CS 603 Exam 3 **Spring 2008** **Solution**

1. Given this Prolog database:

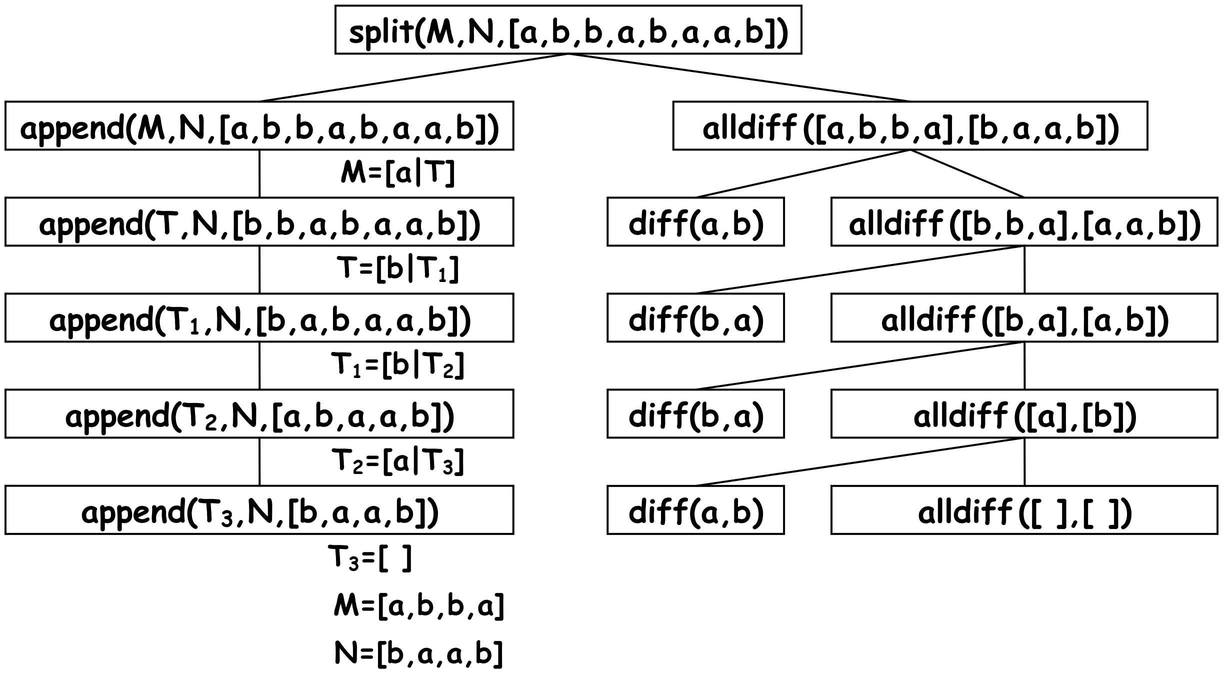
```
split(P,Q,L) :- append(P,Q,L), alldiff(P,Q).

append([ ],Y,Y).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).

alldiff([ ],[ ]).
alldiff([H|T],[X|Y]) :- diff(H,X), alldiff(T,Y).

diff(a,b).
diff(b,a).
```

   Draw the proof tree or search tree that Prolog constructs to satisfy this query: split(M, N, [a,b,b,a,b,a,a,b]). Do not show parts of the search that lead only to dead ends and backtracking, but do show all nodes of the search that contribute to proving the query is satisfied. Also, specify the values of M and N for which this query succeeds.



2. Write this predicate in Prolog: diagonal(M, D) succeeds if M is a matrix stored as a list of row lists, and D is a list of its main diagonal elements. Example: diagonal([[a,b,c],[d,e,f],[g,h,i]], D) succeeds with D = [a,e,i].

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

```
diagonal([ ],[ ]).
diagonal([[H|_]|Rows],[H|T]) :- removeHeads(Rows,X), diagonal(X,T).

removeHeads([ ],[ ]).
removeHeads([[_|T]|Rows],[T|X]) :- removeHeads(Rows,X).
```

3. Write this predicate in Prolog: scan(L, Q) succeeds if L and Q are lists of numbers, and Q is obtained by summing each possible prefix of L. Example: scan([2,3,5,7,11], Q) succeeds with Q = [0,2,5,10,17,28].

```
scan([ ],[0]).
scan([H|T],[0|Z]) :- scan(T,Y), addToAll(H,Y,Z).

addToAll(_,[ ],[ ]).
addToAll(X,[H|T],[A|B]) :- A is X+H, addToAll(X,T,B).
```

4. Write this predicate in Prolog: flatten(L, Q) succeeds if Q is a flat list that contains the same elements as L. Example: flatten([a,[2],[b,[3,[ ]],c],4], Q) succeeds with Q = [a,2,b,3,c,4].

```
flatten([ ],[ ]).
flatten([H|T],Q) :- flatten(H,X), flatten(T,Y), append(X,Y,Q).
flatten(X,[X]) :- not(list(X)).

list([ ]).
list([_|_]).
```

5. Write this predicate in Prolog: eval(E, Z) succeeds if Z is the value of expression E. An expression is essentially a binary tree with leaves labeled "num(_)" and internal nodes labeled "add(_,_)", "sub(_,_)", "mul(_,_)", or "div(_,_)". Example: eval(mul(add(num(4),num(7)), sub(num(5),num(2))), Z) succeeds with Z = (4+7)*(5-2) = 33.

```
eval(num(A),A).
eval(add(A,B),Z) :- eval(A,X), eval(B,Y), Z is X+Y.
eval(sub(A,B),Z) :- eval(A,X), eval(B,Y), Z is X-Y.
eval(mul(A,B),Z) :- eval(A,X), eval(B,Y), Z is X*Y.
eval(div(A,B),Z) :- eval(A,X), eval(B,Y), Z is X/Y.
```

6. Rewrite each logical expression as an equivalent completely simplified expression.

   a. $(a \wedge \neg b) \rightarrow (\neg b \vee c)$

      true

   b. $(a \vee b) \rightarrow (a \wedge b)$

      $a \leftrightarrow b$

   c. $(\forall x)(P(x)) \rightarrow (\exists x)(P(x))$

      $(\exists x)$(true), or just $(\exists x)$

7. Apply resolution and unification to each pair of rules.

   a. a(X,0), b(1,X) ← c(X,2), d(3,X). b(Z,4), c(5,Z) ← d(Z,6), e(7,Z).

      Let X:=5, Z:=2 ⇒ a(5,0), b(1,5), b(2,4) ← d(3,5), d(2,6), e(7,2).

   b. f(W,V), g(q(V),U) ← h(U,W). f(Y,Z) ← g(X,p(Y)), h(Z,X).

      Let X:=q(V), U:=p(Y) ⇒ f(W,V), f(Y,Z) ← h(p(Y),W), h(Z,q(V)).

8. Most Prolog interpreters use a sequential search algorithm based on depth-first search or preorder/postorder traversal of an underlying search tree. However, now consider a parallel implementation of Prolog that instead uses breadth-first search or level-order traversal of the search tree. So, whenever a goal matches several rules in the database, explore all alternatives in parallel (OR-parallelism). Also, whenever a goal has multiple subgoals, explore all subgoals in parallel (AND-parallelism). Identify the major advantages and disadvantages of such a parallel Prolog, as compared to the standard sequential approach.

   Advantages of a parallel Prolog:
   - **Much closer to pure logic programming**
   - **The order of rules and subgoals does not matter**
   - **Backtracking is not needed**
   - **Better for parallelizing or distributing the work to multiple processors**
   - **Speedup if solution would be in a far right subtree using a sequential search**

   Disadvantages of a parallel Prolog:
   - **Slowdown if solution would be in a far left subtree using a sequential search**
   - **Huge amount of memory needed to explore all branches simultaneously**
   - **Subtrees that share variables must synchronize/communicate**
   - **Makes less sense for programs that use arithmetic or side-effects**

9. Write loop invariants that could be used within each algorithm's correctness proof:

   a. $x > 0$ { (set a 0) (set b 1) (while (< b x) (begin (set a b) (set b (+ b 1)))) } $a = x-1$

      $\mathbf{b \le x \;\wedge\; a = b-1}$

   b. $x \ge 0$ { (set a x) (set b x) (while (> b 0) (begin (set a (+ a 1)) (set b (- b 1)))) } $a = 2x$

      $\mathbf{b \ge 0 \;\wedge\; a+b = 2x}$