Objective

The aim of today's lecture is to show you how a simple goal-directed theorem prover can be implemented in Prolog.

For example, we want to be able to get a proof for an argument like

$\vdash \neg(A \wedge \neg A)$

simply by typing the following query into Prolog:

```
?- prove( [], neg (a and neg a)).
```



```
?- prove( [], neg (a and neg a)).
  |
  | rewrite in ready for computation form
  |
[]  |--  a and (a implies false)implies false
  |
  | rule for implication
  |
[a, a implies false]  |--  false
  |
  | rule for atoms
  |
[a]  |--  a
  |
  | success
  |
Yes
```

Implementation of the Proof Rules

We implement the *(refined!) goal-directed rules* as a (recursive!) predicate `prove/3`. The first argument holds the list of data formulas, the second one the current goal formula, and the third one the history of previous goals (for restart).

For example, a Prolog goal like

```
prove( [a implies b, a], c, [b, d])
```

should succeed, iff it is possible to show that C follows from $A \Rightarrow B$ and A, assuming that B and D have been previous goals in that particular proof.

When using `prove/3` the history is initially empty, i.e. `[]`.

Example: First Success Rule

The rule "If the current goal A is atomic and can be found in the data, succeed" can be written in Prolog as follows:

```
prove( Data, A, History) :-
  atom( A),
  member( A, Data), !,
  output( Data, A, History, 'success').
```

Here, `output/4` is a predicate that prints out a proof step on the screen according to the information provided by the four arguments.

Example: Rule for Atoms

"To show $\mathcal{P}, A \Rightarrow B \vdash B$ show $\mathcal{P} \vdash A$."

Note that B has to be atomic (or $\bot$); otherwise $A \Rightarrow B$ wouldn't be in ready for computation form.

```
prove( Data, B, History) :-
  atom( B),
  member( A implies B, Data), !,
  output( Data, B, History, 'rule for atoms'),
  delete( Data, A implies B, NewData),
  add( B, History, NewHistory),
  prove( NewData, A, NewHistory).
```

`add/3` adds `B` to `History`, if it is not a member of that list already.

Example: Restart Rule

An atom B can be restarted, if it can be found in the history and if the next proof step can be different from restart, i.e. if we find either B (success) or $X \Rightarrow B$ (rule for atoms) in the data.

```
prove( Data, A, History) :-
  member( B, History),
  (member( B, Data) ; member( _ implies B, Data)), !,
  output( Data, A, History, 'restart'),
  add( A, History, NewHistory),
  prove( Data, B, NewHistory).
```

Remark. Either list this rule after those for conjunction and implication or add `atom( A)` to the beginning of the rule body!

Ready for Computation Form

We implement the *(systematic!) translation function* $()^*$ as a predicate `ready/2`. Given any propositional formula in the first argument position, the variable in the second argument will be matched with a logically equivalent conjunction of formulas in ready for computation form.

Example:

```
?- ready( neg (a and neg b), X).
X = a and (b implies false)implies false
Yes
```


CS2LAP: Logic and Prolog 2000/2001 Implementing a Goal-directed Theorem Prover

Example

Consider the following rule:

$$(A \Rightarrow (B \wedge C))^* \quad = \quad (A \Rightarrow B)^* \wedge (A \Rightarrow C)^*$$

In Prolog this can be written as follows:

```
ready( A implies (B and C), X and Y) :- !,
  ready( A implies B, X),
  ready( A implies C, Y).
```

Recursion Steps and Base Case

For each of the rules defining the translation function $()^*$ a recursive Prolog rule for `ready/2` needs to be implemented.

The set of all formulas that already are conjunctions of formulas in ready for computation form constitutes the base case. In such a case the formula doesn't have to be changed anymore and the translation process terminates. Therefore, the very last of the Prolog clauses (corresponding to the base case of the recursion) has to be:

```
ready( A, A).
```

Note that in all other rules we have to use a cut to avoid incorrect alternative solutions.

The Program

You can download an incomplete version of the program from the course web site. It consists of four files:

`main.pl`: operators; main predicate `prove/2` (which uses `ready/2` and `prove/3`)

`prover.pl`: `prove/3`; `output/4` for printing one proof step

`ready.pl`: `ready/2`; `ready_list/2` to translate a list of formulas

`auxiliary.pl`: implementation of two auxiliary predicates

Summary

- Unlike most other programming languages, Prolog is very well suited for fast prototyping of automated theorem provers.
- Each goal-directed rule corresponds to a Prolog rule of `prove/3`. Care needs to be taken that we implement the refined version for which *termination* is guaranteed.
- Each translation rule corresponds to a Prolog rule of `ready/2`. In order to allow for automation we have to make sure that every possible formula pattern is captured by our translation rules.
- Use this week's lab sessions to understand and complete the program.