Lecture notes 19.0

First-order logic: clausal form, skolemization

COMP 2411, session1, 2003

Computing by proving

The basic idea behind Logic programming is that *computing can be performed by proving*.

For instance, from the set of formulas
$\varphi = \forall \mathrm{x} \forall \mathrm{y}((\mathrm{mother}(\mathrm{x}) \wedge \mathrm{child}(\mathrm{y}, \mathrm{x})) \rightarrow \mathrm{loves}(\mathrm{x}, \mathrm{y}))$,
$\psi = \mathrm{mother}(\mathrm{mary})$ and $\xi = \mathrm{child}(\mathrm{tom}, \mathrm{mary})$, it is possible to compute whom Mary loves.

This is achieved as follows:

- Prove that $\exists \mathrm{x}\ \mathrm{loves}(\mathrm{mary}, \mathrm{x})$ is a logical consequence of $\{\varphi, \psi, \xi\}$ (which is indeed the case).
- If in this proof, a (stronger) statement of the form $\mathrm{loves}(\mathrm{mary}, \mathit{term})$ is being derived—it is also a logical consequence of $\{\varphi, \psi, \xi\}$—and term (here $\mathit{term} = \mathrm{tom}$) is a result of the computation.

Constructive proofs

To perform the proof, we can use one of the proof systems we know, for instance Natural deduction.

Writing "m" for "mary" and "t" for "tom", we can get:

$$\dfrac{\dfrac{\dfrac{\forall \mathrm{x} \forall \mathrm{y}((\mathrm{mother}(\mathrm{x}) \wedge \mathrm{child}(\mathrm{y}, \mathrm{x})) \rightarrow \mathrm{loves}(\mathrm{x}, \mathrm{y}))}{\forall \mathrm{y}((\mathrm{mother}(\mathrm{m}) \wedge \mathrm{child}(\mathrm{y}, \mathrm{m})) \rightarrow \mathrm{loves}(\mathrm{m}, \mathrm{y}))}\,\forall E}{(\mathrm{mother}(\mathrm{m}) \wedge \mathrm{child}(\mathrm{t}, \mathrm{m})) \rightarrow \mathrm{loves}(\mathrm{m}, \mathrm{t})}\,\forall E \quad \dfrac{\mathrm{mother}(\mathrm{m})\ \ \mathrm{child}(\mathrm{t}, \mathrm{m})}{\mathrm{mother}(\mathrm{m}) \wedge \mathrm{child}(\mathrm{t}, \mathrm{m})}\,\wedge I}{\mathrm{loves}(\mathrm{m}, \mathrm{t})}\,\rightarrow E}{\exists \mathrm{x}\ \mathrm{loves}(\mathrm{m}, \mathrm{x})}\,\exists I$$

Note that this proof is actually a proof in intuitionistic logic: it does not use RAA. It is a constructive proof.

Nonconstructive proofs

The proof that some closed formula that starts with $\exists$ is a logical consequence of a given set X of formulas cannot always be constructive.

For instance, the following proof that $\{\neg\forall x\varphi\} \models \exists x\neg\varphi$ uses RAA, and no alternative proof could avoid using RAA (that is clear intuitively: how could we come up with any term t such that $\varphi[t/x]$ if we only know that $\neg\forall x\varphi$?)

$$\dfrac{\dfrac{\dfrac{[\neg\varphi]^1}{\exists x\neg\varphi}\,\exists I \qquad [\neg\exists x\neg\varphi]^2}{\bot}\,\neg E}{\varphi}\,\mathrm{RAA}_1}{\forall x\varphi}\,\forall I \qquad \neg\forall x\varphi}{\bot}\,\neg E}{\exists x\neg\varphi}\,\mathrm{RAA}_2$$

Finding witnesses

So the key question is: when is a closed formula of the form $\exists x \chi$ a logical consequence of a set of formulas X iff there exists a term t—a witness for $\exists x \chi$—such that $\chi[t/x]$ is a logical consequence of X?

The answer is: iff there exists a constructive proof (a proof in Natural deduction without RAA) of $\exists x \chi$ from X. Moreover, a formula of the form $\chi[t/x]$ will necessarily occur in such a proof.

So we might conclude: if our aim is to compute by proving, let us just study intuitionistic logic rather then classical logic!

What we will do instead is put some syntactic restrictions on X and χ. We will require that X is a set of definite clauses and that $\exists x \chi$ is a definite query (to be defined).

Equivalent logics

If X is a set of definite clauses and φ is a definite query, then the following are equivalent.

- There exists a proof of φ from X in classical logic.
- There exists a proof of φ from X in intuitionistic logic.

So in this particular case, we get the best of both worlds! Indeed, we like the semantics of classical logic, that is close to our intuition (we have not studied and will not study the more complex semantics of intuitionistic logic); but we also like constructive proofs.

Of course, there is a price to pay, namely, to give up some of the expressive power of full first-order logic.

There is another benefit in the above restriction: rather than using Natural deduction without RAA, we can use a more efficient proof procedure, SLD-Resolution.

Clauses, resolution, skolemization

In order to 'glue' first-order logic and Logic programming as tightly as possible, we do not directly jump into the realm of Logic programming, but get close to it by defining more general notions than definite clauses and SLD-resolution: we will first study clauses and resolution.

Though clauses are also syntactically very restrictive, in some sense they still retain the full generality of first-order logic thanks to a syntactic transformation called skolemization that we will also study.

Clauses (1)

A clause is any formula of the form $\varphi_1 \vee \ldots \vee \varphi_n$, where for all $i \leq n$, φ_i is either an atomic formula—an atom, a positive literal—or the negation of an atomic formula—a negative literal.

A clause is also represented by the set of its disjuncts. For instance:

- $p(f(y)) \vee \neg p(g(z)) \vee q(z, y)$ is a clause, also represented as $\{p(f(y)), \neg p(g(z)), q(z, y)\}$;
- $\neg q(z, f(y)) \vee \neg p(g(z)) \vee q(y, y)$ is a clause, also represented as $\{\neg q(z, f(y)), \neg p(g(z)), q(y, y)\}$.

The set representation also gives the empty clause, that does not correspond to any disjunction of literals, but to a contradiction.

Clauses (2)

Remember that by convention, the truth of a clause is equivalent to the truth of its universal closure.

A universal formula, of the form $\forall x_1 \ldots \forall x_n \varphi$ where φ does not contain any occurrence of quantifier, is logically equivalent to a set of clauses: it suffices to put φ in conjunctive normal form: each conjunct is a clause.

More general formulas are not logically equivalent to a set of clauses.

There still exists a relationship between an arbitrary first-order formula and a set of clauses. To find out the nature the nature of the relationship, and which set of clauses is involved, we need to define skolemization.

Skolemization (1)

Skolemization is a technique based on *enrichments* of the vocabulary, that removes existential quantifiers in formulas in prenex form and preserves the notion of satisfiability.

Suppose that we are working with a vocabulary V consisting of the unary predicate symbol p.

The formula $\varphi = \exists x\, p(x)$ is satisfiable: indeed, the structure $\mathfrak{M}$ over V whose domain is $\mathbb{N}$ and such that $p^{\mathfrak{M}} = \{0, 2, 4, 6 \ldots\}$ is a model of φ. (The meaning of φ in $\mathfrak{M}$ is that some number is even.)

Note that V contains no function symbol (in particular, no constant). Let us enrich V with a constant c, which yields a new vocabulary V'.

Skolemization (2)

Then $p(c)$ is satisfiable: indeed, the structure $\mathfrak{N}$ over V' whose domain is $\mathbb{N}$, such that $p^{\mathfrak{N}} = \{0, 2, 4, 6 \ldots\}$, and such that $c^{\mathfrak{N}} = 4$, is a model of φ.
(Of course, $c^{\mathfrak{N}} = 0$, or $c^{\mathfrak{N}} = 12$, would do as well.)

At this stage we might think: let us just consider vocabularies that contain at least one constant, and we will not have to bother using this notion of enrichment.

So let us start again. Suppose that V consists of the unary predicate symbol p and the constant a. As before, $\exists x\, p(x)$ is satisfiable.

Moreover, the formula $p(a)$ is also satisfiable: indeed, the structure $\mathfrak{M}$ over V whose domain is $\mathbb{N}$, such that $p^{\mathfrak{M}} = \{0, 2, 4, 6 \ldots\}$, and such that $a^{\mathfrak{M}} = 4$, is a model of $p(a)$.

Skolemization (3)

But consider the formula $\psi = \neg p(a) \wedge \exists x\, p(x)$. It is satisfiable: indeed, the structure $\mathfrak{N}$ over V whose domain is $\mathbb{N}$, such that $p^{\mathfrak{N}} = \{0, 2, 4, 6 \ldots\}$, and such that $a^{\mathfrak{N}} = 1$, is a model of $\neg p(a) \wedge \exists X p(X)$.

Obviously, $\neg p(a) \wedge p(a)$ is not satisfiable.

Let us enrich V with a constant c, which yields a new vocabulary V'.

Then $\neg p(a) \wedge p(c)$ is satisfiable: indeed, the structure $\mathfrak{S}$ over V' whose domain is $\mathbb{N}$, such that $p^{\mathfrak{S}} = \{0, 2, 4, 6 \ldots\}$, and such that $a^{\mathfrak{S}} = 1$ and $c^{\mathfrak{S}} = 4$, is a model of $\neg p(a) \wedge p(c)$.

Skolemization (4)

Generalizing the argument, we obtain the following result.

Let φ be a formula over a vocabulary V. Set $V' = V \cup \{c\}$ where c is a constant that does not occur in V.

Then $\exists x\varphi$ is satisfiable (i.e., some structure over V is a model of $\exists x\varphi$) if and only if $\varphi[c/x]$ is satisfiable (ie. some structure over V' is a model of $\varphi[c/x]$).

Even if V contains infinitely many constants, enriching V with a *new* constant is necessary to make the previous result hold for any formula over V of the form $\exists x\varphi$.

Skolemization (5)

So now we know how to remove existential quantifiers that occur in front of a formula, using extra constants. But what about the other existential quantifiers ?

Consider the formula $\varphi = \forall x \exists y(q(x,y) \wedge \neg q(x,x))$. If we remove $\exists y$ and replace all other occurrences of y by a constant c we get: $\psi = \forall x(q(x,c) \wedge \neg q(x,x))$.

But $\psi \models q(c,c) \wedge \neg q(c,c)$, which is not satisfiable.

Still φ is satisfiable: indeed, the the structure $\mathfrak{M}$ over V whose domain is $\mathbb{N}$, such that $q^{\mathfrak{M}} = \{(0,1),(1,2),(2,3),(3,4)\ldots\}$ is a model of φ.

Skolemization (6)

Let us enrich V with a unary function symbol f, which yields a new vocabulary V'.

Then $\chi = \forall x(q(x, f(x)) \wedge \neg q(x,x))$ is satisfiable: indeed, the structure $\mathfrak{N}$ over V' whose domain is $\mathbb{N}$, such that $q^{\mathfrak{N}} = \{(0,1),(1,2),(2,3),(3,4)\ldots\}$, and such that $f^{\mathfrak{N}}(n) = n+1$ for all $n \in \mathbb{N}$, is a model of χ.

Skolemization generalizes the previous argument.

For each existential quantifier occurring in a closed formula in prenex form, we count the number n of universal quantifiers that occur before this existential quantifier.

We then enrich the vocabulary with a new n-ary function symbol, and modify the formula accordingly.

Skolemization (7)

The Skolemization technique is justified by the following result, that generalizes the previous examples.

Proposition 1: Let $\forall x_1 \ldots \forall x_n \exists x\varphi$ be a formula over a vocabulary V. Set $V' = V \cup \{f\}$ where f is an n-ary function symbol that does not occur in V.

Then $\forall x_1 \ldots \forall x_n \exists x\varphi$ is satisfiable
(i.e., some structure over V is a model of $\forall x_1 \ldots \forall x_n \exists x\varphi$)
if and only if $\forall x_1 \ldots \forall x_n \varphi[f(x_1,\ldots,x_n)/x]$ is satisfiable
(i.e., some structure over V' is a model of $\forall x_1 \ldots \forall x_n \varphi[f(x_1,\ldots,x_n)/x]$).

Note that the result on slide 11 is the particular case where $n = 0$.

Example

Let us Skolemize
$\varphi = \forall x(\forall y \exists z\, p(x,y,z) \wedge \exists z \forall y \exists w \neg q(x,y,z,w))$.

First, φ has to be transformed into a formula in prenex form:
$\varphi \equiv \forall x(\forall y \exists z\, p(x,y,z) \wedge \exists u \forall v \exists w \neg q(x,u,v,w))$
$\equiv \forall x \forall y \exists z \exists u \forall v \exists w (p(x,y,z) \wedge \neg q(x,u,v,w))$.

Then we enrich the language with 3 function symbols: $f/2$, $g/2$ and $h/3$.

By Skolemization, φ is satisfiable iff

$$\forall x \forall y \forall v (p(x,y,f(x,y)) \wedge \neg q(x,g(x,y),v,h(x,y,v)))$$

is satisfiable.

Back to clauses

It follows from the previous that for any first-order formula φ, there exists a (finite) set X of clauses in an enriched language such that φ is satisfiable iff X is satisfiable. Indeed:

- We can put φ in prenex form and get a formula ψ.
- Using Skolemization, we can remove the existential quantifiers in ψ and get a formula of the form $\forall x_1 \ldots \forall x_n \chi$, where χ is a quantifier free formula.
- We can put χ in conjunctive normal form and get a set X of clauses.
- Then φ is satisfiable iff X is satisfiable.

We examine how to perform all these transformations in Prolog, starting from a closed formula φ.

Eliminate $\leftrightarrow$ and $\oplus$ (1)

To put all quantifiers in front, we first need to eliminate $\leftrightarrow$ (and $\oplus$ in case we include it in the language as a commodity).

Indeed, we have rules to pull out quantifiers for formulas built from $\neg$, $\vee$, $\wedge$ or $\rightarrow$, but not for formulas built from $\leftrightarrow$ or $\oplus$.

We just use the equivalence between:

- $\varphi \leftrightarrow \psi$ and $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$;
- $\varphi \oplus \psi$ and $(\varphi \wedge \neg\psi) \vee (\psi \wedge \neg\varphi)$.

For formulas of a different form, we apply the transformation recursively to the arguments of the boolean operator or quantifier applied last.

Eliminate $\leftrightarrow$ and $\oplus$ (2)

```
eliminate_equiv_xor(A eqv B, (A1 imp B1) and (B1 imp A1)) :- !,
        eliminate_equiv_xor(A, A1),
        eliminate_equiv_xor(B, B1).
eliminate_equiv_xor(A xor B, (A1 and neg B1) or (neg A1 and B1)) :- !,
        eliminate_equiv_xor(A, A1),
        eliminate_equiv_xor(B, B1).
eliminate_equiv_xor(neg A, neg A1) :- !,
        eliminate_equiv_xor(A, A1).
eliminate_equiv_xor(A or B, A1 or B1) :- !,
        eliminate_equiv_xor(A, A1),
        eliminate_equiv_xor(B, B1).
eliminate_equiv_xor(A and B, A1 and B1) :- !,
        eliminate_equiv_xor(A, A1),
        eliminate_equiv_xor(B, B1).
eliminate_equiv_xor(exists X sep A, exists X sep A1) :- !,
        eliminate_equiv_xor(A, A1).
eliminate_equiv_xor(forall X sep A, forall X sep A1) :- !,
        eliminate_equiv_xor(A, A1).
eliminate_equiv_xor(A, A).
```

Rename variables (1)

To pull out quantifiers, we need to make sure that different quantifiers apply to different variables.

For instance, $\exists x(p(x) \wedge \forall x q(x,x))$ is obviously equivalent to $\exists x \forall y(p(x) \wedge q(y,y))$, but not to $\exists x \forall x(p(x) \wedge q(x,x))$ (which note is equivalent to $\forall x(p(x) \wedge q(x,x))$.

To make things simpler, we do not look for variables to which at least two quantifiers are applied, but we just rename all quantified variables as `x1`, `x2`, `x3`,..., using the convenient `gensym` built-in.

As a precondition, we assume that our original formula does not contain any occurrence of `x1`, `x2`, `x3`,...

Basically, with a formula of the form $\exists v \varphi$ or $\forall v \varphi$, we have to replace in φ all free occurrences of v by xi, for the i generated by `gensym`, and put $\exists xi$ or $\forall xi$ in front.

Rename variables (2)

```
rename_variables(Formula, RenamedFormula) :-
        reset_gensym(x),
        rename_variables1(Formula, RenamedFormula).

rename_variables1(exists X sep FormX, exists V sep Form) :- !,
        gensym(x, V),
        instance(X, V, FormX, FormV),
        rename_variables1(FormV, Form).
rename_variables1(forall X sep FormX, forall V sep Form) :- !,
        gensym(x, V),
        instance(X, V, FormX, FormV),
        rename_variables1(FormV, Form).
rename_variables1(Expression, RenamedExpression) :-
        compound(Expression), !,
        Expression =.. [Oper| Args],
        maplist(rename_variables1, Args, Args1),
        RenamedExpression =.. [Oper| Args1].
rename_variables1(ConstantOrVariable, ConstantOrVariable).
```


Rename variables (3)

```
instance(Variable, Term, Variable, Term) :- !.
instance(X, _, exists X sep Form, exists X sep Form) :- !.
instance(X, Term, exists Y sep Form, exists Y sep Form1) :- !,
        instance(X, Term, Form, Form1).
instance(X, _, forall X sep Form, forall X sep Form) :- !.
instance(X, Term, forall Y sep Form, forall Y sep Form1) :- !,
        instance(X, Term, Form, Form1).
instance(X, Term, Expression, Instance) :-
        compound(Expression),
        Expression \= '$VAR'(_), !,
        Expression =..  [Oper| Args],
        maplist(instance(X, Term), Args, Args1),
        Instance =.. [Oper| Args1].
instance(_, _, Atom, Atom).
```


Put into prenex form (1)

We are now ready to put the formula into prenex form, *i.e.*, pull out all quantifiers:

```
prenex(Formula, PrenexFormula) :-
        eliminate_equiv_xor(Formula, Formula1),
        rename_variables(Formula1, Formula2),
        prenex1(Formula2, PrenexFormula).
```

using equivalences like:

- $\forall x \psi \wedge \xi \equiv \forall x(\psi \wedge \xi)$ when x does not occur in ξ;
- $\forall x \psi \rightarrow \xi \equiv \exists x(\psi \wedge \xi)$ when x does not occur in ξ;
- $\psi \rightarrow \forall x \xi \equiv \forall x(\psi \wedge \xi)$ when x does not occur in ψ;
- Etc.

all the conditions to the right being guaranteed by the fact that we started from a closed formula, and that we have renamed all variables.

Put into prenex form (2)

```
prenex1(exists X sep A, exists X sep A1) :- !,
        prenex1(A, A1).
prenex1(forall X sep A, forall X sep A1) :- !,
        prenex1(A, A1).
prenex1(neg exists X sep A, forall X sep C) :-
        prenex1(neg A, C).
prenex1(neg forall X sep A, exists X sep C) :-
        prenex1(neg A, C).
prenex1(exists X sep A or B, exists X sep C) :- !,
        prenex1(A or B, C).
prenex1(forall X sep A or B, forall X sep C) :- !,
        prenex1(A or B, C).
prenex1(exists X sep A and B, exists X sep C) :- !,
        prenex1(A and B, C).
prenex1(forall X sep A and B, forall X sep C) :- !,
        prenex1(A and B, C).
prenex1(exists X sep A imp B, forall X sep C) :- !,
        prenex1(A imp B, C).
prenex1(forall X sep A imp B, exists X sep C) :- !,
        prenex1(A imp B, C).
```


Put into prenex form (3)

```
prenex1(A or exists X sep B, exists X sep C) :- !,
        prenex1(A or B, C).
prenex1(A or forall X sep B, forall X sep C) :- !,
        prenex1(A or B, C).
prenex1(A and exists X sep B, exists X sep C) :- !,
        prenex1(A and B, C).
prenex1(A and forall X sep B, forall X sep C) :- !,
        prenex1(A and B, C).
prenex1(A imp exists X sep B, exists X sep C) :- !,
        prenex1(A imp B, C).
prenex1(A imp forall X sep B, forall X sep C) :- !,
        prenex1(A imp B, C).
```

For the remaining cases, we have to be careful: there might be quantifiers deeper in the formula, in which case we have to pull out the quantifiers in these subformulas, and then pull out the quantifiers again working on the original formula with the previous subformulas being replaced by their prenex form equivalent.

Put into prenex form (4)

```
prenex1(neg A, C) :-
        (   contains_quantifiers(A), !,
            prenex1(A, A1),
            prenex1(neg A1, C)
        ;   C = neg A
        ).
prenex1(A or B, C) :-
        (   contains_quantifiers(A), !,
            prenex1(A, A1),
            prenex1(A1 or B, C)
        ;   contains_quantifiers(B), !,
            prenex1(B, B1),
            prenex1(A or B1, C)
        ;   C = A or B
        ).
```


Put into prenex form (5)

```
prenex1(A and B, C) :-
        (   contains_quantifiers(A), !,
            prenex1(A, A1),
            prenex1(A1 and B, C)
        ;   contains_quantifiers(B), !,
            prenex1(B, B1),
            prenex1(A and B1, C)
        ;   C = A and B
        ).
prenex1(A imp B, C) :-
        (   contains_quantifiers(A), !,
            prenex1(A, A1),
            prenex1(A1 imp B, C)
        ;   contains_quantifiers(B), !,
            prenex1(B, B1),
            prenex1(A imp B1, C)
        ;   C = A imp B
        ).
prenex1(A, A).
```

Put into prenex form (6)

The implementation of `contains_quantifiers` is trivial:

```
contains_quantifiers(_ sep _) :- !.
contains_quantifiers(neg A) :- !,
        contains_quantifiers(A).
contains_quantifiers(A or B) :- !,
        (   contains_quantifiers(A)
        ;   contains_quantifiers(B)
        ).
contains_quantifiers(A and B) :- !,
        (   contains_quantifiers(A)
        ;   contains_quantifiers(B)
        ).
contains_quantifiers(A imp B) :- !,
        (   contains_quantifiers(A)
        ;   contains_quantifiers(B)
        ).
```


Skolemize (1)

Now that we have a formula in prenex form, we can remove the existential quantifiers and replace an existentially quantified variable x by a term of the form $f(x_1, \ldots, x_x)$ where $\{x_1, \ldots, x_n\}$ is the set of all variables such that $\forall x_1, \ldots \forall x_n$ occur in φ before $\exists x$.

So we need to:

- keep track of the universally quantified variables read so far—they will be stored in the second argument of `skolemize/4`;
- state that we want to substitute x by $f(x_1, \ldots, x_n)$—this will be stored in the third argument of `skolemize/4`, and represents a Skolem mapping.

Skolemize (2)

Since f has to be a new function symbol (in the enriched language), we use `gensym` again, and generate `f1`, `f2`, `f3`,... to be used for f. Hence as a precondition, we assume that our original formula does not contain any occurrence of `f1`, `f2`, `f3`,...

Once the Skolem mapping has been built, we can perform the substitution.

```
skolemize(PrenexFormula, SkolemizedFormula) :-
        reset_gensym(f),
        skolemize(PrenexFormula, [], [], SkolemizedFormula).
```


Skolemize (3)

```
skolemize(forall X sep A, UnivVars, SkolMapping, forall X sep B) :- !,
        skolemize(A, [X| UnivVars], SkolMapping, B).
skolemize(exists X sep A, UnivVars, SkolMapping, B) :- !,
        gensym(f, SkolFunctionSymbol),
        SkolFunction =.. [SkolFunctionSymbol| UnivVars],
        skolemize(A, UnivVars, [(X, SkolFunction)| SkolMapping], B).
skolemize(Formula, _, [(X, SkolFunction)| SkolMapping], SkolFormula) :-
        instance(X, SkolFunction, Formula, Instance),
        skolemize(Instance, _, SkolMapping, SkolFormula).
skolemize(Formula, _, [], Formula).
```

Put the matrix into CNF (1)

Having a universal formula, we have to put the matrix (that part of the formula that follows the sequence of universal quantifiers) into conjunctive normal form. This requires:

- eliminating implication;
- applying de Morgan's laws so that negation only applies to atoms;
- distributing disjunction over conjunction;
- using the associativity rules of conjunction and disjunction to have a nice printout without parentheses—in case we want such a printout. . .—, *e.g.*, get $\psi_1 \vee (\psi_2 \vee \psi_3)$ preferably to $(\psi_1 \vee \psi_2) \vee \psi_3$, since the former will be printed out as $\psi_1 \vee \psi_2 \vee \psi_3$.

Put the matrix into CNF (2)

```
cnf(A, A4) :-
        eliminate_imp(A, A1),
        de_morgan(A1, A2),
        distribute_or_over_and(A2, A3),
        right_associate(A3, A4).

eliminate_imp(A imp B, neg A1 or B1) :- !,
        eliminate_imp(A, A1),
        eliminate_imp(B, B1).
eliminate_imp(neg A, neg A1) :- !,
        eliminate_imp(A, A1).
eliminate_imp(A or B, A1 or B1) :- !,
        eliminate_imp(A, A1),
        eliminate_imp(B, B1).
eliminate_imp(A and B, A1 and B1) :- !,
        eliminate_imp(A, A1),
        eliminate_imp(B, B1).
```


Put the matrix into CNF (3)

```
de_morgan(neg neg A, A1) :- !,
        de_morgan(A, A1).
de_morgan(neg (A or B), A1 and B1) :- !,
        de_morgan(neg A, A1),
        de_morgan(neg B, B1).
de_morgan(neg (A and B), A1 or B1) :- !,
        de_morgan(neg A, A1),
        de_morgan(neg B, B1).
de_morgan(A or B, A1 or B1) :- !,
        de_morgan(A, A1),
        de_morgan(B, B1).
de_morgan(A and B, A1 and B1) :- !,
        de_morgan(A, A1),
        de_morgan(B, B1).
de_morgan(A, A).
```


Put the matrix into CNF (4)

```
distribute_or_over_and(A and B, A1 and B1) :- !,
        distribute_or_over_and(A, A1),
        distribute_or_over_and(B, B1).
distribute_or_over_and(A or B and C, ABC) :- !,
        distribute_or_over_and(A or B, AB),
        distribute_or_over_and(A or C, AC),
        distribute_or_over_and(AB and AC, ABC).
distribute_or_over_and(A and B or C, ABC) :- !,
        distribute_or_over_and(A or C, AC),
        distribute_or_over_and(B or C, BC),
        distribute_or_over_and(AC and BC, ABC).
distribute_or_over_and(A or B, C) :- !,
        (   contains_and(A or B), !,
            distribute_or_over_and(A, A1),
            distribute_or_over_and(B, B1),
            distribute_or_over_and(A1 or B1, C)
        ;   C = A or B
        ).
distribute_or_over_and(A, A).
```

Put the matrix into CNF (5)

```
right_associate((A and B) and C, F) :- !,
        right_associate(A and (B and C), F).
right_associate(A and B, A1 and B1) :- !,
        right_associate(A, A1),
        right_associate(B, B1).
right_associate((A or B) or C, F) :- !,
        right_associate(A or (B or C), F).
right_associate(A or B, A1 or B1) :- !,
        right_associate(A, A1),
        right_associate(B, B1).
right_associate(A, A).
```


Get the clauses (1)

Getting the clauses is now easy: we first skolemize the original formula, and then we just have to remove the universal quantifiers in front, put the result into CNF, and create for each conjunct C a list whose members are the disjuncts in C.

We can get a simplified (and logically equivalent) set of clauses by:

- removing duplicates;
- deleting any clause that contains an atom and its negation, since such a clause is obviously valid.

We can also sort the literals in a clause.

We are going to uses clauses in such a way that it is better to represent variables by upper case letters rather than lower case letters.

Get the clauses (2)

Note that it was much cleaner to represent variables by lower case letters so far, otherwise our code would have made a much heavier use of metalogical operators (still we used the metalogical operator `=..`).

We cannot just substitute our variables `x1`, `x2`, `x3` by `X1`, `X2`, `X3` or `A`, `B`, `C`, because if we do that then `X1`, `X2`, `X3` or `A`, `B`, `C` will be automatically translated into anonymous variables, and output as such:

```
?- instance(x1,A,p(x1,x2),R).

A = _G161
R = p(_G161, x2)
```

Of course we would prefer `p(_G161, x2)` to `p(A, x2)` as an output.

Get the clauses (3)

Even worse, we will not be able to replace distinct lower case letters by distinct upper case letters:

```
?- instance(x1,A,p(x1,x2),I1), instance(x2,B,I1,I2).

A = x2
I1 = p(x2, x2)
B = _G166
I2 = p(_G166, _G166)
```

The solution is to use `numbervars/3`: this built-in enables to create terms that are internally represented as `'$VAR'(I)` where `I` is unified with a natural number, and printed out as upper case letters.

Get the clauses (4)

```
?- numbervars([X,Y,Z],0,L).

X = A
Y = B
Z = C
L = 3

?- numbervars([X,Y,Z,V,W],3,L).

X = D
Y = E
Z = F
V = G
W = H
L = 8
```


Get the clauses (5)

```
?- numbervars(X,0,_), var(X).

No

?- numbervars(X,0,_), compound(X).

X = A

?- numbervars(X,0,_), X = '$VAR'(I).

X = A
I = 0
```


Get the clauses (6)

The code to get the clauses from the universal formula obtained from the original formula by skolemization is then:

```
clauses(UniversalFormula, Clauses) :-
        clauses(UniversalFormula, [], [], Clauses).

clauses(forall X sep Form, LowerVars, UpperVars, Clauses) :- !,
        clauses(Form, [X| LowerVars], [_| UpperVars], Clauses).
clauses(Form, LowerVars, UpperVars, Clauses) :-
        numbervars(UpperVars, 0, _),
        change_variables(LowerVars, UpperVars, Form, Form1),
        cnf(Form1, CNFForm),
        cnf_to_clausal(CNFForm, Clauses).

change_variables([LowerX| LowerVars], [UpperX| UpperVars], Form, Form2) :-
        instance(LowerX, UpperX, Form, Form1),
        change_variables(LowerVars, UpperVars, Form1, Form2).
change_variables([], [], Form, Form).
```


Get the clauses (7)

```
cnf_to_clausal(Disjuncts and ConjunctsOfDisjuncts, Clauses) :- !,
        cnf_to_clausal(Disjuncts, FirstClause),
        cnf_to_clausal(ConjunctsOfDisjuncts, OtherClauses),
        append(FirstClause, OtherClauses, Clauses).
cnf_to_clausal(Disjuncts, Clause) :-
        disjunction_to_list(Disjuncts, List),
        sort(List, Set),                    % Remove duplicates and sort
        (   member(Atom, Set),              % Replace clause containing
            member(neg Atom, Set), !,       % an atom and its negation
            Clause = []                     % by the empty clause
        ;   Clause = [Set]
        ).

disjunction_to_list(Atom or Disjuncts, [Atom| Atoms]) :- !,
        disjunction_to_list(Disjuncts, Atoms).
disjunction_to_list(Atom, [Atom]).
```

Examples (1)

The file `to_clauses_tests.pl` enables to test the code we have developed.

For each test we print out:

- the original formula φ;
- a formula ψ in prenex form equivalent to φ;
- a skolemization ξ of ψ;
- a set of clauses equivalent to the matrix of ξ.

Examples (2)

```
?- t1.
forall x: (p(x) -> q(x)) -> forall x: p(x) -> forall x: q(x)
exists x1: exists x2: forall x3: ((p(x1) -> q(x1)) -> p(x2) -> q(x3))
forall x3: ((p(f1) -> q(f1)) -> p(f2) -> q(x3))
[~p(f2), p(f1), q(A)]
[~p(f2), ~q(f1), q(A)]

?- t2.
exists x: forall y: p(x,y) -> forall y: exists x: p(x,y)
forall x1: exists x2: forall x3: exists x4: (p(x1,x2) -> p(x4,x3))
forall x1: forall x3: (p(x1,f1(x1)) -> p(f2(x3,x1),x3))
[~p(B,f1(B)), p(f2(A,B),A)]
```