SLD Module

SLD1 Background to horn clause logic.

SLD2 Goals as Theorems and
Executions as Proofs.

SLD3 Deduction in First Order Logic.

SLD4 Normal Forms.

SLD5 Resolution for Propositional Logics.

SLD6 SLD Resolution for First Order Logics.

SLD7 SLD Trees.

Interpreting Logic Programs as Logical Statements: Horn Clauses

This lecture will show how a logic program can be interpreted as a logical statement about a domain. Each *fact* in the logic program will be interpreted as an *unconditional assertion* about the domain, and each *rule* will be interpreted as a *conditional assertion*.

The *goal* that is being solved can be interpreted as a *theorem* to be proved about the domain from the assertions. Thus the *execution* of a logic program is a *proof* of the theorem from the assertions.

The interpretation is explained with the help of a simple example of family relationships between people. The main concepts brought out are:

- The notion of domain of discourse.
- Facts can be used to represent *types* and *relations*.
- There are many ways of representing the same individual.
- The concept of same-ness is external to logic.
- Universal quantification.

- Simple rules can be used to define derived relationships.

- Conjunction and Disjunction in logic and in logic programming.

- Logical definitions and logic programming definitions are different. (Open-world and closed-world assumptions).

- Rules can be interpreted using existential quantification.

- Logical equivalence.

- Recursive rules pose problems in logic.

- Avoiding recursive definitions.

Background and Motivation:

Interestingly, the above "logical" interpretation predates the usual "procedural" interpretation of logic programs. In fact, the whole domain of logic programming, and the name "logic" in logic programming came out of the observation that we could view theorem proving as another form of computing, and a proof as a program.

Though logic programming arose from logic it has achieved a fair amount of maturity on its own and it is often not necessary to know any logic at all in order to understand and appreciate logic programming. In this context, it is often felt by many students that teaching logic in a logic programming course is a bit of an overkill.

Thus, in this course, logic programming has been introduced without any reference to logic, and this lecture and the ones following it will try to interpret logic programming execution in terms of logic, rather than following the usual approach of explaining how logic programming arose from logic (which is only of historical value).

Why should we try to interpret logic programming in terms of logic? The answer to this question is based on the observation that it is often easy to write logic programs that seem obviously correct at first sight, but which behave very eccentrically.

For example, it commonly happens that the first logic program that is written for a problem goes into a loop on execution. Interpreting logic programs in terms of logic allows us to get a better insight into what we are describing through a program, and to explore the alternative ways of describing something more carefully. It also helps us get a feel for what aspects of logic programming are tricky and should be used cautiously. This should convince the students that it is worth studying the connection between logic programming and logic.

Historical notes:

One of the main objectives of Artificial Intelligence was design automated deduction systems: Programs that could automatically prove theorems given a set of (deduction) rules and (axioms or) facts.

The basis of automated theorem proving were laid by Herbrand, Skolem and Tarski, in the early 20th century, much before computers became widespread. The early work was picked up in the 1960s by Prawitz, Davis, Putnam and Robinson among others who worked out the details necessary to design such theorem provers. The key to theorem proving was the resolution rule of Robinson. However theorem proving was slow and only toy problems were proved.

This changed when Kowalski and Colmerauer argued that logic could be viewed as the central theme of programs. A programmer need only specify the logic of the problem, leaving the details of execution (the control) to the computer. Kowalski argued for the paradigm: `program = logic + control`, and that logic could be viewed as a programming language, ie. *Logic Programming*.

Roussel implemented the first `Prolog` (PROgramming in LOGic) interpreter in ALGOL-W at Marseille in 1972. Central to the idea of prolog was a subset of formulas called *Horn Clauses*. These formulas gave a procedural definition to logic - the formulas could be regarded as facts, rules and queries. The prolog interpreter read in a set of rules and facts, and then it tried to answer (prove) queries in by searching the rules and facts in some predetermined order.

Prolog was slow because it was an interpreted language. Later a new machine called WAM (Warren's Abstract Machine) was defined to execute compiled prolog. Compiled prolog programs executed much faster than did the interpreted version of the same programs.

Bibliography

1. Clocksin, W. F. and Mellish, C. S. *Programming in Prolog* 2nd ed., Springer-Verlag, 1984.

2. Kowalski, R. *Logic for Problem Solving.* The Computer Science Library, North Holland, 1979.

3. Lloyd, J. W., *Foundations of Logic Programming.* Springer, 1984.

4. Gallier, J., *Logic for Computer Science.* John Wiley, Singapore, 1987.

5. Mendelson, E., *Introduction to Mathematical Logic*, 3rd ed., Wadsworth & Brooks/Cole, California, 1987.

SLD1:
Interpreting Logic Programs as Logical Statements SLD1:1
Logic Program example: Family relationships.

Types	Relations
boy(rahul).	parent(rajiv,rahul).
girl(priyanka).	parent(rajiv,priyanka).
man(rajiv).	parent(sonia,rahul).
man(sanjay).	parent(sonia,priyanka).
man(feroze).	parent(indira,rajiv).
man(nehru).	parent(indira,sanjay).
woman(sonia).	parent(nehru,indira).
woman(maneka).	married(rajiv,sonia).
woman(indira).	married(sanjay,maneka).

- Assume girl if age is less than 18, and woman otherwise.

- The *domain* of discourse is the set of people.

- We could also consider a subset of the family members of `rajiv` as the domain of discourse.

- Types represent sets (subsets of the domain).

- The types given above are time invariant.

- The relation `parent` is time invariant.

- The relation `married` is time dependent, and hence there is a problem:
  Consider a person who is divorced.

SLD1:1 On time dependence of facts:

Most relations between objects are captured by the use of verbs in natural language. For example, in the sentence "Rajiv married Sonia", the word "married" is a verb that defines some kind of association between the nouns "Rajiv" and "Sonia". However, verbs are usually complicated to understand semantically because they are usually time dependent.

In the above example, the past tense form "married" obviously indicates that the *event of the relationship* happened in the past. This does not automatically mean that the relationship will hold for ever. When we say, "Rajiv consulted Sonia", we obviously mean that there was an incident in the past in which Rajiv talked with Sonia about something, and this event lasted over a certain interval. It is definitely not an indefinite relationship.

Temporal dependence of relationships poses a very difficult problem for logic programming, which is still in a more or less unsolved stage. In our example, being married is a relationship that is more or less durable (at least in India!). So it is safe to take it as an indefinite relationship. However, if a divorce does take place between two married people X and Y, then we need to *retract* the fact `married(X,Y)` and replace it by a new fact `divorced(X,Y)`. Keeping both these will obviously lead to inconsistency.

Normal logic cannot handle such time variance of assertions. To understand this, we need to use *Modal logic* or to use *non-monotonic logic*.

SLD1:2 Interpretation of Facts.

Constants: `rajiv, sonia, ...`
Constants represent individual elements of the domain unambiguously.
Using names (eg. `rajiv`) is not the only way of doing this. There are other ways of specifying elements:

- 'The green thing that is on the table'.

- 'Sonia's mother'. This is absolutely unambiguous. (Who knows the name of Sonia's mother anyway ?)

  `mother(sonia)` is a *term*,
  and `mother` is a *function*.

  The function `mother` is *total*, i.e. it is defined on all the elements of the domain.

- "Rajiv's wife" .
  We now have two representations for `sonia`: `sonia` and `wife(rajiv)`.
  `wife(rajiv)` is a *term*, and `wife` is a *function*.
  The function `wife` is not total, that is, it is *partial*.

Equality is external to logic:

> '`sonia = wife(rajiv)`' asserts that both `sonia` and `wife(rajiv)` are the *same*.

SLD1:2 On articles:

In English, articles like "the" and "that" are used to fix the reference of some individual element of the domain of discourse.

Thus when we say "Ram took that book", the speaker and the listener are assumed to know which book is being referred to: hence "that book" can really be treated as a constant. To be more specific, we should make the constant dependent on the speaker and the hearer, and the time and place that the sentence was uttered. We could thus exactly capture the book being referred to by using a *constant term* `that_book(a,b,time,place)` for the book where `a` is the name of the speaker, `b` that of the listener, and time and place are either constants or constant terms referring to the time and place when the sentence was uttered.

SLD1:3
Variables and Universal quantification.

Consider the *universal* fact:

```
parent(mother(X),X).
```

Here `X` is a *variable*. It represents any arbitrary but fixed element of the domain.

Interpret the fact with a *universal quantifier*:

```
∀ X . parent(mother(X),X).
```

Read as:

> 'For any person `X`, `X`'s `mother` is a `parent` of `X`.'

`mother` can also be written as a relationship:

```
mother(indira,rajiv).
```

But then, we have to assert that there is only `mother` for every `X`.

How to do this ? .....Later...

SLD1:3 On universal quantifiers:

The importance of the domain of discourse becomes clear only when we look at the meaning of quantifiers. Consider the universal fact:

```
related(X,rajiv).
```

which says that "Everyone is related to Rajiv".

This will be true, if we consider the domain of discourse to be the set of all people in the family tree of Nehru, but will be false if we take the domain to be the set of all people.

Thus it is very important to fix the domain that is being considered before the logic program is written.

SLD1:4
Interpretation of rules.

Better to write the relationship `mother` as *derived relation*.
A *rule* used for this is:

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

Read as:

> 'If `X` is a `parent` of `Y`, and `X` is a `woman`, then `X` is a `mother` of `Y`'.

Or in logic, we write:

```
∀ X,Y.((parent(X,Y) AND woman(X))
   IMPLIES mother(X,Y)).
```

Uniqueness of `mother` guaranteed if the database of facts is consistent,
i.e. only one woman parent for a person is asserted.

A `mother` could also be a girl.
It is Correct to say:

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

where:

```
female(X) :- woman(X).
female(X) :- girl(X).
```

Translate this into English as:

'If `X` is a `woman` or `X` is a `girl`,
then `X` is `female`.'

Or more naturally translated as:

‘All `girls` and `women` are `female`.’

and into logic as:

```
∀ X.((girl(X) OR woman(X))
   IMPLIES female(X)).
```

Note: Conjunction in one rule, but a disjunction splits a rule into many rules.

SLD1:4 Conjunction and Disjunction:

It is somewhat unnatural that logic programming treats conjunction differently from disjunction. This leads to some kind of asymmetry between the two constructs.

We are used in logic to the idea that conjunction and disjunction are duals of each other, and with the help of negation, we could replace one by the other.

The asymmetry between the two is forced basically because it is not easy to capture negation easily in logic programming.

SLD1:5 Definitions.

Note that the two rules for female:

```
female(X) :- woman(X).
female(X) :- girl(X).
```

do not automatically imply that:

'All `females` are either `girls` or `women`.'

Open-world assumption: Addition of a third rule for female will affect this.

Definition of a predicate: Collection of all facts and rules with this predicate as head.

Examples:

```
mother(X,Y) :- parent(X,Y), female(X).
husband(X,Y) :- married(X,Y), male(X).
wife(X,Y) :- husband(Y,X).
sameFamily(X,Y) :- parent(X,Y).
sameFamily(X,Y) :- parent(Y,X).
sameFamily(X,Y) :- married(X,Y).
```

Closed-world assumption is meaningful for complete definitions.
Closed world assumption allows us to *close* a definition.

For the example above, we can interpret the definition of `female` as:

```
∀ X.(female(X)
   IF AND ONLY IF
(girl(X) OR woman(X))).
```

Thus making the one-way implication two-way. Logic programming uses this to get a meaningful definition of Negation.

SLD1:5 Closed-world assumption:

The implementation scheme called *negation as failure* in logic programming concludes that the negative predicate `not(p)` holds, when it has exhausted all possibilities of showing that `p` can hold. If the definition of `p` in the logic program is complete, then this is a meaningful way of defining negation.

SLD1:6 New variables and Existential quantification.

Consider the rule:

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

To define a relation between `X` and `Y` called `grandParent`, we need to bring in a third person `Z`. We are thus introducing a new variable through the rule.

Translate the above as:

> 'If `X` is a parent of some person `Z`,
> and if `Z` is a `parent` of `Y`,
> then `X` is a `grandParent` of `Y`.'

The subgoal `parent(X,Z)` which introduced `Z` for the first time is also called the **generator**, as it is expected to produce alternatives for `Z`.
The subgoal `parent(Z,Y)` is called a **tester**.
The body of the rule thus performs a **generate and test**.
The new variable `Z` gives rise to a **search**.

In logic, we can interpret this as:

$\forall$`X,Y. ([`$\exists$`Z.(parent(X,Z) AND parent(Z,Y))]`
`IMPLIES grandParent(X,Y)).`

The new variable is thus handled with an **existential quantifier**.

Logic is neutral about *how* the element `Z` is to be found.
It merely wants such an element to exist.
On the other hand, for implementation, we need to *search*.
The difference can become a problem when the domain is *infinite*.

Note that the above logical statement is equivalent to:

```
∀X,Y,Z.((parent(X,Z) AND parent(Z,Y))
IMPLIES grandParent(X,Y)).
```

In which, the existential quantifier seems to have vanished!
The left hand side of an implication has *negative import*.

SLD1:6 Existential quantification:

Consider the statement:

```
(∃X.p(X)) IMPLIES q.
```

Since `a IMPLIES b` is logically equivalent to: `(NOT a) OR b`, this reduces to:

```
(NOT (∃X.p(X))) OR q.
```

`NOT ∃X.p(X)` is the same as `∀X.(NOT p(X))`.
Thus we get, `(∀X.(NOT p(X))) OR q`.
which is the same as: `∀X.((NOT p(X)) OR q)`.
which finally gives us: `∀X.(p(X) IMPLIES q)`.

SLD1:7 Negation and equality.

```
sibling(X,Y) :-
    father(Z,X), father(Z,Y),
    mother(W,X), mother(W,Y).
brother(X,Y):-
    sibling(X,Y), male(X).
sister(X,Y) :-
    sibling(X,Y), female(X).
```

Is this complete? ..... No!

`brother(rajiv,rajiv)` will be true,
which is quite meaningless.
We need to modify sibling as follows:

```
sibling(X,Y) :-
    not_same(X,Y),
    father(Z,X), father(Z,Y),
    mother(W,X), mother(W,Y).
```

Since same-ness has to be completely defined, this is a problem. A simple solution is to let:

```
not_same(X,Y) :- not(X=Y).
```

Which considers `X` and `Y` to be the same if they are identical as *terms*.
This is inadequate, as it would lead to

```
not_same(mother(rajiv),indira).
```

SLD1:8 More on negation: Complements.

```
dead(nehru).
dead(indira).
dead(rajiv).
dead(sanjay).
alive(X) :- not(dead(X)).
widow(X) :- wife(X,Y), dead(Y).
```

- Predicates `dead` or `alive` can be defined in terms of each other.

- Predicates `dead` and `alive` are **complements** of each other.

- Thus, using the closed-world assumption, we need only list one of the predicates.

- Efficiency: List the one that is smaller.

- `alive(sonia)` will succeed because `dead(sonia)` will fail.

Existential quantifiers and negation do not mix well. Consider:

```
alive(sonia).
alive(maneka).
alive(priyanka).
alive(rahul).
dead(X):- not(alive(X)).
widow(X):- dead(Y), wife(X,Y).
```

To Search for someone who is `dead`,
we instead search for someone who is `not alive`.

. . . . . This will not work!

Not using negation is more natural:

```
orphan(X) :-
    parent(Y,X), dead(Y),
    parent(Z,X), dead(Z),
    not(Y=Z).
```

is better expressed as:

```
orphan(X) :-
    father(Y,X), dead(Y),
    mother(Z,X), dead(X).
```

The negation is buried deep: male and female are complements.

Procedural use of negation is not logical.

```
guardian(X,Y):-
    father(X,Y), alive(X).
guardian(X,Y):-
    mother(X,Y), alive(X).
```

Assumes that the second rule is tried *after* the first.
Modify the rule to:

```
guardian(X,Y) :-
    father(Z,Y), dead(Z),
    mother(X,Y), alive(X).
```

SLD1:9 Recursively Defined Predicates.

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

Note that to define a relation between `X` and `Y` called `ancestor`,
we need to use the same relation `ancestor` with other arguments. We are thus introducing recursive definitions.

Translate the above as:

> 'If `X` is a parent of `Y`,
> or `X` is the parent of some person `Z`,
> who is an `ancestor` of `Y`,
> then `X` is an `ancestor` of `Y`.'

The order in which these rules are to be tried becomes very important.
Otherwise, the search can loop forever.
Translate this into logic as:

```
∀X,Y.((parent(X,Y) OR
   (∃Z.(parent(X,Z) AND ancestor(Z,Y))))
      IMPLIES ancestor(X,Y)).
```

which is completely indifferent about the order of execution.

Recursive definitions (or mutual definitions) have been viewed suspiciously by logicians.

Non-recursive predicates can be accidentally written recursively.

```
uncle(X,Y):-
   brother(X,Z),parent(Z,Y).
aunt(X,Y) :-
   sister(X,Z), parent(Z,Y).
```

Not complete! Maneka is an aunt of Rahul. Add:

```
uncle(X,Y):-
    married(X,Z), aunt(Z,Y).
aunt(X,Y) :-
    married(X,Z), uncle(Z,Y).
```

Inadvertently, uncle and aunt have become mutually recursive now!

Remove recursion whenever possible.

```
blood_uncle(X,Y):-
    brother(X,Z),parent(Z,Y).
blood_aunt(X,Y) :-
    sister(X,Z),parent(Z,Y).
uncle(X,Y):-
    blood_uncle(X,Y).
uncle(X,Y):-
    married(X,Z), blood_aunt(Z,Y).
aunt(X,Y):-
    blood_aunt(X,Y).
aunt(X,Y):-
    married(X,Z), blood_uncle(Z,Y).
```

This has no recursion at all.

SLD2 Support: Goals are Theorems and Executions are Proofs

This lecture will first summaries the interpretation of facts and rules in a logic program as horn clauses. The goal that is being solved can be interpreted as the negation of a horn clause. If `C` is the set of horn clauses representing the program, and `G` is the horn clause representing the negation of the goal, then the execution of the logic program for the goal can then be seen to be a proof that the set of horn clauses $C \cup \{ G \}$ is inconsistent, i.e. a contradiction can be proved from it. `C` is consistent (it is supposed to be a set of assertions about the domain of discourse), but asserting $\{ G \}$ together with `C` leads to a contradiction. This means that `C` implies `(not G)`, i.e. `C` implies the goal.

Thus the execution of any logic program can be viewed as a *proof by refutation*.

The lecture also addresses the following topics:

- Top-down proofs represent forward reasoning, and bottom-up proofs represent backward reasoning.

- Existential quantifiers and search.

- Recursively defined predicates makes the length of the proof unknown.

- Different ways of proving yield different answers.

Background and Motivation:

It is necessary to understand the connection between logic program executions and proofs by refutations, as this lets us understand a specific execution strategy chosen for a logic programming language. It also allows us to explore alternative ways of implementing logic programming languages.

SLD2: Goals are Theorems and Executions are Proofs

SLD2:1 A Logic Program

The relationship example of the last lecture.

```
male(rahul).          parent(rajiv,rahul).
male(rajiv).          parent(sonia,rahul).
male(sanjay).         parent(indira,rajiv).
male(feroze).         parent(indira,sanjay).
male(nehru).          parent(feroze,rajiv).
female(sonia).        parent(feroze,sanjay).
female(indira).       parent(nehru,indira).
female(maneka).       married(rajiv,sonia).
                      married(sanjay,maneka).
```

```
mother(X,Y) :-    parent(X,Y), female(X).
father(X,Y) :-    parent(X,Y), male(X).
sibling(X,Y) :-   father(Z,X), father(Z,Y),
                  mother(W,X), mother(W,Y).
brother(X,Y) :-   sibling(X,Y), male(X).
sister(X,Y)  :-   sibling(X,Y), female(X).

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

blood_uncle(X,Y) :- brother(X,Z),parent(Z,Y).
blood_aunt(X,Y)  :- sister(X,Z), parent(Z,Y).

uncle(X,Y):- blood_uncle(X,Y).
uncle(X,Y):- married(X,Z), blood_aunt(Z,Y).
aunt(X,Y) :- blood_aunt(X,Y).
aunt(X,Y) :- married(X,Z), blood_uncle(Z,Y).
```

SLD2:2 Horn Clauses

Positive Literal: is a predicate applied to arguments. E.g.

```
male(rahul), female(X),
parent(sonia,X), sister(A,B).
```

Negative literal: is the negation of a positive literal. E.g.

```
NOT parent(indira,rajiv),
NOT brother(rahul,nehru).
```

Clause: is a disjunction of literals. E.g.

```
married(rahul,X)
   OR (NOT parent(X,indira))
   OR blood_uncle(maneka,sanjay).
```

Program Clause: is a clause that contains exactly one positive literal. E.g.

```
married(rahul,X) OR (NOT parent(X,indira)),
parent(nehru,indira).
```

Goal Clause: is a clause that contains no positive literals. E.g.

```
(NOT married(rahul,X))
   OR (NOT parent(X,indira)),
NOT parent(nehru,indira).
```

Horn Clause: is either a program clause or a goal clause. E.g.

```
(NOT married(rahul,X))
   OR (NOT parent(X,indira)),
parent(nehru,indira).
```

Variables in a clause are assumed to be universally quantified.
Thus, `married(rahul,X) OR (NOT parent(X,indira))` stands for:

$\forall$ `X . (married(rahul,X)`
`OR (NOT parent(X,indira))).`

A fact is obviously a horn clause.
A rule is also a horn clause:

"`P :- Q, R`" can be interpreted as
`(Q AND R) IMPLIES P`,
which is equivalent to: `(NOT (Q AND R)) OR P`,
which in turn reduces to: `(NOT Q) OR (NOT R) OR P`, which is a horn clause.

Note: (Ref. SLD1:6): Internal existential quantifiers become external universal quantifiers.

SLD2:3 Horn Clause Program

```
male(rahul).      parent(rajiv,rahul).
male(rajiv).     parent(sonia,rahul).
male(sanjay).     parent(indira,rajiv).
male(feroze).     parent(indira,sanjay).
male(nehru).     parent(feroze,rajiv).
 parent(feroze,sanjay).
female(sonia).    parent(nehru,indira).
female(indira).
female(maneka).   married(rajiv,sonia).
 married(sanjay,maneka).
mother(X,Y)
   OR (NOT parent(X,Y))
   OR (NOT female(X)).
father(X,Y)
   OR (NOT parent(X,Y))
   OR (NOT male(X)).
```

```
sibling(X,Y)
  OR (NOT father(Z,X))
  OR (NOT father(Z,Y))
  OR (NOT mother(W,X))
  OR (NOT mother(W,Y)).
brother(X,Y)
  OR (NOT sibling(X,Y))
  OR (NOT male(X)).
sister(X,Y)
  OR (NOT sibling(X,Y))
  OR (NOT female(X)).

ancestor(X,Y)
  OR (NOT parent(X,Y)).
ancestor(X,Y)
  OR (NOT parent(X,Z))
  OR (NOT ancestor(Z,Y)).
```

```
blood_uncle(X,Y)
  OR (NOT brother(X,Z))
  OR (NOT parent(Z,Y)).
blood_aunt(X,Y)
  OR (NOT sister(X,Z))
  OR (NOT parent(Z,Y)).

uncle(X,Y)
  OR (NOT blood_uncle(X,Y)).
uncle(X,Y)
  OR (NOT married(X,Z))
  OR (NOT blood_aunt(Z,Y)).

aunt(X,Y)
  OR (NOT blood_aunt(X,Y)).
aunt(X,Y)
  OR (NOT married(X,Z))
  OR (NOT blood_uncle(Z,Y)).
```

SLD2:4 Interpreting a Goal

Consider the goal (read as "Is Indira the mother of Sanjay"):

```
?- mother(indira,sanjay).
```

The negation of the goal is:

```
(NOT mother(indira,sanjay)).
```

which is a goal (Horn) clause.

Consider the set of program clauses together with the goal clause.

The relevant subset is listed below (with universal quantifiers):

1. female(indira).
2. parent(indira,sanjay).
3. ∀ X, Y . mother(X,Y)
   OR (NOT parent(X,Y))
   OR (NOT female(X)).
4. (NOT mother(indira,sanjay)).

This set is inconsistent as seen below: From 3 (substituting `indira` for `X` and `sanjay` for `Y`):

```
5. mother(indira,sanjay)
     OR (NOT parent(indira,sanjay))
     OR (NOT female(indira)).
```

from 4 and 5 :

```
6. (NOT parent(indira,sanjay))
      OR (NOT female(indira)).
```

from 2 and 6 :

```
7. NOT female(indira).
```

from 1 and 7, we show a contradiction.

Thus, assuming the negation of the goal leads to a contradiction.
Thus, the program clauses imply the goal, i.e. the goal is true.

SLD2:5 Conjunctive Goals

Consider the goal: "Is Indira the mother of both Sanjay and Rajiv?":

```
?- mother(indira,sanjay),
   mother(indira,rajiv).
```

The goal can be interpreted in logic as:

```
mother(indira,sanjay)
AND mother(indira,rajiv).
```

The negation of the goal is:

```
(NOT mother(indira,sanjay))
OR (NOT mother(indira,sanjay)).
```

which is a goal (Horn) clause.

The subset below is inconsistent too.

```
1.  female(indira).
2.  parent(indira,sanjay).
2'.  parent(indira,rajiv).
3.  ∀X,Y.mother(X,Y)
    OR (NOT parent(X,Y))
    OR (NOT female(X)).
4.  (NOT mother(indira,sanjay))
    OR (NOT mother(indira,rajiv)).
```

To see this, we could repeat the last proof to get:

```
5. mother(indira,sanjay)
     OR (NOT parent(indira,sanjay))
     OR (NOT female(indira)).
6. (NOT parent(indira,sanjay))
     OR (NOT female(indira))
     OR (NOT mother(indira,rajiv)).
7. (NOT female(indira))
     OR (NOT mother(indira,rajiv)).
8. NOT mother(indira,rajiv).
```

The contradiction then arises in the same way as the last proof.

SLD2:6 Existential Quantifiers in a Goal

Consider the goal: "Who is the mother of both Sanjay and Rajiv?":

```
?- mother(X,sanjay),
   mother(X,rajiv).
```

The goal can be interpreted in logic as:

```
∃ X . (mother(X,sanjay)
   AND mother(X,rajiv)).
```

The negation of the goal is:

```
∀ X . ((NOT mother(X,sanjay))
   OR (NOT mother(X,rajiv))).
```

which is also a goal (Horn) clause.

The subset below is inconsistent too.

1. `female(indira).`
2. `parent(indira,sanjay).`
2'. `parent(indira,rajiv).`
3. $\forall$`X,Y.mother(X,Y)`
   `OR (NOT parent(X,Y))`
   `OR (NOT female(X)).`
4. $\forall$ `X.((NOT mother(X, sanjay ))`
   `OR (NOT mother(X,rajiv))).`

Just substitute `indira` for `X` in 4 and use the last proof. The proof of contradiction **requires a substitution**. The substitution also serves as an answer: `X=indira`.

SLD2:7 Top-Down versus Bottom-Up: Forward versus Backward Reasoning.

The proofs that we have seen so far are all top-down: They all start with the goal clause.

Top-down proofs represent backward reasoning.

To show the goal clause is inconsistent (with respect to the program clauses), it is enough to show that the derived goal clause is inconsistent. Note that any clause derived from the goal clause is also a goal clause.

Thus in backward reasoning, we have a proof that goes: $Goal_1 \rightarrow Goal_2 \rightarrow \ldots$, which finally yields a contradiction.

Bottom-up or Forward reasoning

Here, we start with the given program clauses and try to derive more program clauses from them.

Below, we have just a set of program clauses (no goal):

```
1.  female(indira).
2.  parent(indira,sanjay).
3.  ∀ X,Y . mother(X,Y)
      OR (NOT parent(X,Y))
      OR (NOT female(X)).
```

From 3 (substituting `indira` for `X` and `sanjay` for `Y`) we get:

```
5. mother(indira,sanjay)
     OR (NOT parent(indira,sanjay))
     OR (NOT female(indira)).
```

from 1 and 5, we get:

```
6. mother(indira,sanjay)
     OR (NOT parent(indira,sanjay)).
```

from 2 and 6:

```
7. mother(indira,sanjay).
```

Note that we have directly derived the goal. We did not need to use a refutation based proof.

Forward reasoning is harder, because we do not how to proceed.

SLD2:7. On Forward and Backward Reasoning

In *backward reasoning*, we start with a goal clause, and use a program clause to derive another clause. This clause will always be a goal clause. To see this, note that a goal clause contains only negative literals, and a program clause contains exactly one positive literal. In order to derive a clause from two clauses, we need to "cancel" a positive literal from one clause and a corresponding negative literal from the other, and put the remaining literals together. This is the famous *resolution rule* which will be discussed later.

If we remove a positive literal from the program clause, we are left with only negative literals from it (or nothing at all). Since the goal clause contains only negative literals, putting the remaining literals together results in only negative literals, hence a goal clause results.

For example, the goal clause `((NOT P) OR (NOT Q))` and the program clause `(P)` will let us derive `(NOT Q)` which is a goal clause. Similarly, the goal clause `((NOT P) OR (NOT Q))` and the program clause `(P OR (NOT R))` will let us derive `((NOT Q) OR (NOT R))`, which is also a goal clause.

In *forward reasoning*, we start with the program clauses, and derive more clauses. These clauses will always be program clauses. To see this, note that a program clause contains exactly one positive literal. In order to derive a clause from two clauses, we need to "cancel" a positive literal from one clause and a corresponding negative literal from the other, and put the remaining literals together.

If we remove a positive literal from one program clause, we are left with only negative literals from it (or nothing at all). Since the other program clause contains exactly one positive literal, putting the remaining literals together results in exactly one positive literal, hence a program clause results.

For example, the program clause ((NOT P) OR (Q)) and the program clause (P) will let us derive (Q) which is a program clause. Similarly, the program clause ((NOT P) OR (Q)) and the program clause (P OR (NOT R)) will let us derive ((Q) OR (NOT R)), which is also a program clause.

SLD2:8 Many proofs mean many answers are possible

In general, there can be many proofs of a goal.
This may give rise to several answers.
Consider the goal: "Whose mother is Indira?" :

```
?- mother(indira,X).
```

The goal can be interpreted in logic as:

```
∃X.(mother(indira,X)).
```

The negation of the goal is:

```
∀X.(NOT mother(indira,X)).
```

The set of relevant clauses is:

```
1.  female(indira).
2.  parent(indira,sanjay).
2'.  parent(indira,rajiv).
3.  ∀X,Y.mother(X,Y)
    OR (NOT parent(X,Y)) OR (NOT female(X)).
4.  ∀X . (NOT mother(indira,X)).
```

By forward reasoning from 1,2 and 3, we can derive:

```
5. mother(indira,sanjay).
```

Similarly, by forward reasoning from 1,2' and 3, we can derive:

```
5'. mother(indira,rajiv).
```

Both derivations have refuted 4 and hence proved the goal. This gives two alternative answers: `X = sanjay` and `X = rajiv`.

Logic does not give any order among these answers, unlike logic programming, which may announce `X=sanjay` first.

SLD2:9 Recursively defined predicates and lengths of proofs

Absence of recursive predicates implies that forward reasoning will terminate.

Level of a Predicate is the depth of definitions it depends on.

In our example the levels are:

```
Level 0: male, female, parent, married.
Level 1: father, mother.
Level 2: sibling.
Level 3: brother, sister.
Level 4: blood_uncle, blood_aunt.
Level 5: uncle, aunt.
```

Assume that all the facts arising from predicates at level L have been derived.

To derive a predicate at level $(L+1)$, the number of forward reasoning steps will depend only on the sizes of the rules defining the predicate.

For example (as we saw), `mother(indira,sanjay)` can be derived in three forward reasoning steps. The same holds for:

```
mother(indira,rajiv),
father(feroze,sanjay),
father(feroze,sanjay).
```

From this, we can derive `sibling(rajiv,sanjay)` in five steps, and further, we can derive: `brother(rajiv,sanjay)` in three steps.

Thus, we will exhaust all the possible derivations very quickly.

On the other hand, recursive predicates make the length of proofs unknown, since recursive predicates can be re-used many times.

For example, in:

```
1.  parent(rajiv,rahul).
2.  parent(indira,rajiv).
3.  parent(nehru,indira).
4.  ∀X,Y.(ancestor(X,Y)
      OR (NOT parent(X,Y))).
5.  ∀X,Y,Z.(ancestor(X,Y)
      OR (NOT parent(X,Z))
      OR (NOT ancestor(Z,Y))).
```

We can derive `ancestor(nehru,indira)` in three steps using only 3 and 4. Similarly, we can derive `ancestor(indira,rajiv)` in three steps. To derive `ancestor(nehru,rajiv)`, we can use 5 with the substitution:

```
X=nehru, Z=indira, Y=rajiv.
```

To derive `ancestor(nehru,rahul)`, we need to use 5 two times, with the substitutions:

```
X=nehru,  Z=indira, Y=rahul.
X=indira, Z=rajiv,  Y=rahul.
```

SLD3 Deduction in First Order Logic is Hard

This lecture will first review first-order logic, and will describe some simple proof systems for it. The focus will be to bring out the difficulty of automating the proof systems, and to show that first order logic can give rise to ambiguities when we use it for logic programming.

The lecture also addresses the following topics:

- Proof systems.
- Disjunction causes ambiguity.
- Existential queries can be proved non-constructivel

Background and Motivation:

First order logic provides the context in which logic programming concepts are dealt with. Besides, the main application area of logic programming, namely Artificial Intelligence programming, has traditionally used first-order logic as a knowledge representation mechanism, and automated deduction as a knowledge inference method. Thus it is necessary to bring out the logic programming issues associated with first-order logic to see where logic programming stands with respect to full first-order deduction.

SLD3: Deduction in First order Logic is Hard

SLD3:1 Definitions used in first-order logic

The *symbols* used in first-order logic can be divided into:

- Logical Symbols.
- Non-logical symbols.
- Variable symbols. (Variable symbols start with upper case).

Logical symbols are application independent and can be divided into two types:

- Logical Connectives: AND, OR, NOT, IFF, IMPLIES.

- Quantifiers: $\forall$ ("forall"), $\exists$ ("there exists").

For a given application, we fix the domain of discourse, and choose a set of non-logical symbols to describe it. This set is called the Language.
The symbols in the language can be split into:

- Constant symbols.
- Function symbols.
- Predicate symbols.

All these symbols start with lower case.
e.g. Language = { `sanjay, rajiv, indira, male, female, mother, parent` }.

The **terms** of first-order logic (with respect to a Language) are:

- Constants.
- Variables.
- function symbols applied to other terms.

e.g. `sanjay`, `X`, `mother(sanjay)`, `mother(X)`, `mother(mother(sanjay))`.

The **atomic formulas** of first-order logic are: 'predicates applied to terms'.

e.g. `parent(indira,sanjay)`, `parent(X,Y)`,
`parent(mother(mother(X)),rajiv)`.

The **formulas** of first-order logic are:

- Atomic Formulas.
- Logical connectives applied to other formulas.
- Quantifiers applied to other formulas.

e.g. ∀`Y.(NOT (`∃`X.(parent(mother(X),Y)`
`OR (NOT parent(Y,X))))).`

SLD3:2 Axioms, Proof Rules and Theorems

There are some formulas that will always be true, whichever domain of discourse we are describing.

e.g. `P OR (NOT P).`

$\forall$`X.(p(X) IMPLIES p(X)).`

Some formulas are true for specific domains.
e.g. $\forall$`X.(male(X) OR female(X)).`

An important part of logic is to exactly capture these two sets.

One way to do this is to use a **Proof System**. A proof system consists of **Axioms** and **Proof Rules**.

Axioms are formulas that are unconditionally true. They are analogous to facts of a logic program.

We usually write **Axiom Schemas** to capture axioms.
e.g. α `OR (NOT` α`).`
$\forall$`X.(`α`(X) IMPLIES` α`(X)).`

where α can be replaced by any formula with one free variable.

Proof Rules allow other true formulas to be derived from the axioms. They are conditional, similar to logic programming rules.

Proof Rules are generic similar to axiom schemas. e.g.

$$\frac{\alpha \qquad \alpha \texttt{ IMPLIES } \beta}{\beta}$$

True formulas derived from the axioms and proof rules are called **Theorems**.

SLD3:3 An Example Proof in Propositional Logic

Consider the following axioms and definitions of propositional logic:

A1. α IMPLIES (β IMPLIES α).

A2. (α IMPLIES (β IMPLIES γ)) IMPLIES
((α IMPLIES β)
IMPLIES (α IMPLIES γ)).

A3. (NOT β IMPLIES NOT α) IMPLIES
(NOT β IMPLIES α) IMPLIES β.

Df-OR. (α OR β) is defined as
((NOT α) IMPLIES β).

Df-AND. (α AND β) is defined as
NOT(α IMPLIES NOT β).

Df-IFF. (α IFF β) is defined as
((α IMPLIES β) AND
(β IMPLIES α)).

and the Rule of Inference:

$$\frac{\alpha \qquad \alpha \text{ IMPLIES } \beta}{\beta} \text{ MP.}$$

Example: To prove: ‘P IMPLIES P’

1. ‘(P IMPLIES ((P IMPLIES P) IMPLIES P))
   IMPLIES
      (P IMPLIES (P IMPLIES P)) IMPLIES
         (P IMPLIES P)’

   This is Axiom A2, with α=P,
      β=(P IMPLIES P), and γ=P.

2. ‘P IMPLIES ((P IMPLIES P) IMPLIES P)’

   This is Axiom A1, with α=P and
      β=(P IMPLIES P).

3. ‘(P IMPLIES (P IMPLIES P))
IMPLIES (P IMPLIES P)’

Using Rule MP on 1 and 2.

4. ‘P IMPLIES (P IMPLIES P)’

This is Axiom A1, with α=P and β=P.

5. ‘P IMPLIES P’

(By applying Rule MP on 4 and 5).

Axioms for FOL

Besides the axioms of PL, we now add axioms and a definition to account for the meaning of quantifiers:

A4. (∀X. A(X)) IMPLIES A(t)
where term t is free of X.

A5. (∀X. (A IMPLIES B))
IMPLIES (A IMPLIES ∀ X.B)
where A is free of X.

Df-∃: ∃X.α(X) is defined as: NOT ∀X. NOT α(X)

A new rule of inference for generalisation is also included:

```
   A
-----          Gen.
∀X.A
```

Deduction Theorem:

> If we can prove β, by assuming α, then we have proved 'α IMPLIES β'.

This holds under the condition that no free variable of α was generalised during our proof of β.

SLD3:4 FOL formulas are harder to prove

To prove: ‘‘∀X.∀Y.P(X,Y) IMPLIES ∀Y.∀X.P(X,Y).’’

1. ∀x.∀y.P(X,Y)
   Assume.
2. (∀x.∀y.P(X,Y)) IMPLIES ∀y.P(X,Y)
   this is axiom A4.
3. ∀y.P(X,Y)
   From 1,2 by MP.
4. (∀y.P(X,Y)) IMPLIES P(X,Y)
   this is axiom A4.
5. P(X,Y)
   From 3,4 by MP.
6. ∀x.P(X,Y)
   Gen.5.
7. ∀y.∀x.P(X,Y)
   Gen.6.

Using the Deduction Theorem on 1 and 6, with:

α = ∀X.∀Y.P(X,Y),
β = ∀Y.∀X.P(X,Y)

we have proved the formula.

SLD3:5 FOL is too Hard, and Besides may not be Useful

- Where do we start?
- What lemmas and derived rules do we prove first?
- Is it possible to automate this procedure?
- Is it a meaningful exercise to prove formulas like this?

- Does proving arbitrary formulas give us useful answers to queries?

- Proving existential formulas is meaningful.

- Proving $\exists \mathtt{X}.\alpha(\mathtt{X})$, may yield an answer to the query:
  Find an object $\mathtt{X}$ that satisfies $\alpha(\mathtt{X})$.

- But there is a danger that we may get ambiguous answers.

SLD3:6 Disjunction causes unpredictability

Consider the formula: `P(a) OR P(b)`.

Now, we can prove $\exists$`X.P(X)` from the above formula. The proof goes like this:

The following axiom follows from the meaning of existential quantification.

$$\alpha\texttt{(t) IMPLIES } \exists\texttt{X.}\alpha\texttt{(X)},$$

for any term `t`.

The following two formulas can be derived from the above axiom.

```
p(a) IMPLIES ∃X.p(X).
p(b) IMPLIES ∃X.p(X).
```

The following derived rule can be proved:

$$\frac{\alpha \texttt{ IMPLIES } \gamma \qquad \beta \texttt{ IMPLIES } \gamma}{(\alpha \texttt{ OR } \beta) \texttt{ IMPLIES } \gamma}$$

From which, we can derive:

```
(p(a) OR p(b)) IMPLIES ∃X.p(X).
```

Given our initial formula, we can now prove: `∃X.p(X).`

What is the answer to the question:
Find an `X` such that `p(X)` ?

Disjunction in `p(a) OR p(b)` gives rise to *ambiguity*.

Solution: Disallow disjunction of positive information.

For the same reason, disallow facts like: ∃`X.p(X)`.

SLD3:7 Some justification for Horn Clauses

- Facts should contain only positive information.

- Facts should not be disjunctive.

- Facts should not contain existential quantifiers. i.e. Allow only conjunction and universal quantifiers.

- Universal quantification distributes over conjunction. i.e.

  $\forall$X.(α(X) AND β(X)) IMPLIES
  ($\forall$X.α(X)) AND ($\forall$X.β(X)).

- This is the same as allowing only logic programming facts! (i.e. Atomic formulas, with possibly universal quantification).

- Queries should be only existential.

- Proof rules should allow facts to be used to answer queries.

- Should be able to automate the proof system.

- Narrows the scope of formulas that can be allowed.

SLD4: Normal forms are better

This lecture will introduce how first order logic formulas can be transformed into formulas in clause form in such a way that the transformation does not affect the truth of the formula. Clause form formulas are more amenable to automated deduction.

The lecture also addresses the following topics:

- Skolemisation.
- Resolution.
- The Herbrand Universe.

Background and Motivation:

We have already seen that first order logic formulas are far too hard to prove. Besides this, it may not be meaningful to try to prove any first order logic formula from the point of view of logic programming.

Normalising first order logic formulas to clause form allows us to simplify enormously the task of trying to prove them, without losing anything by way of descriptive power. It also lets us get a better insight into what each construct of first order logic really provides us. For example, Skolemisation gives us a very good understanding of what existential quantifiers gives us.

SLD4: Normal Forms are Better

SLD4:1 Clause Form Formulas

Review:
Atomic formulas are predicates applied to terms. e.g.

```
parent(indira,sanjay),
parent(X,Y),
parent(mother(mother(X)),rajiv).
```

A **literal** is either an atomic formula or the negation of an atomic formula. e.g.

```
parent(indira,sanjay),
NOT parent(mother(mother(X)),rajiv).
```

A **clause** is a disjunction of literals.

A **Prime** formula is a formula whose main logical symbol (if any) is not a logical connective.

Atomic formulas are obviously prime formulas (no logical symbols).

Other examples:

$\forall$ `X. (p(X) OR (NOT q(X))),`
$\exists$ `X.` $\forall$ `Y. (p(X,Y) IMPLIES p(Y,X))`

Every formula can be seen to be made from prime formulas using only logical connectives.

Hence prime formulas are like propositional symbols. e.g.

$$(\exists X.\forall Y.p(X,Y))\ \texttt{IMPLIES}\ (\forall Y.\exists X.p(X,Y))$$

contains two prime formulas:
$\exists X.\forall Y.p(X,Y)$ and
$\forall Y.\exists X.p(X,Y)$.

Let `P1` denote the former and `P2` the latter respectively. Then the given formula can be written as a propositional logic formula: `P1 IMPLIES P2`.

SLD4:2 Clause form formulas in Propositional Logic

A **clause form formula** in propositional logic is a conjunction of clauses.

e.g. `(P1 OR P2) AND`
`((NOT P3) OR (NOT P2) OR P4)`

Sometimes, a clause is simply written as *a set of literals* and a clause form formula as *a set of clauses*.

e.g. { { `P1, P2` }, { `NOT P3, NOT P2, P4` } }

Any propositional logic formula can be converted into clause form.

Use equivalent definitions to eliminate some logical connectives.

`P IMPLIES Q` is equivalent to `(NOT P) OR Q`.
`P IFF Q` is equivalent to:

```
(P AND Q) OR ((NOT P) AND (NOT Q)).
```

We can then distribute `OR` over `AND` to get the conjunctions out. The formula will then be in clause form.
e.g. `(P AND Q) OR ((NOT P) AND (NOT Q))`

Distribute `(P AND Q)`:

```
  ((P AND Q) OR (NOT P)) AND
  ((P AND Q) OR (NOT Q))
```

Distribute `(NOT P)`:

```
((P OR (NOT P)) AND (Q OR (NOT P)))
 AND ((P AND Q) OR (NOT Q))
```

Distribute `(NOT Q)`:

```
  ((P OR (NOT P)) AND (Q OR (NOT P))) AND
  ((P OR (NOT Q)) AND (Q OR (NOT Q)))
```

which is in clause form.

We could drop the two tautologies:

```
(P OR (NOT P)) and (Q OR (NOT Q)).
(Q OR (NOT P)) AND (P OR (NOT Q))
```

This can be written in set form as:

```
{{Q, NOT P}, {NOT Q, P}}
```

SLD4:3 Pulling Quantifiers Out

We can extract quantifiers out of a formula using the fact:
α `OR` $\forall$`X.`β`(X)` is equivalent to $\forall$`Y.(`α `OR` β`(Y)),`
where `Y` is any variable symbol that is not used in α.

Similarly:
α `OR` $\exists$`X.`β`(X)` is equivalent to $\exists$`Y.(`α `OR` β`(Y)).`

α `AND` $\forall$`X.`β`(X)` is equivalent to $\forall$`Y.(`α `AND` β`(Y)).`

α `AND` $\exists$`X.`β`(X)` is equivalent to $\exists$`Y.(`α `AND` β`(Y)).`

Finally, we can extract a quantifier out of a negation symbol using:
`NOT` $\forall$`X.`α`(X)` is equivalent to $\exists$`X.(not` α`(X)).`
`NOT` $\exists$`X.`α`(X)` is equivalent to $\forall$`X.(not` α`(X)).`

Consider the formula:

$$(\exists X.\forall Y.p(X,Y))\ \texttt{IMPLIES}\ (\forall Y.\exists X.p(X,Y))$$

Writing propositional symbols for prime formulas, we get: `P1 IMPLIES P2`.
which in clause form is simply: `(NOT P1) OR P2`.

Replacing `P1` and `P2` by the prime formulas, we get:

$$(\texttt{NOT}\ (\exists X.\forall Y.p(X,Y)))\ \texttt{OR}\ (\forall Y.\exists X.p(X,Y))$$

Extracting the first two quantifiers out of the negation symbol:

$$(\forall X.\exists Y.(\texttt{NOT p(X,Y)}))\ \texttt{OR}\ (\forall Y.\exists X.\texttt{p(X,Y)})$$

Extracting the first two quantifiers out of the `OR` symbol:

$$\forall A.\exists B.((\texttt{NOT p(A,B)})\ \texttt{OR}\ (\forall Y.\exists X.\texttt{p(X,Y)}))$$

Extracting the last two quantifiers out of the `OR` symbol:

$$\forall A.\exists B.\forall C.\exists D.\ ((\texttt{NOT p(A,B)})\ \texttt{OR}\ \texttt{p(C,D)})$$

SLD4:4 Skolemising Existential Quantifiers

Consider the formula $\alpha = \forall$`X.`$\exists$`Y.P(X,Y)`, and the formula $\beta = \forall$`X.P(X,f(X))`, where `f` is a function symbol.

When do α and β have the same meaning?

Let the domain be `D` $= \{d_1, d_2, \ldots\}$.

Formula α asserts that for each d_i, there is a d_{m_i} such that P(d_i,d_{m_i}) holds.
i.e. it sets up a *map* m from `D` to `D`:

d_1 is mapped to d_{m_1},
d_2 is mapped to d_{m_2}, ...

Formula β asserts that for each d_i, `P(`d_i`, f(`d_i`))` holds, where `f` is a *function* from `D` to `D`.

Now α and β would have exactly the same import if the map m and the function `f` are exactly the same:
i.e. `f`$(d_i) = d_{m_i}$ for all d_i in `D`.

This is the basic idea behind Skolemization:

> We can replace an existential quantifier by a carefully chosen function

Thus, we could convert ∀`X.`∃`Y.P(X,Y)` into ∀`X.P(X,f(X))`, where `f` is a new function symbol, *provided we agree that we will fix the definition of* `f` as given above.

Skolemisation tells us what existential quantification gives us:

> Existential quantification allows us to use *anonymous* functions in our description.

Alternatively, we can see that if a language gives us the facility to use anonymous functions, then we really don't need existential quantifiers.

SLD4:5 Converting FOL Formulas into Clause Form

A formula is said to be in clause form if it is in the form: `Q1.Q2...Qn.A`, where `A` is a conjunction of clauses, and `Qi` are universal quantifiers.

To convert any first order formula to clause form:

- Express the formula in terms of logical connectives and prime formulas. (i.e. as a propositional logic formula whose propositional symbols represent prime formulas).

- Reduce the propositional formula to clause form.

- For each prime formula that is not atomic, Suppose the prime formula is in the form `Q1.Q2...Qn.A`. (Let `A` be called the *body* of the prime formula).
  Pull out the quantifiers `Q1...Qn`.

- The formula will now be in the form:
  `Q1.Q2...Qn.(C1 AND C2 AND ... AND Cn)`,
  where each `Ci` is a disjunction of atomic formulas and the bodies of prime formulas.

- (Recursively) Perform the normalisation procedure for each `Ci`.

- Pull out any quantifiers that may arise from this.

- Skolemise all the existential quantifiers.

SLD4:6 The resolution rule for propositional logic

For clause form formulas, all proofs can be carried out using only one simple rule called the *resolution rule*.

Consider two clauses `C1` and `C2` such that:
the literal `P` is in `C1` and the literal `(NOT P)` is in `C2`.

> Basically, there is something *conflicting* between `C1` and `C2`.

Clause `C1` is of the form `(P OR C1')` which we can equivalently write as: `(NOT P) IMPLIES C1'`. `C2` is of the form `((NOT P) OR C2')` which we can equivalently write as: `P IMPLIES C2'`.

Thus both `C1` and `C2` together say:

```
IF P THEN C2' ELSE C1'.
```

The "conflict" between `C1` and `C2` is because they rely on two different values of `P` to proceed.

Since we do not the value of `P`, we cannot know which branch to take. But:

> Whichever branch is taken, the formula `(C1' OR C2')` will be true.

Thus the **Resolution Rule** says:

```
P OR C1'          (NOT P) OR C2'
-------------------------------
        C1' OR C2'
```

Resolution is usually used in refutation proofs, i.e. to show that some set of clauses together give a contradiction.

As we saw in SLD1 and SLD2, this is a very useful proof system from the point of view of logic programming.

SLD4:7 An Example of a Resolution Based Proof

Prove that: `NOT Q IMPLIES NOT P` is a logical consequence of `P IMPLIES Q`.

That is, prove: `(P IMPLIES Q) IMPLIES (NOT Q IMPLIES NOT P)`.
In other words, show that `(P IMPLIES Q) AND NOT (NOT Q IMPLIES NOT P)` leads to an empty clause by resolution.

Converting to clausal form:

```
NOT p OR q AND NOT (NOT NOT q OR NOT p)
NOT p OR q AND NOT ( q OR NOT p)

   we get three clauses:

C1= NOT p OR q, C2= NOT q, C3= p
```

Resolving `C1` and `C2` we get: `C4= NOT p`.
Resolving `C4` and `C3` we get: `C3=` $\square$.
Hence we are done.

SLD4:8 Extending to FOL: The Herbrand Universe

First order logic clauses can contain variables. e.g. `p(X) OR (NOT q(X,Y)) OR r(X,X).`

The meaning of such a clause is:

$\forall$`X.`$\forall$`Y.p(X) OR (NOT q(X,Y)) OR r(X,X).`

If we are given another clause:

```
(NOT p(a)) OR r(a,b).
```

Then there is a conflict between this clause and the earlier one.

To see the conflict, substitute `a` for `X` in the first clause:

```
p(a) OR (NOT q(a,Y)) OR r(a,a).
```

Then, we have the required pre-condition for resolution:

> One clause contains `p(a)` and the other its negation `(NOT p(a))`.

This will allow us to derive the following result from the two clauses:

```
(NOT (q(a,Y)) OR r(a,a) OR r(a,b)).
```

Note that resolution has resulted in an universally quantified clause.

Each universally quantified clause actually represents a whole set of clauses. Each element of the set is obtained by making a substitution of "meaningful" terms for variables in the clause.

> What is a meaningful term to substitute for a variable?

The term should be relevant to the clauses participating in the resolution process.

Let S be the set of clauses participating in the resolution. The set of **Herbrand terms**, (also called the **Herbrand Universe**) is the set obtained as follows:

> If '`a`' is any constant symbol appearing in any of the clauses of `S`, then '`a`' is a Herbrand term.
> If `t1,...,tn` are Herbrand terms, and `f` is a function of n arguments that appears in any clause of `S`, then `f(t1,...,tn)` is a Herbrand term.

Thus, the only Herbrand terms that we can make from the two clauses above are `a` and `b`. Thus their Herbrand Universe is {`a`,`b`}.

The set of clauses that are "represented" by the formula:

```
∀X.∀Y.p(X) OR (NOT q(X,Y)) OR r(X,X)
```

in this herbrand universe is:

```
{ p(a) OR (NOT q(a,a)) OR r(a,a),
  p(a) OR (NOT q(a,b)) OR r(a,a),
  p(b) OR (NOT q(b,a)) OR r(b,b),
  p(b) OR (NOT q(b,b)) OR r(b,b)  }
```

Note that all these clauses obtained by substituting Herbrand terms are free of variables, and hence can be treated as propositional clauses. We will call these clauses **ground clauses**.

Thus we can extend resolution to first order logic as follows:

- Let `S` be the set of clauses.
- Let `C1` and `C2` be any two clauses in `S`.
- Let `C1'` be a ground clause that `C1` represents.
- Let `C2'` be a ground clause that `C2` represents.
- Then, we can apply the resolution rule to `C1'` and `C2'`.

The rule can be simplified through unification. We will discuss this in SLD resolution.

SLD4:9 A FOL Example

Consider the following FOL clauses (without any free variables):

```
C1= P(a) C2= NOT D(b) OR L(a,b) C3= NOT
P(a) OR NOT Q(b) OR NOT L(a,b) C4= D(b)
C5= Q(b)
```

Resolving we get:

C6= L(a,b)
From C4, C2.
C7= NOT Q(b) OR NOT L(a,b)
From C3, C1.
C8= NOT L(a,b)
From C5, C7.
C9= □
From C6, C8.

SLD5: SLD Resolution for Propositional Logic

This lecture will show why it is easier to deal with horn clauses for deduction. It will also deal with the SLD resolution technique that provides us with a way of easily automating the proof procedure. The lecture concentrates only on propositional logic.

The lecture also addresses the following topics:

- Goals and subgoals.
- Computation rule.
- Search rule.
- Backtracking.

Background and Motivation:

SLD resolution lies at the heart of logic programming. It has been the main vehicle for understanding the operational behaviour of logic programs. It has also provided the guideline for implementing interpreters and compilers for logic programming languages.

Though propositional logic programs have no practical use, concentrating only on propositional programs helps us focus our attention on the *control* aspects of logic programs. the first order part can then be seen to add the *data* part. Since understanding logic program behaviour mainly amounts to understanding their control part, it is important to study fully the propositional logic case before attending to the additional issues that come out due to first order logic.

Resolution Interpreter

This lecture comes with a floppy containing a Prolog program that implements an interpreter for SLD resolution for propositional logic. The interpreter is called "`sldpi`", and you can start it by running the goal `sldpi`.

`sldpi` allows you to define the computation and search rules that you will want to use. The definitions can be entered in a file that can be read in and installed with the command:

```
loadrules(FileName).
```

The definitions have to be written in Prolog with the help of some auxiliary predicates that `sldpi` provides to access parts of a clause.

A number of example files containing some computation and search functions are also provided for guidance.

Exercises:

The first laboratory exercise accompanying this lecture will consist of trying out the interpreter on the example clause sets given after installing each of the given computation and search rules, and to tabulate the trace of the SLD resolution in a meaningful way, explaining how each step cam about.

The second exercise will consist of finding an example clause set through experimentation that will produce a given trace for a given computation and search rule.

The third exercise will consist of writing a computation and search rule that will produce the given trace for a given set of clauses.

SLD5: SLD Resolution for Propositional Logic

SLD5:1 Horn clauses

Review:
Program Clause: is a clause that contains exactly one positive literal. e.g.

```
married(rahul,X) OR NOT parent(X,indira),
parent(nehru,indira).
```

Goal Clause: is a clause that contains no positive literals. e.g.

```
(NOT married(rahul,X)) OR
(NOT parent(X,indira)),
NOT parent(nehru,indira).
```

Each negative literal in the goal clause constitutes a *subgoal* to be proved.

Horn Clause: is either a program clause or a goal clause. e.g.

```
(NOT married(rahul,X)) OR
(NOT parent(X,indira)),
 parent(nehru,indira).
```

Looking at Propositional Logic: Horn Clauses look like one of the following:

```
Fact:  P.  (A fact)
Rule: P OR (NOT Q1) OR (NOT Q2)
        OR  ......  OR (NOT Qn).
Goal: (NOT Q1) OR (NOT Q2) OR
       ......  OR (NOT Qn).
```

We cannot apply resolution on two facts.
Similarly, we cannot apply resolution on two goals.

So the only ways in which we can apply resolution are:
(1) A fact can be resolved with a rule (Forward reasoning), resulting in another fact or rule.

e.g. `(P)` resolved with `(Q OR (NOT P))` results in `(Q)`.
e.g. `(P)` resolved with `(Q OR (NOT P) OR (NOT R))` results in `(Q OR (NOT R))`.

(2) A rule can be resolved with another rule (Rule expansion) results in another rule.

e.g. `(P OR (NOT Q))` resolved with `(Q OR (NOT R))` results in `(P OR (NOT R))`.

(3) A goal can be resolved with a rule (Backward reasoning). results in another goal.

e.g. `(NOT Q)` resolved with `(Q OR (NOT P))` results in `(NOT P)`.

(4) A goal can be resolved with a fact (subgoal satisfaction). results in a reduced goal or the empty clause (representing a contradiction).

e.g. `(NOT P)` resolved with `(P)` results in `()`.
e.g. `((NOT P) OR (NOT Q))` resolved with `(P)` results in `(NOT Q)`.

Note:
(1) and (2) are unnecessary, as we can replace proofs that use (1) and (2) with proofs that use only (3) and (4).

Example

Converting proof to have only *Backward Reasoning*.
Consider the following prolog facts, rules and a goal:

```
1. a :- b,c.
2. b :- d.
3. c :- e.
4. e.
5. d.
G. ?- a.
```

Proof 1 which picks clauses at random:

```
6.   a :- d,c. from 1, 2.
7.   a :- d,e. from 6, 3.
8.   a :- d.   from 7, 4.
9.   a.        from 8, 5.
10.  EMPTY.    from 9, G.
```

Proof 2 with only backward reasoning (prolog style):

```
6’.  ?- b,c.    from G, 1.
7’.  ?- d,c.    from 6’,2.
8’.  ?- c.      from 7’,5.
9’.  ?- e.      from 8’,3.
10’. ?- EMPTY.  from 9’,4.
```

SLD5:2 Start at the Goal

The resolution rules for propositional Horn Clauses are:

1. For Facts:

```
((NOT P) OR (NOT R1) OR ... OR (NOT Rn)),
     (P)
-----------------------------------------
           (NOT R1) OR ... OR (NOT Rn)
```

2. For Rules:

```
((NOT P) OR (NOT R1) OR ... OR (NOT Rn)),
     (P  OR (NOT Q1) OR ... OR (NOT Qn))
-----------------------------------------
((NOT Q1) OR ... OR (NOT Qn) OR
 (NOT R1) OR ... OR (NOT Rn))
```

The second assumption (a program clause) is called the side clause.

The remaining part of the program clause is substituted for the resolved literal.

Note that both rules have the first assumption as a goal, and both produce a goal as the result. This gives us a clear way of kicking off the resolution process.

Start with one of the goal clauses given.

Since, we cannot resolve a goal clause with a goal clause, this also implies that:

We can use at most one of the given goal clauses in the proof.

This is why it is enough in logic programming to take one query at a time.

This also explains how we should handle disjunctive queries:
Handle each separately.

There are still some ambiguities about how the resolution should proceed after a goal is chosen for starting.

Two specific questions:

1. Which subgoal of the goal clause should be chosen first for solving.

2. Which program clause should be chosen for resolution.

These are addressed next.

SLD5:3 The Computation Rule: Choose a subgoal for proving

An example:
Consider the set of program clauses:

```
(1) P OR (NOT Q) OR (NOT R).
(2) Q OR (NOT R) OR (NOT S).
(3) Q OR (NOT T) OR (NOT S).
(4) R OR (NOT S).
(5) R OR (NOT T).
(6) S.
(7) T.
```

and the goal `G1 = (NOT P).`

Since `G1` has only subgoal, we have to select this one for solving.

Also, the only program clause which can be used for resolution with `G1` is (1).

The result of resolution is:

```
G2:  (NOT Q) OR (NOT R).
```

Now, we have a choice: Do we choose `(NOT Q)` or `(NOT R)` first?

A **computation rule** is a function that maps goal clauses to literals. i.e., we need a computation rule to tell us which literal to pick now.

The following are examples of computation rules:

S1. Pick the leftmost literal from the clause.
This rule is also called the leftmost-first computation rule, and is the rule used in Prolog.

S2. Pick the rightmost literal from the clause.
This rule is also called the rightmost-first computation rule.

S3. Pick the first literal which can be resolved with a fact.
If such a literal cannot be found, choose the leftmost literal.

Thus, S1 will pick `(NOT Q)` and S2 will pick `(NOT R)`. Since neither `Q` nor `R` is a fact, S3 will pick `(NOT Q)`. However, for the goal `(NOT R) OR (NOT S)`, S3 will pick `(NOT S)`.

A key theorem in the theory of logic programming is:

> *The choice of the computation rule does not affect the ability to prove a contradiction from a set of clauses.*

In fact, a more stronger theorem states that:

> *The choice of the computation rule does not affect the length of the shortest proof of contradiction from a set of clauses.*

Thus, in effect, we can fix any computation rule based on convenience, and the proof system can work without any problem.

SLD5:4 The Search Rule: Choosing a Clause for Resolution

Consider the program:

```
(1) P OR (NOT Q) OR (NOT R).
(2) Q OR (NOT R) OR (NOT S).
(3) Q OR (NOT T) OR (NOT S).
(4) R OR (NOT S).
(5) R OR (NOT T).
(6) S.
(7) T.
```

and the goal, `G2 = (NOT Q) OR (NOT R).`

Once, we have picked a subgoal using a fixed computation rule, we need to choose a program clause that can be used for resolution. Thus, if we have S1 as our computation rule, and have picked `(NOT Q)`, we can choose either clause (2) or (3) for resolution.

A **search rule** is a function that maps subgoal literals to program clauses. i.e., we need a search rule to tell us which program clause to pick now.

The following are examples of search rules:

SR1. Pick the topmost program clause that can be used for resolution from the database. This rule is also called the top-first search rule, and is the rule used in Prolog.

SR2. Pick the bottom-most program clause that can be used for resolution from the database. This rule is also called the bottom-first search rule.

SR3. Pick the first fact that can be resolved with the literal. If such a fact cannot be found, choose the topmost program clause.

Thus, SR1 will pick (2) and SR2 will pick (3). Since `(Q)` is not a fact, S3 will pick (1). However, if `(Q)` were listed as a fact, then SR3 would have picked this first.

SLD5:5 An Example

Using search rule S1 and search rule SR1, we get the following proof:

```
G2: (NOT Q) OR (NOT R)
     Select (NOT Q) and (2).
G3: (NOT R) OR (NOT S) OR (NOT R)
     Select (NOT R) and (4).
G4: (NOT S) OR (NOT S) OR (NOT R)
     Select (NOT S) and (6).
G5: (NOT S) OR (NOT R)
     Select (NOT S) and (6).
G6: (NOT R)
     Select (NOT R) and (4).
G7: (NOT S)
     Select (NOT S) and (6).
G8: ()
```

Using search rule S1 and search rule SR2, we get the following proof:

```
G2: (NOT Q) OR (NOT R)
     Select (NOT Q) and (3).
G3: (NOT T) OR (NOT S) OR (NOT R)
     Select (NOT T) and (7).
G4: (NOT S) OR (NOT R)
     Select (NOT S) and (6).
G5: (NOT R)
     Select (NOT R) and (5).
G6: (NOT T)
     Select (NOT T) and (7).
G7: ()
```

Using search rule S2 and search rule SR1, we get the following proof:

```
G2: (NOT Q) OR (NOT R)
     Select (NOT R) and (4).
G3: (NOT Q) OR (NOT S)
     Select (NOT S) and (6).
G4: (NOT Q)
     Select (NOT Q) and (2).
G5: (NOT R) OR (NOT S)
     Select (NOT S) and (6).
G6: (NOT R)
     Select (NOT R) and (4).
G7: (NOT S)
     Select (NOT S) and (6).
G8: ()
```

SLD5:6 Backtracking

Consider now the modified program:

```
(1) P OR (NOT Q) OR (NOT R).
(2) Q OR (NOT R) OR (NOT T).
(3) Q OR (NOT T).
(4) R OR (NOT S).
(5) R OR (NOT T).
(6) T.
```

and the goal `G1 = (NOT P).`

Using search rule S1 and search rule SR1, we get:

```
G2: (NOT Q) OR (NOT R);
   Select (NOT Q) and (2).
G3: (NOT R) OR (NOT S) OR (NOT R);
   Select (NOT R) and (4).
G4: (NOT S) OR (NOT S) OR (NOT R);
   Select (NOT S).
```

Now, we are stuck because there is no clause that we can select to resolve with the selected literal.

The search path is said to have *failed*, but the search itself has not yet failed.
We need to look at the other alternatives.

Now, we **backtrack** to the last point at which another alternative for the search rule is available.

In this case, goal G3 with the selected literal (NOT R) provides another alternative for resolution, namely (5). Hence,

```
G2: (NOT Q) OR (NOT R);
    Select (NOT Q) and (2).
G3: (NOT R) OR (NOT T) OR (NOT R);
    Select (NOT R) and (5).
G4: (NOT T) OR (NOT T) OR (NOT R);
    Select (NOT T) and (6).
G5: (NOT T) OR (NOT R);
    Select (NOT T) and (6).
G6: (NOT R);
    Select (NOT R) and (4).
G7: (NOT S);
    Select (NOT S).
```

Again, we are stuck and we backtrack to G6 and choose (5).

```
G2: (NOT Q) OR (NOT R);
    Select (NOT Q) and (2).
G3: (NOT R) OR (NOT T) OR (NOT R);
    Select (NOT R) and (5).
G4: (NOT T) OR (NOT T) OR (NOT R);
    Select (NOT T) and (6).
G5: (NOT T) OR (NOT R);
    Select (NOT T) and (6).
G6: (NOT R);
    Select (NOT R) and (5).
G7: (NOT T);
    Select (NOT T) and (6).
G8: ()
```

SLD6: Extending SLD Resolution to FOL

This lecture will show how the SLD resolution technique can be extended to Horn clauses of first order logic, and hence can be applied to logic programming.

The lecture also addresses the following topics:

- Generalisation.
- Unification.

Lecture SLD6: Extending SLD Resolution to FOL

SLD6:1 An Example

Consider the program:

```
(1) p(X) OR (NOT q(X))
    OR (NOT r(X,X)).
(2) q(X) OR (NOT r(X,Y))
    OR (NOT s(Y,X)).
(3) q(X) OR (NOT t(X))
    OR (NOT s(X,X)).
(4) r(X,Y) OR (NOT s(a,X)).
(5) r(X,a) OR (NOT t(X)).
(6) s(a,b).
(7) s(a,a).
(8) t(a).
```

and the goal, `G1 = (NOT p(X)).`

Substituting the Herbrand terms {a,b} for variables in the clauses, we get the following ground clauses:

```
(1a)  p(a) OR (NOT q(a))
      OR (NOT r(a,a)).
(1b)  p(b) OR (NOT q(b))
      OR (NOT r(b,b)).
(2aa) q(a) OR (NOT r(a,a))
      OR (NOT s(a,a)).
(2ab) q(a) OR (NOT r(a,b))
      OR (NOT s(b,a)).
(2bb) q(b) OR (NOT r(b,b))
      OR (NOT s(b,b)).
(2ba) q(b) OR (NOT r(b,a))
      OR (NOT s(a,b)).
```

```
(3a)  q(a) OR (NOT t(a))
      OR (NOT s(a,a)).
(3b)  q(b) OR (NOT t(b))
      OR (NOT s(b,b)).
(4aa) r(a,a) OR (NOT s(a,a)).
(4ab) r(a,b) OR (NOT s(a,a)).
(4bb) r(b,b) OR (NOT s(a,b)).
(4ba) r(b,a) OR (NOT s(a,b)).
(5a)  r(a,a) OR (NOT t(a)).
(5)   r(b,a) OR (NOT t(b)).
(6)   s(a,b).
(7)   s(a,a).
(8)   t(a).
G1a:  (NOT p(a)).
G1b:  (NOT p(b)).
```

Let us pick goal G1a, and try SLD resolution with computation rule S1 and search rule SR1.

```
G1a: (NOT p(a));
     Select (NOT p(a)) and (1a).
G2: (NOT q(a)) OR (NOT r(a,a));
    Select (NOT q(a)) and (2aa).
G3: (NOT r(a,a)) OR (NOT s(a,a))
      OR (NOT r(a,a));
    Select (NOT r(a,a)) and (4aa).
G4: (NOT s(a,a)) OR (NOT s(a,a))
      OR (NOT r(a,a));
    Select (NOT s(a,a)) and (7).
```

```
G5: (NOT s(a,a)) OR (NOT r(a,a));
    Select (NOT s(a,a)) and (7).
G6: (NOT r(a,a));
    Select (NOT r(a,a)) and (4aa).
G7: (NOT s(a,a));
    Select (NOT s(a,a)) and (7).
G8: ()
```

Luckily, we were able to complete the proof without using backtracking.

SLD6:2 Generalisation

Let `((NOT p(X)) OR G1)` be a goal clause from which the ground goal clause:
`((NOT p(a)) OR G1'[a/X])` is obtained.

Here `G1'` is a clause obtained by substituting Herbrand terms for variables other than `X` in `G1`, and `G1'[a/X]` denotes the clause obtained by substituting `a` **for** `X` **in** `G1'`.

Similarly, let `(p(X) OR G2)` be a program clause and `(p(a) OR G2'[a/X])` be a ground clause obtained from it.

E.g. `((NOT p(a)) OR (NOT q(a,b)))` is derived from `((NOT p(X)) OR (NOT q(X,Y)))`, and `(p(a) OR (NOT r(a,c)))` is derived from `(p(X) OR (NOT r(X,Y)))`.

Let `(NOT p(a))` be the selected literal from the ground goal clause, and `(p(a) OR G2'[a/X])` be the selected ground program clause.

Then, by SLD resolution, we can derive the goal `(G1'[a/X] OR G2'[a/X])` from the two ground clauses.

For the example, we can derive:
`((NOT q(a,b)) OR (NOT r(a,c)))`.

Similarly, if we had chosen to substitute '`b`' for '`X`' in both the goal and the program clause, we would have derived: `(G1'[b/X] OR G2'[b/X])`.

For the example, we could derive:
`((NOT q(b,b)) OR (NOT r(b,c)))`.

The proof step itself does not depend on which ground substitution is used so long as the two clauses have conflicting literals.

We can thus *generalise* this step to apply for a whole set of ground clauses simultaneously by using the first order clause with variables directly.

The generalised step is:

> Resolve ((NOT p(X)) OR G1') with (p(X) OR G2') to get the ground clause (G1' OR G2').

Note that this step now captures the whole set of resolution steps:

```
((NOT p(a)) OR G1'[a/X])
   with (p(a) OR G2'[a/X]).
((NOT p(b)) OR G1'[b/X])
   with (p(b) OR G2'[b/X]).
...
```

For the example the generalised step is:
Resolve `((NOT p(X)) OR (NOT q(X,b)))` with `(p(X) OR (NOT r(X,c)))` and derive:
`((NOT q(X,b)) OR (NOT r(X,c)))`.

Since the substitutions to the other variables in `G1'` and `G2'` play no role at all in the resolution step, we can generalise further:

> Resolve `((NOT p(X)) OR G1)` with `(p(X) OR G2)` to get `(G1 OR G2)`.

In the example:
Resolve `((NOT p(X)) OR (NOT q(X,Y)))`
with `(p(X) OR (NOT r(X,Y)))`
and derive `((NOT q(X,Y)) OR (NOT r(X,Y)))`.

This is not completely correct!

The same variable (say `Y`) may occur in both `G1` and `G2`.
Whereas, we were able to substitute different values for `Y` in `G1` and `G2`, we will not be able to do so in `(G1 OR G2)`. So we have thus lost some information.

In the example above, this generalisation gives

```
((NOT q(X,Y)) OR (NOT r(X,Y))),
```

from which we cannot get

```
((NOT q(a,b)) OR (NOT r(a,c)))
```

that we could derive earlier.

We thus need to apply a **separating pair of substitutions**, that will make the variable names other than `X` different in `G1` and `G2`, before applying the resolution.

Let `G1''` and `G2''` be the clauses obtained by applying such a substitution.

Then the resolution step in its most generalised form is:

> Resolve `((NOT p(X)) OR G1'')` with `(p(X) OR G2'')` to get the goal clause `(G1'' OR G2'')`.

For the example above, the rule will then allow us to derive:

```
((NOT q(X,Z)) OR (NOT r(X,W))),
```

from which we can get:

```
((NOT q(a,b)) OR (NOT r(a,c))).
```

Here the separating pair of substitutions:
replaces `Y` by `Z` in `(NOT q(X,Y))`
and `Y` by `W` in `(NOT r(X,Y))`.

To summarise, **generalisation** is a method of lifting a proof step to first order logic so that it applies to a set of pairs of clauses simultaneously.

SLD6:3 Generalisation example

Let us look at the generalisation of the proof we did earlier.

```
G1a: (NOT p(a)).;
   Select (NOT p(a)) and (1a).
G2: (NOT q(a)) OR (NOT r(a,a));
   Select (NOT q(a)) and (2aa).
G3: (NOT r(a,a)) OR (NOT s(a,a))
     OR (NOT r(a,a));
   Select (NOT r(a,a)) and (4aa).
G4: (NOT s(a,a)) OR (NOT s(a,a))
     OR (NOT r(a,a));
   Select (NOT s(a,a)) and (7).
```

```
G5: (NOT s(a,a)) OR (NOT r(a,a));
   Select (NOT s(a,a)) and (7).
G6: (NOT r(a,a));
   Select (NOT r(a,a)) and (4aa).
G7: (NOT s(a,a));
   Select (NOT s(a,a)) and (7).
G8: ()
```

of the program:

```
(1) p(X) OR (NOT q(X))
    OR (NOT r(X,X)).
(2) q(X) OR (NOT r(X,Y))
    OR (NOT s(Y,X)).
(3) q(X) OR (NOT t(X))
    OR (NOT s(X,X)).
(4) r(X,Y) OR (NOT s(a,X)).
(5) r(X,a) OR (NOT t(X)).
(6) s(a,b).
(7) s(a,a).
(8) t(a).
```

The Generalisation of the proof yields:

```
G1: (NOT p(X)).;
   Select (NOT p(X)) and (1).
G2: (NOT q(X)) OR (NOT r(X,X));
   Select (NOT q(X)) and (2).
G3: (NOT r(X,Y)) OR (NOT s(Y,X)) OR (NOT r(X,X));
   Select (NOT r(X,Y)) and (4).
G4: (NOT s(a,X)) OR (NOT s(Y,X)) OR (NOT r(X,X));
   Select (NOT s(a,X)) and (7).
G5: (NOT s(Y,a)) OR (NOT r(a,a));
   Select (NOT s(Y,a)) and (7).
G6: (NOT r(a,a));
   Select (NOT r(a,a)) and (4).
G7: (NOT s(a,a));
   Select (NOT s(a,a)) and (7).
G8: ()
```

Note that at each step, the generalisation that we have chosen happened to be the correct one, as it allowed the proof to proceed.

Also note that, the generalised proof is not the one we would have got if we used the generalised proof step, but applied the selection rule and search rule directly. This proof is given next.

The Direct Proof:

```
G1: (NOT p(X)).;
    Select (NOT p(X)) and (1).
G2: (NOT q(X)) OR (NOT r(X,X));
    Select (NOT q(X)) and (2).
G3: (NOT r(X,Y)) OR (NOT s(Y,X))
      OR (NOT r(X,X));
    Select (NOT r(X,Y)) and (4).
G4: (NOT s(a,X)) OR (NOT s(Y,X))
      OR (NOT r(X,X));
    Select (NOT s(a,X)) and (6).
G5: (NOT s(Y,b)) OR (NOT r(b,b));
    Select (NOT s(Y,b)) and (6).
G6: (NOT r(b,b));
    Select (NOT r(b,b)) and (4).
G7: (NOT s(a,b));
    Select (NOT s(a,b)) and (6).
G8: ()
```

The proof differs from the earlier one from G5, where we can select (6) for resolution, unlike the ground case which had to choose (7).

Thus, the SLD resolution from first order logic can be derived from propositional logic, but could have a different control flow.

SLD6:4 Unification

Consider the program:

```
(1) p(X) OR (NOT q(X,f(X))).
(2) q(a,f(f(a))).
(3) q(f(a),f(a)).
(4) q(f(f(a)),f(f(f(a)))).
```

The Herbrand terms are

$\{$`a, f(a), f(f(a)),...`$\}$.

Since this is an infinite set, the set of ground clauses that can be obtained is also infinite.

Besides (2), (3) and (4), the following clauses can be derived:

```
(1a)   p(a) OR (NOT q(a,f(a))).
(1fa)  p(f(a))
       OR (NOT q( f(a), f(f(a)))).
(1ffa) p(f(f(a)))
       OR (NOT q( f(f(a)), f(f(f(a))))).
          ...
```

Consider the goal: `(NOT p(X))`.

Going through the SLD proof on the ground clauses with the computation rule S1 and the search rule SR1:
We first select `(1a)`:
This results in `(NOT q(a,f(a)))`.
This fails, as there is no clause that can be selected for this.

Next, we backtrack and select `(1fa)`:
This results in `(NOT q( f(a), f(f(a))))` .
This also fails, as there is no clause that can be selected for this.

Again, we backtrack and select `(1ffa)`:
This results in `(NOT q( f(f(a)), f(f(f(a)))))`.
This succeeds, as it can be resolved with `(4)`.

Thus, working with ground clauses leads to a search, which can be very costly.

Unification can be used to substantially reduce the search.

Working directly with first order logic:
We start with the goal `(NOT p(X))`.
Resolve it with (1) to get `(NOT q(X, f(X)))`.

Now, we search for a clause that can participate in the resolution. This is done by looking for a clause whose positive literal can unify with the selected literal.

Since `q(a,f(f(a)))` and `q(f(a),f(a))` fail to unify with `q(X,f(X))`, the first clause that succeeds with unification is `(4)`. Unification yields: `X = f(f(a))`.

Thus, we directly complete the proof, and no search for values is required.

SLD6:5 The SLD Resolution Rule for First Order Logic

The resolution step for first order logic is as follows:
Suppose we have a goal clause ((NOT L1) OR G1) and a program clause (L2 OR G2), such that L1 and L2 can be unified.

Assume that the two clauses have no variables in common, otherwise we first make a separating substitution for variables in the two clauses.

Let ρ be the most-general-unifier (mgu) of L1 and L2. i.e. L1[ρ] = L2[ρ].
(G[ρ] denotes the clause obtained by applying the substitution ρ on G).

Then, we can derive (G1[ρ] OR G2 [ρ]) from these two clauses.

Thus the SLD resolution for first order logic works as follows:

Given a set of program clauses `P`, computation rule `S`, search rule `SR`, and a goal `G`:

Start with the goal `G0 = G`, and let $i = 0$.
Repeat the following steps till the empty clause is reached:

> **Choose** a literal `(NOT L)` from `Gi` using the computation rule `S`.
> **Find** the first program clause `C` using `SR` whose positive literal unifies with `L`.
> **If** no such clause can be found, then
>
> > **backtrack** to an earlier goal, where another choice for `SR` exists.
>
> **Resolve** `C` with `Gi` to get `G(i+1)`.
> **Set** i to $i + 1$.

Background and Motivation:

SLD resolution can be extended very easily to first order logic by using unification. This lecture first shows how a propositional logic derivation from ground clauses can be "generalised" to get a derivation in first order logic. Then, the unification based rule is presented as the best compromise between getting an useful derivation and getting a narrow derivation.

SLD resolution interpreter

This lecture comes with a floppy containing a Prolog program that implements an interpreter for SLD resolution. The interpreter is called "`sldpi`", and you can start it by running the goal `sldpi`. The interpreter `sldpi` is an extension of the interpreter `sldpi` that you would have used in the last exercise.

Exercises

The first laboratory exercise accompanying this lecture will consist of trying out the interpreter on the example clause sets given after installing each of the given computation and search rules, and to tabulate the trace of the SLD resolution in a meaningful way, explaining how each step came about.

The second exercise will consist of finding an example clause set through experimentation that will produce a given trace for a given computation and search rule.

The third exercise will consist of writing a computation and search rule that will produce the given trace for a given set of clauses.

SLD7

SLD7: SLD Trees

This lecture will introduce SLD trees, which are diagrammatic representations that "display" all the possibilities available during SLD resolution.

Using SLD trees, we can get a very clear understanding of what is the effect of choosing a computation rule. SLD trees also helps us visualise the search rule as a way of searching in the SLD tree. For example, we can easily see that the top-first search rule is a depth-first search strategy in the SLD tree. This in turn helps us get a picture of why a derivation ends in failure or goes into an infinite loop.

SLD trees also provide a way to understand the more complex control structures implemented in logic programming languages. For example, the "negation as failure" rule and the "cut" of Prolog can be easily understood in terms of SLD trees.

Background and Motivation:

As we have seen before, two rules govern the way SLD resolution works: the computation rule that selects the subgoal to work on, and the search rule that tells us how to search for clauses that can be used for resolution. The choice of the computation rule does not affect the ability to obtain a certain derivation, but it can make a big difference in the searching procedure. So it is essential to get an understanding of what is the effect of choosing a certain computation or search rule.

Lecture SLD7: SLD Trees

SLD7:1 Definition

Let P be a set of program clauses, G a goal clause, and S a computation rule.

The *SLD tree* for $\mathtt{P} \cup \{\mathtt{G}\}$ via S is defined as follows:

- Each node of the tree is a goal (possibly the empty clause).

- The root node is G.

- Let `(NOT A1)OR...(NOT Ak)OR...(NOT An)`, $n > 0$, be a node.
  Let `(NOT Ak)` be the literal selected by `S`.
  Then the node has a descendant corresponding to each program clause `(A OR (NOT B1) OR ... (NOT Bm))` such that `A` and `Ak` can be unified.
  The descendant is

  ```
  ((NOT A1) OR ... (NOT B1)
     OR ... (NOT Bm) OR ... (NOT An))[ρ],
  ```

  where ρ is the most general unifier of `A` and `Ak`.

- Nodes which are the empty clause do not have any descendants.

Each path in the SLD tree is a derivation from $\mathtt{P} \cup \{\mathtt{G}\}$ using the computation rule $\mathtt{S}$.

- Paths which end at the empty clause are called **success paths**.

- Paths which end at a non-empty clause are called **failure paths**.

- Paths which do not have an end are called **infinite paths**.

SLD7:1 Example of Leftmost-First Computation Rule

Consider the program:

```
(1) p(X,Z) OR (NOT q(X,Y))
    OR (NOT p(Y,Z)).
(2) p(X,X).
(3) q(a,b).
```

and the goal:

```
G1:  (NOT p(X,b)).
```

Let the computation rule be the leftmost-first computation rule.