Simon Fraser University
School of Computing Science

CMPT 383

Assignment 3 (Prolog)

Due date: November 22, 2005

1) **(7 marks)** Convert the following predicate calculus to Horn clause(s).
`∀g ((logician(g) ∩ ∀a(argument(l,a)⊃ sound(a))) ⊃ happy(g))`

Predicate
∀g ((logician(g) ∧ ∀a(argument(l,a)⊃ sound(a))) ⊃ happy(g))

Simplify
∀g ((logician(g) ∧ ∀a (¬argument(l,a) ∨ sound(a))) ⊃ happy(g))
∀g (¬ (logician(g) ∧ ∀a(¬argument(l,a) ∨ sound(a))) ∨ happy(g))

Move negations in
∀g (¬ logician(g) ∨ ¬(∀a(¬argument(l,a)∨ sound(a))) ∨ happy(g))
∀g (¬ logician(g) ∨ ∃a(¬ (¬argument(l,a) ∨ sound(a))) ∨ happy(g))
∀g (¬ logician(g) ∨ ∃a(argument(l,a) ∧ ¬sound(a)) ∨ happy(g))

Skolemize
∀g (¬ logician(g) ∨ (argument(l,c) ∧ ¬sound(c)) ∨ happy(g))

Remove universal quantifier
¬ logician(g) ∨ (argument(l,c) ∧ ¬sound(c)) ∨ happy(g)

Distribute disjunctions
¬ logician(g) ∨ happy(g) ∨ (argument(l,c) ∧ ¬sound(c))
(¬ logician(g) ∨ happy(g) ∨ argument(l,c)) ∧ (¬ logician(g) ∨ happy(g) ∨ ¬sound(c))

Convert to Clause Normal Form
¬ logician(g) ∨ happy(g) ∨ argument(l,c)
¬ logician(g) ∨ happy(g) ∨ ¬sound(c)

Convert to Horn Clauses
The first Normal Form Clause cannot be converted to a Horn clause because it contains more than one positive literal.

2) **(8 marks)** Given the following Prolog program

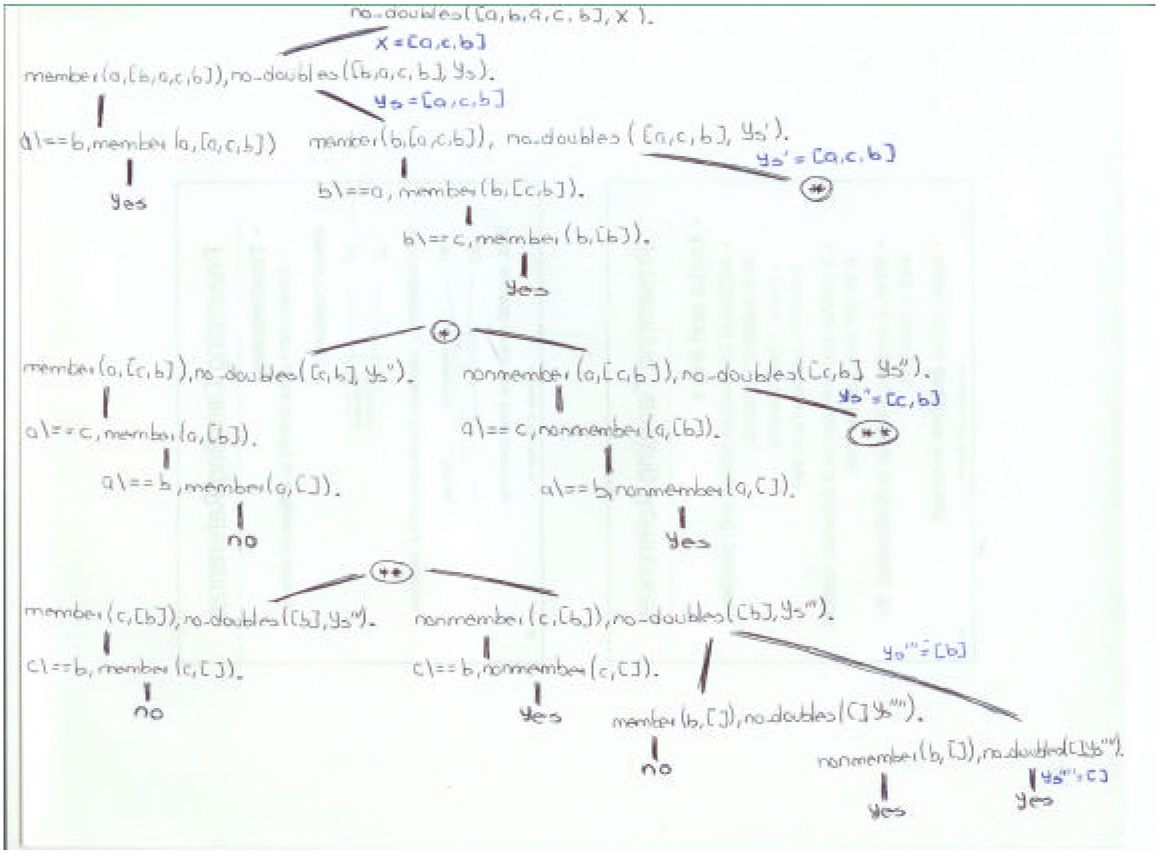
```
no_doubles([],[]).
no_doubles([X|Xs],Ys) :- member(X,Xs), no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :- nonmember(X,Xs), no_doubles(Xs,Ys).

nonmember(X,[]).
nonmember(X,[Y|Ys]) :- X\==Y, nonmember(X,Ys).

member(X,[X|_]).
member(X,[Y|T]) :- X \== Y, member(X,T).
```

Describe the complete execution trace, using a graphic representation, of the following goal:

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



```
?- no_doubles([a,b,a,c,b],X).
     1 Call: no_doubles([a,b,a,c,b],_X)
     2 Call: member(a,[b,a,c,b])
     3 Call: a\==b
     3 Exit: a\==b
     3 Call: member(a,[a,c,b])
     3 Exit: member(a,[a,c,b])
     2 Exit: member(a,[b,a,c,b])
     2 Call: no_doubles([b,a,c,b],_251)
     3 Call: member(b,[a,c,b])
     4 Call: b\==a
     4 Exit: b\==a
     4 Call: member(b,[c,b])
```

```
     5 Call: b\==c
     5 Exit: b\==c
     5 Call: member(b,[b])
     5 Exit: member(b,[b])
     4 Exit: member(b,[c,b])
     3 Exit: member(b,[a,c,b])
     3 Call: no_doubles([a,c,b],_251)
     4 Call: member(a,[c,b])
     5 Call: a\==c
     5 Exit: a\==c
     5 Call: member(a,[b])
     6 Call: a\==b
     6 Exit: a\==b
     6 Call: member(a,[])
     6 Fail: member(a,[])
     5 Fail: member(a,[b])
     4 Fail: member(a,[c,b])
     4 Call: nonmember(a,[c,b])
     5 Call: a\==c
     5 Exit: a\==c
     5 Call: nonmember(a,[b])
     6 Call: a\==b
     6 Exit: a\==b
     6 Call: nonmember(a,[])
     6 Exit: nonmember(a,[])
     5 Exit: nonmember(a,[b])
     4 Exit: nonmember(a,[c,b])
     4 Call: no_doubles([c,b],_7108)
     5 Call: member(c,[b])
     6 Call: c\==b
     6 Exit: c\==b
     6 Call: member(c,[])
     6 Fail: member(c,[])
     5 Fail: member(c,[b])
     5 Call: nonmember(c,[b])
     6 Call: c\==b
     6 Exit: c\==b
     6 Call: nonmember(c,[])
     6 Exit: nonmember(c,[])
     5 Exit: nonmember(c,[b])
     5 Call: no_doubles([b],_10986)
     6 Call: member(b,[])
     6 Fail: member(b,[])
     6 Call: nonmember(b,[])
     6 Exit: nonmember(b,[])
     6 Call: no_doubles([],_13451)
     6 Exit: no_doubles([],[])
     5 Exit: no_doubles([b],[b])
     4 Exit: no_doubles([c,b],[c,b])
     3 Exit: no_doubles([a,c,b],[a,c,b])
     2 Exit: no_doubles([b,a,c,b],[a,c,b])
     1 Exit: no_doubles([a,b,a,c,b],[a,c,b])

X = [a,c,b] ?
```

3) **(5 marks)** Write the predicate **`difference/3`** that defines the set subtraction relation, where all three sets are represented as lists. For example:

```
?- difference( [a,b,c,d], [b,d,e,f], D ).
D = [a,c]

difference([],_,[]).
difference([H|T],L2,[H|L3]) :- non_member(H,L2), difference(T,L2,L3).
difference([H|T],L2,L3) :-     member(H,L2), difference(T,L2,L3).
```

4) **(5 marks)** Write the predicate **`merge/3`** to merge two sorted lists producing a third list. For example:

```
?- merge( [2,5,6,6,8], [1,3,5,9], L ).
L = [1,2,3,5,5,6,6,8,9]

merge([],L,L).
merge(L,[],L).
merge([H1|T1],[H2|T2],[H1|T3]) :-  H1 =< H2, merge(T1,[H2|T2],T3).
merge([H1|T1],[H2|T2],[H2|T3]) :-  H2 < H1, merge([H1|T1],T2,T3).
```

5) **(5 marks)** Write the predicate **`split/3`** to split a list of numbers into two lists: positive ones (including zero) and negative ones. For example:

```
?- split( [3,-1,0,5,-2], P, N ).
P = [3,0,5]
Q = [-1,-2]

split([],[],[]).
split([H|T],[H|P],N) :-    H >= 0, split(T,P,N).
split([H|T],P,[H|N]) :-    H < 0, split(T,P,N).
```

6) **(5 marks)** Define the predicate **`palindrome(List)`**.A list is a palindrome if it reads the same in the forward and in the backward direction. For example:

```
?- palindrome([m,a,d,a,m]).
yes

palindrome(L) :- reverse(L,L).
```

7) **(5 marks)** Define two predicates **`evenlength(List)`** and **`oddlength(List)`** so that they are true if their argument is a list of even or odd length respectively. For example, the list `[a,b,c,d]` is ‘evenlength’ and `[a,b,c]` is ‘oddlength’.

```
evenlength([]).
evenlength([_,_|T]) :- evenlength(T).

oddlength([_]).
oddlength([_,_|T]) :- oddlength(T).
```

or

```
evenlength2([]).
evenlength2([_|T]) :-      oddlength2(T).
oddlength2([H|T]) :-       evenlength2(T).
```

8) **(5 marks)** Assume that a rectangle is represented by the term `rectangle(P1,P2,P3,P4)` where the `P’s` are the vertices of the rectangle

positively ordered. Define the predicate **`regular(R)`**, which is true if `R` is a rectangle whose sides are vertical and horizontal.

```
regular(rectangle(point(X1,Y1),point(X2,Y1),point(X2,Y2),point(X1,Y2))).
regular(rectangle(point(X1,Y1),point(X1,Y2),point(X2,Y2),point(X2,Y1))).
```

9) **(10 marks)** Write the predicate **`simplify/2`** to symbolically simplify summation expressions with numbers and symbols (lower-case letters). Let the predicate to rearrange the expressions so that all the symbols precede numbers. For example:

```
?- simplify( 1+1+a, E ).
E = a+2
?- simplify( 1+a+4+2+b+c, E ).
E = a+b+c+7
?- simplify( 3+x+x, E ).
E = 2*x+3
```

```
simplify(Sum,SimpExp) :- flat_exp(Sum,List), addition(List,SumL),
                         expression(SumL,SimpExp).

% Converts from a summation (structure) to a flatten list
flat_exp(X+Y,[Y|T]) :- !, flat_exp(X,T).
flat_exp(X,[X]).

% Converts from a list to a summation (structure) without parentheses
expression([H],H).
expression([H|T],Exp) :- expression(T,E), Exp = E+H.

% Does the simplification of the summation
addition(List,[SumNum|SumList]) :- addition(List,SumList,0,SumNum).
addition([],[],Num,Num).
% Numbers: sums the value
addition([H|T],SumT,SumNum,NewSum) :-  number(H), !, NewS is SumNum+H,
                                       addition(T,SumT,NewS,NewSum).
% Symbols: counts the number of times
addition([H|T],[Hs|SumT],SumNum,NewSum) :- count(H,T,1,Times,T1),
                  times(H,Times,Hs), !, addition(T1,SumT,SumNum,NewSum).

% Counts the number of times a symbols is in a list and eliminates them
% from the list
count(_,[],N,N,[]).
count(H,[H|T],N,N2,T1) :- !, N1 is N+1, count(H,T,N1,N2,T1).
count(X,[H|T],N,N1,[H|T1]) :- count(X,T,N,N1,T1).

times(H,1,H).
times(H,N,N*H).
```

10) **(5 marks)** Define the predicate **`between(N1,N2,X)`** which, for two given integers `N1` and `N2`, generates through backtracking all integers `X` that satisfy the constraints `N1≤X≤N2`.

```
between(N1,N2,N1):- N1 =< N2.
between(N1,N2,X) :- N1 < N2, Y is N1+1, between(Y,N2,X).
```

Programming Assignment: *Kinship Relations*

The relationships you must define are the following:

- **(1 marks)** *child(X,Y)* - true if X is a child of Y.

```
child(X,Y) :- parent(Y,X).
```

- **(2 marks)** *daughter(X,Y)* - true if X is a daughter of Y.

```
daughter(X,Y) :- child(X,Y), female(X).
% or
daughter(X,Y) :- parent(Y,X), female(X).
```

- **(1 marks)** *parent(X,Y)* - true if X is a parent of Y.

```
% Facts
parent(javier,karla).
…
```

- **(2 marks)** *mother(X,Y)* - true if X is the mother of Y.

```
mother(X,Y) :- parent(X,Y), female(X).
```

- **(4 marks)** *sibling(X,Y)* - true if X and Y are siblings (i.e. have the same biological parents). Be sure your definition does not lead to one being one's own sibling.

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X =\= Y.
```

- **(2 marks)** *brother(X,Y)* - true if X is a brother of Y.

```
brother(X,Y) :- sibling(X,Y), male(X).
```

- **(1 marks)** *grandparent(X,Y)* - true if X is a grandparent of Y.

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

- **(1 marks)** *grandmother(X,Y)* - true if X is a grandmother of Y.

```
grandmother(X,Y) :- grandparent(X,Y), female(X).
```

- **(1 marks)** *grandfather(X,Y)* - true if X is a grandfather of Y.

```
grandfather(X,Y) :- grandparent(X,Y), male(X).
```

- **(3 marks)** *uncle(X,Y)* - true if X is an uncle of Y. Be sure to include uncles by marriage (e.g. your mother's husband's brother) as well as uncles by blood (e.g. your mother's brother).

```
uncle(X,Y) :- parent(Z,Y), brother(X,Z).
uncle(X,Y) :- parent(Z,Y), married(Z,Z1), brother(X,Z1).

% married is defined as:
married(X,Y) :- spouse(X,Y).
married(X,Y) :- spouse(Y,X).
```

- **(2 marks)** *sister-in-law(X,Y)* - true if X is a sister-in-law of Y.

```
sister_in_law(X,Y) :- married(Y,Z), sister(X,Z).

% sister is defined as:
sister(X,Y) :- sibling(X,Y), female(X).
```

- **(2 marks)** *mother-in-law(X,Y)* - true if X is a of Y.

```
mother_in_law(X,Y) :- married(Y,Z), mother(X,Z).
```

- **(1 marks)** *spouse(X,Y)* - true if X and Y are married.

```
% Facts
spouse(javier,carmen).
…
```

- **(2 marks)** *wife(X,Y)* - true if X is the wife of Y.

```
wife(X,Y) :- married(X,Y), female(X).
```

- **(1 marks)** *ancestor(X,Y)* - true if X is a direct ancestor of Y (i.e. a parent or an ancestor of a parent).

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

- **(2 marks)** *descendant(X,Y)* - true if X is a descendant of Y.

```
descendant(X,Y) :- child(X,Y).
descendant(X,Y) :- child(X,Z), descendant(Z,Y).
```

- **(4 marks)** *relative-by-blood(X,Y)* - true if X is a blood relative of Y (i.e. related through some combination of offspring relations).

```
% At least:
relative_by_blood(X,Y) :- ancestor(X,Y).
relative_by_blood(X,Y) :- descendant(X,Y).
relative_by_blood(X,Y) :- sibling(X,Y).
relative_by_blood(X,Y) :- niece_or_nephew(X,Y).
relative_by_blood(X,Y) :- cousin(X,Y).

% niece_or_nephew and cousin are defined as:
niece_or_nephew(X,Y) :- sibling(X1,Y), child(X,X1).
cousin(X,Y) :- parent(X1,X), parent(Y1,Y), sibling(X1,Y1).
```

- **(4 marks)** *relative(X,Y)* - true if X and Y are related somehow (i.e. through some combination of offspring and marriage relations).

```
relative(X,Y) :- relative_by_blood(X,Y).
relative(X,Y) :- relative_by_marriage(X,Y).

relative_by_marriage(X,Y) :- married(Y,X1), relative_by_blood(X,X1).
relative_by_marriage(X,Y) :- relative_by_blood(X1,Y), married(X,X1).
```

- **(4 marks)** *young_parent(X)* – true if X has a child but does not have any grand children.

```
young_parent(X) :- grandparent(X,_), !, fail.
young_parent(X) :- parent(X).
```