DM552: Programming Languages

Project 1 Reexam Report

Torvald Johnson

Table of Contents

Implementation:

The following is a concise description of the project, and the predicates I have defined in this project to meet the requirements.

The project itself was based around prepositional logic, with part one requiring me to create predicates in prolog to test different aspects of a prepositional formula. The system is able to test if a given valuation can satisfy a given formula, produce the different valuations that can satisfy a given formula, and test if a given formula is a tautology, satisfiable, or unsatisfiable.

Part two of the project was similar, but employing prepositional tableaus to solve the problems. The system is able to provide all leafs of a tableau for a given prepositional formula, find all values that satisfy the leafs of a given formula, and test if a formula is a tautology, satisfiable, or unsatisfiable using a tableau.

Predicates in part 1:

wff/1 tests if the input is comprised of the connectivesneg(A),impl(A,B),and(A,B),or(A,B),equiv(A,B), and xor(A,B).If these connectives all contain p(_), the given input is a prepositional formula.

satisfies/2accepts a valuation and a prepositional formula. To prevent backtracking and incorrect answers, satisfies/2 uses the auxiliary predicate check_satisfies/2 to test the valuation against the propositional formula, followed by a cut. If the answer comes back true, the valuation satisfies the formula and there is no need to backtrack.

find_val_tt/2 accepts a formula, and using the auxiliary gen_valuation_combos/2 predicate, builds a list of all p(_) integer values contained in the formula and then finds all possible combinations of these values starting with the empty list. It then checks if any of these combinations satisfies the formula, using satisfies/2.

taut_tt/1 accepts a formula, and uses the built-in predicate findall/3 along with the auxiliary predicate gen_valuation_combos/2 to generate a list of all possible valuation combinations. The auxiliary predicate loop_valuations_tt/2 then checks if every valuation satisfies the given formula. This predicate contains a cut at the end, because by that point every possible valuation has been checked against the formula, and if it is true there is no need to backtrack.

sat_tt/1 accepts a formula, and tests if it is satisfiable using find_val_tt/2. This predicate employs a cut at the end to prevent backtracking and wrong answers. If the formula has been found to be satisfiable once, it is known to be satisfiable and there is no need to backtrack.

unsat_tt/1 accepts a formula, and tests if it is not satisfiable using the built in predicate not/1 and find_val_tt/2.

Auxiliary predicates in part 1:

check_satisfies/2is an auxiliary predicate whichaccepts a valuation and a prepositional formula. It uses recursion and the built-in member/2 predicate to check if the value in each p(_) is contained in the given valuation, or not, based on what would make the current prepositional formula true.

gen_valuation_combos/2 is an auxiliary predicate which uses the auxiliary predicates make_list/2, remove_dups/2, and combinations/3 to generate a list of all possible valuations which could solve a given formula

make_list/2 is an auxiliary predicate which recursively builds a list of all the p(_) int values contained in a given formula.

remove_dups/2 is an auxiliary predicate which, together with helper predicate remove_dups/3, recursively builds a list containing all values of a given list, but without duplicates.

combinations/3 is an auxiliary predicate which, when backtracked, will provide all possible combinations of a given list, including the empty list.

Predicates in part 2:

tableau/2 accepts a formula, and using recursion, returns a list containing a leaf pertaining to the formula. Using backtracking, all leafs of the formula can be produced. The leafs contain relevant lists of p(X) or neg(p(X)), as outlined by the project description.

find_val_tab/2 first uses tableau/2 to find the first leaf pertaining to the given formula. Then the built-in predicate findall/3 produces all possible valuation combinations for the formula. Auxiliary predicate loop_valuations_tab/3 then loops through the possible valuation combinations, and uses loop_leaf_valuations_tab/2 to loop through every p(X) within the leaf, to check if this valuation satisfies this leaf. If the valuation satisfies every p(X) within the leaf, then it is returned as the answer. The user can backtrack to try every possible valuation against that leaf, and then continue backtracking further to try every possible leaf returned by the tableau/2 predicate.

taut_tab/1 accepts a formula, and using the built-in predicate findall, generates a list of all possible leafs of the formula and a list of all possible valuations. The auxiliary predicate loop_taut_tab/2 then loops through every valuation. For each valuation, loop_leafs_tab/2 loops through every leaf, and loop_leaf_valuations_tab/2 loops through every p(X) value within the leaf and tests if the given valuation satisfies it. If every possible valuation satisfies every p(X) value within every leaf, the formula is a tautology. This predicate contains a cut at the end, because by that point every possible valuation has been checked against every leaf, and if it is true there is no need to backtrack.

sat_tab/1 accepts a formula, and tests if it is satisfiable using find_val_tab/2. This predicate employs a cut at the end to prevent backtracking and wrong answers. If the formula has been found to be satisfiable once, it is known to be satisfiable and there is no need to backtrack.

unsat_tab/1 accepts a formula, and tests if it is not satisfiable using the built in predicate not/1 and find_val_tab/2.

Auxiliary predicates in part 2:

loop_valuations_tab/3 is an auxiliary predicate, used to recursively loop through a list of valuations and, using the loop_leaf_valuations_tab/2 predicate, find which valuations satisfy a given leaf.

loop_leaf_valuations_tab/2 is an auxiliary predicate which accepts a list of p(X) values and a valuation. loop_leaf_valuations_tab/2 recursively loops through the list of p(X) values provided as a leaf of a tableau, and tests if the given valuation satisfies each p(X) value.

loop_taut_tab/2 accepts a list of leafs and valuations, and recursively loops through the given list of valuations, and calls loop_leafs_tab/2 on each one to check if every valuation satisfies every leaf.

loop_leafs_tab/2 accepts a list of leafs and a valuation, and recursively loops through the given list of leafs, calling loop_leaf_valuations_tab/2 on each one to check if the valuation satisfies every leaf.

Example executions(using the compiler at http://swish.swi-prolog.org/):

```
wff(and(p(1),or(p(2),p(3)))).
true
```

```
satisfies([1,2,3],or(and(p(1),p(2)),p(3))).
true
```

```
find_val_tt(and(p(1),xor(p(2),neg(p(3)))),V).
V = [1]
V = [1, 2, 3]
false
```

```
taut_tt(impl(and(p(1),p(2),p(2)))).
true
```

```
sat_tt(xor(p(1),p(1))).
false
```

```
unsat_tt(xor(p(1),p(1))).
true
```

```
tableau(impl(equiv(p(1),p(2)),and(p(3),or(p(1),neg(p(3))))),V).
V = [p(1), neg(p(2))]
V = [neg(p(1)), p(2)]
V = [p(3), p(1)]
V = [p(3), neg(p(3))]
false
```

```
find_val_tab(impl(equiv(p(1),p(2)),and(p(3),or(p(1),neg(p(3))))),V).
V = [1]
V = [1, 3]
V = [2]
V = [2, 3]
V = [1, 3]
V = [1, 2, 3]
false
```

```
taut_tab(equiv(impl(and(p(1),p(2)),p(3)),impl(p(1),impl(p(2),p(3))))).
true
```

```
sat_tab(equiv(p(1),impl(p(1),and(p(2),p(3))))).
true
```

```
unsat_tab(and(and(p(1),p(2)),xor(neg(p(1)),neg(p(2))))).
true
```

Full program listing:

```
/* 1.1
 * wff/1 tests if the input is comprised of the connectives
 * neg(A),impl(A,B),and(A,B),or(A,B),equiv(A,B),and xor(A,B).
 * If these connectives contain p(_), the given input is a
 * prepositional formula.
 */
wff(p(_)).
wff(neg(A)):-wff(A).
wff(impl(A,B)):-wff(A),wff(B).
wff(and(A,B)):-wff(A),wff(B).
wff(or(A,B)):-wff(A),wff(B).
wff(equiv(A,B)):-wff(A),wff(B).
wff(xor(A,B)):-wff(A),wff(B).



/* 1.2
 * satisfies/2 accepts a valuation and a prepositional formula.
 * To prevent backtracking and incorrect answers, satisfies/2
 * uses the auxiliary predicate check_satisfies/2 to test the
 * valuation against the propositional formula, followed by a
 * cut.
 */

satisfies(V,A):- check_satisfies(V,A), !.

/* check_satisfies/2 is an auxiliary predicate, which
 * uses recursion and the built-in member/2 predicate to
 * check if the value in each p(_) is contained in
 * the given valuation, or not, based on what
 * would make the current prepositional formula
 * true.
 */
check_satisfies(V,p(A)):-member(A,V).

check_satisfies(V,neg(A)):-not(check_satisfies(V,A)).

check_satisfies(V,and(A,B)):-check_satisfies(V,A), check_satisfies(V,B).

check_satisfies(V,or(A,B)):-check_satisfies(V,A) ; check_satisfies(V,B).

check_satisfies(V,equiv(A,B)):-check_satisfies(V,A), check_satisfies(V,B).
check_satisfies(V,equiv(A,B)):-not(check_satisfies(V,A)), not(check_satisfies(V,B)).

check_satisfies(V,impl(A,B)):-not(check_satisfies(V,A)) ; check_satisfies(V,B).

check_satisfies(V,xor(A,B)):-check_satisfies(V,A), not(check_satisfies(V,B)).
check_satisfies(V,xor(A,B)):-not(check_satisfies(V,A)), check_satisfies(V,B).



/* 1.3
 * find_val_tt/2 accepts a formula, and using the gen_valuation_combos/2
 * predicate, builds a list of all p(_) integer values contained in the
 * formula and then finds all possible combinations of these values starting
 * with the empty list. It then checks if any of these combinations
```

```
 * satisfies the formula, using satisfies/2.
 */
find_val_tt(F,V):-gen_valuation_combos(F,V), satisfies(V,F).

/*
 * gen_valuation_combos/2 is an auxiliary predicate which
 * uses the auxiliary predicates make_list/2, remove_dups/2,
 * and combinations/3 to generate a list of all possible
 * valuations which could solve a given formula
 */
gen_valuation_combos(F,R):-make_list(X,F), remove_dups(X,L), combinations(L,_,R).

/* make_list/2 is an auxiliary predicate which
 * recursively builds a list of all the p(_)
 * int values contained in a given formula.
 */
make_list([A],p(A)).
make_list(R,neg(A)):-make_list(X,A), append(X,[],R).
make_list(R,and(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,or(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,equiv(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,impl(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,xor(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).

/* remove_dups/2 is an auxiliary predicate which,
 * together with helper predicate remove_dups/3,
 * recursively builds a list containing all
 * values of a given list, but without duplicates.
 */
remove_dups([],[]).
remove_dups([H|T],[H|R]):-remove_dups(H,T,S), remove_dups(S,R).

remove_dups(_,[],[]).
remove_dups(X,[X|T],R):-remove_dups(X,T,R).
remove_dups(X,[H|T],[H|R]):-not(X = H), remove_dups(X,T,R).

/* combinations/3 is an auxiliary predicate which,
 * when backtracked, will provide all possible
 * combinations of a given list, including the
 * empty list.
 */
combinations([],[],[]).
combinations([H|T],[H|L],R):-combinations(T,L,R).
combinations([H|T],L,[H|R]):-combinations(T,L,R).



/* 1.4
 * taut_tt/1 accepts a formula, and uses the built-in
 * predicate findall/3 along with the auxiliary predicate
 * gen_valuation_combos/2 to generate a list of all
 * possible valuation combinations. The auxiliary predicate
 * loop_valuations_tt/2 then checks if every valuation satisfies the
 * given formula.
 */
taut_tt(F):-findall(V,gen_valuation_combos(F,V),R), loop_valuations_tt(R,F), !.

/* loop_valuations_tt/2 accepts a list of valuations and a prepositional
 * formula, then recursively tests if every valuation in the
 * list satisfies the formula.
 */
```

```
loop_valuations_tt([],_).
loop_valuations_tt([H|T],F):-satisfies(H,F), loop_valuations_tt(T,F).

/* sat_tt/1 accepts a formula, and tests if it is satisfiable
 * using find_val_tt/2.
 */
sat_tt(F):-find_val_tt(F,_),!.

/* unsat_tt/1 accepts a formula, and tests if it is not
 * satisfiable using the built in predicate not/1 and
 * find_val_tt/2.
 */
unsat_tt(F):-not(find_val_tt(F,_)).



/* PART 2 */



/* 2.5
 * tableau/2 accepts a formula, and using recursion, returns a list
 * containing a leaf pertaining to the formula. Using backtracking,
 * all leafs of the formula can be produced. The leafs contain relevant
 * lists of p(X) or neg(p(X)), as outlined by the project description.
 */
tableau(p(X),[p(X)]).
tableau(neg(p(X)),[neg(p(X))]).

tableau(neg(neg(A)),R):-tableau(A,R).

tableau(impl(A,_),R):-tableau(neg(A),R).
tableau(impl(_,B),R):-tableau(B,R).
tableau(neg(impl(A,B)),R):-tableau(A,X), tableau(neg(B),Y), append(X,Y,R).

tableau(and(A,B),R):-tableau(A,X), tableau(B,Y), append(X,Y,R).

tableau(neg(and(A,_)),R):-tableau(neg(A),R).
tableau(neg(and(_,B)),R):-tableau(neg(B),R).

tableau(or(A,_),R):-tableau(A,R).
tableau(or(_,B),R):-tableau(B,R).

tableau(neg(or(A,B)),R):-tableau(neg(A),X), tableau(neg(B),Y), append(X,Y,R).

tableau(equiv(A,B),R):-tableau(A,X), tableau(B,Y), append(X,Y,R).

tableau(equiv(A,B),R):-tableau(neg(A),X), tableau(neg(B),Y), append(X,Y,R).

tableau(neg(equiv(A,B)),R):-tableau(A,X), tableau(neg(B),Y), append(X,Y,R).

tableau(neg(equiv(A,B)),R):-tableau(neg(A),X), tableau(B,Y), append(X,Y,R).

tableau(xor(A,B),R):-tableau(A,X), tableau(neg(B),Y), append(X,Y,R).

tableau(xor(A,B),R):-tableau(neg(A),X), tableau(B,Y), append(X,Y,R).

tableau(neg(xor(A,B)),R):-tableau(A,X),tableau(B,Y), append(X,Y,R).

tableau(neg(xor(A,B)),R):-tableau(neg(A),X), tableau(neg(B),Y), append(X,Y,R).
```

```
/* 2.6
 * find_val_tab/2 first uses tableau/2 to find the first leaf pertaining
 * to the given formula. Then the built-in predicate findall/3 produces
 * all possible valuation combinations for the formula. Auxiliary
 * predicate loop_valuations_tab/3 then loops through the possible valuation
 * combinations, and uses loop_leaf_valuations_tab/2 to loop through every p(X) within
 * the leaf, to check if this valuation satisfies this leaf. If the valuation
 * satisfies every p(X) within the leaf, then it is returned as the answer.
 * The user can backtrack to try every possible valuation against that leaf, and
 * then continue backtracking further to try every possible leaf returned by the
 * tableau/2 predicate.
 */
find_val_tab(F,V):-tableau(F,X), findall(C,gen_valuation_combos(F,C),R),
loop_valuations_tab(X,R,V).


/* loop_valuations_tab/3 is an auxiliary predicate, used to recursively loop through a
 * list of valuations and, using the loop_leaf_valuations_tab/2 predicate, find which
valuations
 * satisfy a given leaf.
 */
loop_valuations_tab(X,[H|_],H):-loop_leaf_valuations_tab(X,H).
loop_valuations_tab(X,[_|T],R):-loop_valuations_tab(X,T,R).

/* loop_leaf_valuations_tab/2 is an auxiliary predicate which accepts a list of p(X)
values
 * and a valuation. loop_leaf_valuations_tab/2 recursively loops through the list of
p(X)
 * values provided as a leaf of a tableau, and tests if the given valuation
 * satisfies each p(X) value.
 */
loop_leaf_valuations_tab([],_).
loop_leaf_valuations_tab([H|T],V):-satisfies(V,H), loop_leaf_valuations_tab(T,V).



/* 2.7
 * taut_tab/1 accepts a formula, and using the built-in predicate
 * findall, generates a list of all possible leafs of the formula and
 * a list of all possible valuations. The auxiliary predicate loop_taut_tab/2
 * then loops through every valuation. For each valuation, loop_leafs_tab/2
 * loops through every leaf, and loop_leaf_valuations_tab/2 loops through every p(X)
 * value within the leaf and tests if the given valuation satisfies it.
 * If every possible valuation satisfies every p(X) value within every
 * leaf, the formula is a tautology.
 */
taut_tab(F):-
findall(X,tableau(F,X),R1),findall(Y,gen_valuation_combos(F,Y),R2),loop_taut_tab(R1,R2)
, !.

/* loop_taut_tab/2 accepts a list of leafs and valuations, and recursively
 * loops through the given list of valuations, and calls loop_leafs_tab/2 on
 * each one to check if every valuation satisfies every leaf.
 */
loop_taut_tab(_,[]).
loop_taut_tab(L,[H|T]):-loop_leafs_tab(L,H),loop_taut_tab(L,T).

/* loop_leafs_tab/2 accepts a list of leafs and a valuation, and recursively
 * loops through the given list of leafs, calling loop_leaf_valuations_tab/2 on
```

```
 * each one to check if the valuation satisfies every leaf.
 */
loop_leafs_tab([H|_],V):-loop_leaf_valuations_tab(H,V).
loop_leafs_tab([_|T],V):-loop_leafs_tab(T,V).

/* sat_tab/1 accepts a formula, and tests if it is satisfiable
 * using find_val_tab/2.
 */
sat_tab(F):-find_val_tab(F,_),!.

/* unsat_tab/1 accepts a formula, and tests if it is not
 * satisfiable using the built in predicate not/1 and
 * find_val_tab/2.
 */
unsat_tab(F):-not(find_val_tab(F,_))./* 1.1
 * wff/1 tests if the input is comprised of the connectives
 * neg(A),impl(A,B),and(A,B),or(A,B),equiv(A,B),and xor(A,B).
 * If these connectives contain p(_), the given input is a
 * prepositional formula.
 */
wff(p(_)).
wff(neg(A)):-wff(A).
wff(impl(A,B)):-wff(A),wff(B).
wff(and(A,B)):-wff(A),wff(B).
wff(or(A,B)):-wff(A),wff(B).
wff(equiv(A,B)):-wff(A),wff(B).
wff(xor(A,B)):-wff(A),wff(B).



/* 1.2
 * satisfies/2 accepts a valuation and a prepositional formula.
 * To prevent backtracking and incorrect answers, satisfies/2
 * uses the auxiliary predicate check_satisfies/2 to test the
 * valuation against the propositional formula, followed by a
 * cut. If the answer comes back true, the valuation satisfies
 * the formula and there is no need to backtrack.
 */

satisfies(V,A):- check_satisfies(V,A), !.

/* check_satisfies/2 is an auxiliary predicate, which
 * uses recursion and the built-in member/2 predicate to
 * check if the value in each p(_) is contained in
 * the given valuation, or not, based on what
 * would make the current prepositional formula
 * true.
 */
check_satisfies(V,p(A)):-member(A,V).

check_satisfies(V,neg(A)):-not(check_satisfies(V,A)).

check_satisfies(V,and(A,B)):-check_satisfies(V,A), check_satisfies(V,B).

check_satisfies(V,or(A,B)):-check_satisfies(V,A) ; check_satisfies(V,B).

check_satisfies(V,equiv(A,B)):-check_satisfies(V,A), check_satisfies(V,B).
check_satisfies(V,equiv(A,B)):-not(check_satisfies(V,A)), not(check_satisfies(V,B)).

check_satisfies(V,impl(A,B)):-not(check_satisfies(V,A)) ; check_satisfies(V,B).
```

```
check_satisfies(V,xor(A,B)):-check_satisfies(V,A), not(check_satisfies(V,B)).
check_satisfies(V,xor(A,B)):-not(check_satisfies(V,A)), check_satisfies(V,B).



/* 1.3
 * find_val_tt/2 accepts a formula, and using the gen_valuation_combos/2
 * predicate, builds a list of all p(_) integer values contained in the
 * formula and then finds all possible combinations of these values starting
 * with the empty list. It then checks if any of these combinations
 * satisfies the formula, using satisfies/2.
 */
find_val_tt(F,V):-gen_valuation_combos(F,V), satisfies(V,F).

/*
 * gen_valuation_combos/2 is an auxiliary predicate which
 * uses the auxiliary predicates make_list/2, remove_dups/2,
 * and combinations/3 to generate a list of all possible
 * valuations which could solve a given formula
 */
gen_valuation_combos(F,R):-make_list(X,F), remove_dups(X,L), combinations(L,_,R).

/* make_list/2 is an auxiliary predicate which
 * recursively builds a list of all the p(_)
 * int values contained in a given formula.
 */
make_list([A],p(A)).
make_list(R,neg(A)):-make_list(X,A), append(X,[],R).
make_list(R,and(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,or(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,equiv(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,impl(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).
make_list(R,xor(A,B)):-make_list(X,A),make_list(Y,B), append(X,Y,R).

/* remove_dups/2 is an auxiliary predicate which,
 * together with helper predicate remove_dups/3,
 * recursively builds a list containing all
 * values of a given list, but without duplicates.
 */
remove_dups([],[]).
remove_dups([H|T],[H|R]):-remove_dups(H,T,S), remove_dups(S,R).

remove_dups(_,[],[]).
remove_dups(X,[X|T],R):-remove_dups(X,T,R).
remove_dups(X,[H|T],[H|R]):-not(X = H), remove_dups(X,T,R).

/* combinations/3 is an auxiliary predicate which,
 * when backtracked, will provide all possible
 * combinations of a given list, including the
 * empty list.
 */
combinations([],[],[]).
combinations([H|T],[H|L],R):-combinations(T,L,R).
combinations([H|T],L,[H|R]):-combinations(T,L,R).



/* 1.4
 * taut_tt/1 accepts a formula, and uses the built-in
 * predicate findall/3 along with the auxiliary predicate
 * gen_valuation_combos/2 to generate a list of all
```

```
 * possible valuation combinations. The auxiliary predicate
 * loop_valuations_tt/2 then checks if every valuation satisfies the
 * given formula.
 */
taut_tt(F):-findall(V,gen_valuation_combos(F,V),R), loop_valuations_tt(R,F), !.

/* loop_valuations_tt/2 accepts a list of valuations and a prepositional
 * formula, then recursively tests if every valuation in the
 * list satisfies the formula.
 */
loop_valuations_tt([],_).
loop_valuations_tt([H|T],F):-satisfies(H,F), loop_valuations_tt(T,F).

/* sat_tt/1 accepts a formula, and tests if it is satisfiable
 * using find_val_tt/2.
 */
sat_tt(F):-find_val_tt(F,_),!.

/* unsat_tt/1 accepts a formula, and tests if it is not
 * satisfiable using the built in predicate not/1 and
 * find_val_tt/2.
 */
unsat_tt(F):-not(find_val_tt(F,_)).



/* PART 2 */



/* 2.5
 * tableau/2 accepts a formula, and using recursion, returns a list
 * containing a leaf pertaining to the formula. Using backtracking,
 * all leafs of the formula can be produced. The leafs contain relevant
 * lists of p(X) or neg(p(X)), as outlined by the project description.
 */
tableau(p(X),[p(X)]).

tableau(neg(p(X)),[neg(p(X))]).
tableau(neg(neg(A)),R):-tableau(A,R).

tableau(and(A,B),R):-tableau(A,X), tableau(B,Y), append(X,Y,R).
tableau(neg(and(A,_)),R):-tableau(neg(A),R).
tableau(neg(and(_,B)),R):-tableau(neg(B),R).

tableau(or(A,_),R):-tableau(A,R).
tableau(or(_,B),R):-tableau(B,R).
tableau(neg(or(A,B)),R):-tableau(neg(A),X), tableau(neg(B),Y), append(X,Y,R).

tableau(equiv(A,B),R):-tableau(A,X), tableau(B,Y), append(X,Y,R).
tableau(equiv(A,B),R):-tableau(neg(A),X), tableau(neg(B),Y), append(X,Y,R).
tableau(neg(equiv(A,B)),R):-tableau(A,X), tableau(neg(B),Y), append(X,Y,R).
tableau(neg(equiv(A,B)),R):-tableau(neg(A),X), tableau(B,Y), append(X,Y,R).

tableau(impl(A,_),R):-tableau(neg(A),R).
tableau(impl(_,B),R):-tableau(B,R).
tableau(neg(impl(A,B)),R):-tableau(A,X), tableau(neg(B),Y), append(X,Y,R).

tableau(xor(A,B),R):-tableau(A,X), tableau(neg(B),Y), append(X,Y,R).
tableau(xor(A,B),R):-tableau(neg(A),X), tableau(B,Y), append(X,Y,R).
tableau(neg(xor(A,B)),R):-tableau(A,X),tableau(B,Y), append(X,Y,R).
```

```
tableau(neg(xor(A,B)),R):-tableau(neg(A),X), tableau(neg(B),Y), append(X,Y,R).



/* 2.6
 * find_val_tab/2 first uses tableau/2 to find the first leaf pertaining
 * to the given formula. Then the built-in predicate findall/3 produces
 * all possible valuation combinations for the formula. Auxiliary
 * predicate loop_valuations_tab/3 then loops through the possible valuation
 * combinations, and uses loop_leaf_valuations_tab/2 to loop through every p(X) within
 * the leaf, to check if this valuation satisfies this leaf. If the valuation
 * satisfies every p(X) within the leaf, then it is returned as the answer.
 * The user can backtrack to try every possible valuation against that leaf, and
 * then continue backtracking further to try every possible leaf returned by the
 * tableau/2 predicate.
 */
find_val_tab(F,V):-tableau(F,X), findall(C,gen_valuation_combos(F,C),R),
loop_valuations_tab(X,R,V).


/* loop_valuations_tab/3 is an auxiliary predicate, used to recursively loop through a
 * list of valuations and, using the loop_leaf_valuations_tab/2 predicate, find which
valuations
 * satisfy a given leaf.
 */
loop_valuations_tab(X,[H|_],H):-loop_leaf_valuations_tab(X,H).
loop_valuations_tab(X,[_|T],R):-loop_valuations_tab(X,T,R).

/* loop_leaf_valuations_tab/2 is an auxiliary predicate which accepts a list of p(X)
values
 * and a valuation. loop_leaf_valuations_tab/2 recursively loops through the list of
p(X)
 * values provided as a leaf of a tableau, and tests if the given valuation
 * satisfies each p(X) value.
 */
loop_leaf_valuations_tab([],_).
loop_leaf_valuations_tab([H|T],V):-satisfies(V,H), loop_leaf_valuations_tab(T,V).



/* 2.7
 * taut_tab/1 accepts a formula, and using the built-in predicate
 * findall, generates a list of all possible leafs of the formula and
 * a list of all possible valuations. The auxiliary predicate loop_taut_tab/2
 * then loops through every valuation. For each valuation, loop_leafs_tab/2
 * loops through every leaf, and loop_leaf_valuations_tab/2 loops through every p(X)
 * value within the leaf and tests if the given valuation satisfies it.
 * If every possible valuation satisfies every p(X) value within every
 * leaf, the formula is a tautology.
 */
taut_tab(F):-
findall(X,tableau(F,X),R1),findall(Y,gen_valuation_combos(F,Y),R2),loop_taut_tab(R1,R2)
, !.

/* loop_taut_tab/2 accepts a list of leafs and valuations, and recursively
 * loops through the given list of valuations, and calls loop_leafs_tab/2 on
 * each one to check if every valuation satisfies every leaf.
 */
loop_taut_tab(_,[]).
loop_taut_tab(L,[H|T]):-loop_leafs_tab(L,H),loop_taut_tab(L,T).
```

```
/* loop_leafs_tab/2 accepts a list of leafs and a valuation, and recursively
 * loops through the given list of leafs, calling loop_leaf_valuations_tab/2 on
 * each one to check if the valuation satisfies every leaf.
 */
loop_leafs_tab([H|_],V):-loop_leaf_valuations_tab(H,V).
loop_leafs_tab([_|T],V):-loop_leafs_tab(T,V).

/* sat_tab/1 accepts a formula, and tests if it is satisfiable
 * using find_val_tab/2.
 */
sat_tab(F):-find_val_tab(F,_),!.

/* unsat_tab/1 accepts a formula, and tests if it is not
 * satisfiable using the built in predicate not/1 and
 * find_val_tab/2.
 */
unsat_tab(F):-not(find_val_tab(F,_)).
```