Prolog Laborationer

Programmet som används är SWI-Prolog. Några grundläggande tips för att lära sig grunderna i SWI-Prolog:

Programmet startas genom att först logga in i **Linux**-miljö.

Skriv sedan vid commando-prompten: >**pl**

Då är Du i frågeläge och promten är: **?-**

?- consult('filnamn'). % Laddar in programmet som är sparat i **filnamn**, t.ex. lab2, (Obs! reconsult/1 odefinierad i SWI-Prolog.)
? - consult(user). % Man kan även mata in fakta och regler direkt från tangentbordet.
|: …
|: …
|: **Ctrl-z** % Avslutar inmatningen från tangentbordet.
?- % Åter i frågeläge.

?- listing. % Visar alla predikat som är laddade.
?- listing(P). % Visar bara fakta och regler med namnet **P**, t.ex. free_in.

?- trace. % Sätter på tracing-mekanism för att, t.ex. se hur Prolog exekverar ett program.

?- spy(P). % Lägger in "spy-points" på alla **P**.
?- nospy(P). % Tar bort alla spy-points på **P**.

?- debugging. % Visar debug-status och spy points.
?- nodebug. % Tar bort debug-mode.

?- help(listing). % On-line hjälp om olika "topics" i SWI-Prolog, t.ex. **listing**.

Under trace-mode kan man välja olika "options" för varje prompt. Genom att välja en lämplig option vid varje steg, kan debuggingsarbete bli mer effektivt.
c(reep = godtyckligt tangentslag): visar fullständig tracing för aktuell fråga;
s(kip): hoppar över tracing för aktuell fråga och stannar vid nästa fråga;
l(eap): hoppar över tracing för aktuell fråga och stannar vid nästa spy-point;
r(etry): satisfierar aktuell fråga en gång till på exakt samma sätt som första gången (för att visa hur den satisfierades);
f(ail);
a(bort): avbryter exekveringen;
h(alt): lämnar Prolog.

Det är viktigt att inte glömma punkten '**.**' efter ett kommando, t.ex. punkten i **consult('filnamn').** måste vara med.

Enklaste sättet att utveckla program är att växla mellan en texteditor och SWI-prolog, dvs editera och spara sitt program i editorn och läsa in filen i SWI-Prolog med consult.

SWI-Prolog för Windows (plus manualen) kan hämtas som freeware från t.ex.

http://www.swi.psy.uva.nl/projects/SWI-Prolog/

Lab1 och Lab2 är **obligatoriska**. Lab3 är inte obligatorisk, men en godkänd lösning ger **3 bonus** poäng på tentamen (max 60 poäng). Lösningarna utförs enskilt eller i grupper om två personer.
Krav:

1. Enkel dokumentation (omslag, programutskrift, kortfattad beskrivning).
2. Vid inlämning av en upprättad version efter retur skall den returnerade versionen med handledarens anmärkningar bifogas.
3. Laborationerna skall provköras (helst under schemalagd tid) av handledaren.

Deadline för Lab1 är 070314, Lab2 070326 och Lab3 070330. Vid ev. retur skall labbarna åter-inlämnas inom en vecka.

Lab1.

1. Exercise 1.3 och 1.4 i läroboken *Programming in Prolog*, Clocksin & Mellish, sid. 20. Testa gärna med en liten ”databas”.

2. Enkla aritmetiska program. Skriv prologprogram

sum(+Number1, +Number2, -Summa)

som uttrycker att *Summa* är summan av två talen *Number1* och *Number2*. Dvs programmet returnerar summan av de talen som *Summa*.

product(+Number1, +Number2, -Produkt)

som uttrycker att *Produkt* är produkten av två talen *Number1* och *Number2*;

factorial(+Number, -Fakultet)

som uttrycker att *Fakultet* är fakulteten av talet *Number*.

3. Enkel listhantering. Skriv prologprogram

smaller_than_all(+Elem, +List)

som gäller om elementet *Elem* är mindre än alla element i listan *List*;

exchange_all(+Elem1, +Elem2, +List, -NewList)

som byter alla förekomster av *Elem1* i listan *List* mot *Elem2*. Resultatet är *NewList*;

exchange_nth(+N, +Elem, +List, -NewList)

som byter ut N:te elementet i *List* mot *Elem*. Resultatet är *NewList*;

subset(+List1, +List2)

som gäller om listan *List1* är en ”delmängd” av listan *List2*, dvs alla element i *List1* förekommer i *List2*.

Lab2.

1. Skriv ett prologprogram

free_in(+Variable, +TermOrFormula)

som avgör om varabeln *Variable* är fri i *TermOrFormula* som är antingen en term, dvs

en variabel t.ex. *x1*,
en konstant t.ex. *c*, eller
en funktion med ett godtyckligt antal argument t.ex. *f(x1, x2)*, eller

en formel i predikatlogik med ett godtyckligt antal argument, dvs

en atomär t.ex. *p(x2, c, f(x1, x2)),* eller
en formel t.ex. $\forall x\alpha$, $\exists x\alpha$, $\neg\alpha$, $\alpha\wedge\beta$, $\alpha\vee\beta$, $\alpha\rightarrow\beta$, eller $\alpha\leftrightarrow\beta$

där α och β i sin tur också är en godtycklig formel rekursivt.

2. Skriv ett prologprogram

rename_var(+Variable, +TermOrFormula, +NewVariable, -NewTermOrFormula)

som ersätter alla förekomster av *Variabel* i *TermOrFormula* med en ny variabel *NewVariable*. Resultatet returneras som *NewTermOrFormula*.

Förutsättningar:

1. Endast *x1, x2, ..., xn* (n>1) får användas som variabelsymboler.
2. En godtycklig motsägelseformel, dvs ett falsum, betecknas med atomären *falsum*.
3. En variabel *x1* är fri i *x1* själv, men inte fri i någon konstant eller i *falsum*.
4. Kvantorerna ∀ och ∃ skall beskrivas som 2-ställiga sturukturer *all/2* respektiv *exists/2*, och predikat och funktioner som strukturer med en lista som argument.
5. Logiska konnektiven ¬, ∧, ∨, → och ↔ skall beskrivas som prefixoperator *~/1*, infixoperatorer *&/2*, *#/2*, *->/2* resp *<->/2*.

Extra Krav för Lab2: Både *free_in/2* och *rename_var/4* skall skrivas genom att använda endast *free_in/2* och *rename_var/4* i sig själva samt fördefinierade predikat.

Följande operatorer behöver deklareras i Ditt program:

```
:- op(200, fx, ~).
:- op(400, xfy, #).
:- op(400, xfy, &).
:- op(700, xfy, ->).
:- op(700, xfy, <->).
```

T.ex. Formeln ∀x1(¬P(x1, F(x2,x3)) ∧ R(x3) → ∃x2Q(x1,x2)) skall beskrivas som

all(x1, ~p(x1, f(x2,x3)) & r(x3) -> exists(x2, q(x1,x2)))

Tips. En godtycklig struktur, t. ex. p(x1, f(x2,x3)), kan omvandlas till en lista mha en fördefinierad predikat *=../2* som presenterats i boken *Programming in Prolog*, Clocksin & Mellish, sid. 120:

p(x1, f(x2,x3)) =.. X.	X=[p, x1, f(x2,x3)]
p(x1, f(x2,x3)) =.. [Funk | Arg].	Funk=p, Arg=[x1, f(x2,x3)]
p(x1, f(x2,x3)) =.. [Funk, Arg1, Arg2].	Funk=p, Arg1=x1, Arg2=f(x2,x3)
all(x1, p(x1, f(x2,x3))) =.. X.	X=[all, x1, p(x1, f(x2,x3))]
~p(x1, f(x2,x3)) & r(x3) =.. X.	X=[&, ~p(x1, f(x2,x3)), r(x3)]

Med hjälp av det predikatet *=../2* kan man även omvandla en lista ”tillbaka” till en struktur:

X =.. [all, x1, p(x1, f(x2,x3))]	X= all(x1, p(x1, f(x2,x3)))
X =.. [&, ~p(x1, f(x2,x3)), r(x3)]	X= ~p(x1, f(x2,x3)) & r(x3)

”Eller”-operatorn ;/2 kan användas för att beskriva en disjunktion:

```
P:- Q; R.   % P gäller om antingen Q eller R gäller.
```

Ovanstående regel skrivas om till följande två regler:

```
P:- Q.
P:- R.
```

Lab3.

Med hjälp av bl.a. predikaten *free_in/2* och *rename_var/4* som definierats ovan skriv ett prologprogram

prenex(+Formula, +NewVariableList, -PNF)

som omvandlar en godtycklig predikatlogikformel *Formula* till en prenex normalform *PNF* med hjälp av en lista *NewVariableList* of nya variabler. Predikaten, eller idéerna med, *translate/1, implout/2* och *negin/2* (Clocksin & Mellish, sid. 254-259), kan gärna utnyttjas på ett lämpligt sätt.

Föutsättningar:

1. Som i Lab2.
2. En lista [*y1, y2, ..., y10*] variabler ges som *NewVariableList*. Endast variablerna i listan får användas som nya vid variabelutbyte och de skall räcka.
3. Du skall studera och *förstå* hur de predikaten från boken fungerar *innan* Du använder dem.

Extra krav för Lab3: Körningsexempel med följande 4 fall

```
p(x1, f(x2,x1))
all(x1, p(x1, x2)) -> exists(x2, q(x1, x2))
all(x1, p(x1) -> (exists(x2, q(x1, x2)) & all(x3, r(x2, x3))))
(exists(x1, p(x1, x2)) & all(x2, q(x2, x3))) ->
        (exists(x3, r(x3, f(x1, x2))) # s(x1, x2, x3, x4))
```

skall skapas som en skärmdumpningsfil och en utskrift av filen skall bifogas.

Förslag och tips:

```
prenex(Formula, NewVariableList, PNF):-
    implout(Formula, WithoutImpl),
        % ingen ”->” som var i Formula är nu kvar i WithoutImpl
    negin(WithoutImpl, NegIn),
        % alla ”–” i WithoutImpl har nu flyttats in
    quantout(NegIn, NewVariableList, PNF).
        % alla kvantorer i NegIn har nu flyttats ut
```

NegIn är en av följande alternativ dvs,

1. en atomär t.ex. *p(x1, c, f(x2, x3)),*
2. en negation av en atomär t.ex. *¬p(x1, c, f(x2, x3)),*
3. $\forall x\alpha$,
4. $\exists x\alpha$,
5. $\alpha \wedge \beta$ eller
6. $\alpha \vee \beta$

där α och β i sin tur också är en av de 6 alternativen rekursivt.