Mathematical Logic for Computer Science
(Third Edition)

Prolog Programs

Version 3.0.0

Mordechai Ben-Ari
`http://www.weizmann.ac.il/sci-tea/benari/`

March 14, 2012

1 Source archive

The programs have been tested with SWI-Prolog Version 6.0.0 `http://www.swi-prolog.org/`.
The source files use the extension `pro` instead of `pl` to avoid conflict with program in Perl.
The archive is a zip file structured into directories (see Appendix A for the list of files):

- `common` - modules and files used by other programs
- `prop` - propositional logic
- `fol` - first-order logic
- `tl` - temporal logic

2 Common modules

The operators in the logics are represented within the Prolog programs using the following operators defined in the file `ops`:

```
:- op(650, xfy, xor).         /* exclusive or */
:- op(650, xfy, eqv).         /* equivalence  */
:- op(650, xfy, nor).         /* nor          */
:- op(650, xfy, nand).        /* nand         */
:- op(640, xfy, imp).         /* implication  */
:- op(630, xfy, or).          /* disjunction  */
:- op(620, xfy, and).         /* conjunction  */
:- op(610, fy,  neg).         /* negation     */
:- op(600, xfy, eq).          /* equality     */
:- op(610, fy,  always).      /* always       */
:- op(610, fy,  eventually).  /* eventually   */
:- op(610, fy,  next).        /* next         */
```

Formulas written with this notation are not easy to read or write. Instead, the following operators are used to for input and output:

```
:- op(650, xfy,+).           /* exclusive or */
:- op(650, xfy,<->).         /* equivalence  */
:- op(640, xfy,-->).         /* implication  */
:- op(630, xfy,v).           /* disjunction  */
:- op(620, xfy,^).           /* conjunction  */
:- op(610, fy, ~).           /* negation     */
:- op(610, fy, #).           /* always       */
:- op(610, fy, <>).          /* eventually   */
:- op(610, fy, @).           /* next         */
```

Module `intext` contains predicates that translate between the notations.

Here is a formula followed by its external and internal representations:

- $(p \oplus q) \leftrightarrow (\neg(p \rightarrow q) \vee \neg(q \rightarrow p))$.
- `(p xor q)  eqv (neg (p imp q) or neg (q imp p))`
- `(p + q) <-> (~(p -> q) v ~(q -> p))`

Other modules in the directory `common` are `defs` which contain the semantic definitions of the Boolean operators and `io` which performs input and output of the various logical formalisms.

3 Propositional logic

3.1 Truth tables

The predicate `tt(Fml, V, TV)` returns the truth value `TV` of formula `Fml` under the assignment `V`. The assignment is a list of pairs `(A,TV)`, where `A` is an atom and `TV` is `t` or `f`, for example, `[(p,f),(q,t)]`. `tt` recurses on the structure of the formula. For atoms, it returns the truth value by lookup in the list; for negations, `neg` is called to negate the value; for formulas with a binary operator, `opr` is called to compute the truth value from the truth values of the subformulas.

`create_tt(Fml)` prints the truth table for `Fml`.

```
create_tt(Fml) :-
  to_internal(Fml, IFml),
  get_atoms(IFml, Atoms),
  write_tt_title(IFml, Atoms),
  generate(Atoms, V),
  tt(IFml, V, TV),
  write_tt_line(IFml, V, TV),
  fail.
create_tt(_).
```

`get_atoms(Fml,Atoms)` returns a sorted list of the atoms occurring in `Fml`. The assignments for this set of atoms are generated by `generate(Atoms,V)`. As each assignment is generated, `tt(Fml, V, TV)` is called, the value of `TV` is printed and then the predicate `fail` causes backtracking into `generate` in order to print the entire truth table.

3.2 Semantic tableaux

A tableau will be represented by a predicate `t(Fmls, Left, Right)`, where `Fmls` is a list of the formulas labeling the root of the tableau, and `Left` and `Right` are the subtrees of the root which recursively contain terms on the same predicate. `Right` is ignored for an α-rule. Here is the term for the tableau for $p \wedge (\neg q \vee \neg p)$.

```
t([p and (neg q or neg p)],
  t([p, neg q or neg p],
    t([p,neg q],open,empty),
    t([p,neg p],closed,empty)
  ),
  empty
).
```

The tableau for a formula `Fml` is created by starting with `t([Fml],_,_)` and then extending the tableau by instantiating the logical variables for the subtrees.

```
create_tableau(Fml, Tab) :-
  Tab = t([Fml], _, _),
  extend_tableau(Tab).
```

The predicate `extend_tableau` performs one step of the tableau construction. First, it checks for a pair of contradictory formulas (an optimization, we don't wait until there are only literals) in `Fmls`, and then it checks if `Fmls` contains only literals. Only then does it perform an alpha or a beta rule, with alpha rules given precedence. To check if a branch is open, check if all elements of the label are literals. To check if a branch is closed, a formula is chosen from the list of formulas labeling the node, and then a search is made for its complement. Backtracking will ensure that all possibilities are tried.

```
check_closed(Fmls) :-
  member(F, Fmls), member(neg F, Fmls).
```

To perform an α- or β-rule, we nondeterministically select a formula, pattern-match it against the database of rules, delete the formula from the node and add the subformulas. The rule for double negation is implemented separately.

3.3 Proof checker

Just as we wrote a program to *generate* a semantic tableau from a formula, it would be nice if we could write a program to generate a proof of a formula in $\mathscr{H}$. However, this is far from straightforward as quite a lot of ingenuity goes into producing a concise proof. In this section we present a *proof checker* for $\mathscr{H}$: a program which receives a list of formulas and their assumptions as its input, and checks if the list is a correct proof. It checks that each element of the list is either an axiom or assumption, or follows from previous elements by *MP* or deduction. The program writes out the justification of each element in the list.

The axioms are facts with the axiom number as an additional argument.

The data structure used is a list whose elements are of the form `deduce(A,F)`, where `A` is a list of formulas that is the current set of assumptions, and `F` is the formula that has been proved. The predicate `proof` has two additional arguments, a line number used on output and a list of the formulas proved so far.

```
proof(List) :- proof(List, 0, []).
```

Checking an axiom or an assumption is trivial and involves just checking the database of axioms or the list of assumptions.

To check if `A` can be justified by *MP*, the predicate `nth1` nondeterministically searches `SoFar` for a formula of the form `B imp A` and then for the formula `B`. The head of List is the Line'th line in the proof, so the N'th element of the list is the Line-N'th line.

```
proof([Fml | Tail], Line, SoFar) :-
  Line1 is Line + 1,
  Fml = deduce(_, A),
  nth1(L1, SoFar, deduce(_, B imp A)),
  nth1(L2, SoFar, deduce(_, B)),
  MP1 is Line1 - L1,
  MP2 is Line1 - L2,
  write_proof_line(Line1, Fml, ['MP ', MP1, ',', MP2]),
  proof(Tail, Line1, [Fml | SoFar]).
```

A formula can be justified by the deduction rule if it is an implication `A imp B`. Nondeterministically choose a formula from `SoFar` that has `B` as its formula, and check that `A` is in its list of assumptions. The formula `A` is deleted from `Assump`, the list of assumptions of `A imp B`.

3.4 Conversion to CNF

There are two programs: `cnfprop` for propositional logic and `cnffol` that contains the additions for first-order logic.

The program for conversion to CNF performs three steps one after another: elimination of operators other than negation, disjunction and conjunction, reducing the scope of negations using De Morgan's laws and double negation, and finally distribution of disjunction over conjunction.

Elimination of operators is done by a recursive traversal of the formation tree. The first three clauses eliminate `imp`, `eqv` and `neqv`. The next four clauses simply traverse the tree for the other operators.

The application of De Morgan's laws is similar. Two clauses apply the laws for negations of conjunction and disjunction and the next two clauses traverse the formula for non-negated formulas. The final two clauses eliminate double negation and terminate the recursion at a literal.

Distribution of disjunction over conjunction is more tricky, because one step of the distribution may produce additional structures that must be handled. For example, one step of distribution applied to $p \vee ((q \wedge r) \wedge r)$ gives $(p \vee (q \wedge r)) \wedge (p \vee r)$ and the distribution rule called recursively not just on the subformulas but also on the new formula that results.

The predicate `cnf_to_clausal` transforms from formulas as terms with operators to formulas as sets of sets. Both lists, clauses and sets of clauses, are converted are converted to sets to remove duplicates, and the literals in the clauses are sorted so that duplicate clauses can be identified.

3.5 Resolution

To perform resolution we first convert a CNF formula to clausal form. Trivial clauses like $\{p, r, \bar{p}\}$ are discarded. First, a check is made if the set of clauses is empty (the formula is valid) or if the set contains the empty clause (the formula is unsatisfiable). Resolution is performed by nondeterministically selecting two clauses in the set, creating their resolvent, adding it to the set of clauses and recursively calling the predicate. The predicate will fail and backtrack in three cases: there are no clashing literals, the resolvent is trivial, the resolvent already exists in the set.

```
resolve(S) :-
  member(C1, S),              % choose two clauses
  member(C2, S),
  clashing(C1, L1, C2, L2),   % check that they clash
  delete(C1, L1, C1P),        % delete the clashing literals
  delete(C2, L2, C2P),
  union(C1P, C2P, C),         % new clause is their union
  \+ clashing(C, _, C, _),    % don't add trivial clauses
  \+ member(C, S),            % don't add an existing clause
  write_clauses(S), nl,
  resolve([C | S]).           % add the resolvent to the set
```

3.6 Binary decision diagrams

Atoms are represented by integers: `N` stands for the atom p_N. BDDs are represented by the predicate `bdd(N,False,True)`, where `N` is the atom labeling the root, `False` is the sub-BDD when `N` is assigned F and `True` is the sub-BDD when `N` is assigned T.

The module `bddwrite` contains predicates for formatting a BDD.

3.6.1 Reduce

Calling the predicate `reduce(B, BR)` with a BDD `B` returns the reduced BDD in `BR`. It does a recursive traversal of the BDD. A cache of reduced BDDs is maintained as a dynamic database, so that it is easy to check if a BDD already exists as required by the second type of reduction. `retractall` should be called before executing `reduce` in order to initialize the database.

The first clause checks if the current BDD is in the cache; if so, it is returned.

```
reduce(B, B) :- B, !.
```

Next we check if the BDD is a leaf; if so, it is placed into the cache and returned. The third clause recurses on the sub-BDDs, but before returning it calls `remove` to perform the first type of reduction. If the two edges from this node `N` point to the same subBDD, `N` must be removed and one copy of the subBDD returned instead. Otherwise, the new BDD formed by `N` and the subBDDs is returned after storing in the cache.

```
remove(bdd(_, SubBDD, SubBDD), SubBDD) :- !.
remove(B, B) :- assert(B).
```

3.6.2 Apply

`apply(B1, Opr, B2, B` applies the operator `Opr` to the BDDs `B1` and `B2` and returns the result in `B`. A cache is used for optimization: the predicate `bdd_pair(B1, B2, B)` is asserted if applying the operator to the pair of of BDDs `B1` and `B2` and returns the result `B`. An additional optimization is to integrate `reduce` with `apply` instead of first creating an unreduced BDD.

```
apply1(B1, _, B2, Result) :-
  bdd_pair(B1, B2, Result), !.
apply1(B1, Opr, B2, Result1) :-
  create(B1, Opr, B2, Result),
  check_reduced(Result, Result1),
  assert(bdd_pair(B1, B2, Result1)).
```

The algorithm requires a simultaneous recursive traversal of two BDDs. The base case is if both BDDs are leaves; in this case, simply apply the operator to the values in the leaves. If the same atom is at the root of both BDDs, a simultaneous recursion is done and the resultant BDD constructed from the BDDs that are returned.

When one of the BDDs has an atom at the root and the other is a leaf, or when the roots are labeled with different atoms, the first clause is taken if the right-hand sub-BDD is a leaf or has a higher-numbered atom; in this case, the sub-BDDs of the left-hand node are applied to the *entire* right-hand node `Node2`. The two cases can be treated together, using the `;` operator which succeeds if either of its operand does. The check that `N1` is not a leaf is not needed by the algorithm; it just ensures that we don't try to evaluate `leaf<N2` which is illegal.

```
create(bdd(N1, False1, True1), Opr,     % True node is leaf or smaller
       Node2,
       bdd(N1, FalseResult, TrueResult)) :-
  Node2 = bdd(N2, _, _),
  (N2 = leaf ;
   (N1 \= leaf, N1 < N2)), !,          % Check N1\=leaf before "<"
  apply1(False1,  Opr, Node2, FalseResult),
  apply1(True1, Opr, Node2, TrueResult).
```

There is another, symmetrical, clause if the left-hand node is a leaf or has a higher-numbered atom.

Another optimization is to check for a *controlling operand* for the operator if one of the BDDs is a leaf. A value is controlling if the result of the operation does not depend on the other operand. T is controlling for $\vee$ and F is controlling for $\wedge$.

The restriction and quantification operations are also implemented.

4 DPLL algorithm

The DPLL algorithm dpll works on a set of clauses represented as a list of lists. After checking for the empty set of clauses and a set containing the empty clause, unit_propagate is called. If it succeeds, the result is recursively passed to dpll. If not, the user is prompted for a literal to be assigned true and the set of clauses partially evaluated for this assignment by the predicate evaluate. Both predicates use eliminate to delete clauses that contain the literal and delete_complement to delete occurrences of the complement of the literals.

The test program the four-queens problem given in the book. The predicate put_clauses displays the set of clauses with each clause on a separate row. If the literal entered is p12, the predicate terminates with the empty set of clauses and the assignments to positive literals give the answer, and, similarly, if the literal entered is p13. If the literals entered are p11 followed by p23, the predicate terminates with a set containing the empty clause, showing that the set is unsatisfiable.

5 First-order logic

5.1 Semantic tableaux

The predicate check_closed that checks if a node is closed must use the operator for syntactical identity == to prevent unification of atomic formulas.

```
check_closed(Fmls) :-
  member(F1, Fmls), member(neg F2, Fmls), F1 == F2.
```

To implement the systematic search, the rules are ordered so that rules for α-, β- and δ-formulas are performed before attempting the rule for a γ-formula. The tableau predicate has an extra argument C to hold the list of constants. This is updated whenever the rule for a δ-formula is used.

The rules for the α- and β-formulas are straightforward.

For a δ-formula, gensym generates a new constant symbol. instance(A1,A2,X,C) returns in A2 the instance of A1 that can be obtained by replacing the variable X by the constant C.

```
delta_rule(Fmls, [A2 | Fmls1], C) :-
  member(A, Fmls),
  delta(A, X, A1), !,
  gensym(a, C),
  instance(A1, A2, X, C),
  delete(Fmls, A, Fmls1).
```

For a γ-formula, first we have to identify if there exists a γ-formula in the set of formulas. Then, each constant is used to instantiate the γ-formula. The list of formulas is re-ordered: the instantiated formulas are placed at the head of the list so that non-γ-rules can be used if possible and the γ-formula is placed at the end of the list so that other γ-formulas will be used.

```
gamma_rule(Fmls, Fmls4, C) :-
  member(A, Fmls),
  is_gamma(A), !,               % Check if gamma rule
  gamma_all(C, A, AList),       % Apply gamma for all constants C
  delete(Fmls, A, Fmls1),       % Re-order: gamma(c), Fmls, gamma
  append(Fmls1, [A], Fmls2),    %   so that substitutions are taken
  append(AList, Fmls2, Fmls3),  %   and universal fml is taken last
  list_to_set(Fmls3, Fmls4).    % Remove duplicates introduced by gamma
```

`gamma_all(C, A, AList)` applies the predicate `gamma` to `A` with all of the constants in the list `C` and returns the list of formulas in `AList`. `gamma` recognizes the γ-formulas and returns instances using `instance(A, A1, X, C)` in module `instance`, where $A1$ is obtained from A by instantiating X by the constant C. To create an instance, the predicate `instance` recursively traverses the formula until an atomic formula is reached; then the substitution is performed. To implement substitution, the operator == must be used to prevent unification instead of substitution. `instance` is more complex than it needs to be here because it performs other tasks for proof checking.

5.2 Proof checker

The proof checker for the Hilbert system in propositional logic is extend to first-order logic by adding Axioms 4 and 5, and the generalization rule. Axiom 5 requires that the quantified variable not occur as a free variable in the antecedent.

```
axiom(all(X, A1) imp A2, 4) :-
  instance(A1, A2, X, _).
axiom(all(X, A imp B) imp (A imp all(X, B)), 5) :-
  \+ free_in(A, X).
```

Here, the predicate `instance(A, A1, X, C)`traverses the formulas `A` and `A1` *together*; when an atomic formula is reached, the arguments are compared to see if they are the same variablev or if one is a constant and the other the variable `X`. To check if a variable is free in a formula, simply traverse the formula and for every quantifier, check that the variable is different from the quantified variable.

An additional argument `Gens` is added to the predicate `proof` to store a list of the constants to which Generalization has been applied. When the deduction rule is used, two things must be checked: that the new set of assumptions is the same as the previous one without the formula `A` and the proviso that no constant of `A` appears in `Gens`. To check the proviso, the list of constants is traversed and a check is made that each one does not appear (free) in `A`.

```
proviso([], _).
proviso([C|Rest], A) :-
  \+ free_in(A,C),
  proviso(Rest,A).
```

5.3 Conversion to CNF

For first-order logic, there are additional predicates to rename the bound variables and then extract the quantifiers, which is easy to do once the variables have been renamed.

`rename` works by traversing the formula, keeping a list of variable substitutions. The call is `rename(A,List,List1,A1)`, where `A1` is `A` after the variables have been renamed, and `List` and `List1` are lists of pairs of variables. On the way down the recursive traverse, `List` stores all the variables that have been encountered and the new variable names. At the bottom, `List` is unified with the variable `List1` and the substitutions are made on the way up the recursive traverse. When a quantified variable is the same as one previously encountered, `copy_term` is used to create a new variable. This is a predicate that makes a copy of its first argument with a fresh variable and places it in the second argument.

```
rename(all(X, A), List, List1, all(Y, A1)) :-
  member_var((X, _), List), !,
  copy_term(X, Y),
  rename(A, [(X, Y) | List], List1, A1).
```

When a quantified variable is encountered for the first time, an identity substitution is created. The clauses for `ex` are similar and the clauses for the Boolean operators are elementary. The clause terminating the recursion performs the substitution on the atomic formulas using `subst_var`.

A simple transversal is not sufficient for extracting quantifiers. A simple traversal of the formula $(p_1 \vee \forall x q_1(x)) \wedge (p_2 \vee \forall y q_2(y))$ will give $\forall x(p_1 \vee q_1(x)) \wedge \forall y(p_2 \vee q_2(y))$, but the extraction has to be applied again to this formula. To do this, the result of a traversal is checked to see if the formula has changed and if so the traversal is done again.

5.4 Skolemization

The call to `skolem(A,A1)` first transforms the formula A into CNF and then calls `skolem(A, ListA, ListE,A1)` to obtain `A1`, the Skolemized version of `A`. `ListA` is the list of universally quantified variables that have appeared so far (initially, the empty list) and `ListE` is a list of pairs: the first element is an existentially quantified variable and the second element is itself a pair that contains the Skolem function and a list of its arguments that are to be substituted for the existential variable. `gensym` is used to create new Skolem function symbols. Here are the clauses for quantified formulas.

```
skolem(all(X, A), ListA, ListE, all(X, B)) :- !,
  skolem(A, [X | ListA], ListE, B).

skolem(ex(X, A), ListA, ListE, B) :- !,
  gensym(f, F),
  Function =.. [F | ListA],
  skolem(A, ListA, [(X, Function) | ListE], B).
```

The clauses for Boolean formulas are elementary. When an atomic proposition is encountered in the recursive traversal, the operator `=..` (read *univ*) is used to decompose the formula into a predicate symbol and a list of variables. Then, `subst_var` is called to replace existentially quantified variables by the Skolem functions, and `=..` is called again to recompose the formula.

5.5 Unification

To unify a pair of atomic propositions, check that the predicate symbols are identical, create a set of equations from the arguments and call `solve`.

```
unify(A1, A2, Subst) :-
  A1 =.. [Pred | Args1],
  A2 =.. [Pred | Args2],
  create_equations(Args1, Args2, Eq),
  solve(Eq, Subst).
```

The predicate `solve` is called with a list of equations and returns a list of substitutions written as equations $x = t$, where x is a variable and t is a term. The list is traversed, attempting to apply each of the rules to the current equation. It is convenient to maintain the list in two parts, one to the left of the current equation and one that includes the current equation as its head and the rest of the equations as its tail. The `solve` predicate will have four parameters: two for the equation list, a third for status information and the fourth will return the solved set.

The status is passed down the recursive calls to `solve` and is used to terminate the recursion as necessary, for example, if either rule 3 or rule 4 fails. The equation that caused the failure is returned together with the failure indication for printing. Each rule is applied in turn; if successful, it sets the status to `modified` as an indication to the list traversal clauses described below.

```
solve(Head, [Current | Tail], _, Result) :-
  rule1(Current, Current1), !,
  solve(Head, [Current1 | Tail], modified, Result).

solve(Head, [Current | Tail], _, Result) :-
  rule2(Current), !,
  solve(Head, Tail, modified, Result).

solve(Head, [Current | Tail], _, Result) :-
  rule3(Current, NewList, Status), !,
  append(NewList, Tail, NewTail),
  solve(Head, NewTail, Status, Result).
```

The set of equations returned by rule 3 replaces the current equation and is appended in front of the remaining equations.

When a substitution is performed in rule 4, it must be performed on *all* the equations, including those that have already been checked.

```
solve(Head, [Current | Tail], _, Result) :-
  append(Head, Tail, List),
  rule4(Current, List, NewList, Status), !,
  solve([Current], NewList, Status, Result).
```

The next three clauses of `solve` traverse the list. If no equation applies, the traverse goes to the next one. When the end of the list is reached, another traversal is initiated if the previous one made any modifications to the list.

In the rules, we must prevent confusion between the equality operator of the term equation and the Prolog equality operator, so the former is explicitly defined using the operator `eq`. Rules 1 and 2 are straightforward. Rule 3 compares the outermost functors and fails if they are not the same. Otherwise, it calls `new_equations` to pair the subterms. Rule 4 reports failure if the occurs-check fails. Otherwise, it calls `subst_list` to perform the substitutions. If nothing is changed, the predicate fails, initiating traversal to the next equation in the list.

`occur(X,T)` traverses the list and succeeds as soon as it finds an occurrence of the variable `X` in the term `T`. `occur_list` is used to check the list of subterms.

5.6 Resolution

A formula is first transformed into a set of clauses using `skolem` and `skolem_to_clauses` and then resolution is performed.

The empty set of clauses is valid and a set containing the empty clause is unsatisfiable. Otherwise, choose two *different* clauses and resolve. `copy_term` standardizes apart. After resolving a check is made that the clause is not trivial and that it is a new clause.

```
resolve(S)  :-
  member(C1, S),
  member(C2, S),
  C1 \== C2,
  copy_term(C2, C2_R),
  clashing(C1, L1, C2_R, L2, Subst),
  delete_lit(C1, L1, Subst, C1P),
  delete_lit(C2_R, L2, Subst, C2P),
  clause_union(C1P, C2P, Resolvent),
  \+ clashing(Resolvent, _, Resolvent, _, _),
  \+ member(Resolvent, S),
  resolve([Resolvent | S]).
```

The predicate `clashing(C1, L1, C2, L2, Subst)` checks if the clauses clash and if so it returns the clashing literals and the mgu that unifies them. `delete_lit(Clause, Literal, Subst, Result)` deletes from `Clause` all the literals that are equal to `Literal` under the substitution `Subst` and returns the `Result`. `clause_union(C1, C2, Result)` takes the union of the literals in the two clauses to form the resolvent.

6 Temporal logic

6.1 Semantic tableaux

The decision predicate for satisfiability is implemented in four stages:

- extend_tableau performs the tableau construction until it terminates.
- check_tableau decides if the tableau is opened, closed or contains cycles.
- create_states constructs the state diagram from the tableau.
- check_fulfillment constructs the component graph and checks fulfillment.

Each node of the tableau contains five fields t(Fmls, Left, Right, N, Path): the list of formulas and the links to the left and right children. A node number that is generated by get_num when a new node is created and the ancestor path. This is used to check if a new state should be created or if a node should be connected to an ancestor. α- and β-nodes add themselves to the ancestor path by appending the term pt(Fmls,N) to Path, where Fmls is the label and N is the node number. The rule for a β-formula is:

```
extend_tableau(t(Fmls, Left,   Right, N, Path)) :-
  beta_rule(Fmls, Fmls1, Fmls2), !,
  get_num(N1),
  get_num(N2),
  Left  = t(Fmls1, _, _, N1, [pt(Fmls,N)|Path]),
  Right = t(Fmls2, _, _, N2, [pt(Fmls,N)|Path]),
  extend_tableau(Left),
  extend_tableau(Right).
```

The predicate next_rule is called to get the formulas in a node created by the rule for a next formula, but before the node is created, search is called to search the Path for a node with the same set of formulas in its label. If successful, it returns the node number N; this branch of the tableau is terminated and marked connect(N). Otherwise, a new node is created.

```
extend_tableau(t(Fmls, connect(N), empty, _, Path)) :-
  next_rule(Fmls, Fmls1),
  search(Path, Fmls1, N), !.

extend_tableau(t(Fmls, Left,   empty, N, Path)) :-
  next_rule(Fmls, Fmls1), !,
  get_num(N1),
  Left = t(Fmls1, _, _, N1, [pt(Fmls,N)|Path]),
  extend_tableau(Left).
```

After the tableau construction terminates, `check_tableau` is called. It traverses the tableau from the root down to the leaves and then returns back up, computing the status of the tableau: open, closed or with a cycle that must be checked for fulfillment.

`create_states` takes a tableau and constructs the structure: states, state paths which are transitions, and state labels which are the union of the labels on the state paths. It returns a list of terms `st(Fmls, N)`, where `Fmls` is the state label and `N` the node number of the state, and a list of terms `tau(From, To)`, where `From` and `To` are the node numbers of states.

These lists are the input to `component_graph`, which returns a list of MSCCs (a MSCC is a list of its states) and a list of edges of the form `e(From,To)`, where `From` and `To` are MSCCs. `fulfil1` selects a MSCC `S` with no outgoing edges and calls `self-fulfil` to check if `S` is self-fulfilling. If successful, it returns `ok(S)`; if not, it deletes `S` from the list of MSCCs, adds `notok(S,Result)` to the list of results and calls itself recursively. `Result` is the future formula that could not be fulfilled in `S`. `self-fulfil` checks each future formula such as `<>F` to see if `F` occurs in some state in the SCC.

A List of files

For each program `p.pro`, there is a file `p-t.pro` which contains test programs. The programs use the predicate `file_search_path` to access the modules in the `common` directory.

Directory `common`	
`ops.pro`	declaration of operators with precedence and associativity.
`def.pro`	correspondence between symbols and internal operators.
	semantic definition of Boolean operators.
`intext.pro`	conversion from external to internal format and conversely.
`io.pro`	display predicates for all programs (except BDDs).
Directory `prop`	(propositional logic)
`tt.pro`	truth tables.
`cnfpro.pro`	conversion of a formula to CNF
`tabl.pro`	semantic tableaux.
`check.pro`	Hilbert proof checker.
`resolv.pro`	resolution.
`bdd.pro`	BDD algorithms.
`bddwrite.pro`	display of BDDs.
`dpll.pro`	DPLL algorithm.
Directory `fol`	(first-order logic)
`cnffol.pro`	conversion of a formula to CNF
`tabl.pro`	semantic tableaux.
`check.pro`	Hilbert proof checker.
`skolem.pro`	skolemize a formula.
`unify.pro`	unification algorithm.
`resolv.pro`	resolution.
`instance.pro`	create an instance by substitution.
Directory `tl`	(temporal logic)
`tl.pro`	semantic tableaux.