ett anrop kan komma att se ut så här:

  ```
  ?- vik.
  Papperet måste vikas svaret gånger!
  ```

- En anekdot berättar att schacket uppfanns åt en maharadja som var trött på alla sina gamla spel, och ville ha något nytt. Han blev så nöjd när schacket presenterades för honom att han sade att uppfinnaren kunde önska sig vad som helst som belöning. Denne svarade då: ”Jag begär bara att ett sädeskorn lägges på schackbrädets första ruta, två på den andra, fyra på den tredje, åtta på den fjärde, och så vidare. Jag nöjer mig med säden som sedan ligger på brädet!”

  Skriv ett predikat som tar två predikat:

  ```
  dubblera(SiffraSomSkallDubbleras,AntalDubbleringar).
  ```

Ett schackbräde har 64 rutor, och om man lägger ett korn på den första rutan så krävs alltså 63 dubbleringar. Hur många korn ligger på den 64:e rutan? Hur många på hela brädet (ytterligare en dubblering minus ett).

OBS! Alla prologer klarar inte av så stora tal. I så fall kan ni låtsas att ett schackbräde har 49 eller 36 rutor! Vissa prologer har möjlighet att uttrycka tio-potenser, i vilket fall svaret kommer att dyka upp i en lite "lustig" variant.

10 UNIFIERING

En viss mängd systempredikat berör den för prolog så väsentliga egenskapen *unifiering*.[1] Unifiering går i princip ut på att prolog tar två strukturer, jämför dessa och försöker göra dem identiska genom att instantiera variablerna i de båda termerna. Således försöker prolog matcha strukturen **`man(X)`** i en fråga med t ex faktumet **`man(bertil)`** i databasen genom att substituera **`X`** med **`bertil`**. Om detta lyckas har unifiering skett.

Nedan följer några operatorer som används vid unifiering:

= Unifieringspredikatet. Till skillnad från **`is`** så ger detta inte värden åt aritmetiska uttryck. Ett uttryck som **`X is Y`** skulle utläsas som *ge X värdet av den aritmetiska operationen Y* , medan ett uttryck som **`X = Y`** skulle kunna utläsas på följande sätt:

Är X lika med Y?
Om X är lika med Y, så...
Gör så att X blir lika med Y!

Således instantieras inte:

```
X = 2 + 2
```

... som:

```
4 = 2 + 2
```

... utan som:

```
2 + 2 = 2 + 2
```

\= Lyckas om unifiering inte sker, och utgör alltså motsatsen till =. Vi skriver ett litet program för att exemplifiera.

```
olika(Tal1, Tal2) :-
Tal1 \= Tal2.

?- olika(2,2).
  no

?- olika(2,3).
  yes
```

== Kollar för *ekvivalens*. Detta innebär att prolog kontrollerar om två argument är identiska. Medan = ger värden åt variabler, kollar == om två prologtermer är identiska. Således skulle **`X = hej`** ge resultatet **`hej = hej`**, medan **`X == hej`** skulle ge resultatet **`no`** om **`X`** inte är identiskt med **`hej`**, och **`yes`** om **`X`** är identiskt med **`hej`**.

\== Lyckas om inte ekvivalens, dvs motsatsen till ==.

---

[1]Egentligen arbetar de flesta system inte med *unifiering* i strikt bemärkelse, utan termen *matchning* anses som adekvatare. Ur en praktiskt synvinkel spelar denna distinktion ingen roll, utan jag kommer att använda termen *unifiering* i detta kompendium.

Det finns flera andra predikat, och eftersom dessa kan skilja sig lite åt från prolog till prolog gör man bäst i att se efter i manualen vilka som finns i den prolog man använder, och vad de betyder.

Nya termer och begrepp

unifiering
=
\=
==
ekvivalens
\==
stoppvillkor
matchning

Övningar:

- Inga!

11 RETRACT/ASSERT – DYNAMISK ÄNDRING AV DATABASER

Tidigare har vi sett hur smidigt prolog arbetar med databaser. De program vi har sett har dock arbetat med "färdiga" databaser och vi har inte kunnat ändra något i dem. Ibland har man emellertid behov av att kunna ändra innehållet i en databas, vilket detta kapitel skall beskriva.

Det finns två predefinierade predikat som lägger in respektive plockar bort data ur baser. Predikatet **`assert/1`** tar en klausul som argument och lägger in den i databasen. **`asserta/1`** gör detsamma, men lägger dessutom klausulen först i raden av predikat med samma namn. **`assertz/1`** dito, men sist i raden av predikat med samma namn i databasen (*z* är ju sist i engelska alfabetet). **`retract/1`** tar också en klausul som argument, men tar bort argumentet ur databasen. Dessa predikat kan vara bra om man dynamiskt vill förändra något i en databas. Om man till exempel har en databas över sin bankbok och man omväxlande sätter in/tar ut pengar, kan en kombination av **`assert/1`** och **`retract/1`** göra den aktuella förändringen i databasen genom att sättas efter varandra.

En viktig detalj att lägga märke till rörande detta är att klausuler som läggs in med **`assert/1`** är "flyktiga" i den bemärkelsen att de inte skrivs in *fysiskt* i regelfilen, utan bara läggs in i prologs arbetsminne för tillfället. Om man avslutar prolog så kommer det man lagt in med **`assert/1`** att försvinna.

För att exemplifiera ovanstående utläggning skapar vi nu predikatet **`bankbok/1`** vars argument säger oss hur rika vi är. I databasen lägger vi således in vår bankbok:

```
bankbok(8000).
```

Nu har vi skrivit in det på det sätt vi är vana vid – direkt i regelfilen. För att exemplifiera användningen av **`assert/1`**, går vi i stället ut i Query Mode och lägger in bankboken därifrån:

```
?-  assert(bankbok(8000)).
  yes
```

Lägg märke till att **`assert/1`** tar ett *predikat* som argument. Därför måste man avsluta med två högerparenteser, vilket kan vara lätt att glömma!

Vi kollar nu hur mycket pengar vi har:

```
?-  bankbok(X).
X = 8000
```

Vi simulerar nu en bankomat, och skapar till den ändan predikatet **`ta_ut/1`** som vi använder för att ta ut pengar med.

```
ta_ut(Pengar) :-
  bankbok(Förmögenhet),
  NyFörmögenhet is Förmögenhet - Pengar,
  retract(bankbok(Förmögenhet)),
  assert(bankbok(NyFörmögenhet)).
```

Det första vi behöver veta när vi skall ta ut pengar är om vi har en bankbok. Således kollas detta överst i kroppen. **`Förmögenhet`** instantieras då som **`8000`**. **`NyFörmögenhet`** sätts till det som blir över när man dragit ifrån **`Pengar`** från **`8000`**. Om vi till exempel anropar med **`2000`**, på detta sätt:

```
?- ta_ut(2000).
```

...så sätts **NyFörmögenhet** till **Förmögenhet** minus **2000**, dvs **8000 - 2000**, vilket ger oss **6000**. Därefter lyfter vi ut hela predikatet **bankbok(8000)** ur databasen med predikatet **retract**, men sätter i stället in predikatet **bankbok(6000)** med predikatet **assert**. På detta sätt har vi förändrat innehållet i databasen.

Nya termer och begrepp

```
assert/1
asserta/1
assertz/1
retract/1
```

Övningar:

- Inga!

12 SUBRUTINER

En av de viktigaste sakerna att förstå vid programmering är hur program "ser" och använder sig av andra *delprogram* för att göra saker. Ytterst sällan gör ett och endast ett program[1] allt det man vill. Om inte annat blir sådana program oerhört långa och svårläsliga. Ett bättre alternativ är att skriva flera små delprogram som tar hand om varsin liten del av allt det som skall göras, och sedan låta ett stort *skalprogram* knyta samman de mindre delprogrammen och låta de ta hand om sina egna små specialiteter. En av fördelarna med detta är att programmet blir lättare att överskåda. En annan fördel är att det är lättare att se exakt var något klickar om programmet inte beter sig som man vill.

Vi börjar med ett enkelt aritmetiskt exempel:

```
/* Detta program tar en siffra som argument, dubblerar siffrans värde
och skriver ut resultatet på skärmen. */

dubblera(Siffra) :-
  NySiffra is Siffra + Siffra,
  write(NySiffra),
  nl.

/* Detta program tar en siffra som argument, kvadrerar dess värde och
skriver ut resultatet på skärmen. */

kvadrera(Siffra) :-
  NySiffra is Siffra * Siffra,
  write(NySiffra),
  nl.

/* Detta program tar en siffra som argument, tar fram kuben på dess
värde och skriver ut resultatet på skärmen. */

kuben(Siffra) :-
  NySiffra is Siffra * Siffra * Siffra,
  write(NySiffra),
  nl.
```

Om vi nu har en siffra, t ex 8, som vi vill ha det dubbla värdet, kvadraten och kuben för, så kan vi ju göra tre anrop som:

```
?- dubblera(8).
16
  yes

?- kvadrera(8).
64
  yes

?- kuben(8).
512
  yes
```

Men vi kan också göra ett överordnat program som använder de tre ovanstående programmen:

---

[1] I betydelsen *klausul*. Ett program kan bestå av en eller flera klausuler.

```
/* Detta program tar tre siffror som argument, dubblerar den första
siffran, kvadrerar den andra siffran och tar ut kuben på den tredje
siffran. Resultaten skrivs ut på skärmen. */

du_kva_kub(Siffra1, Siffra2, Siffra3) :-
  dubblera(Siffra1),
  kvadrera(Siffra2),
  kuben(Siffra3).
```

Om vi anropar detta program så tar det först **Siffra1**, letar upp **dubblera/1** i programfilen, kör dubblera med **Siffra1**. Därefter går **du_kva_kub/1** vidare i sin programkropp, hittar **kvadrera/1** med **Siffra2** som argument. Prolog letar då upp kvadrera i programfilen, kör det programmet. Därefter körs **kuben/1** på samma sätt.

```
?- du_kva_kub(2,3,4).
4
9
64
  yes
```

Om vi vill ha de tre värdena för samma siffra så anropar vi helt enkelt med samma siffra:

```
?- du_kva_kub(2,2,2).
4
4
8
  yes
```

Om vi tycker att det är en ”osnygg” lösning så kan vi skriva ett skalprogram, dvs ett program på en högre nivå som gör det enklare och snyggare att anropa. Man kallar här **du_kva_kub/3** för ett *topp-predikat*, dvs ett predikat som ”på toppen” anropar andra predikat ”längre ned” i strukturen.

```
/* Detta program tar en siffra som argument och anropar du_kva_kub med
den siffran som argument för alla du_kva_kub:s tre funktioner. */

dkb(Siffra) :-
  du_kva_kub(Siffra,Siffra,Siffra).
```

Observera hur **Siffra** här ”kopieras”! Detta är ett bra exempel på hur information transporteras i prolog!

Ett anrop:

```
?- dkb(3).
6
9
21
  yes
```

Vi har nu ett program, **dkb/3**, som anropar ett program, **du_kva_kub/3**, som i sin tur anropar tre andra program, **dubblera/1**, **kvadrera/1** och **kuben/1**. Självfallet kan även **dkb/1** användas som del av ett större program på ännu ”högre” nivå.

Givetvis så krävs det inte att varje predikat måste finnas ”utanför” funktionskroppen, man kan mycket väl blanda de två. Vi kan således specificera programmet **funktion/1**, som använder sig av både **dkb/1** och en ”egen” aritmetisk operation:

```
/* funktion/1 tar en siffra, adderar 2 till denna siffra, anropar
sedan dkb/1 med Siffra som argument. */

funktion(Siffra) :-
  NySiffra is Siffra + 2,
  write(NySiffra), nl,
  dkb(Siffra).
```

Vi visar hur detta "ser ut" genom att åskådliggöra hur programmen ligger i samma fil, och således "ser varandra".

Regelfil	Interpretator
dubblera(Siffra) :- NySiffra is Siffra + Siffra, write(NySiffra), nl. kvadrera(Siffra) :- NySiffra is Siffra * Siffra, write(NySiffra), nl. kuben(Siffra) :- NySiffra is Siffra * Siffra * Siffra, write(NySiffra), nl. du_kva_kub(Siffra1, Siffra2, Siffra3) :- dubblera(Siffra1), kvadrera(Siffra2), kuben(Siffra3). dkb(Siffra) :- du_kva_kub(Siffra,Siffra,Siffra). funktion(Siffra) :- NySiffra is Siffra + 2, write(NySiffra), nl, kuben(Siffra).	?- dubblera(8). 16 yes ?- kvadrera(8). 64 yes ?- kuben(8). 512 yes ?- du_kva_kub(2,3,4). 4 9 64 yes ?- du_kva_kub(2,2,2). 4 4 8 yes ?- dkb(3). 6 9 21 yes

En nyttig övning kan vara att "följa" anropen i interpretatorn, och se hur den letar sig igenom de olika klausulerna i regelfilen till vänster.

I ett tidigare kapitel såg vi hur man med inbäddade IF-THEN-ELSE-uttryck hanterade alternativ. Detta kan, som då nämndes, i prolog uttryckas på ett sätt som skiljer sig från andra programmeringsspråks syntax. Som framgått kan man i prolog använda samma predikatsnamn flera gånger, dvs samma regel kan existera i flera "versioner" – klausuler – som känner igen olika, specifika situationer. Om en regel inte passar i en viss given situation, provar prolog de andra ända tills det att en regel passar i situationen. Som tidigare nämnts så kallas denna inbyggda egenskap hos prolog att alltid söka alla lösningar *backtracking*, och behöver implementeras i andra programmeringsspråk, men finns tillhanda redan i prologs grundläggande struktur. Prolog kan ha arbetat sig aldrig så långt ned i en regelstruktur, om denna misslyckas så "backar" prolog upp i strukturen ("sökträdet") och försöker igen, antingen genom att prova en alternativ väg i samma klausul, eller, om en alternativa vägar inte ges, en annan klausul (med samma namn).

Detta gör att vi i stället för att i en regelfil lägga in en IF-THEN-ELSE-struktur kan ställa våra villkor medelst flera olika regler.

Vi definierar programmet **hejsan/0**, men i stället för att använda oss av inbäddade uttryck så definerar vi en *subrutin* **namnkoll/1** som tar hand om namnet, och sedar letar efter ett namnkoll som passar namnet i fråga, dvs uppfyller villkoret som ställs i koden.

```
/* Hejsan/0 ber om ett namn (efter att ha hälsat småartigt!), och
anropar sedan subrutinen namnkoll/1 med detta namn som argument.
namnkoll/1 skriver sedan ut "skräddarsydda" hälsningar till ett antal
personer. Om ingen av dessa motsvarar Namn, skrivs ett
"skönsmeddelande" ut. */

hejsan :-
  write('Hej, vad heter du?), nl,
  read(Namn),
  namnkoll(Namn).

namnkoll(Namn) :-
  Namn == gunnel,
  write('Arbetat hårt?'), nl.

namnkoll(Namn) :-
  Namn == robert,
  write('Sovit gott?'), nl.

namnkoll(Namn) :-
  Namn == janne,
  write('Är du hungrig?'), nl.

namnkoll(Namn) :-
  Namn == eva,
  write('Lagt schemat?'), nl.

namnkoll(Namn) :-
  write('Trevligt namn!'), nl.
```

Subrutinen **namnkoll/1** tar, som nämnts, här med sig namnet (dvs, **Namn**), och om detta motsvarar villkoren **gunnel**, **robert**, **janne** eller **eva** så skrivs respektive meddelande ut, annars lyckas den sista klausulen, eftersom den inte har något villkor för att lyckas.

Vi provar med ett anrop:

```
?- hejsan.
Hej, vad heter du?
```

... och svarar:

```
robert.
```

Nu letar prolog efter ett **namnkoll** som uppfyller kravet på att det lästa namnet är lika med **robert**. Det första **namnkoll** som prolog hittar (eftersom prolog läser uppifrån och ned) har "villkoret" **gunnel**. Därför misslyckas predikatet. Prolog "backar upp", och provar nästa klausul **namnkoll** det ser. Detta andra **namnkoll** har just villkoret **robert**, och programmet lyckas alltså och skriver ut det skräddarsydda meddelandet.

Det sätt på vilket namnkoll här är uppställt kan beskivas så att varje **namnkoll** med ett villkor kan sägas utgöra en IF-THEN-ELSE, dvs *Om Gunnel så skriv X, annars leta vidare*! Den sista **namnkoll**-klausulen kan ses som det ultimativa ELSE-uttrycket som utförs när inget annat lyckats.

Hur detta fungerar kan åskådliggöras med kommentarer på följande sätt:

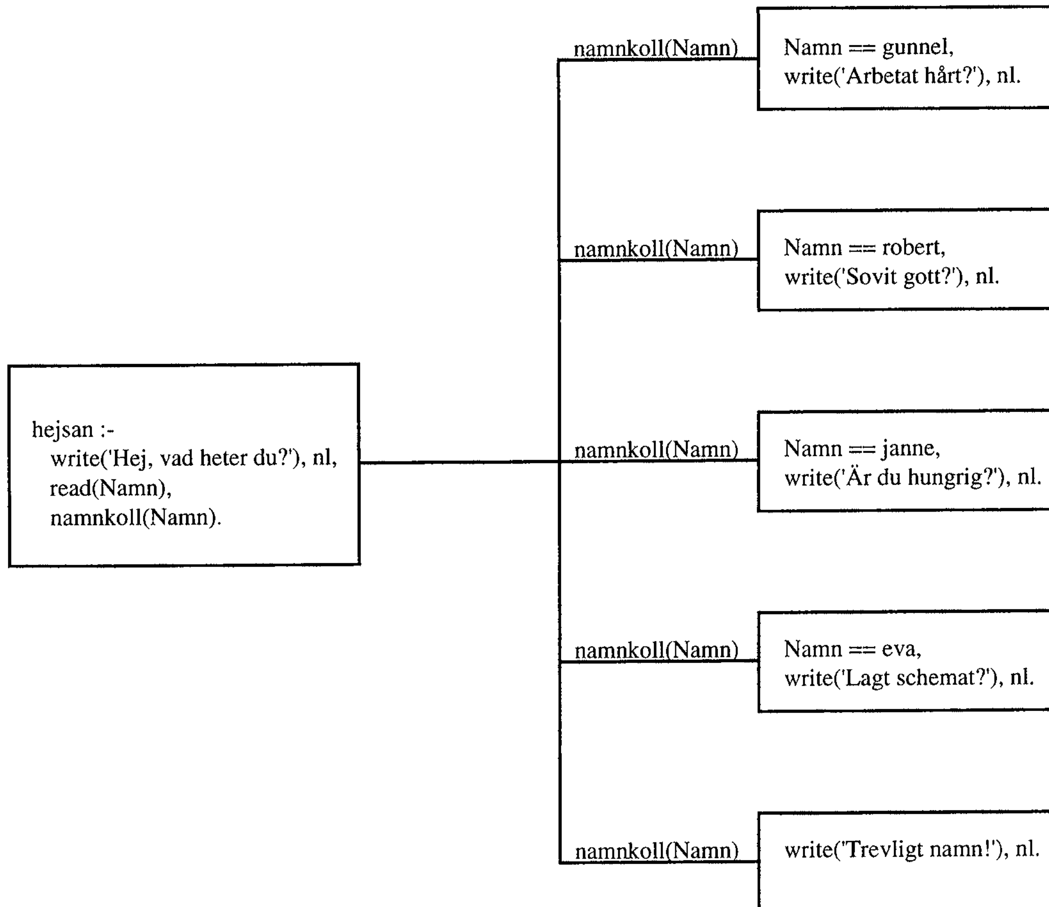
```
% Kommenterad version av hejsan/0 med subrutinen namnkoll/1.

hejsan :-                                  % Anrop (nollställigt)
  write('Hej, vad heter du?), nl,          % Skriv ut fråga
  read(Namn),                              % Läs in ett namn
  namnkoll(Namn).                          % Anropa namnkoll med namnet


namnkoll(Namn) :-
  Namn == gunnel,                          % Om namnet är Gunnel...
  write(Arbetat hårt?'), nl.               % ...så

                                           % Annars, leta vidare...

namnkoll(Namn) :-
  Namn == robert,                          % Om namnet är Robert...
  write('Sovit gott?'), nl.                % ...så

                                           % Annars, leta vidare...

namnkoll(Namn) :-
  Namn == janne,                           % Om namnet är janne...
  write('Är du hungrig?'), nl.             % ...så

                                           % Annars, leta vidare...

namnkoll(Namn) :-
  Namn == eva,                             % Om namnet är eva...
  write('Lagt schemat?'), nl.              % ...så

                                           % Annars, leta vidare...
namnkoll(Namn) :-
  write('Trevligt namn!'), nl.             % Om inget ovan fungerat!
```

Medan man i andra programmeringsspråk gärna arbetar med IF-THEN-ELSE-strukturer som liknar de först nämnda i detta kapitel (med eller utan det predefinierade **ifthenelse/3**), föredrar man i prolog att skriva program enligt den struktur som **namnkoll/1** har, dvs med flera åtskilda klausuler, vilka var och en täcker varsitt fall. Detta beror främst på två saker:

Det ena är att program som arbetar med "grunda" strukturer, dvs där villkoren inte ligger inbäddade, är snabbare. Det går helt enkelt ofta fortare för prolog att leta igenom flera klausuler efter olika alternativ än att "gräva ned sig" i djupa IF-THEN-ELSE-strukturer.

Dessutom blir programmen väldigt mycket mer lättlästa om varje alternativ på detta sätt får en egen klausul med en väl avgränsad uppgift.

Ett bra sätt att åskådliggöra hur ett program arbetar kan vara att rita ett *flödesdiagram* över programmets struktur. Ett flödesdiagram består av en mängd boxar förenade av linjer som visar hur de olika boxarna hänger ihop. Ofta använder man pilar i stället för linjer, för att visa just "flödet" av information i programmet. I detta fall, då vi inte har att göra med något som helst slags rekursion, är det inte lika nödvändigt. Vi läser helt enkelt programmet från vänster till höger. Nedan visas ett flödesdiagram över **hejsan/0**.



Ett bra tips när man skall lösa en programmeringsuppgift kan vara att rita ett flödesdiagram (som vi redan stött på) över uppgiften, dels för att man väldigt lätt ser exakt hur många moduler (dvs, delar) som behövs, dels för att man på detta sätt lätt kan se hur de olika modulerna hänger ihop.

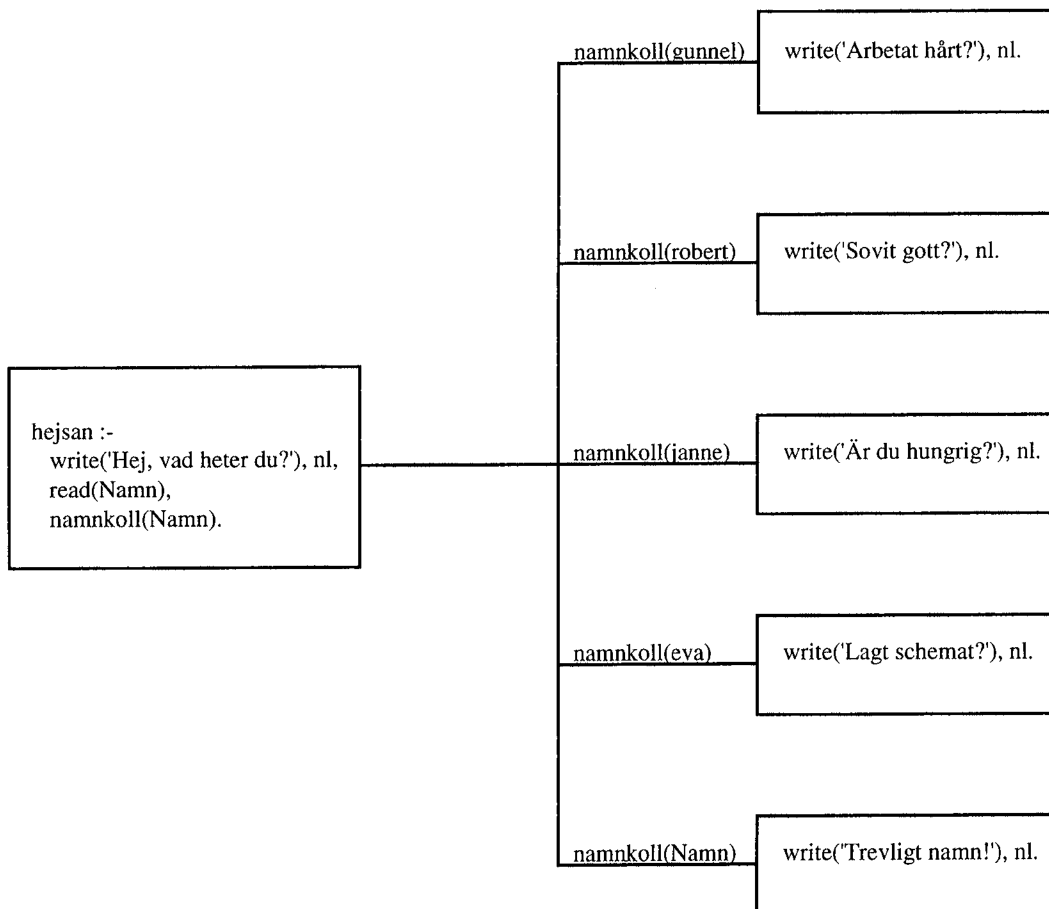
Ett sätt att snabba upp programmet är att lägga in namnen direkt i predikathuvudet, vilket benämnes *indexering*. Programmet får då följande utseende:

```
% Indexerad version av hejsan/0 med subrutinen namnkoll/1.

hejsan :-                              % Anrop (nollställigt)
  write('Hej, vad heter du?), nl,      % Skriv ut fråga
  read(Namn),                          % Läs in ett namn
  namnkoll(Namn).                      % Anropa namnkoll med namnet


namnkoll(gunnel) :-                    % Om namnet är Gunnel...
  write(Arbetat hårt?'), nl.           % ...så

                                       % Annars, leta vidare...

namnkoll(robert) :-                    % Om namnet är Robert...
  write('Sovit gott?'), nl.            % ...så

                                       % Annars, leta vidare...

namnkoll(janne) :-                     % Om namnet är janne...
  write('Är du hungrig?'), nl.         % ...så
```

```
                                           % Annars, leta vidare...

namnkoll(eva) :-                           % Om namnet är eva...
  write('Lagt schemat?'), nl.              % ...så

                                           % Annars, leta vidare...
namnkoll(Namn) :-
  write('Trevligt namn!'), nl.             % Om inget ovan fungerat!
```

Orsaken till att denna version blir snabbare än den tidigare beror på att prolog nu slipper ett anrop, dvs ekvivalenskontrollen.

Ett flödesdiagram över detta program får följande utseende:



Nya termer och begrepp

delprogram
skalprogram
topp-predikat
backtracking
sökträd
flödesdiagram
indexering

Övningar:

- Gör flödesdiagram till alla tidigare program i kompendiet. ("Typiskt en sån övning man inte gör" sa en f d student (numera lärare). OK! Men det är rätt bra att göra den, faktiskt!!)

- Skriv ett program som kollar accessrätt (till t ex en datorsal!). Skapa ett predikat **`open/0`** som ber om ett namn, medelst en subrutin kollar om namnet finns med i en lista över personer som har accessrätt. Om namnet finns med i listan så skall programmet meddela att "dörren nu är öppen", om namnet inte finns med i listan så skall programmet meddela att man inte har accessrätt. **IF-THEN-ELSE**-strukturer skall inte användas.

- I programmet **`namnkoll/1`** så används predikatet **`write/1`**. Skriv om **`hejsan/0`** så att man ersätter alla dessa write-anrop med en enda, generell write-sats i toppanropet.

13 EN INTRODUKTION TILL GENERATIV GRAMMATIK

Ett av de ursprungliga skälen bakom skapandet av prolog var att man ville ha ett programmeringsspråk som lämpade sig för lingvistiska uppgifter. Man ville helt enkelt ha ett språk i vilket man lätt kunde implementera formaliserade lingvistiska beskrivningar. Jag ämnar här ge en kortfattad beskrivning av en *generativ grammatik*, vilket är det slags grammatik som man oftast använder sig av inom teoretisk lingvistik.

De flesta har någoin gång i sitt liv utsatts för begreppet grammatik, antingen vid undervisning i modersmålet eller vid studiet av främmande språk. Begreppet grammatik kan dock innebära lite olika saker. Lite förenklat kan man skilja på tre slags grammatiker:

En *normativ* (eller *preskriptiv*) grammatik ger "regler" för hur språk skall användas, och tillhandahåller sålunda rätt-eller-fel-bedömningar rörande språkbruk. Ett exempel på detta skulle kunna vara att en normativ grammatik lär ut att det på svenska heter *bättre än jag* och att det är fel att säga *bättre än mig*. Det är den här sortens grammatik man stöter på när man går kurser i främmande språk, och den tjänar där sitt syfte med den äran. (Att mycket av det som lärs ut inte stämmer lär man upptäcka själv när man besöker det främmande landet i fråga och upptäcker att t ex alla fransmän pratar "fel"!)

En *deskriptiv* grammatik söker helt enkelt beskriva (därav namnet) hur ett språk faktiskt ser ut, och om vi använder exemplet ovan skulle en deskriptiv beskrivning av svenskt talspråk kanske ge svaret att det i 95 % av fallen heter *bättre än mig*, och i 5 % av fallen heter *bättre än jag* i svenska. En deskriptiv grammatik uttalar sig inte normativt om rätt eller fel – det som talare av språket förstår och inte reagerar emot är rätt. Däremot kan man inom den deskriptiva grammatiken säga att former som *mer bättre än mig äro* är ogrammatiska, eftersom ingen grupp av talare av språket (svenska i detta fall) skulle gå med på att formen är korrekt svenska. Likaså skulle man betrakta former som *ett grön bilar* som ogrammatiska.

En *generativ grammatik* försöker beskriva ett språk *in extenso*, så att säga. Man vill att grammatiken dels skall kunna generera, skapa, ett språks meningar och yttranden. Dessutom vill att man att grammatiken skall kunna skapa ett språks *samtliga* meningar. Ett ytterligare krav man har är att en generativ grammatik för svenska inte bara skall kunna generera alla svenskans meningar, utan dessutom inte skall kunna generera en enda mening som *inte* är svenska. Man behöver inte fundera särskilt länge innan man inser att det finns vissa problem med detta. Det första man kanske kommer att tänka på är kravet "svenskans samtliga meningar"! Man behöver inte tänka länge på hur språk ser ut innan man inser att ett av språks drag är att de är oändliga. Detta innebär att hur många svenska meningar man än har sagt eller genererat, kan man alltid generera en till, eller flera till, eller, i själva verket mer än dubbelt så många till. Hur skall man kunna skriva en grammatik som beskriver ett oändligt antal meningar?

Jag skall här i korthet redogöra för grunderna i hur en generetiv grammatik för svenska kan se ut.

Vi tittar på några "typiska" svenska satser:

> *Anna sover.*
> *Bo ser Anna.*
> *En liten pojke slöar.*
> *Den lilla flickan ger katten ett fat mjölk.*

Man ser här att svenska satser (nästan) alltid kan delas upp i två delar: vad som händer och vem/vad som orsakar eller är föremål för händelsen:

Anna	*sover.*
Bo	*ser Anna.*
En liten pojke	*slöar.*
Den lilla flickan	*ger katten ett fat mjölk.*

Den första kolumnen cirklar kring ett *nomen* (eller substantiv) och den andra kolumnen cirklar kring ett verb. En gängse syn är därför att svenska satser kan delas upp i *nominalfraser*, NP:n[1], och *verbfraser*, VP:n. Dessa kan i sin tur ha en inre struktur, men lite förenklat kan vi åskådliggöra strukturen på följande sätt:

NP	VP
Katten	*sover*
En pojke	*skrattar mycket*
En liten flicka	*ser en pojke*
Den fula ankungen	*grät när han var ensam*
Några väldigt elaka vargar	*skulle vilja äta upp några får*

... osv.

Nedan följer en kort sammanfattning de fraskategorier man brukar räkna med i svenskan, samt de *regler* som beskriver deras struktur. Lägg märke till att denna beskrivning inte är utan problem, och att det krävs mer för att få en fullt fungerande grammatisk beskrivning av svenska. Denna beskrivning tjänar mest syftet som förberedelse till kapitel 14 om parsning.

Innan vi presenterar fraskategorierna måste vi emellertid förklara lite av de symboler som används.

Teckenförklaringar

→ "kan utläsas som", "kan skrivas om som", "kan expanderas som"

Dvs, symbolen till vänster om pilen kan göras om till symbolerna till höger om pilen. Om vi t ex har

S → NP VP

... så kan vi göra om S till NP följt av VP. NP och VP måste komma i den ordning de står i regeln.

* "noll eller flera"

Om det sitter en asterisk vid en kategori så innebär detta att man kan använda kategorin ingen gång eller hur många gånger som helst.

( ) valfritt element, dvs "noll eller ett"

Kategorier inom parenteser innebär att de är optionella element, dvs kan användas, men behöver inte användas. I praktiken är det detsamma som att säga att de kan användas noll eller en gång.

---

[1] Att det heter *NP*, i stället för *NF*, beror givetvis på att termen/begreppet är importerad/t från engelska *Noun Phrase*.

{ } betyder att man kan välja en av de horisontella raderna

Regler som står inom bågparenteser indikerar alternativa regler. Man måste välja en av de horisontella rader som står inom parenteserna.

Man räknar i svenskan med följande syntaktiska kategorier:

Nominalfraser

Har som huvudord ett *nomen*, dvs substantiv. Före detta kan det stå *determinerare* (artiklar m m), *kvantifierare* eller *adjektivfraser*. Substantivet kan ocksa följas av en bisats (S').

		(Det)			
NP	→	{ }	**(AP)***	**N**	**(S')**
		(Kvant)			
		En	*liten*	*pojke*	*som sover gott*
		Den	*lilla*	*flickan*	*som är rik*
		Några	*små*	*katter*	

Verbfraser

Verbfraser har som huvudord ett *verb*, som kan föregås av ett eller flera *hjälpverb*. Beroende på om verbet är *intransitivt* (noll-ställigt), *transitivt* (ett-ställigt) eller *bitransitivt* (två-ställigt) kan det följas av ingen, en eller två NP:n, samt en optionell *adverbfras*.

			Vitr			
VP	→	**(Aux)***	**Vtr**	**NP**		**(AdvP)***
			Vbtr	**NP**	**NP**	
			sover			
		vill	*sova*			
			ser	*små pojkar*		*ibland*
		skulle	*se*	*pojken*		*mycket sällan*
			ger	*pojken*	*boken*	*nu*
		hade velat	*ge*	*flickan*	*stora A*	*varje dag*

Adjektivfraser

Har som huvudord ett *adjektiv*, som kan föregås av ett optionellt *adverb*. Adjektiv har den egenskapen att de kan användas flera i rad, som i t ex *En fin liten bil.*

AP	→	**(Adv)***	**Adj**	**(AP)**
			fint	
			litet ,	*fint*
		mycket	*liten*	
		ganska	*stort* ,	*oerhört gulligt*

Prepositionsfraser

Har som huvudord en *preposition* som följs av en NP.

PP	→	**Prep**	**NP**
		på	*det stora bordet*
		under	*tiden*
		genom	*Europas längsta undervattenstunnel*

Adverbfraser

Består av *lexikala adverb* (dvs oböjliga adverb, inte verb *avledda* från adjektiv), optionellt föregångna av adverb.

AdvP	→	**(Adv)***	**Adv**
			fortfarande
		mycket	*sällan*

Ovanstående genomgång är givetvis mycket kortfattad och förenklad. För djupare kunskap i området rekommenderas en titt i någon av alla de böcker om modern lingvistik som finns att tillgå.

I kapitel 14 kommer huvudsakligen nominalfraser att behandlas.

Nya termer och begrepp

implementering
generativ grammatik
normativ grammatik
preskriptiv grammatik
deskriptiv grammatik
nomen
nominalfras, NP
verbfras, VP
→
*
{ }
()
regler
determinerare, Det
kvantifierare, Kvant
adjektivfras, AP
intransitiva verb
transitiva verb
bitransitiva verb
prepositionsfras, PP
adverbfras, AdvP
lexikala adverb
avledda adverb

Övningar:

- Inga.

14 PARSNING

På grund av sina inbyggda backtrackingmekanismer är prolog ytterst lämpat för att *parsa* satser, dvs ta en språklig sats och analysera den i dess beståndsdelar enligt en grammatisk modell man valt för sin beskrivning. Jag skall i detta avsnitt visa hur man smidigt kan bygga upp en svensk minigrammatik.

Vi börjar med att presentera en enkel grammatik:

S → *NP VP*
NP → *pojken*
VP → *sover*

Att göra om detta till prologkod är inte svårt :

```
s((np(N),vp(V))) :-
  np(N),
  vp(V).

np(pojken).

vp(sover).
```

Vi gör nu ett anrop:

```
?- s(X).
     X = np(pojken), vp(sover)
```

Som synes så kan vi här binda en variabel till en grammatik som vi definierat. För att göra grammatiken lite mer sofistikerad än ovanstående (prova gärna att provköra den!) får vi dock lägga till lite saker. Således skriver vi en ny grammatik:

```
s((np,vp),FörstaOrd,SistaOrd) :-
  np(N,FörstaOrd,NästaOrd),
  vp(V,NästaOrd,SistaOrd).

np(pojken,FörstaOrd,NästaOrd).

vp(sover,NästaOrd,SistaOrd).
```

FörstaOrd och **NästaOrd** är här variabler som visar att satsen analyseras ord för ord från vänster till höger. Man skulle även kunna beskriva dem som ett slags *länkad kedja*, där **FörstaOrd** är *pojken* och **NästaOrd1** är *sover*, och **NästaOrd2** är den tomma listan, som vi senare skall lägga till i reglerna.

Schematiskt skulle det kunna åskådliggöras sålunda:

```
FörstaOrd          NästaOrd1          NästaOrd2
           pojken              sover              []
```

Detta vill säga att det första ordet är *pojken*, det andra ordet är *sover*, och nästa ord är den tomma listan, vilket betyder att meningen är fullt genomgången. Den tomma listan i slutet av meningen kan sägas motsvara punkt i vanlig text. Innan vi förklarar hur detta fungerar i detalj så kan vi beskriva hur själva meningen kommer in i prolog, och vad den tomma listan egentligen har där att göra.

Meningar läses lättast i form av listor (dessa har redogjorts för tidigare), och en typisk mening skulle då kunna ha utseendet

[det,grå,huset,står,på,heden]

Som vi redan sett består listor av listans första element och resten av listan. Denna mening läses sedan av prolog ord-för-ord tills det inte finns fler ord att läsa, dvs nästa ord är lika med den tomma listan. Detta görs på ett smidigt sätt i prolog med något som kallas *differenslistor*. Differenslistor är två relaterade listor där den första listan är en fylld lista som man för tillfället tittar på, och den andra listan är den ännu inte kända listan (resten av listan). Detta kan skrivas som två helt vanliga argument, sålunda:

([FörstaElementet|Resten],Resten)

Andra notationsformer som förekommer är bland andra:

([FörstaElementet|Resten]-Resten)

...och:

([FörstaElementet|Resten]/Resten)

Dessa är helt ekvivalenta betydelsemässigt.

Om vi skulle ta exempelmeningen ovan och läsa den med ett predikat som arbetar med en grammatik som den ovan beskrivna (även om vi ännu inte skrivit regler som täcker in just denna meningstyp), så skulle vi börja med att plocka in det första ordet, **det**, och då få utseendet:

[det|[grå,huset,står,på,heden]],[grå,huset,står,på,heden]

...där:

FörstaElementet = **det**

Resten = **[grå,huset,står,på,heden]**

Vi har således plockat av resten av listan och gjort den "direkt tillgänglig". Om det första ordet, dvs det ord som instantierats som **FörstaElementet**, hittas i grammatiken så letar grammatiken automatiskt vidare med **Resten** och läser nästa ord, **grå**, på samma sätt. När vi nått situationen:

[heden|[]], []

där:

FörstaElementet = **heden**

Resten = **[]**

...dvs, nästa ord är den tomma listan, så har vi kommit igenom hela meningen, och satsen har således lyckats instantieras, ord för ord, och svarar alltså mot en regel i vår grammatik.

Formellt skulle detta kunna beskrivas enligt mönstret:

[FörstaOrd, $\mathbf{NästaOrd_1}$**,** $\mathbf{NästaOrd_2}$**, ... ,** $\mathbf{NästaOrd_n}$**]**

...tills:

$\mathbf{NästaOrd_n}$ = **[]**

På grund av prologs inherenta backtracking – att alltid pröva alla möjliga sätt att lösa något innan den ger upp – så räcker det med att någon regel kan lösa upp satsen i dess beståndsdelar. Eller, sagt på ett annat sätt, om det finns någon regel som svarar mot input-satsen, så kommer prolog att hitta den.

Nåväl, även om några detaljer säkert är oklara så torde principen nu vara klar. Vi skriver en liten grammatik för svenska nominalfraser för att visa mera fullständigt hur det går till. Grammatiken skall klara av att parsa fraser som:

stuga
hus
en grön stuga
ett grönt hus
den gröna stugan
det gröna huset
några gröna stugor
några gröna hus

Vi börjar med att skriva själva satsregeln på toppnivå:

```
%SATSREGEL
s(s(NP),FörstaOrd,NästaOrd) :-
  np(NP,FörstaOrd,NästaOrd).
```

Denna regel svarar mot den grammatiska omskrivningsregeln

S --> *NP*

Vi förklarar denna satsregel i detalj.

```
s(s(NP),FörstaOrd,NästaOrd) :-
  np(NP,FörstaOrd,NästaOrd).
```

s är här det översta predikatet, satsen. Satsen har som "FörstaOrd" en NP, enligt mönstret:

```
  FörstaOrd          NästaOrd
            np(NP)
```

Att vi här kallar det för "ord" är måhända lite missvisande, men strukturen är densamma som i beskrivningen ovan, vi har bara inte kommit ner till själva orden ännu. Att vi skriver **np(NP,** i stället för bara **NP** är att vi behöver ett predikat **np/3** för NP:n också.

Vi skriver nu in några NP-regler:

```
%NP-REGEL 1
np(np(N),FörstaOrd,NästaOrd) :-
  n(N,FörstaOrd,NästaOrd).
```

Ovanstående regel svarar mot den grammatiska omskrivningsregeln:

NP --> *N*

... och motsvarar situationen:

```
  FörstaOrd          NästaOrd
            pojken              []
```

Vi skriver en regel till:

```
%NP-REGEL 2
np(np(Det,N),FörstaOrd,NästaOrd2) :-
  det(Det,FörstaOrd,NästaOrd1),
  n(N,NästaOrd1,NästaOrd2).
```

... som svarar mot:

NP --> Det N

... och mönstret:

FörstaOrd		NästaOrd1		NästaOrd2	
	en		*pojke*		*[]*

Vi lägger till ytterligare en regel:

```
%NP-REGEL 3
np(np(Det,Adj,N),FörstaOrd,NästaOrd3) :-
  det(Det,FörstaOrd,NästaOrd1),
  adj(Adj,NästaOrd1,NästaOrd2),
  n(N,NästaOrd2,NästaOrd3).
```

... som svarar mot:

NP --> Det Adj N

... och mönstret:

FörstaOrd		NästaOrd1		NästaOrd2		NästaOrd3	
	en		*liten*		*pojke*		*[]*

Nu har vi skrivit in tre stycken omskrivningsregler för svenska nominalfraser. Vad som saknas är nu ett litet lexikon. Vi skriver då vårt lexikon:

```
% LEXIKON
n(n(pojke),[pojke|NästaOrd],NästaOrd).
n(n(flicka),[flicka|NästaOrd],NästaOrd).
n(n(hus),[hus|NästaOrd],NästaOrd).
n(n(djur),[djur|NästaOrd],NästaOrd).

det(det(en),[en|NästaOrd],NästaOrd).
det(det(ett),[ett|NästaOrd],NästaOrd).

adj(adj(liten),[liten|NästaOrd],NästaOrd).
adj(adj(litet),[litet|NästaOrd],NästaOrd).
```

Detta lexikon motsvarar reglerna:

N --> pojke, flicka, hus, djur
Det --> en, ett
Adj --> liten, litet

Lägg märke till att vi lagt in orden som atomer, och alltså inte med citationstecken.

Som synes är hela grammatiken treställig - dvs alla i grammatiken ingående predikat är treställiga. Som första argument har vi själva grammatiken, där regler försöker unifieras med det som läses som andra argument, vilket är första ordet i en lista som innehåller en

sats. Som tredje argument har vi resten av listan, som kommer att läsas om det andra argumentet hittas i lexikon.

En sak som kan verka förbryllande är utseendet på det första argumentet. Varför skriver vi:

```
n(n(pojke),[pojke|NästaOrd],NästaOrd).
```

... och inte helt enkelt:

```
n(pojke,[pojke|NästaOrd],NästaOrd).
```

Detta vore väl mycket klarare! Vi läser ju, som andra argument, just ordet *pojke*, och vill bara kolla om det finns i lexikon, innan vi fortsätter med nästa ord. Saken är den att vi inte bara vill ha ut ordet i prologs svar, utan också en grammatisk analys. Vi vill kort sagt veta vilken ordklass ordet har. Därför skriver vi in den i lexikonet. Detta leder till att vi som utdata inte bara får:

```
pojke
```

... utan:

```
n(pojke)
```

Om vi inte lägger in analysen i lexikon, så kommer vi av frasen *en pojke* att få ut analysen:

```
s(np(en,pojke))
```

... medan vi i själva verket vill ha hela analysen, sålunda:

```
s(np(det(en),n(pojke)))
```

Att vi på detta sätt skriver in analysen explicit i lexikonet innebär att variablerna en efter en "fylls på" med var sin lilla delanalys. Sålunda kommer en fras med strukturen

```
np(Det,Adj,N)
```

... att instantieras, i tur och ordning, som:

```
np(Det,Adj,N)
     ↓
np(det(den),Adj,N)
             ↓
np(det(den),adj(lilla),N)
                       ↓
np(det(den),adj(lilla),n(pojken))
```

Vi behöver nu bara skriva ett *parsningspredikat*, dvs ett predikat som tar en sats som argument och försöker matcha den mot reglerna och lexikonet i grammatiken. Detta predikat behöver inte vara treställigt, utan själva satsen räcker fint:

```
% PARSNINGSPREDIKAT
parsa(Sats) :-
  s(Analys,Sats,[]),
  write(Analys).
```

Ett annat sätt att få analysen utskriven är att göra parsningspredikatet tvåställigt:

```
% ALTERNATIVT PARSNINGSPREDIKAT
parsa(Sats,Analys) :-
  s(Analys,Sats,[]).
```

I detta fall får man analysen returnerad automatiskt och "slipper" skriva ut den med ett write-uttryck.

I dessa predikat motsvarar `Sats` den sats man vill parsa, t ex `[en,liten,pojke]`. Predikatet `parsa/1` "skickar ned" satsen till predikatet `s/3` som via regler ned till lexikonet försöker hitta en väg sådan att `NästaOrd` i lexikonet till slut är den tomma listan. Således motsvarar `Analys` i predikatet den ackumulerade analysen ord-för-ord, och `Sats` den sats man läser ord-för-ord. Den tomma listan matchas mot den situation där resten av listan är tom. Eftersom vi vill att analysen skrivs ut när den väl är funnen så lägger vi till ett `write`-anrop.

Vi gör nu ett anrop, varefter vi går igenom parsningen steg för steg.

```
?- parsa([en,flicka]).
s(np(det(en),n(flicka)))
  yes
```

Det gick ju bra! Vi provar igen:

```
?- parsa([ett,hus]).
s(np(det(ett),n(hus)))
  yes
```

Hur går nu detta till? Vi "följer" ett anrop steg för steg. Vi anropar predikatet `parsa` med en `Sats` och tittar på och kommenterar (i liten stil) ett `trace` av processningen. (Extra radmatningar är inlagda för tydlighetens skull!)

```
?- parsa([en,flicka]).
```
Anrop.

```
  (1:0) Call: parsa([en,flicka])
```
Predikatet `parsa/1` anropas med satsen `[en,flicka]`.

```
  |(2:1) Call: s(_1,[en,flicka],[])
```
Predikatet `s/1` anropas med satsen `[en,flicka]`. Lägg märke till att `Analys` är oinstantierat samt att argument tre redan i anropet är den tomma listan.

```
  | (3:2) Call: np(_1,[en,flicka],[])
```
Predikatet `np/3` anropas med satsen `[en,flicka]`.

```
  | |(4:3) Call: n(_6,[en,flicka],[])
```
Predikatet `n/3` anropas med satsen `[en,flicka]`.

```
  | |(4:3) Fail: n(_6,[en,flicka],[])
```
Prolog lyckas inte hitta satsens första element (`en`) i lexikon som ett nomen, och "failar".

```
  | (3:2) Redo: np(_1,[en,flicka],[])
```
Prolog backtrackar upp till första närmast liggande alternativa väg: NP-regel 2.

```
  | |(4:3) Call: det(_6,[en,flicka],_10)
```
Predikatet `det/3` anropas.

```
|  |(4:3) Exit: det(det(en),[en,flicka],[flicka])
```
Prolog lyckas instantiera satsens första element (`en`) i lexikonet som `det(en)`.[1] `NästaOrd` instantieras då som resten av satsen: `[flicka]`.

```
|  |(4:4) Call: n(_7,[flicka],[])
```
Predikatet **n/3** anropas nu (nästa steg i NP-regel 2) med det som är kvar av satsen: **[flicka]**.

```
|  |(4:4) Exit: n(n(flicka),[flicka],[])
```
Prolog lyckas hitta en "match" i lexikonet: `n(flicka)`.

```
|  (3:2) Exit: np(np(det(en),n(flicka)),[en,flicka],[])
```
`Sats` i anropet har instantierats, och prolog backar upp med den i lexikonet instantierade NP-strukturen `np(np(det(en),n(flicka))` till närmaste högre nivå: `np/3`.

```
|(2:1) Exit: s(s(np(det(en),n(flicka))),[en,flicka],[])
```
Denna struktur hämtas upp till nästa nivå: `s/3`. Prolog har nu lyckats instantiera anropets sats med en struktur i lexikonet. Således kan man säga att predikatet `parsa`:s kropp nu "ser ut" på följande sätt:
```
parsa([en,flicka]) :-
      s(s(np(det(en),n(flicka))),[en,flicka],[]),
      write(np(det(en),n(flicka))).
```

```
|(2:5) Call: write(s(np(det(en),n(flicka))))
s(np(det(en),n(flicka)))
```
`Analys` skrivs ut.

```
Exit: write(s(np(det(en),n(flicka))))
```
Prolog backar ur `write`-predikatet.

```
 (1:0) Exit: parsa([en,flicka])
yes
```
Prolog backar ur topp-predikatet.

Sammanfattningsvis kan det sägas att predikatet **`parsa/1`** anropas med en sats, och söker efter ett sätt att matcha de i satsen ingående orden i den samling regler som grammatiken innehåller.

Så långt allt väl. Men vi skall nu se att den här grammatiken bjuder på vissa problem! Vi provar ett anrop till:

```
?- parsa([en,hus]).
s(s(np(det(en),n(hus)))
  yes
```

Denna sats är ju inte riktig, men ges ändå en *parse*. Orsaken är att så som våra regler och vårt lexikon är specificerade tas ingen hänsyn till vissa restriktioner i språket. Vår grammatik uppvisar således en total negligens för *genus*. Alltså måste vi på något sätt lägga in den preciseringen att satser bara kan ges en godkänd parse om artikeln och nomenet har samma genus.

Det görs helt enkelt genom att lägga in särdraget **`Genus`** i grammatiken. I reglerna lägger vi in det som en variabel, och i lexikonet lägger vi in genus för varje ord. Vår utbyggda grammatik – som nu tar hänsyn till genus – ser alltså ut sålunda:

```
% Satsregel
s(s(NP),Genus,FörstaOrd,Nästaord) :-
  np(NP,Genus,FörstaOrd,NästaOrd).
```

[1]OBS! Om vi hade i stället hade skrivit lexikonet som **det(en,[en|Rest],Rest)** så hade prolog här returnerat enbart **en**. Eftersom vi nu skrev **det(det(en),[en|Rest],Rest)** så får vi även ut kategorin.

```
% NP-regler
np(np(N),Genus,FörstaOrd,NästaOrd) :-
  n(N,Genus,FörstaOrd,NästaOrd).

np(np(Det,N),Genus,FörstaOrd,NästaOrd2) :-
  det(Det,Genus,FörstaOrd,NästaOrd1),
  n(N,Genus,NästaOrd1,NästaOrd2).

% Lexikon
n(n(pojke),utrum, [pojke|Rest],Rest).
n(n(flicka),utrum, [flickaRest],Rest).
n(n(hus),neutrum, [hus|Rest],Rest).
n(n(djur),neutrum, [djur|Rest],Rest).

det(det(en),utrum, [en|Rest],Rest).
det(det(ett),neutrum, [ett|Rest],Rest).

% Parsningspredikat
parsa(Sats) :-
  s(Analys,Genus,Sats,[]),
  write(Analys), nl,
  write(Genus).
```

Som synes så ligger genus specificerat på samma position, dvs som det andra argumentet, i både regler och lexikon. Detta krävs för att det skall kunna unifieras. I parsningspredikatet lägger vi till ett **write**-uttryck för att även få genus utskrivet.

Om vi nu provkör:

```
?- parsa([en,flicka]).
s(np(det(en),n(flicka)))
utrum
  yes

?- parsa([ett,flicka]).
  no
```

Orsaken till att den första satsen lyckas, men inte den andra, är att när det första ordet i satsen, determineraren, instantieras i lexikonet, så byts varibeln **Genus** samtidigt ut mot det genus som står specificerat i lexikon för den aktuella determineraren. I första satsen så binds alltså **Genus** till **utrum** när **en** hittas i lexikon, eftersom **en** står specificerat som **utrum**. Detta "följer sedan med" när prolog söker vidare i lexikonet. Medan prolog från början letade efter ett ord **en**, som skulle ha ett **Genus**, så letar prolog nu alltså efter ett ord **flicka** som skall vara **utrum**. Den andra satsen misslyckas således därför att när **ett** hittas så binds **Genus** till neutrum, varefter prolog letar efter ett ord **flicka** i lexikon som har specifikationen **neutrum** i andra position. Något sådant ord finns inte i lexikon, och hela parsningen misslyckas alltså, eftersom det inte gick att hitta någon sats i grammatiken som motsvarande satsen *en flicka*, med orden i den ordningen och bägge orden specificerade som neutrum. Om vi vill ha ett roligare meddelande än **no** när en sats inte lyckas så kan vi lägga till ett parsningspredikat som talar om för oss lite artigare att satsen var ogrammatisk. Vi lägger således till, *efter* det förra predikatet[2], med följande utseende:

```
parsa(Sats) :-
  write('Satsen är inte grammatisk!'), nl.
```

[2]Om du inte förstår varför, så prova att lägga det före och provköra med några satser, grammatiska såväl som ogrammatiska enligt din grammatik!

Här behöver vi inte göra något annat än att meddela att satsen var ogrammatisk. Om den var grammatisk så hade den ju analyserats av det första **parsa/1**. Anropen enligt ovan får nu utseendet:

```
?- parsa([en,flicka]).
s(np(det(en),n(flicka)))
utrum
  yes

?- parsa([ett,flicka]).
Satsen är inte grammatisk!
  yes
```

Som vår grammatik hittills är specificerad klarar den bara av obestämda NP:n som *en flicka*, *ett hus* osv. Vi vill nu bygga ut den till att klara av även bestämda NP:n som *den flickan*, *det huset* osv. Detta sker lätt genom att lägga till de nya orden i lexikon, men som vi nyss sett måste vi undvika att parsern accepterar satser som *den flicka*, *ett huset*. Detta sker givetvis genom att vi efter **Genus** i reglerna och lexikon specificerar *species* (dvs *bestämdhet*) i såväl regler (som en variabel **Species**) som lexikon (som atomer **obest** och **best**).

Vår nya utbyggda grammatik får då utseendet:

```
% Satsregel
s(s(NP),Genus,Species,FörstaOrd,NästaOrd) :-
  np(NP,Genus,Species,FörstaOrd,NästaOrd).

% NP-regler
np(np(N),Genus,Species,FörstaOrd,NästaOrd) :-
  n(N,Genus,Species,FörstaOrd,NästaOrd).

np(np(Det,N),Genus,Species,FörstaOrd,NästaOrd2) :-
  det(Det,Genus,Species,FörstaOrd,NästaOrd1),
  n(N,Genus,Species,NästaOrd1,NästaOrd2).

np(np(Det,Adj,N),Genus,Species,FörstaOrd,NästaOrd3) :-
  det(Det,Genus,Species,FörstaOrd,NästaOrd1),
  adj(Adj,Genus,Species,NästaOrd1,NästaOrd2),
  n(N,Genus,Species,NästaOrd2,NästaOrd3).

% Lexikon
n(n(pojke),utrum,obest, [pojke|NästaOrd],NästaOrd).
n(n(pojken),utrum,best, [pojken|NästaOrd],NästaOrd).
n(n(flicka),utrum,obest, [flicka|NästaOrd],NästaOrd).
n(n(flickan),utrum,best, [flickan|NästaOrd],NästaOrd).
n(n(hus),neutrum,obest, [hus|NästaOrd],NästaOrd).
n(n(huset),neutrum,best, [huset|NästaOrd],NästaOrd).

det(det(en),utrum,obest, [en|NästaOrd],NästaOrd).
det(det(den),utrum,best, [den|NästaOrd],NästaOrd).
det(det(ett),neutrum,obest, [ett|NästaOrd],NästaOrd).
det(det(det),neutrum,best, [det|NästaOrd],NästaOrd).

adj(adj(liten),utrum,obest, [liten|NästaOrd],NästaOrd).
adj(adj(litet),neutrum,obest, [litet|NästaOrd],NästaOrd).
```

```
% Parsningspredikat
parsa(Sats) :-
  s(Analys,Genus,Species,Sats,[]),
  write(Analys), nl,
  write(Genus), nl,
  write(Species).
```

Som var fallet tidigare med genus så lägger vi species på samma plats – som tredje argument – i reglerna och i lexikonet.

Vi prövar nu att parsa:

```
?- parsa([en,flicka]).
s(np(det(en),n(flicka)))
utrum
obest
  yes

?- parsa([det,huset]).
s(np(det(det),n(huset)))
neutrum
best
  yes

?- parsa([en,flickan]).
Satsen är inte grammatisk!
  yes

?- parsa([den,huset]).
Satsen är inte grammatisk!
  yes
```

Som framgår måste kraven på överensstämmelse gälla både genus och species för att satsen skall kunna parsas. I satsen *en flickan* överensstämmer de bägge orden vad gäller genus, men inte vad gäller species, och i satsen *den huset* så har orden samma species men inte genus. Eftersom unifiering inte kan ske ”över hela linjen”, så att säga, så misslyckas predikatet.

Ett annat kännetecken för svenska nominalfraser är att de har *numerus*. Vi lägger nu till detta i såväl regler som lexikon.

```
%Satsregel
s(s(NP),Genus,Species,Numerus,FörstaOrd,NästaOrd) :-
  np(NP,Genus,Species,Numerus,FörstaOrd,NästaOrd).

%NP-regler
np(np(N),Genus,Species,Numerus,FörstaOrd,NästaOrd) :-
  n(N,Genus,Species,Numerus,FörstaOrd,NästaOrd).

np(np(Det,N),Genus,Species,Numerus,FörstaOrd,NästaOrd2) :-
  det(Det,Genus,Species,Numerus,FörstaOrd,NästaOrd1),
  n(N,Genus,Species,Numerus,NästaOrd1,NästaOrd2).

np(np(Det,Adj,N),Genus,Species,Numerus,FörstaOrd,NästaOrd3):-
  det(Det,Genus,Species,Numerus,FörstaOrd,NästaOrd1),
  adj(Adj,Genus,Species,Numerus,NästaOrd1,NästaOrd2),
  n(N,Genus,Species,Numerus,NästaOrd2,NästaOrd3).
```

```
% Lexikon
n(n(pojke),utrum,obest,sing,[pojke|Rest],Rest).
n(n(pojken),utrum,best,sing,[pojke|Rest],Rest).
n(n(pojkar),utrum,obest,plur,[pojkar|Rest],Rest).
n(n(pojkarna),utrum,best,plur,[pojkarna|Rest],Rest).

n(n(flicka),utrum,obest,sing,[flicka|Rest],Rest).
n(n(flickan),utrum,best,sing,[flicka|Rest],Rest).
n(n(flickor),utrum,obest,plur,[flickor|Rest],Rest).
n(n(flickan),utrum,best,plur,[flickorna|Rest],Rest).

n(n(hus),neutrum,obest,_,[hus|Rest],Rest).
n(n(huset),neutrum,best,sing[huset|Rest],Rest).
n(n(husen),neutrum,obest,plur,[husen|Rest],Rest).

det(det(en),utrum,obest,sing,[en|Rest],Rest).
det(det(den),utrum,best,sing,[den|Rest],Rest).
det(det(ett),neutrum,obest,sing,[ett|Rest],Rest).
det(det(det),neutrum,best,sing,[det|Rest],Rest).
det(det(de),_,best,plur,plur,[de|Rest],Rest).

adj(adj(liten),utrum,obest,sing,[liten|NästaOrd],NästaOrd).
adj(adj(litet),neutrum,obest,sing,[litet|NästaOrd],NästaOrd).
adj(adj(lilla),_,obest,sing,[liten|NästaOrd],NästaOrd).

% Parsningspredikat
parsa(Sats) :-
   s(Analys,Genus,Species,Numerus,Sats,[]),
   write(Analys), nl,
   write(Genus), nl,
   write(Species), nl,
   write(Numerus).
```

Innan vi provar ett anrop skall vi titta närmare på ett par ”udda” detaljer i lexikonet.

Vi ser något skumt vid artikeln *de*. Ordet är `best` och `plur`, men på genuspositionen finner vi enbart en understrykningslinje! Orsaken därtill är helt enkelt att den bestämda artikeln i plural är gemensam för båda genusformerna, och därför inte behöver specificeras. Understrykningslinjen är helt enkelt en variabel som fylls av den information som resten av meningen tillhandahåller. Vi hade således lika gärna kunnat skriva `SpelarIngenRoll` i stället för understrykningslinjen! Detta – att skriva en understrykningslinje – är en generell metod i kodskrivning i prolog att ange att en viss parameter inte är viktig eller inte behöver specificeras i en given regel. En variabel som inte specificeras benämnes den *anonyma variabeln.* Givetvis hade vi kunnat skriva två regler, en för utrum och en för neutrum, sålunda

```
det(det(de),utrum,best,plur,[de|Rest],Rest).
det(det(de),neutrum,best,plur,[de|Rest],Rest).
```

...vilket i praktiken hade varit ekvivalent betydelsemässigt, men man bör hålla regelantalet nere där detta är möjligt.

Således skönjer vi en skillnad mellan hur *flicka* och *hus* är specificerade:

```
n(n(flicka),utrum,obest,sing,[flicka|Rest],Rest).
n(n(flickor),utrum,obest,plur,[flickor|Rest],Rest).

n(n(hus),neutrum,obest,_,[hus|Rest],Rest).
```

Medan *flicka* har en egen obestämd pluralform, saknas en sådan för *hus* (ett av femte deklinationens karaktärsdrag), och man kan därför låta en regel täcka bägge fallen genom

att helt enkelt inte specificera numerus. Detsamma gäller adjektiven, där *lilla* är gemensam bestämd form för både utrum och neutrum.

Vi provar nu ett anrop:

```
?- parsa([den,flickan]).
s(np(det(den),n(flickan)))
utrum
best
sing
  yes

?- parsa([de,husen])
s(np(det(de),n(husen)))
neutrum
best
plur
  yes

?- parsa([de,flickor])
Satsen är inte grammatisk!
  yes
```

Precis som ovan nämnts så måste alla specifikationerna överensstämma för att ett anrop skall lyckas. Vi har nu i grammatiken lagt in dels regler för hur NP:n får se ut, dels ord i lexikon. Dessutom har vi specificerat de *särdrag* som präglar svenska nominalfraser: genus, species och numerus. Vi har lagt in dessa i såväl regler som lexikon i vad som benämnes en *särdragsmatris*.

Svenska nominalfraser kan även innehålla *kvantifierare* (ord som *några*, *någon*), så vi lägger till dessa. Vi passar samtidigt på att "dryga ut" kommentarerna lite, samt lägga till lite fler ord i lexikon.

```
% REGEL FÖR ENKLA NOMINALFRASER
s(s(NP),Gen,Spec,Num,FörstaOrd,NästaOrd) :-
  np(NP,Gen,Spec,Num,FörstaOrd,NästaOrd).

% FRASREGLER: NOMINALFRASER
np(np(N),Gen,Spec,Num,FörstaOrd,NästaOrd) :-
  n(N,Gen,Spec,Num,FörstaOrd,NästaOrd).

np(np(Det,N),Gen,Spec,Num,FörstaOrd,NästaOrd2) :-
  det(Det,Gen,Spec,Num,FörstaOrd,NästaOrd1),
  n(N,Gen,Spec,Num,NästaOrd1,NästaOrd2).

np(np(Det,Adj,N),Gen,Spec,Num,FörstaOrd,NästaOrd3) :-
  det(Det,Gen,Spec,Num,FörstaOrd,NästaOrd1),
  adj(Adj,Gen,Spec,Num,NästaOrd1,NästaOrd2),
  n(N,Gen,Spec,Num,NästaOrd2,NästaOrd3).

np(np(Kvant,N),Gen,Spec,Num,FörstaOrd,NästaOrd2) :-
  kvant(Kvant,Gen,Spec,Num,FörstaOrd,NästaOrd1),
  n(N,Gen,Spec,Num,NästaOrd1,NästaOrd2).

np(np(Kvant,Adj,N),Gen,Spec,Num,FörstaOrd,NästaOrd3) :-
  kvant(Kvant,Gen,Spec,Num,FörstaOrd,NästaOrd1),
  adj(Adj,Gen,Spec,Num,NästaOrd1,NästaOrd2),
  n(N,Gen,Spec,Num,NästaOrd2,NästaOrd3).


% DETERMINERARE
det(det(en),utrum,obest,sing,[en|NästaOrd],NästaOrd).
```

```
det(det(ett),neutrum,obest,sing,[ett|NästaOrd],NästaOrd).
det(det(ena),_,obest,plur,[ena|NästaOrd],NästaOrd).
det(det(den),utrum,best,sing,[den|NästaOrd],NästaOrd).
det(det(det),neutrum,best,sing,[det|NästaOrd],NästaOrd).
det(det(de),_,best,plur,[de|NästaOrd],NästaOrd).

% ADJEKTIV
adj(adj(liten),utrum,obest,sing,[liten|NästaOrd],NästaOrd).
adj(adj(litet),neutrum,obest,sing,[litet|NästaOrd],NästaOrd).
adj(adj(lilla),_,best,sing,[lilla|NästaOrd],NästaOrd).
adj(adj(små),_,_,plur,[små|NästaOrd],NästaOrd).

% KVANTIFIERARE
kvant(kvant(någon),utrum,obest,sing,[någon|NästaOrd],NästaOrd).
kvant(kvant(något),neutrum,obest,sing,[något|NästaOrd],NästaOrd).
kvant(kvant(några),_,obest,plur,[några|NästaOrd],NästaOrd).

% NOMEN
n(n(pojke),utrum,obest,sing,[pojke|NästaOrd],NästaOrd).
n(n(pojken),utrum,best,sing,[pojken|NästaOrd],NästaOrd).
n(n(pojkar),utrum,obest,plur,[pojkar|NästaOrd],NästaOrd).
n(n(pojkarna),utrum,best,plur,[pojkarna|NästaOrd],NästaOrd).

n(n(hus),neutrum,obest,_,[hus|NästaOrd],NästaOrd).
n(n(huset),neutrum,best,sing,[huset|NästaOrd],NästaOrd).
n(n(husen),neutrum,best,plur,[husen|NästaOrd],NästaOrd).

% PARSNINGSPREDIKAT. Om satsen kan ges en analys enligt grammatiken.
parsa(Sats) :-
  s(Analys,Gen,Spec,Num,Mening,[]),
  write(Analys),
  nl, tab(5),
  write(Gen),
  nl, tab(5),
  write(Spec),
  nl, tab(5),
  write(Num),
  nl, nl.

% PARSNINGSPREDIKAT. Om satsen inte kan ges en analys..
parsa(Sats) :-
  write('Satsen är inte grammatisk!'), nl.
```

Den grammatik vi nu skrivit in motsvarar i omskrivningsregler följande:

S	-->	*NP*		
NP	-->	*N*	t ex	*stuga, hus*
NP	-->	*Det NP*	t ex	*en stuga, ett hus, det huset, de stugorna*
NP	-->	*Det Adj NP*	t ex	*en grön stuga, ett grönt hus, de gröna husen*
NP	-->	*Kvant NP*	t ex	*några hus, någon stuga*
NP	-->	*Kvant Adj NP*	t ex	*några gröna hus, någon grön stuga*

Det vi nu skrivit är en så kallad *top-down*-parser (eller på svenska: *hypotesstyrd* parser). Detta innebär att den börjar "uppifrån" och letar sig nedåt bland reglerna i sökandet efter en lösning. Det motsatta förhållandet benämnes *bottom-up* (*datastyrt*) och innebär att man börjar med läsning av orden och arbetar sig uppåt i strukturen. Detta senare går vi emellertid inte in på i detta kompendium, även om det har många fördelar gentemot top-down-filosofin, såväl lingvistiska som datalogiska.

Nämnas bör att det finns ytterligare parsningsmetoder vilka inte tas upp här, eftersom detta kapitel är ämnat som en introduktion till parsning i prolog.

Nya termer och begrepp

parsa
grammatik
länkad kedja
differenslistor
nominalfras
omskrivningsregel
lexikon
grammatisk analys
parsningspredikat
genus
species
numerus
anonym variabel
deklination
särdrag
särdragsmatris
kvantifierare
top-down
hypotesstyrd
bottom-up
datastyrd

Övningar:

- Skriv en grammatik (parser) som klarar av även intransitiva VP:n, och således klarar av satser som t ex *pojken sover, flickan springer, hunden ligger* osv.

- Dryga ut grammatiken med transitiva VP:n. Den skall således klara av satser som *pojken ser flickan, hunden jagar katten* osv.

- Dryga ut grammatiken med rekursiva NP:n. Den skall således klara av NP:n som:

  Jag, han, pojken och flickan

- Gör detsamma med rekursiva VP:n:

  springer, sover, leker och slåss och stojar

  ERGO:

  Satser som:

  Jag, pojken, han, hon och flickan och pojken slåss och leker, stojar och bråkar

  ... skall klaras av.

Vilka ord lexikonet skall innehålla avgör ni själva.

15 DEFINITE CLAUSE GRAMMAR (DCG)

I förra kapitlet såg vi hur man lätt kan skriva en parser i prolog. Prolog tillhandahåller även ett speciellt verktyg för att skriva grammatiska regler enligt ovan. Detta benämnes DCG för *Definite Clause Grammar*, och har följande utseende.

I en DCG skriver man reglerna på ett sätt som ännu mer påminner om vanliga omskrivningsregler i en generativ grammatik.[1] En liten minigrammatik skulle kunna se ut på följande sätt, t ex:

```
% En minigrammatik i DCG-notation.

s       --> n(N),
            v(V).

n(N)    --> [N],  {n(N)}.
v(V)    --> [V],  {v(V)}.

n(flickan).
v(ler).
```

Regler som dessa skrivs av prolog om till regler av den typ vi såg i kapitel 13 vid körning. Sålunda så görs pilen **-->** av prolog automatiskt om till :- . Detta gör att man får en "renare" kod, där de grammatiska reglerna syns tydligare. Om man vill lägga till vanliga prologpredikat i koden så görs detta genom att lägga dem inom "måsvingeparenteser". Som synes i koden ovan ligger lexikonet specificerat som vanliga prologpredikat. Således betyder

```
n(N)  -->  [N],  {n(N)}.
```

... att ett nomen **n** ska unifieras med ett **n/1** i lexikon, specificerat som ett vanligt prologpredikat.

Om ens prolog inte skulle tillhandahålla en DCG-tolkare, men man ändå skulle vilja använda en pil som ovan, kan man lätt specificera en ny operator genom det predefinierade predikatet **op/3**. Sålunda skulle vi lätt kunna tillverka en pil genom att skriva på följande sätt:

```
:- op(1200, xfx, -->).
```

Det första argumentet, **1200**, är "styrkan" på operatorn. Alla operatorer som används i prolog har en viss styrka, som inte är densamma - vissa operatorer är starkare än andra. (För jämförelse kan manualen konsulteras!) Det andra argumentet anger operatorns *associativitet*. Sålunda står **xfx** för en *symmetrisk* associativitet, där **f** står för funktorn, och de båda **x**:en visar att argumenten står på varsin sida om funktorn, samt att de är av samma typ. Det sista argumentet är den symbol man vill definiera.

Att köra en DCG är som att köra vilket program som helst. Dock kräver vissa prologer (som AAIS) att man läser in en "tolkningsfil" for DCG-notation innan interpretering kan ske. I de flesta prologer behöver man inte göra detta, emellertid.

Ett anrop med denna lilla minuskula grammatik skulle kunna se ut så här:

---

[1]Vilket inte är en slump – DCG utformades just med detta syfte!

```
?- s(S,[]).
 S = [flickan,ler] ;
 no
```

Om vi vill kan vi skriva ett parsningspredikat i vanlig prolog:

```
% Parsningspredikat för DCG:n ovan. Om anropet lyckas.
p(Sats) :-
  s(Sats,[]),
  write('Satsen är grammatisk!'), nl.

% Parsningspredikat för DCG:n ovan. Om anropet misslyckas.
p(Sats) :-
  write('Satsen är inte grammatisk!'), nl.
```

Mycket enklare DCG än ovan kan inte skrivas, och som synes belyser den inte särskilt mycket i det svenska språket. Vi försöker därför skriva en DCG som tar hänsyn till företeelser som genus, species och dylikt. Denna får en lite annan grundstruktur. I stället för att bara beskriva NP:n, så skriver vi den för fullständiga satser direkt.

Vi börjar med reglerna:

```
/* EN DCG FÖR SVENSKA*/

/* FRASREGLER */

s(s(NP,VP))         --> np(NP),  vp(VP).

np(np(Det,N))       --> det(det(Det),Genus,Numerus,Species),
                        n(n(N),Genus,Numerus,Species).

vp(vp(v(V)))        --> v(v(V),intrans).

vp(vp(v(V),NP))     --> v(v(V),trans),  np(NP).
```

Som synes finns det både likheter och skillnader jämfört med den ”vanliga” prolog-parsern i kapitel 10. Likheter är själva regelstrukturen (vilket säger sig självt!), samt särdragsmatriserna för substantiven och nomena. En skillnad är att vi inte behöver specificera något som motsvarar `FörstaOrd` i parsern i kapitel 10, eller dylikt.

Vi specificerar nu vårt lexikon:

```
/* LEXIKON */

/* Determinerare */
det(det(en),utrum,sing,obest)       --> [en].
det(det(ett),neutrum,sing,obest)    --> [ett].
det(det(den),utrum,sing,best)       --> [den].
det(det(det),utrum,sing,best)       --> [det].
det(det(de),utrum,plur,best)        --> [de].

/* Nomen */
n(n(bil),utrum,sing,obest)          --> [bil].
n(n(bilen),utrum,sing,best)         --> [bilen].
n(n(hus),neutrum,sing,obest)        --> [hus].
n(n(huset),neutrum,sing,best)       --> [huset].
n(n(pojke),utrum,sing,obest)        --> [pojke].
n(n(pojken),utrum,sing,best)        --> [pojken].
n(n(pojkar),utrum,plur,obest)       --> [pojkar].
n(n(kakor),utrum,plur,obest)        --> [kakor].
```

```
/* Verb */
v(v(ser),trans)                  --> [ser].
v(v(äter),trans)                 --> [äter].
v(v(sover),intrans)              --> [sover].
```

För att förenkla anropet skriver vi ett anropspredikat:

```
parsa(Sats,Struktur) :-
  s(Struktur,Sats,[]).
```

Lägg märke till att vi genom att lägga "stoppvillkoret" (dvs, "håll på tills den tomma listan dyker upp!") i kroppen, så behöver vi inte skriva in den i anropet. I själva verket är vi för slöa för att skriva in `Struktur` (eller `X`, eller vad man nu skriver i anropet) också, så vi gör ett topp-predikat för topp-predikatet, dvs ett anropspredikat som anropar anropspredikatet, som i sin tur anropar grammatiken. Vi kallar detta `p/1`.

```
/* Om anropet lyckas */
p(Sats) :-
  parsa(Sats,Struktur),
  write(Struktur), nl, nl.

/* Annars */
p(Sats) :-
  write('Satsen är inte grammatisk!'), nl, nl.
```

Lägg märke till att vi *skriver ut* `Struktur`, i stället för att få den *returnerad* av prolog. Vad vi har vunnit är att vi i stället för att behöva skriva ett `X` i anropet varje gång, och få resultatet returnerat som `X =` *något*, behöver vi bara skriva ett `write`-uttryck en enda gång, och sedan alltid få strukturen utskriven.

Vi provar ett par anrop:

```
?- p([en,pojke,ser,ett,hus]).
s(np(det(en),n(pojke)),vp(v(ser),np(det(ett),n(hus))))

  yes

?- p([en,pojke,ser,en,hus]).
Satsen är inte grammatisk!

  yes
```

Som synes kräver denna grammatik också att särdragen överensstämmer och unifieras. Däremot får vi dem inte utskrivna.

Nya termer och begrepp

DCG
Definite Clause Grammar
generativ grammatik
`op/3`
associativitet
returnering

Övningar:

- Dryga ut parsern ovan så att den klarar av även prepositionsfraser (PP:n) i stil med *i Stockholm*, *på bordet* osv.

- Dryga ut parsern med adverb.

16 DIFFERENSLISTOR

Ett användbart verktyg i prolog är *differenslistor* (vi har tidigare stött på dem när vi gjorde parsern). De är helt enkelt listor där man "häktar av" resten (dvs, **Rest**) av listan från en lista, och på så sätt gör den direkt tillgänglig. Ett exempel skulle vara att om vi har listan:

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

... och delar upp den på följande sätt:

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

... så kommer **Rest** att motsvara:

```
Rest = [d,e]
```

Om vi i det här fallet skulle vilja sklija mellan Rest och den första delen av listan kan vi skapa en lista som redan från början markerar denna skillnad:

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

... eller:

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

... eftersom inte alla prologer accepterar syntaxen med ett bindestreck. I bägge fallen arbetar man emellertid med differenslistor, dvs listor där man separerat en del av listan och lagt den i en egen lista.

Här har vi som sagt "lyft ut" **Rest** och gjort den direkt åtkomlig. Detta kan vara synnerligen användbart i vissa fall. Ett klart lingvistiskt fall är ändelser! Om vi beaktar parsern i kapitel 14, kanske vi gör reflektionen att det verkar onödigt att lägga in varje *ordform* i lexikon (dvs, *pojke*, *pojken*, *pojkar*, *pojkarna*) i stället för att bara lägga in den *invarianta stammen* – i detta fall *pojk* – och sedan lägga till ändelser. På så sätt skulle man kunna lägga in bara de invarianta stammarna i lexikon och specificera ändelse-informationen någon annanstans.

Vad som krävs för detta är ett lexikon med annat utseende. Vi måste dels lägga in orden i form av listor, dels ge plats för ändelsen i form av en differenslista, och dessutom specificera vilken *deklination* (substantiv, dvs *böjningsparadigm*) ordet i fråga tillhör.

För att ge ett exempel på hur detta kan användas skriver vi ett program **info/1** som ger information om vilken böjningsform ett substantiv har. Om vi således frågar programmet:

```
?- info(bilarna).
```

... så vill vi få svaret:

```
deklination_2
[utrum, bestämd, plural]
```

Dvs, ordet är av andra deklinationen, och en så kallad *särdragsmatris* ger oss information om vilken form ordet står i. (Vi kan skriva ut medlemmarna i listan med predikatet **skriv_ut_lista/1**, som förekommer i kapitel 4, om vi inte vill ha dem i listform).

Vi skall alltså skriva ett program som fungerar på detta sätt.

Vi börjar med lexikonet, där vi alltså, som sagt, bara vill lägga in ordstammar. Ett ord som *bil* tillhör andra deklinationen i svenska. Vi specificerar alltså ordet som *deklination_2* i lexikon:

```
ord(deklination_2,[b,i,l|Ändelse],Ändelse).
```

Att vi lägger in ordet i formen `[b,i,l]` är för att differenslistor arbetar med listor (vilket ju hörs på namnet!). I fallet med parsern så skrev vi in hela ord som atomer, och det vore ju praktiskt om man kunde göra som i anropet ovan, skriva:

```
?-  info(bilarna).
```

... som en atom, i stället för att behöva skriva:

```
?-  info([b,i,l,a,r,n,a]).
```

Detta är nu inget problem, eftersom det finns ett predefinierat predikat som heter `explode/2`, som tar som första argument en atom, delar upp den i sina beståndsdelar, och lägger in den i en lista (det andra argumentet).[1]

Ett anrop skulle kunna se ut så här:

```
?-  explode(bilarna,X).
X  =  [b,i,l,a,r,n,a]
```

Det gör att när vi skriver predikatet `info/1`, som tar ett ord som argument, så är det första vi gör att använda `explode/2` för att "översätta" ordet till lexikonets format. Vi börjar alltså på detta sätt (observera att det sista tecknet är komma, eftersom vi ämnar fortsätta ett tag!):

```
info(Ord) :-
   explode(Ord,ExploderatOrd),
```

Sedan matchar vi detta "exploderade" ord mot lexikonet:

```
info(Ord) :-
   explode(Ord,ExploderatOrd),
   ord(Ordklass,ExploderatOrd,Ändelse),
```

... där `Ordklass` instantieras med deklinationen som är specificerad som första argument i lexikonet. `ExploderatOrd` instantieras som ordet vi skrivet in, och `Ändelse` som den "avlyfta" ändelsen. Att vi väljer att kalla det `Ordklass` i stället för `Deklination` beror på att deklinationer hör till substantiven och således kan betraktas som en lite "precisare" ordklassbeskrivning än t ex substantiv. Om vi dessutom vill lägga in verb, adjketiv och andra ordklasser i lexikonet så kommer det ju inte stå `Deklination` som första argument i predikatet `ord/3`.

Om vi till exempel anropar `info/1` med ordet *bilarna*, så instantieras `ord/3` som:

```
ord(deklination_2,[b,i,l,a,r,n,a],[a,r,n,a]).
```

OK! Nu har vi hittat ordet *bil* i lexikon, vi har också fått fram vilken deklination ordet har – `deklination_2` – och vi har också lyft av ändelsen, i detta fall `[a,r,n,a]`. Nu gäller det alltså att skriva ett predikat som givet en deklination och en ändelse tillhandahåller en särdragsmatris som säger oss vilken form ord står i. Vi döper detta predikat till `matris`. Vad behöver vi nu ha med oss för argument i detta predikat? Jo,

---

[1]Eftersom ett flertal prologer inte har `explode/2` som predefinerat predikat tillhandahålls koden till `explode/2` i slutet av kapitlet.

dels behöver vi ordklassen, dels behöver vi ändelsen. Själva ordet är inte längre intressant – hela poängen är ju att givet en deklination och en ändelse så vet vi ordets form! Vi behöver dessutom ett argument **Matris**, som skall "fyllas i" med den information vi söker efter. Alltså får predikatet **matris** följande utseende, när vi lägger till det på **info/1**:

```
info(Ord) :-
  explode(Ord,ExploderatOrd),
  ord(Ordklass,ExploderatOrd,Ändelse),
  matris(Ordklass,Ändelse,Matris),
```

Innan vi skriver själva **matris/3** så kan vi avsluta **info/1**, så är det jobbet gjort. Det enda vi behöver göra är att skriva ut **Ordklass** och **Matris**. Vi lägger alltså till ett par **write**-uttryck på följande sätt:

```
/* info/1 tar ett ord, lägger det i en lista, slår upp det i lexikon
och hämtar upp information om ordets deklination och information om
genus, species och numerus samt häktar av ändelsen, skriver ut
ordklassen samt särdragsmatrisen. */

info(Ord) :-
  explode(Ord,ExploderatOrd),
  ord(Ordklass,ExploderatOrd,Ändelse),
  matris(Ordklass,Ändelse,Matris),
  write(Ordklass), nl,
  write(Matris), nl.
```

Detta ger exakt det svar som vi angav i exemplet ovan. Om vi inte vill ha matrisen i form av en lista så kan vi ju använda **skriv_ut_lista/1** i stället! Vi behöver nu bara ett extra predikat som klarar av situationen om det ord man anropar med inte skulle finnas i lexikon:

```
/* Om anropet ovan inte lyckas, dvs om det sökta ordet inte finns
inte belagt i lexikonet. */

info(Ord) :-
  write('Ordet finns inte lexikon!').
```

Det borde vara klart vid det här laget (som jag har tjatat!) att detta predikat måste ligga efter det första predikatet i regelfilen. Om inget ord hittas så går prolog vidare till nästa predikat – det nyss definierade – och skriver ut *Ordet finns inte i lexikon!*. Om det senare predikatet skulle ligga först i regelfilen skulle alla ord få det meddelandet!

Nu återstår alltså bara att skriva matrisprogrammet. Detta skall alltså som första argument ta en deklination, som andra argument en ändelse, och med hjälp av dessa bägge instantera det tredje argumentet, en särdragsmatris. Detta är inte svårare än som följer (vi tar andra deklinationens hela böjningsparadigm!):

```
/* matris/3 anropas med en ordklass och en ändelse, och hämtar upp den
särdragsmatris som matchar ändelsen. */

matris(Ordklass,Ändelse,Matris) :-
  Ordklass = deklination_2,
  Ändelse = [],
  Matris = [utrum,obestämd,singular].

matris(Ordklass,Ändelse,Matris) :-
  Ordklass = deklination_2,
  Ändelse = [e,n],
  Matris = [utrum,bestämd,singular].
```

```
matris(Ordklass,Ändelse,Matris) :-
  Ordklass = deklination_2,
  Ändelse = [a,r],
  Matris = [utrum,obestämd,plural].

matris(Ordklass,Ändelse,Matris) :-
  Ordklass = deklination_2,
  Ändelse = [a,r,n,a],
  Matris = [utrum,bestämd,plural].
```

Hur ändelserna hänger ihop med matriserna torde vara självförklarande. Det är helt enkelt andra deklinationens böjningsparadigm.

Vi provar nu några anrop med programmet, så långt det nu är definierat:

```
?- info(bil).
deklination 2
[utrum,obestämd,singular]
  yes

?- info(bilarna).
deklination 2
[utrum,bestämd,plural]
  yes

?- info(hus).
Ordet finns inte i lexikon!

?- info(bilen).
deklination 2
[utrum,bestämd,singular]
  yes
```

Vi har nu skrivit embroyt till ett program som ger oss information om ordformer, samtidigt som vi drastiskt minskat lexikonets storlek. För att göra detta fick vi visserligen skriva till ett antal predikat **matris/3**, men när de väl finns där så kan vi dryga ut lexikonet med hur många ord vi vill, och behöver bara skriva in de invarianta stammarna.

Som var och en inser kryllar det dock av oregelbundna ord, som ställer till problem för en sådan här "snygg" modell (språk tenderar alltid "förstöra" vackra beskrivningsmodeller!), och dessa måste tas om hand på annat sätt. Dock är mycket vunnet i överskådlighet och snabbhet med en sådan här implementering.

Nya termer och begrepp

ordform
invariant stam
deklination
särdragsmatris
explode/2
böjningsparadigm

Predikatet explode/2

```
% explode/2 tar en atom, delar upp den i sina beståndsdelar samt
% lägger dessa i en lista. Exempel på anrop:
% ?- explode(hej,X).
% X = [h,e,j]

explode(X,X1) :-
  var(X1), !,
  name(X,Ascii),
  name1(Ascii,X1).

explode(X,X1) :-
  var(X), !,
  name1(Ascii,X1),
  name(X,Ascii).

name1([],[]).
name1([X|Xs],[X1|X1s]) :-
  name(X1,[X]),
  name1(Xs,X1s).
```

Övningar:

- Skriv ut **info/1** så att det klarar av substantiv av alla deklinationer. En böjningstabell tillhandahålles nedan.

DEKLINATION	OBEST, SING	BEST, SING	OBEST, PLUR	BEST, PLUR
1	*flicka*	*flickan*	*flickor*	*flickorna*
2	*bil*	*bilen*	*bilar*	*bilarna*
3	*dam*	*damen*	*damer*	*damerna*
4	*hjärta*	*hjärtat*	*hjärtan*	*hjärtana*
5	*hus*	*huset*	*hus*	*husen*

- Utöka **info/1** så att det klarar av verb också. Dessa böjs enligt olika *konjugationer*. Dessa kan lämpligen kallas *k1*, *k2* osv. En tabell med böjningsmönstret tillhandahålles nedan:

TEMPUS	K1	K2	K3	K4
Imperativ	*kalla*	*väg*	*sy*	*spring*
Infinitiv	*kalla*	*väga*	*sy*	*springa*
Presens	*kallar*	*väger*	*syr*	*springer*
Preteritum	*kallade*	*vägde*	*sydde*	*sprang*
Presens particip	*kallande*	*vägande*	*syende*	*springande*
Perfekt particip	*kallad*	*vägd*	*sydd*	*sprungen*
Supinum	*kallat*	*vägt*	*sytt*	*sprungit*

Nota bene! Ta inte denna tabell som någon absolut sanning. Den innehåller – såsom alla kortfattade lingvistiska beskrivningar – grova förenklingar! Som ni kommer att märka kommer era program (i bästa fall!) att fungera för just de ord som ni har i åtanke när ni skriver programmet – och några till. Undantag florerar emellertid!!

- Dryga ut `info/1` så att det även klarar av ord ur andra klasser. Lägg märke till att inte alla av dessa har ändelser, och alltså måste programmet klara av att skilja på ord som har och ord som inte har ändelser. Både anropen och lexikonet måste således ha olika utseenden för de olika fallen. I vilket fall som helst skall dock programmet signalera när det sökta ordet inte finns i lexikonet!

- Tracea `explode/2` och försök förstå hur det fungerar.

17 FILLÄSNING OCH FILSKRIVNING

Vi har i tidigare avsnitt sett hur prolog kan läsa från interpretatorn (dvs Query Mode). Detta är nu inte det enda som prolog kan läsa, utan även *textfiler* kan läsas. För detta ändamål finns de predefinierade predikaten **see/1** och **seen/0**.

När man vill öppna en fil för läsning använder man **see/1**. Vi skapar predikatet **läs/1**, som vi vill ska läsa en fil (för något ändamål):

```
läs(Filnamn) :-
   see(Filnamn),
   gör_något.
```

Filnamn måste anges med rätt *sökväg* i datorn. Ett anrop i Macintosh skulle kunna se ut på följande sätt:

```
?-  läs('HD:Program:Brev').
```

HD är i detta fall namnet på hårddisken, **Program** namnet på en mapp i vilken dokumentet **Brev** ligger.[1]

I en PC skulle motsvarande anrop kunna se ut på följande sätt:

```
?-  läs('C:\PROGRAM\BREV.TXT').
```

I detta fall är **C:** namnet på hårddisken, **PROGRAM** namnet på ett direktorium, och **brev.txt** namn på ett dokument i detta direktorium.[2]

Efter att **gör_något** gjort något med filen, måste vi stänga filen igen (man skall ju inte lämna dörrar öppna!). Detta gör vi med predikatet **seen/0**.

```
läs(Filnamn) :-
   seen.
```

Skall vi då hitta på något att göra med vår öppnade textfil!

En lagom lingvistisk tillämpning vore väl att plocka ut alla ord och meningar från en text. Till skillnad från LISP finns det i prolog inga lämpliga predefinierade predikat som läser t ex ord eller rader från indata (t ex en textfil), utan vi får skapa dessa predikat själva.

Vi skriver om **läs/1** så att det läser in ord och meningar från en textfil. Vi döper därför om **gör_något/1** till **read_file/0**.[3]

```
% Öppna en textfil och gör read_file/0.
läs(Filnamn) :-
   see(Filnamn),
   read_file.

% Stäng textfilen.
läs(Filnamn) :-
   seen.
```

---

[1] I *din* Macintosh har du säkert andra namn på hårddisk, mappar och dokument!

[2] I *din* PC har du säkert andra namn på direktorier och dokument (fast troligen inte på hårddisk).

[3] Att vi här övergår till engelska beror på att varianter av **read_file**, **get_letters**, **get_word** och **get_sentence** finns i de flesta prologböcker, och att bruket av samma namn gör det lättare att jämföra olika implementeringar.

```
/* Plocka ut satser från den öppna filen och skriv ut dem på skärmen. */

read_file :-
  get_sentence(Sentence), nl,
  write(Sentence), nl,
  read_file.

/* get_sentence/1 plockar av ASCII-tecken efter ASCII-tecken från de
lästa tecknen. */
get_sentence(WordList) :-
  get0(Char),
  get_rest(Char,WordList).
```

I **get_sentence/1** så stöter vi på det predefinierade predikatet **get0/1**, som returnerar ASCII-numret för sitt argument. ASCII-nummer är de nummer som tecken motsvaras av i datorn. Således är t ex numret för ”mellanslag” 32 osv.

```
/* get_rest/2 säger oss att vi är klara med en sats när vi finner
följande tecken: */
get_rest(Char,[]) :-
  (Char=46                    % .
  ;
  Char=33                     % !
  ;
  Char=63),                   % ?
  !.                          % CUT!

/* get_rest/2 säger oss att vi är klara med ett ord när vi finner
följande tecken: */
get_rest(Char,WordList) :-
  (Char=13                    % Radmatning
  ;
  Char=32                     % Mellanslag
  ;
  Char=10),                   % Radmatning
  !,                          % CUT!
  get_sentence(WordList).

get_rest(Letter,[Word|WordList]) :-
  get_letters(Letter,Letters,NextChar),
  name(Word,Letters),
  get_rest(NextChar,WordList).

get_letters(Char,[],Char) :-
  (Char=46                    % .
  ;
  Char=13                     % Radmatning
  ;
  Char=33                     % !
  ;
  Char=63                     % ?
  ;
  Char=32),                   % Mellanslag
  !.

get_letters(Let,[Let|Letters],NextChar) :-
  get0(Char),
  get_letters(Char,Letters,NextChar).
```

Dessa program är som de står här snålt kommenterade. Jag föreslår att ni tracar en mening och ser hur de fungerar.

För att pröva det hela skapar vi en liten textfil som vi vill läsa. Vi döper den till **Text** och lägger den i samma mapp (i Macintosh-termer) eller direktorium (i PC-termer) som själva prologen. Textfilen får följande utseende:

Det var en gång en man. Han hette Harald. Det var en gammal person.

Vi anropar nu programmet enligt ovan. (Lägg märke till att anropet är för Macintosh!):

```
?-  läs('HD:Program:AAIS:Text').

[Det,var,en,gång,en,man]

[Han,hette,Harald]

[Det,var,en,gammal,person]
    yes
```

Som argument till **läs/1** anger vi sökvägen i datorns hierarki. **HD** representerar här hårddisken. I mappen **HD** finns mappen **Program**, i vilken mappen **AAIS** ligger. I mappen **AAIS**, slutligen, ligger textfilen **Text**.

Ett anrop i PC skulle kunna se ut på följande sätt:

```
?-  läs('C:\PROGRAM\AAIS\PROV.TXT')
```

... där **C:** är hårddisken, **PROGRAM** och **AAIS** är direktorier, och **PROV.TXT** är textfilen.

På samma sätt som vi i stället för att läsa Query Mode har läst en fil, kan vi, i stället för att få resultatet av en körning returnerat till Query Mode få det skrivet till en fil. Detta sker med hjälp av predikaten **tell/1** och **told/0**.

Om man således i ett predikat använder **tell/1**, så skapas filen och man kan då skriva på den. Strukturen är således **tell(+Filnamn)**.

Nya termer och begrepp

see/1
seen/0
sökväg
get0/1
ASCII-nummer
mapp
direktorium
tell/0
told/0

Övningar:

- Skriv ett program som räknar ord och meningar i en fil.
- Dryga ut programmet så att det dessutom räknar ut medellängden på orden i den text som lästs.
- Skriv ett program **`parsa_text/1`** som läser en textfil och parsar en efter en av textfilens meningar. Vi kan tänka oss en textfil Provtext:

  Snälla pojkar ser små flickor. Flickorna spelar boll.

  Ett anrop skulle kunna se ut på följande sätt:

```
?-  parsa_text('HD:Program:AAIS:Provtext').

Sats: [snälla,pojkar,ser,små,flickor]

Parse:

s(np(adj(snälla)n(pojkar))vp(vtr(ser)np(adj(små)n(flickor)
)))

Sats: [flickorna,spelar,boll]

Parse:

s(np(n(flickorna))vp(vtr(spelar)np(n(boll))))

    yes
```

  ... eller på något liknande sätt.

TERMINOLOGI

Här presenteras ett antal termer som förekommer i samband med prolog. De står i bokstavsordning för att lättare kunna användas som referensdel. Motsvarande ord på engelska ges i parentes efter uppslagsordet.

Vissa termer har inte i detalj gåtts igenom i kompendiet, men kan vara bra att kunna slå upp eftersom man vid vidare läsning kan komma att stöta på dem. I dessa fall har jag försökt förklara dem i lite mer detalj här.

Anonym variabel (Anonymous Variable)

En variabel vars värde är oviktigt i en viss situation. Skrivs oftast med en understrykningslinje. Två anonyma variabler i samma regel betraktas som skilda variabler.

Argument (Argument)

I samklang med predikatlogik tar ett predikat ett antal argument, där strukturen kan beskrivas:

<predikat>(<argument$_1$>,<argument$_2$>, ... ,<argument$_n$>)

... där argumenten kan vara vilket antal som helst, inklusive noll. Argument skiljs åt medelst kommatecken.

Exempel:

```
knattar(knatte,fnatte,tjatte).
```

Om man tycker att det blir lättare att läsa om man har mellanslag mellan argumenten så är detta tillåtet (däremot är det bra om man är konsekvent). Sålunda är detta också ett syntaktiskt uttryck:

```
knattar(knatte,  fnatte,  tjatte).
```

Aritet (Arity)

Antalet argument ett predikat tar. Så tar exempelvis predikatet **`gifta`** två argument. Man säger då att predikatet **`gifta`** har ariteten 2. Detta skrivs som

```
gifta/2
```

Atom (Atom)

Odelbara prologobjekt. I programmet

```
pappa(albert).
```

är t ex **`albert`** en atom.

En atom kan delas upp i sina beståndsdelar med predikatet **`explode/2`**.

Backtracking (Backtracking)

Prologs inbyggda sätt att alltid leta sig igenom hela sökträden (dvs, söka alla möjliga lösningar) innan den ger upp benämnes backtracking. På grund därav spelar regelordningen ibland roll för vissa programs effektivitet. Kallas även *backning*.

Cuts (Cuts)

Ibland vill man undvika prologs inbyggda egenskap att alltid söka alla lösningar i ett program. Orsaken kan vara att man vill ha ett effektivare program, men även att om man inte "stoppar" prolog i vissa fall så beter sig programmet på ett felaktigt sätt. Cuts skrivs med utropstecken (**`!`**) och förhindrar backtracking. Kallas även *snitt*.

Det finns två slags *cuts* i prolog: *Green Cuts* och *Red Cuts*. Medan *Green Cuts* gör program deterministiska så ändrar *Red Cuts* programs deklarativa innebörd.

Ett exempel på ett grönt cut ges i följande exempel. Vi definierar ett program **min**, som kollar om en siffra är mindre än en annan siffra:

```
min(X,Y,X) :-
   X =< Y.

min(X,Y,Y) :-
   X > Y.
```

Den första regeln säger att en siffra **X** är mindre än en siffra **Y** om **X** är mindre än eller lika med (för att klara av fallet där två siffror är lika stora) siffran **Y**. Den andra regeln säger att en siffra **X** är större än en siffra **Y** om siffra **X** är högre än siffran **Y** (detta kan låta som ett tautologiskt resonemang, men det är så programmen testar en relation).

Eftersom vi vet att båda fallen inte kan vara sanna samtidigt kan vi lägga till ett cut för att undvika att prolog prövar bägge fallen. Om den ena regeln har lyckats så vet vi att den andra regeln inte kan lyckas, och således inte behöver kollas.

```
min(X,Y,X) :-
   X =< Y, !.

min(X,Y,Y) :-
   X > Y, !.
```

Röda cuts ändrar programs deklarativa innebörd.

Databas (Database)

En samling regler och fakta som beskriver relationer mellan objekt.

Dubbel rekursion (Double Recursion)

När ett predikat anropar sig självt två gånger i programkroppen.

Fakta (Facts)

Generellt har fakta strukturen:

<predikat>(<argument$_1$>,<argument$_2$>, ... ,<argument$_n$>)

... och kan ha t ex följande utseende:

```
man(albert).
kvinna(hanna).
gifta(albert,hanna).
knattar(knatte,fnatte,tjatte).
```

Ett faktum är i själva verket (i princip) en förkortad klausul, där kroppen är **true**.

Frågor (Queries)

Den struktur med vilken man befrågar sitt prologprogram. Exempel (se databasen under *Fakta*).

`?- man(X).`	*Finns det någon man i databasen?*
`X = albert`	*Ja, Albert!*
`?- gifta(X,Y).`	*Finns det några gifta i databasen?*
`X = albert`	
`Y = hanna`	*Ja, Albert och Hanna är gifta!*

```
?- gifta(albert,X).          Vem är Albert gift med?
X = Hanna                    Albert är gift med Hanna
```

Funktor (Functor)

Predikatnamnet benämnes funktor. Generellt har prologtermer strukturen

<funktor>/<argument>

Grund(ad)fråga (Ground Query)

Fråga utan variabel. Exempel:

```
?- gifta(albert,hanna).
  yes
```

Gröna cuts (Green Cuts)

Se *Cuts*.

Hals (Neck)

Implikationspilen i regler, dvs :-

Huvud (Head)

Delen till vänster om **:-** i en regel. I följande regel är **son(X,Y)** huvudet:

```
son(X,Y) :-
  förälder(Y,X),
  man(X).
```

Icke-grund(ad)fråga (Non-Ground Query)

En fråga med en eller flera variabler. Exempel:

```
?- gifta(X,Y).
X = albert
Y = hanna
  yes

?- man(X).
X = albert
  yes
```

Instantiering (Instantiation)

När en variabel ges ett värde så instantieras den. Om man t ex ställer en fråga

```
?- man(X)
```

... letas databasen igenom tills prolog finner ett faktum **man**, där **X** är **albert**. **X** byts då ut mot **albert**, dvs instantieras som **albert**.

Interpretator

Den del av prolog som tolkar (läser in) de regler man skrivit. När man skriver regler i en editor har de samma värde som vilken text som helst i en textfil. En interpretator tolkar om reglerna enligt prologs syntax så att man kan köra sitt program.

Iteration (Iteration)

Är en benämning på att alla objekt i en struktur underkastas samma behandling i tur och ordning.

Klausul (Clause)

En klausul består av huvud (*Head*) och kropp (*Body*) skilda åt av en hals (*Neck*). Följande program består av *ett* predikat med *tre* klausuler:

```
byggnad(Svenska).

byggnad(Svenska,Engelska).

byggnad(Svenska,Engelska,Franska).
```

Kompilator

Skriver om regler till maskinkod (datorns eget interna språk). Kompilerade regler går inte att läsa. Kompilerad kod går mycket snabbare att köra än interpreterad kod eftersom den arbetar "djupare" i datorn.

Konstant (Constant)

Konstanter markerar individer. Det finns tre slags konstanter:

strängar - $Robert Eklund$

atomer - lärare

siffror - 3

Kropp (Body)

Delen till höger om :- i en regel. I följande program

```
son(X,Y) :-
  förälder(Y,X),
  man(X).
```

... utgörs kroppen av

```
förälder(Y,X),
man(X).
```

Lista (List)

Är en av prologs grundläggande strukturer. Listor i prolog skrivs med hakparenteser. Lodlinjer skiljer resten av argumenten i en lista (oavsett dessas antal) från ett eller flera argument i början av samma lista.

Exempel:

```
knattar([ knatte, tjatte, fnatte ]).
knattar([ knatte | FleraKnattar ]).
```

Logikprogram (Logic Program)

En uppsättning axiom/regler som definierar relationer mellan objekt. En körning av ett sådant program är en deduktion av dess konsekvenser.

Mål (Goal)

Det man vill uppnå med sina frågor. En fråga som

```
pappa(X,ann), pappa(X,sofia).
```

... har som mål att tillfredsställa konjunktionen av de båda frågorna.

Operator (Operator)

Funktorer i prolog. Exempel på predefinierade operatorer är t ex:

```
+ - , ; :-
```

I prolog finns möjligheten att specificera nya operatorer med predikatet `op/3`.

Predikat (Predicate)

I samklang med predikatlogik består ett predikat av predikatsnamn och noll eller flera argument, där enligt strukturen:

<predikatsnamn>(<argument>)

... där argumenten kan vara vilket antal som helst, inklusive noll.

Program (Program)

En samling regler och fakta som beskriver relationer mellan objekt.

Regler (Rules)

Definierar relationer mellan olika fakta. Om vi t ex har en struktur

```
far(X,Y).
```

... så kan vi definiera regeln

```
farfar(X,Z) :-
  far(X,Y),
  far(Y,Z).
```

Rekursion (Recursion)

Kan beskrivas som en regel som kan expanderas som sig själv. Ett av de mest karakteristiska dragen i naturliga språk är att de är rekursiva. En minigrammatik som

S -> NP VP
NP -> N
VP -> V
N -> *pojken*
N -> *flickan*
V -> *sover*
V -> *läser*

... är inte rekursiv, dvs vi kan lätt räkna antalet möjliga meningar i språket. Om vi däremot lägger till en regel som

S -> S KONJ S
KONJ -> och

... så erhåller vi en rekursiv grammatik, som kan generera att oändligt antal meningar, t ex:

Pojken läser och flickan läser
Pojken läser och pojken sover och flickan läser
Flickan läser och pojken sover och flickan läser och pojken sover

Röda cuts (Red Cuts)

Se *Cuts*.

Sammansatta frågor (Compound Queries)

Sammansatta frågor uppträder när man ställer flera frågor på en gång. Exempel är (se databasen under *Fakta*):

```
?- man(X), gifta(X,Hanna)
X = albert
```

... vilket kan läsas ut som frågan *Finns det en man sådan att denna man är gift med Hanna?*

Man ställer inte sammansatta frågor i Query-fönstret särskilt ofta, men däremot ställs ofta sammansatta frågor (dvs, villkor) i reglers funktionskroppar.

Sammansatt Term (Compound Term)

En struktur med en funktor och ett eller flera argument vilka är termer. Ett exempel är:

```
son(X,Y).
```

Se även *Term*.

Svansrekursion (Tail Recursion)

När det rekursiva anropet i ett program ligger sist i programkroppen. Ett exempel är programmet **skriv_ut_lista** i kapitlet om rekursion. Svansrekursion benämnes även *iteration*.

Term (Term)

Prologs enda datastruktur. Prologs grundläggande termer är *siffror*, *atomer*, *variabler* och *strukturer*.

Unifiering (Unification)

Unifiering kan beskrivas som "överlappning" av mönster. Det lyckas om ett helt mönster "matchar" en fråga. Vi har ett faktum:

```
gifta(albert,hanna).
```

... i databasen. Om vi då frågar:

```
?- gifta(albert,hanna).
```

... så kan detta unifieras med faktum i databasen. Om vi i stället skriver:

```
?- gifta(albert,josefina).
```

... så kan detta inte unifieras. Visserligen kan **albert** instantieras, men inte med **josefina** som andra argument. Unifering är dessutom huvudkonstruktionen för datatransport i prolog.

Variabel (Variable)

Skrivs med inledande versal eller understrykningslinje. En variabel kan antingen vara instantierad eller oinstantierad.

Vänsterrekursion (Left Recursion)

Inträffar när den rekursiva delen av ett program står först i kroppen:

```
funktion(Argument) :-
   funktion(Argument),
   något_annat(Argument).
```

Detta leder i bästa fall till ineffektivitet och i värsta fall till en *loop*, dvs ett program som anropar sig självt oändligt, och således inte avbryter. Motsatsen är *svansrekursion*.

LITTERATUR

Det är inte särskilt svårt att hitta litteratur som introducerar prolog. Nedan presenteras några böcker med en (subjektiv!) beskrivning av deras respektive innehåll:

Bratko, Ivan: *PROLOG, Programming For Artificial Intelligence*, Addison-Wesley Publishing Company 1990 (second edition).

Denna bok har jag själv (och vänner) funnit vara den bästa introduktionen. Den är omfattande (596 sidor), men exemplen är mycket överskådliga och pedagogiskt upplagda. Den innehåller en hel del om AI, men man behöver ju inte läsa allt.

Clocksin, W F och Mellish, C S: *Programming in Prolog*, Springer Verlag, 1981.

En av de första prologböckerna. Otillräcklig idag.

Coelho, Helder och Cotta, José: *Prolog by example*, Springer Verlag, 1988.

En bok som vänder sig till prologlärare. Beskriver hur man kan använda prolog till att lösa olika problem.

Filipic, Bogdan: *Prolog User's Handbook*, Ellis Horwood Limited, 1988.

Denna bok innehåller en hel mängd användbara predikat som inte finns predefinierade i de flesta prologer. Inkluderar användbara predikat för list-processning, aritmetik, räknare, in- och utdata-hantering, skärmhantering m m.

Gal, Annie; Lapalme, Guy; Saint-Dizier, Patrick och Somers, Harold: *Prolog for Natural Language Processing*, John Wiley & Sons, Chicester 1991.

En intruduktion till prolog med stor tonvikt på lingvistiska tillämpningar. Går bland annat igenom logikgrammatik, semantiska representationer, automatisk textgenerering, svarssystem med mera. Innehåller dessutom en stor mängd användbar programkod. Inte så användbar om man inte är intresserad av just de lingvistiska specialområden som boken behandlar.

Gazdar, Gerald och Mellish, Chris: *Natural Language Processing in PROLOG – An Introduction to Computational Linguistics*, Addison-Wesley Publishing Company 1989.

Denna bok är ämnad för dataloger och lingvister med en mer allvarligt menad vilja att lära sig att använda prolog inom lingvistik. Går igenom grammatiker, parsrar, *ATN*:s, finita automater, semantikrepresentation med mera. Graderade övningar och svar till vissa av övningarna.

Kluzniak, Feliks och Szpakowitz, Stanislaw: *Prolog for programmers*, Academic Press, 1985.

Skriven för folk som redan är väl förtrogna med andra programmeringsspåk, och nu vill lära sig prolog. Kräver omfattande kunskaper om datavetenskap i allmänhet.

Merrit, Dennis: *Adventure in Prolog*, Springer Verlag, New York 1990.

En utsökt introduktion. I 15 kapitel gås olika områden igenom grundläggande. Boken djupdyker inte i något område, men tjänar som sagt som en introduktion till prologs olika användningsområden. Övningar och spel (sic!).

Mueller, Robert A. och Page, Rex L.: *Symbolic Computing with LISP and PROLOG*, John Wiley & Sons, 1988.

En grundläggande introduktion i både LISP och prolog. Innehåller många exempelprogram, snarare än att diskutera teoretiska strukturer. Alla nya termer och predikat sammanfattas i rutor i slutet av varje kapitel. Övningar med svar.

Nilsson, Ulf och Maluszynski, Jan: Logic, Programming in Prolog, John Wiley & Sons, 1990.

En välskriven bok för de som vill förstå hur prolog förhåller sig till logik i närmare detalj. Inte lämpad som bok om man vill lära sig praktiskt prologprogrammering, emellertid.

O'Keefe, Richard: *The Craft of Prolog*, MIT Press, 1990.

O'Keefe är en riktig prolog-guru, och detta är en bok för de som handskas med prolog på högsta nivå. I prolog - i högre grad än andra språk - kan små förändringar i koden leda till dramatiska förändringar i effektivitet. Denna bok handlar om hur man skriver effektiva program. En lämplig tredje bok i prolog för riktiga hackers!

Pereira, Fernando och Shieber, Sterling: *Prolog and Natural-Language Analysis*, Center for the Study of Language and Information 1987.

Denna bok börjar också med en "från-scratch"-presentation av PROLOG, varefter den mest är inriktad på lingvistiska användningar av prolog, såsom olika parsningsmetoder, DCG och dylikt. Presenterar även CFG:s, *Transition Networks*, *Recognizers* och liknande formalismer. Inga övningar (och inga svar, heller!).

Spencer-Smith, Richard: *Logic and Prolog*, Harvester Wheatsheaf, 1991.

En bok som behandlar lika mycket logik som prolog, och hur de två relaterar till varandra. Inte en bok för de som främst vill lära sig programmera i prolog.

Sterling, Leon och Shapiro, Ehud: *The Art of Prolog, Advanced Programming Techniques*, The MIT Press, Cambridge, Massachusetts 1986.

Denna bok anser jag vara en lämplig andra bok i prolog, eftersom dess formaliseringar ofta förutsätter att man redan vet vad det hela går ut på. Exemplen som ges är ofta inte i form av ren programkod, utan är formaliserade versioner av sådan, vilket kan vara frustrerande om man inte redan är säker på hur man skriver program.

INDEX

Jag skäms! Inget här. Men du kan ju själv fylla i saker du slår upp, så hittar du dem lättare nästa gång! (Väl?)

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

Z

Å

Ä

Ö