Lecture 9
1) Cut & Negation in PROLOG

2) Alternative semantics for negation: Perfect, Well-founded and Stable Models

20/12/2006
Cut: 

- Example: 2 implementations of \texttt{max}:

\begin{verbatim}
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y) :- X < Y.
\end{verbatim}

- Goal: cut down search space, gain efficiency
- Disadvantage: against declarativity.
The working of the cut operator:

Clause C: \( A : - B_1, ..., B_k, !, B_{k+1}, ..., B_n \)

Goal: ?- Z.

- If A unifies with Z and subgoals \( B_1, ..., B_k \) can be proven, then ! succeeds and fixes the solutions to the clause C, i.e. no further backtracking for any alternative wrt. C (below C in the program).

- \( B_{k+1}, ..., B_n \) are proven normally, but in case \( B_i \) (i>k) fails, backtracking only upto '!'  

- If backtracking reaches '!' a second time then it fails and computation returns to the point before clause Z C was chosen.

*Behavior: if ... then ... elseif ... else, can be emulated.*
Cut

• ... in other words:
  – '!' cuts off all alternative clauses which follow
  – '!' cuts off all alternatives on its left.
  – '!' does not influence subgoals on its right.
Example:

% mymerge(X,Y,Z) merges two sorted lists X and Y:

mymerge([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, mymerge(Xs,[Y|Ys],Zs).
mymerge([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, mymerge(Xs,Ys,Zs).
mymerge([X|Xs],[Y|Ys],[Y|Zs]) :- X > Y, mymerge([X|Xs],Ys,Zs).

mymerge(Xs,[]),Xs).
mymerge([],Ys,Ys).

- Backtracking over exclusive alternatives can be avoided with cuts:

% mymerge(X,Y,Z) merges two sorted lists X and Y:

mymerge([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, !, mymerge(Xs,[Y|Ys],Zs).
mymerge([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, !, mymerge(Xs,Ys,Zs).
mymerge([X|Xs],[Y|Ys],[Y|Zs]) :- X > Y, !, mymerge([X|Xs],Ys,Zs).

mymerge(Xs,[]),Xs) :- !.
mymerge([],Ys,Ys) :- !.

In summary: Cut makes sense if you model some deterministic choices, but a bit dirty ;-) compared to pure Prolog.
What the cut is usable for:

- Useful to make programs more efficient.
- Sometimes useful to avoid additional or duplicate answers,
- But often a sign of "dirty" non-declarative PROLOG hacking and should be avoided.
- You should know what you're doing and have to understand the working of PROLOG when using cuts!
Negation (not or \+) in Prolog.

- Prolog has the built-in fail which never succeeds.
- This can be used to emulate a restricted form of negation, so called "negation as finite failure".

- Recall: whenever PROLOG could not find a solution to a query in finite time, it answered 'No'.

- We also want to reuse this in rules... for this, there exists the predicate not also written \+
- can be emulated more or less as follows:
  
  not X :- X, !, fail.
  not X.
Negation as failure and SQL:

We said in Lecture 4 that we can use PROLOG as a Query language similar to SQL… Now we can also express negative queries:

"Give me all persons without a father"

SELECT name FROM person p WHERE NOT EXISTS (SELECT * FROM child_of c, male m WHERE c.child=p.name AND c.parent=m.name);

In PROLOG:
no_father(X) :- person(X), \+ has_father(X).
has_father(X) :- child_of(X,Y), male(Y).
Problems with this form of not:

- `not` in PROLOG often written `\+` does not correspond with classical negation!!!

Example: 

```
a :- not b.
```

Minimal Herbrand Models: `{a} {b}`

- i.e., Success of `{not G}` does not mean: $P \models \neg G$ but: $P \not\models G$
Non-monotonic reasoning 1: Default rules

- This form of negation allows some limited form of non-monotonic reasoning.
- Classical logic is monotonic, i.e. whenever I add knowledge, the set of consequences increases.
- Horn Logic is classical, i.e. monotonic.

- This is not the case if negation as failure is added to Horn logic!
- In non-monotonic reasoning, previous conclusions can be invalidated by additional knowledge.
- Non-monotonic reasoning often important in common-sense reasoning: "Default" reasoning

Example: "Birds normally fly, unless they are penguins"

\begin{verbatim}
flys(X) :- bird(X), \+ penguin(X).
bird(tweety).
\end{verbatim}

Does Tweety fly?
What if I add \texttt{penguin(tweety)}. to the facts?
Non-monotonic reasoning 2: Closed World Assumption

- Everything which is not explicitly known, is assumed to be false.
- Example: Train Schedule.

- This is the motivation for using a minimal model semantics.

- ?- \+ train(vienna, bregenz, 0500, X).

- ‘No’ means there really is no such train under the closes world assumption.
All the examples had

**Negation as failure in rule bodies/queries:**

- Prolog makes a “practical” assumption about this: Negation as (finite) failure to proof.
- What happens to the Semantics? (rule with negation in the body are no longer Horn)
- PROLOG cannot deal with negation and recursion at once!
Alternative semantics for negation: Perfect, Well-founded and Stable Models

• We will now try to define formal semantics for programs with non-monotonic negation in rule bodies!

• To keep things simple, we now talk about function-free programs only, i.e., no nested terms.

• We learned already that the Herbrand Base is finite for such programs.

• For-such programs, the $T_{P}^{\infty}(\emptyset)$ operator defines an algorithm to compute the minimal Herbrand model:
  – First ground the program using the Herbrand Base
  – Then compute (in finite time) the minimal Herbrand model
Bottom-up computation:
Alternative definition of $T_P$:

Let $I$ be a Herbrand interpretation and $\Phi$ a definite program:
We define by $\text{Ground}(\Phi)$ the set of all ground instances of rules of $\Phi$

$T_P(I) = \{ A \in B_P : A \leftarrow A_1, \ldots A_n \text{ is a rule in } \text{Ground}(\Phi) \text{ such that } A_1, \ldots, A_n \in I \}$

Since HB(P) is finite, also $\text{Ground}(\Phi)$ is finite
The ground Instantiation of a program:

\[ \mathcal{P}_r = \{ \text{arc}(a, b), \text{arc}(b, c), \text{reachable}(a), \text{reachable}(Y) \leftarrow \text{arc}(X, Y), \text{reachable}(X). \} \]

\[ \text{HU}(\mathcal{P}_r) = \{ a, b, c \} \]

\[ \text{HB}(\mathcal{P}_r) = \{ \text{arc}(a, a), \text{arc}(a, b), \text{arc}(a, c), \text{arc}(b, a), \text{arc}(b, b), \text{arc}(b, c), \text{arc}(c, a), \text{arc}(c, b), \text{arc}(c, c), \text{reachable}(a), \text{reachable}(b), \text{reachable}(c) \} \]
Ground($\mathcal{P}$) – The ground Instantiation of a program:

\[
Ground(\mathcal{P}_r) = \{ \text{arc}(a, b), \text{arc}(b, c), \text{reachable}(a).
\]

\[
\begin{align*}
\text{reachable}(a) & \leftarrow \text{arc}(a, a), \text{reachable}(a). \\
\text{reachable}(b) & \leftarrow \text{arc}(a, b), \text{reachable}(a). \\
\text{reachable}(c) & \leftarrow \text{arc}(a, c), \text{reachable}(a). \\
\text{reachable}(a) & \leftarrow \text{arc}(b, a), \text{reachable}(b). \\
\text{reachable}(b) & \leftarrow \text{arc}(b, b), \text{reachable}(b). \\
\text{reachable}(c) & \leftarrow \text{arc}(b, c), \text{reachable}(b). \\
\text{reachable}(a) & \leftarrow \text{arc}(c, a), \text{reachable}(c). \\
\text{reachable}(b) & \leftarrow \text{arc}(c, b), \text{reachable}(c). \\
\text{reachable}(c) & \leftarrow \text{arc}(c, c), \text{reachable}(c). \\
\}
\]
Normal logic programs

• Negation in the body allowed, rules of the form:

\[ h \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n. \]
\[ 1 \leq m \leq n \]

\[ B^+(r) = \{b_1, \ldots, b_m\} \]
\[ B^-(r) = \{\text{not } b_{m+1}, \ldots, \text{not } b_m\} \]
\[ B(r) = B^+(r) \cup B^-(r) \]

Recall: we already had one form of negation (negation as failure) in Prolog!
Normal Logic Programs

• Recall: Problems with Semantics:

• In general there is no unique minimal Herbrand Model anymore:
  \[ a \leftarrow \text{not } b. \]

• Two Herbrand Models: \( M_1 = \{a\}, M_2 = \{b\}, \)
  \( M_2 \) is less intuitive.
Normal Logic Programs

- Negation as failure in Prolog was fine, as long as negation was non-recursive, but we had problems with evaluating things like:

\[
\text{male}(X) \leftarrow \text{person}(X), \neg \text{female}(X).
\]

\[
\text{female}(X) \leftarrow \text{person}(X), \neg \text{male}(X).
\]

- When evaluating this bottom-up, we would get an alternating fixpoint.

- Practical solution: forbid recursion over negation!
Stratified Programs:

• Let the dependency graph be defined as in the previous lecture:
  
  • Nodes: Predicates
  • Edges: for each rule from head to body literals.
  • Edges with negation are marked
  • Components: maximal sets of nodes such that each node is reachable from each other node.
  • Partial order between components is induced by the edges.

• A program is called **stratifiable**, if there are no cycles with a marked (negative) edge.
Example:
Stratifiable vs. non-stratifiable

- Non-stratifiable:
  \[ a \leftarrow b, c. \]
  \[ c \leftarrow \neg b. \]
  \[ b \leftarrow a \]

- Stratifiable:
  \[ a \leftarrow b. \]
  \[ c \leftarrow \neg b. \]
  \[ b \leftarrow a \]
The Perfect Model:

- Components induced by the dependency graph are inducing a stratification, i.e. you can evaluate stratifiable programs in a leveled fashion, first grounding, where negation only occurs between the levels.

- This stratification implies an order for the evaluation. Similar idea as component-wise evaluation.

- Let \( \langle P_1, \ldots, P_n \rangle \) be the strata (levels) of a stratifiable normal program \( P \), then the sequence:

\[
M_1 = T_{P_1}^\infty(\emptyset), \quad M_2 = T_{P_2}^\infty(M_1), \ldots, \quad M_n = T_{P_n}^\infty(M_{n-1})
\]

defines the **perfect model** \( M_n \) of \( P \).
The Perfect Model:

• Operator $T_P$ has to be slightly modified, since $P$ can now contain negation in rule bodies:

$$T_P(I) = \{ A \in B_P : A \leftarrow A_1, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_m \text{ is a rule } r \text{ in } \text{Ground}(\mathcal{P}) \text{ such that } B^+(r') \subseteq I \text{ and } B^-(r') \cap I = \emptyset \}$$
The Perfect Model:

- Perfect Model Semantics only defined for stratifiable programs
- Each stratifiable program has a unique perfect Model
- Non-recursive Programs are always stratifiable
- Remark: Non-recursive safe programs with negation under the perfect model semantics have the same expressivity as Relational Algebra.
- Componentwise evaluation methods as shown last lecture are directly applicable.
Non-stratifiable programs:

\[\text{person}(\text{nicola}).\]
\[\text{alive}(X) \leftarrow \text{person}(X).\]
\[\text{male}(X) \leftarrow \text{person}(X), \neg \text{female}(X).\]
\[\text{female}(X) \leftarrow \text{person}(X), \neg \text{male}(X).\]

The perfect model is not defined here, but at least we would like to conclude \text{alive}(\text{nicola}).
How to proceed with non-stratifiable Programs, i.e. recursive negation?

- Partial Interpretations
- Unfounded sets
- Well-founded Model Semantics
- Stable Model Semantics
Recursive Negation:

person(nicola).
alive(X) ← person(X).
male(X) ← person(X), ¬ female(X).
female(X) ← person(X), ¬ male(X).

What happens if we apply $T_P$?

$$T_P(\emptyset) = \{\text{person(nicola)}, \text{alive(nicola)}, \text{male(nicola)}, \text{female(nicola)}\}$$

$$T_P(T_P(\emptyset)) = \{\text{person(nicola)}, \text{alive(nicola)}\}$$

$$T_P(T_P(T_P(\emptyset))) = T_P(\emptyset)$$

$$T_P(T_P(T_P(T_P(\emptyset)))) = T_P(T_P(\emptyset))$$

$$T_P(T_P(T_P(T_P(T_P(\emptyset)))))) = T_P(\emptyset)$$

...
Recursive Negation:

- **But:** There are two fixpoints of $T_P$:

  \[
  T_P(\{\text{person(nicola), alive(nicola), male(nicola)}\}) = \\
  \{\text{person(nicola), alive(nicola), male(nicola)}\} \\
  T_P(\{\text{person(nicola), alive(nicola), female(nicola)}\}) = \\
  \{\text{person(nicola), alive(nicola), female(nicola)}\}
  \]

- **Two ways to deal with this in the semantics:**
  1. don't state anything about $\text{male(nicola)}$ and $\text{female(nicola)}$
  2. accept both possible scenarios $\text{male(nicola)}$ and $\text{female(nicola)}$

Needs an additional truth-value: $\{\text{true, false, unknown}\}$ to express that no statement is made on a certain atom.

Allow several models, one where $\text{male(nicola)}$ holds and one where $\text{female(nicola)}$ holds.
Three-valued interpretation:

- Literals are of the form: $a$ or $\text{not } a$
  (where $a$ is an atom)
- A set of literals is called consistent iff
  $L \cap \text{not } L = \emptyset$ *
  thus if no atom occurs positively and negatively.
- A three-valued interpretation is a consistent set of ground literals.

Example: $I = \{\text{not } a, c\}$
  - $a$ is $\text{false}$ wrt. $I$
  - $b$ is $\text{unknown}$ wrt. $I$
  - $c$ is $\text{true}$ wrt. $I$

* $\text{not } l$ is defined for $l=a$ as $\text{not } l$ and for $l=\text{not } a$ as $l$
Three-valued interpretation:

Difference to Herbrand Interpretations:

- Negative Information explicitly mentioned.
- Negative information has to be explicitly derived during Fixpoint-Computation.
- No direct consequences from undefined literals in the rule body!
Unfounded Sets - Example

• **Goal:** Derive as much negative information as possible.

$$ a \leftarrow \text{not } b. $$

*b doesn't occur in any rule head*

⇒ **thus** *b cannot be true*

⇒ **thus** *a is true.*

"**Intended interpretation**"  $ I = \{\text{not } b, a\}$
Unfounded Sets - Example

**Goal:** Derive as much negative information as possible.

\[ a \leftarrow b. \]

\[ c \leftarrow \text{not } a. \]

*b doesn't occur in any rule head*

\( \rightarrow \) *thus b cannot be true*

\( \rightarrow \) *thus b is false*

\( \rightarrow \) *thus a cannot be made true*

\( \rightarrow \) *thus a is false*

\( \rightarrow \) *Thus c should be true*

*"Intended interpretation"* \( I = \{ \text{not } a, \text{not } b, c \} \)
Unfounded Sets - Example

- **Goal:** Derive as much negative information as possible.

\[
\begin{align*}
  a & \leftarrow b. \\
  b & \leftarrow a. \\
  c & \leftarrow \text{not } a.
\end{align*}
\]

*a occurs in a rule head, but can only be caused "by itself"

→ thus a cannot be true (same for b)
→ thus c is true.

"Intended interpretation" \( I = \{ \text{not } a, \text{not } b, c \} \)
Unfounded Sets - Definition:

• The previous scenarios should be covered wrt. to a definition of what the "intended interpretation" should be:

• A Set $U \subseteq B_{\emptyset}$ is called **unfounded** wrt. a partial interpretation $I$ if:

  $\text{For every atom } a \in U \text{ and any rule } r \in \text{Ground}(\hat{\mathcal{F}}) \text{ with head } a \text{ one of the following conditions holds:}$

  1. $\exists l \in B(r) : \text{not}.l \in I$
  2. $B^+(r) \cap U \neq \emptyset$

  Informally: "For each element of $U$ there is no rule which justifies believing $a."
Unfounded Sets – examples:

\[ a \leftarrow \text{not } b. \]

With \( I=\emptyset \) \( \{b\} \) is an unfounded set.

\[ a \leftarrow b. \]
\[ c \leftarrow \text{not } a. \]

With \( I=\{\text{not } b\} \) \( \{a\} \) is an unfounded set, due to condition 1. \{a,c\} is not unfounded wrt. \textit{I since neither condition holds for the second rule.}

\[ a \leftarrow b. \]
\[ b \leftarrow a. \]
\[ c \leftarrow \text{not } a. \]

With \( I=\emptyset \) \( \{a,b\} \) is an unfounded set, due to condition 2.
Greatest unfounded Set:

- **Theorem**: There is always a greatest unfounded set $GUS_{\mathcal{A}}(I)$ which contains all other unfounded sets.

- **Idea**: Use $GUS_{\mathcal{A}}(I)$ to derive negative information.

- **Definition**:
  
  Operator $U_{\mathcal{P}}(I) := \{ \text{not}.a \mid a \in GUS_{\mathcal{P}}(I) \}$
Well-Founded Operator

• Now we generalize $T_{\mathcal{I}}(I)$ to three-valued interpretations:

$$T_{\mathcal{I}}(I) = \{ A \in B_P : A \leftarrow A_1, \ldots, A_n \text{ is a rule } r \text{ in } \text{Ground}(\mathcal{P}) \text{ such that } B(r) \in I \}$$

• Now we define the well-founded Operator $W_{\mathcal{I}}(I)$ as a combination of $T_{\mathcal{I}}(I)$ and $U_{\mathcal{I}}(I)$:

Definition:

$$W_{\mathcal{P}}(I) := T_{\mathcal{P}}(I) \cup U_{\mathcal{P}}(I)$$
Well-Founded Model

Allen Van Gelder, Kenneth Ross, John Schlipf 1988
Well-founded Model:

• Theorem:
  \( \mathcal{W}_\mathcal{F} \) is monotonic, thus, there exists a least fixpoint \( \mathcal{W}^\infty_{\mathcal{F}}(\emptyset) \)

\( \mathcal{W}^\infty_{\mathcal{F}}(\emptyset) \) is called the **Well-Founded Model** of a normal Program \( \mathcal{F} \)

A three-valued Interpretation \( I \) is called **total** if no atom has the value "unknown", i.e. each element of \( \mathcal{B}_\mathcal{F} \) is either assigned true or false.
Well-founded Model:

- **Theorem:** Each Program has a unique well-founded model

- **Theorem:** The well-founded model for definite (negation-free logic programs is total and corresponds to the least Herbrand-Interpretation

- **Theorem:** The well founded model of a stratifiable normal logic program is total and corresponds to the perfect model
Well-founded Model: Example

\[
\begin{align*}
\text{person(nicola).} \\
\text{alive}(X) & \leftarrow \text{person}(X). \\
\text{male}(X) & \leftarrow \text{person}(X), \text{not female}(X). \\
\text{female}(X) & \leftarrow \text{person}(X), \text{not male}(X).
\end{align*}
\]

- The well founded model is not total:

\[
\{\text{person(nicola), alive(nicola)}\}
\]
Similar to the previous program, but:

\[
\begin{align*}
\text{male}(X) & \leftarrow \text{person}(X), \textbf{not} \ \text{female}(X). \\
\text{female}(X) & \leftarrow \text{person}(X), \textbf{not} \ \text{male}(X). \\
\text{alive}(X) & \leftarrow \text{female}(X). \\
\text{alive}(X) & \leftarrow \text{male}(X). \\
\text{person}(\text{n}icola). \\
\end{align*}
\]

Similar to the previous program, but:

The well-founded model only consists of \{\text{person}(\text{n}icola)\}

Intuitively, many people would also expect \text{alive}(\text{n}icola) to be true.
Stable Models

Michael Gelfond, Vladimir Lifschitz 1988
Stable Models

- Allow more than one model
- Stability condition instead of Fixpoint-Semantics
Gelfond-Lifschitz Transformation:

- The GL-Transformation $\hat{\mathcal{P}}^I$ of a program $\mathcal{P}$ wrt. a total interpretation $I$ is defined by transforming $\text{Ground}(\hat{\mathcal{P}})$ as follows:

1. Remove all rules $r$ for which $B^-(r) \cap I \neq \emptyset$ holds.
2. Remove $B^-(r)$ from all remaining rules.

⇒ From this transformation you achieve a definite (negation-free) program!
GL-Transformation: Example

\[ \mathcal{P} = \{ \text{male}(n) \leftarrow \text{not female}(n). \}
\]

\[ \text{female}(n) \leftarrow \text{not male}(n). \} \]

\[ I_1 = \emptyset, \mathcal{P}^{I_1} = \{ \text{male}(n). \text{female}(n). \} \]
\[ I_2 = \{ \text{male}(n) \}, \mathcal{P}^{I_2} = \{ \text{male}(n). \} \]
\[ I_3 = \{ \text{female}(n) \}, \mathcal{P}^{I_3} = \{ \text{female}(n). \} \]
\[ I_4 = \{ \text{male}(n). \text{female}(n) \}, \mathcal{P}^{I_4} = \emptyset \]
Stable Models

- **Observation**: $\hat{\Phi}^I$ is a definite (negation-free) program, thus it has a least Herbrand model.

- **Definition**: A total Herbrand interpretation $M$ is called **stable model** of $\hat{\Phi}$ if $M$ is the least Herbrand model of $\hat{\Phi}^M$. 
Stable Model: Examples

\[ P = \{ \text{male}(n) \leftarrow \text{not female}(n). \]
\[ \quad \text{female}(n) \leftarrow \text{not male}(n). \}\]

\[ I_1 = \emptyset, P^{I_1} = \{\text{male}(n), \text{female}(n).\}, \text{MM}(P^{I_1}) \neq I_1 \]
\[ I_2 = \{\text{male}(n)\}, P^{I_2} = \{\text{male}(n).\}, \text{MM}(P^{I_2}) = I_2 \]

\[ I_3 = \{\text{female}(n)\}, P^{I_3} = \{\text{female}(n).\}, \text{MM}(P^{I_3}) = I_3 \]

\[ I_4 = \{\text{male}(n), \text{female}(n)\}, P^{I_4} = \emptyset, \text{MM}(P^{I_4}) \neq I_4 \]

\[ I_2 \text{ and } I_3 \text{ are stable models.} \]
Stable Model: Examples

\[ \mathcal{P} = \{ \text{weird} \leftarrow \text{not weird.} \} \]

\[ I_1 = \emptyset, \mathcal{P}^I_1 = \{ \text{weird.} \}, \text{MM}(\mathcal{P}^I_1) \neq I_1 \]

\[ I_2 = \{ \text{weird} \}, \mathcal{P}^I_2 = \emptyset, \text{MM}(\mathcal{P}^I_2) \neq I_2 \]

Has no stable model!
Stable Models:

- Each normal datalog program has one, several or no stable models.

- **Theorem:** On stratifiable programs stable model semantics, well-founded semantics and perfect model semantics coincide, i.e. there is a single stable model which is equal to the perfect model.

- On the whiteboard:
  - The example from slide 30.
  - An elegant formulation of 3-colorability.
Answer Set Programming:

- Stable Models are the basis for a powerful logic programming paradigm, as an alternative to non-declarative PROLOG: ANSWER SET PROGRAMMING (ASP)

- ASP = LP under stable model semantics plus useful extensions
  - More in the rest of my lectures after Christmas!
Feliz navidad & muchos regalos!

;(