Chapter 4. Operational Semantics

- Last updated: May 11, 2016 -

Refutation Proofs
Resolution Principle
Unification
SLD-Resolution
Negation as Failure
Incompleteness of SLD-Resolution
Refutation via Resolution

Refutation Proofs
Resolution
Translation
Role of Unification
Proof by Refutation

- **Undecidability** of first order logic
  - There is no automated proof system that always answers **yes** if a goal is provable from the available clauses and answers **no** otherwise.

- **Semi-decidability** of first order logic
  - It is possible to determine unsatisfiability of a formula by showing that it leads to a contradiction (an empty clause)

**Implication of Semi-Decidability**

- We cannot prove a goal directly but must show that adding the negation of the goal to the program $P$ makes $P$ unsatisfiable
  
  $$P \models G \quad \text{is proven by showing that} \quad (P \cup \neg G) \models \{\}$$

- Proving a formula by showing that its negation is wrong is called **proof by refutation**.
Refutation via Resolution

Prolog does **refutation** proofs by

1. Translating the program to a set (conjunction) of Horn clauses
2. Translating the query to a Horn clause
3. Negating the query and adding it to the program
4. Proving unsatisfiability of the formula corresponding to the extended program.

Unsatisfiability of a set of Horn clauses (step 4.) is proved by a process called **resolution**.
Resolution via Horn Clauses

This section is just for the particularly interested students (not an exam topic). It explains the logic background of what Prolog does. You do not need to know this.

For those who have a look at it, remember that the resolution of one positive literal with one negative literal that is performed to derive a logical contradiction corresponds to the resolution of a goal with a clause head at the level of Prolog.
Translation of Prolog Programs to First Order Logic (repeated)

- A Prolog program is translated to a set of formulas, with each clause in the program corresponding to one formula:

\[
\begin{align*}
\{ & \text{bigger( elephant, horse ),} \\
& \text{bigger( horse, donkey ),} \\
& \forall x. \forall y. ( \text{bigger}(x, y) \rightarrow \text{is}_\text{bigger}(x, y) ), \\
& \forall x. \forall y. ( \exists z. ( \text{bigger}(x, z) \land \text{is}_\text{bigger}(z, y)) \rightarrow \text{is}_\text{bigger}(x, y) ) \\
\}
\end{align*}
\]

- Such a set is to be interpreted as the conjunction of all the formulas in the set:

\[
\begin{align*}
& \text{bigger( elephant, horse ) } \land \\
& \text{bigger( horse, donkey ) } \land \\
& \forall x. \forall y. ( \text{bigger}(x, y) \rightarrow \text{is}_\text{bigger}(x, y) ) \land \\
& \forall x. \forall y. ( \exists z. ( \text{bigger}(x, z) \land \text{is}_\text{bigger}(z, y)) \rightarrow \text{is}_\text{bigger}(x, y) )
\end{align*}
\]
Horn Clauses

- The formula we get when translating a Prolog clause has the structure:

\[ a_1 \land a_2 \land \cdots \land a_n \rightarrow b \]

- Such a formula can be rewritten as follows:

\[
\begin{align*}
  a_1 \land a_2 \land \cdots \land a_n & \rightarrow b \\
\neg(a_1 \land a_2 \land \cdots \land a_n) & \lor b \\
\neg a_1 \lor \neg a_2 \lor \cdots \lor \neg a_n & \lor b
\end{align*}
\]

- By law \( a \rightarrow b \equiv \neg a \lor b \) we get:

\[
\neg(a_1 \land a_2 \land \cdots \land a_n) \lor b
\]

- By law \( \neg(a \land b) \equiv \neg a \lor \neg b \) we get:

\[
\neg a_1 \lor \neg a_2 \lor \cdots \lor \neg a_n \lor b
\]

- Hence, every Prolog clause can be translated as a disjunction of negative literals with at most one positive literal.

- This is called a Horn clause.
Horn Clauses ▶ Relevance

- Expressiveness
  - Every closed first order logic formula can be translated to Horn clause form.
  - This translation preserves (un)satisfiability: If the original formula is (un)satisfiable, the translated one is (un)satisfiable too and vice versa.

- Efficiency
  - Satisfiability is the problem of determining if the variables of a Boolean formula can be assigned in such a way as to make the formula true.
    - Satisfiability is an NP-complete problem. ☹
  - There exists an efficient automated way to prove the unsatisfiability of a set of Horn clauses: SLD-Resolution.
  - It is the basis for practical implementations of Prolog interpreters and compilers.
Translation of Closed First Order Logic Formulas to Horn Clause Form

The translation of FOL formulas that are closed (= only contain variables bound by quantifiers) proceeds in four steps:

1. **Variable Disjunct Form**: If different quantors bind variables with the same name, rename the variables so that each is named differently.

2. **Elementary Junctor Form**: Translate implication to disjunction as shown two slides ago.

3. **Prenex Normal Form**: Replace negated quantifiers by their opposite (e.g. \( \neg \exists x \) by \( \forall x \)) and move all remaining positive quantifiers to the front.

4. **Skolemized Form (implicit \( \forall \))**: Replace all existentially quantified variables by some unique constant. Eliminate remaining \( \forall \) quantors.

5. **Horn Clause Form (implicit \( \forall \))**: Turn remaining conjunction into a set of formulas, where each formula is a Horn clause.

**Example**: See next slide.

**Exercise**: Translate the formula shown two slides ago to Horn clauses.
Proof by Refutation via Resolution

- **Formula that we want to prove**
  
  \[(\exists x \forall y. R(x, y)) \rightarrow \forall y \exists x. R(x, y)\]

- **Its negation**
  
  \[-(\exists x \forall y. R(x, y)) \rightarrow \forall y \exists x. R(x, y)\]

- **Variable Disjunct Form**

- **Elementary Junctor Form**

- **Prenex Normal Form**

- **Skolemized Form (implicit \(\forall\))**

- **(Horn) Clause Form (implicit \(\forall\))**

- **Unification with mgu \(\{ w \leftarrow c_0, y \leftarrow c_1 \}\)**

- **Resolution of clause 2 with clause 1**

  \[\{\{R(c_0, c_1)\}, \{\neg R(c_0, c_1)\}\}\]

  **contradiction \(\rightarrow\) \{\}**
Chapter 4: Operational Semantics

Unification

Equality
Variable bindings, Substitutions, Unification
Most general unifiers
Resolution and Unification

- **Resolution** is the process by which Prolog derives answers (successful substitutions) for queries.

- During **resolution**, clauses that can be used to prove a goal are determined via **unification**.

```
?- isFatherOf(paul,Child).
```

This step corresponds to the resolution of a negative literal with a positive one in the Horn Clause form (prev. section).

```
isFatherOf(F, C) :- isMarriedTo(F,M), isMotherOf(M,C).
```

**Unification is the basic operation of any Prolog interpreter.**
Equality (1)

Testing equality of terms

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?- europe = europe.</code></td>
<td>yes</td>
</tr>
<tr>
<td><code>?- 5 = 2.</code></td>
<td>no</td>
</tr>
<tr>
<td><code>?- 5 = 2 + 3.</code></td>
<td>no</td>
</tr>
<tr>
<td><code>?- 2 + 3 = +(2, 3).</code></td>
<td>yes</td>
</tr>
</tbody>
</table>

- Terms are not evaluated!

- Terms are equal if they are **structurally equal**!!

- **Structural equality** for ground terms:
  - functors are equal and ...
  - ... all argument values in the same position are structurally equal.

**Constants are just functors with zero arity!**
Unifiability (2)

Testing equality of terms with variables

?- person(peter, Name, date(27, 11, 2007))
   = person(peter, mueller, date(27, MM, 2007)).

- These terms are obviously not equal.

Idea

- A variable can take on any value.
- If we substitute Name by mueller and MM by 11 the two terms above will be equal.

Principle

- Equality = terms are equal
- Unifiability = terms can be made equal via a substitution.

Prolog doesn’t test equality but unifiability!
Unifiability (3)

Two terms are unifiable if there is a substitution that makes them equal!

```
?- person(peter, Name, date(27, 11, 2007))
  =
  person(peter, mueller, date(27, MM, 2007)) .
```

Definitions: Bindings, substitutions and unifiers

- A binding is an association of a variable to a term different from the
  - Name ← mueller
  - Y ← Z  ← The right-hand-side term can be a variable!
  - MM ← month(M)  ← The right-hand-side term can contain variables!

- A substitution is a set of bindings with pairwise different variables
  - \{Name ← mueller, MM ← 11\}

- A unifier is a substitution that makes two terms equal
  - The above substitution is a unifier for the two person/3 terms above
Unifiability (2)

- Can you find out the unifiers for these terms?

\[
\begin{align*}
\text{date}(1, 4, 1985) &= \text{date}(1, 4, \text{Year}) \\
\text{date}(\text{Day}, \text{Month}, 1985) &= \text{date}(1, 4, \text{Year}) \\
a(b, c, d(e, F, g(h, i, J))) &= a(B, c, d(E, f, g(H, i, j))) \\
[[\text{the, } Y]|Z] &= [[X, \text{dog}], [\text{is, here}]] \\
X &= Y + 1
\end{align*}
\]

- What about

\[
p(X) = p(q(X))
\]

produces a cyclic substitution
Application of a Substitution to a Term (1)

Notation
- Substitutions are denoted by greek letters: $\gamma$, $\sigma$, $\tau$, ...
  - $\tau = \{\text{Year} \leftarrow 1985, \text{Month} \leftarrow 4\}$

Definition: Application of a Substitution
Let $T$ be any term.
Let $\tau = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ be any substitution.
$T_{\tau}$, the application of $\tau$ to $T$, replaces (for $i = 1..n$) each occurrence of $V_i$ in $T$ by $t_i\tau$.

Examples
- $\text{date (Day, Month, 1985)}_{\tau} \equiv$
  - $x = y + 1_{\{x \leftarrow y + 1\}} \equiv$
  - $f(x, 1)_{\{y \leftarrow 2, x \leftarrow g(y)\}} \equiv$
Application of a Substitution to a Term (2)

Important

For $\tau = \{V_1 \leftarrow t_1, \ldots, V_n \leftarrow t_n\}$ and $i = 1..n$

$T\tau$ replaces all the occurrences of $V_i$ in $T$ by $t_i\tau$.

Substitutions are applied to their own right-hand-sides too!

Therefore:

$f(X, 1) \{Y \leftarrow 2, X \leftarrow g(Y)\} \equiv f(g(2), 1)$

This would be wrong:

$f(X, 1) \{Y \leftarrow 2, X \leftarrow g(Y)\} \equiv f(g(Y), 1)$

Resulting Problem

- Application of cyclic substitutions creates infinite terms

$p(X) \{X \leftarrow q(X)\} \equiv p(q(q(q(q(q(q(q(...)))))...)$

- Prevention: Don’t create cyclic substitutions in the first place!
  - “Occurs Check” verifies whether unification would create cyclic substitutions
“Occurs Check“ (1)

Theory
- Unification must fail if it would create substitutions with cyclic bindings

\[ p(X) = p(q(X)) \] \% must fail

Problem
- Unification with “occurs-check” has \textit{exponential} worst-case run-time
- Unification without “occurs-check” has \textit{linear} worst-case run-time

Practical Prolog implementations
- Prolog implementations do not perform the occurs check

\[ p(X) = p(q(X)) \] \% succeeds

- … unless you explicitly ask for it

\[
\text{unify_with_occurs_check}(p(X), p(q(X))) \] \% fails
Cyclic Substitutions

- Cyclic substitutions result from lack of „occurs“ check:

  ?- X=f(X).
  X = f(X).

- SWI-Prolog: Graceful printing of cyclic terms

  ?- X=f(X), write('This is what we get: '), write(X).
  'This is what we get: @(S_1,[S_1=f(S_1)])
  X = f(X).


- Other Prolog implementations: Infinite loop

  - X=f(X), write('This is what we get: '), write(X).
  'This is what we get: f(f(f(f(f(f(f(f(f(...

- SWI-Prolog has an occurs-check version of unification:

  ?- unify_with_occurs_check(X,f(X)).
  fail.
Unification

- Unification of terms T1 and T2
  - finds a substitution $\sigma$ for the variables of T1 and T2 such that ...
  - ... if $\sigma$ is applied to T1 and T2 then the results are equal

- Unification satisfies equations
  - ... but only if possible

Question

- How to unify two variables?
  - Problem: Infinitely many unifying substitutions possible!!!

Solution

- Unification finds the most general unifying substitution
  - “most general unifier” (mgu)

Example 1:

?- p(X, f(Y), a) = p(a, f(a), Y).
X = a, Y = a.

Example 2:

?- p(X, f(Y), a) = p(a, f(b), Y).
fail.

Example 3:

?- p(X, f(Y), a) = p(a, f(b), Y).
fail.

Example 4:

?- p(X) = p(Y).
X = a, Y = a;
X = b, Y = b;
...

Example 5:

?- p(X) = p(Z).
X = _G800, Z = _G800;
true.
Unification yields Most General Unifier (MGU)

- Unification of terms $T_1$ and $T_2$
  - finds a substitution $\sigma$ for the variables of $T_1$ and $T_2$ such that …
  - … if $\sigma$ is applied to $T_1$ and $T_2$ then the results are equal
  - if $\sigma$ is a most general substitution

Theorem (Uniqueness of MGU): The most general unifier of two terms $T_1$ and $T_2$ is uniquely determined, up to renaming of variables.

- If there are two different most general unifiers of $T_1$ and $T_2$, say $\sigma$ and $\tau$, then there is also a renaming substitution $\gamma$ such that $T_1\sigma\gamma \equiv T_2\tau$

- A renaming substitution only binds variables to variables
  \[ f(A) \{A\leftarrow B, B\leftarrow C\} \equiv f(C) \]
Computing the Most General Unifier
\texttt{mgu}(T_1, T_2)

- **Input:** two terms, \( T_1 \) and \( T_2 \)
- **Output:** \( \sigma \), the most general unifier of \( T_1 \) and \( T_2 \) 
  (only if \( T_1 \) and \( T_2 \) are unifiable)

- **Algorithm**

  1. If \( T_1 \) and \( T_2 \) are the same constant or variable then \( \sigma = \{ \} \)
  2. If \( T_1 \) is a variable not occurring in \( T_2 \) then \( \sigma = \{ T_1 \leftarrow T_2 \} \)
  3. If \( T_2 \) is a variable not occurring in \( T_1 \) then \( \sigma = \{ T_2 \leftarrow T_1 \} \)
  4. If \( T_1 = f(T_{11}, \ldots, T_{1n}) \) and \( T_2 = f(T_{21}, \ldots, T_{2n}) \) are function terms with the same functor and arity

     1. Determine \( \sigma_1 = \text{mgu}(T_{11}, T_{21}) \)
     2. Determine \( \sigma_2 = \text{mgu}(T_{12}\sigma_1, T_{22}\sigma_1) \)
     3. \ldots
     4. Determine \( \sigma_n = \text{mgu}(T_{1n}\sigma_1\ldots\sigma_{n-1}, T_{2n}\sigma_1\ldots\sigma_{n-1}) \)
     5. If all unifiers exist then \( \sigma = \sigma_1\ldots\sigma_{n-1}\sigma_n \)
        (otherwise \( T_1 \) and \( T_2 \) are not unifiable)
  6. Occurs check: If \( \sigma \) is cyclic fail, else return \( \sigma \)
  5. If none of the above applies, fail.
Composition Defined

Let $\sigma_1 = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ and $\sigma_2 = \{w_1 \leftarrow u_1, \ldots, w_m \leftarrow u_m\}$ be two substitutions.

- Then $\sigma_1 \sigma_2$ (called the composition of $\sigma_1$ and $\sigma_2$) is the substitution obtained from the set

\[
\{v_1 \leftarrow t_1 \sigma_2, \ldots, v_n \leftarrow t_n \sigma_2, w_1 \leftarrow u_1, \ldots, w_m \leftarrow u_m\}
\]

by deleting any trivial binding $v_i \leftarrow t_i \sigma_2$ for which $v_i = v_i$ and by deleting any conflicting binding $w_i \leftarrow u_i$ for which $w_i \in \{v_1, \ldots, v_n\}$.

- Operationally, we get the composition $\sigma_1 \sigma_2$ by
  - applying $\sigma_2$ to each right-hand-side of $\sigma_1$,
  - deleting trivial bindings resulting from the previous step and
  - appending non-conflicting bindings from $\sigma_2$.\]
Composition Comments

- Note the difference
  - $\sigma_1 \sigma_2$ is the composition of two substitutions
  - $t_1 \sigma_2$ is the application of a substitution to a term
Composition Example (1)

Let $\sigma_1 = \{ Y \leftarrow Z, \ X \leftarrow f(Y) \}$ and $\sigma_2 = \{ X \leftarrow a, \ Y \leftarrow b, \ Z \leftarrow Y \}$.

1. By applying $\sigma_2$ to the right-hand sides of $\sigma_1$ we get
   \[ \{ Y \leftarrow Y, \ X \leftarrow f(b) \} \]

2. By appending $\sigma_2$ we get
   \[ \{ Y \leftarrow Y, \ X \leftarrow f(b), \ X \leftarrow a, \ Y \leftarrow b, \ Z \leftarrow Y \} \]

3. By deleting the trivial binding $Y \leftarrow Y$ we get
   \[ \{ X \leftarrow f(b), \ X \leftarrow a, \ Y \leftarrow b, \ Z \leftarrow Y \} \]

4. By deleting the contradicting binding $X \leftarrow a$ we get
   \[ \{ X \leftarrow f(b), \ Y \leftarrow b, \ Z \leftarrow Y \} \]

Thus $\sigma_1 \sigma_2 = \{ X \leftarrow f(b), \ Y \leftarrow b, \ Z \leftarrow Y \}$
Composition Example (2)

Let $\sigma_1 = \{ Y \leftarrow 2, \ X \leftarrow g(Y, Z) \}$ and $\sigma_2 = \{ Y \leftarrow 3, \ Z \leftarrow 4 \}$.

1. By applying $\sigma_2$ to the right-hand sides of $\sigma_1$ we get
   \[
   \{ Y \leftarrow 2, \ X \leftarrow g(3, 4) \}
   \]

2. By appending $\sigma_2$ we get
   \[
   \{ Y \leftarrow 2, \ X \leftarrow g(3, 4), \ Y \leftarrow 3, \ Z \leftarrow 4 \}
   \]

3. By deleting the contradicting binding $Y \leftarrow 3$ we get
   \[
   \{ Y \leftarrow 2, \ X \leftarrow g(3, 4), \ Z \leftarrow 4 \}
   \]

Thus $\sigma_1 \sigma_2 = \{ Y \leftarrow 2, \ X \leftarrow g(3, 4), \ Z \leftarrow 4 \}$

Note that the conflicting binding $Y \leftarrow 3$ is not part of the result but has been applied nevertheless, yielding $X \leftarrow g(3, 4)$.
Unification Examples

?- food(bread,X,beer) = food(Y,sausage,X).
false.

Because \{X ← beer, X ← sausage\} is impossible.

?- food(bread,X,beer) = food(Y,kahuna_burger).
false.

Because we have 3 arguments on the left side but only 2 on the right

?- meal(food(bread),drink(beer)) = meal(X,Y).
X = food(bread),
Y = drink(beer).
Unification Examples (continued)

- \(-\) loves\( (X,X) = loves\( (marsellus,mia). \)
  
  false.

  Because \(\{X \leftarrow marsellus, X \leftarrow mia\}\) is impossible.

- \(-\) \(k(s(g),Y) = k(X,t(k)).\)
  
  \(X=s(g),\)
  
  \(Y=t(k).\)

- \(-\) \(k(s(X),t(Y,g)) = k(s(g),t(k,X)).\)
  
  \(X = g,\)
  
  \(Y = k.\)
Unification Examples (continued)

?- T1 = k(s(X),t(Y,g)),
   T2 = k(s(g),t(k,X)),
   writeln('Before unification: '),
   write('T1 = '), writeln(T1), write('T2 = '), writeln(T2),
   T1 = T2,
   write('After unification: '),
   write('T1 = '), writeln(T1), write('T2 = '), writeln(T2).

Before unification:
T1 = k(s(_G5975),t(_G5977,g))
T2 = k(s(g),t(k,_G5975))

After unification:
T1 = k(s(g),t(k,g))
T2 = k(s(g),t(k,g))
T1 = T2, T2 = k(s(g), t(k, g)),
X = g,
Y = k.

Output created by the writeln/1 and nl/0 predicate calls.

Output from the Prolog system:
All computed bindings.
Unification Examples (continued)

?- T1 = k(s(X), t(Y,g)),
    T2 = k(s(g), t(k,X)),
    writeln('Before unification: '),
    write('T1 = '), writeln(T1), write('T2 = '), writeln(T2),
    T1 = T2,
    write('After unification: '),
    write('T1 = '), writeln(T1), write('T2 = '), writeln(T2).

Before unification:
T1 = k(s(_G5975), t(_G5977, g))
T2 = k(s(g), t(k, _G5975))

After unification:
T1 = k(s(g), t(k, g))
T2 = k(s(g), t(k, g))
T1 = T2, T2 = k(s(g), t(k, g)),
X = g,
Y = k.
SLD-Resolution
Resolución

- Resolución Princípio

  La prueba del objetivo \( G \rightarrow P, L, Q \).

  Si existe una cláusula \( L_0 \rightarrow L_1, \ldots, L_n \) \((n \geq 0)\)

  tal que \( \sigma = \text{mgu}(L, L_0) \)

  puede ser reducida a provar \( \neg P, L \sigma, \ldots, L_n \sigma \) \( Q \sigma \).

- Informal “resolución algorítmica”

  Para probar el objetivo \( G \rightarrow P, L, Q \).

  Selecciona uno de los literales en \( G \), digamos \( L \).

  Selecciona una copia de una cláusula \( L_0 \rightarrow L_1, \ldots, L_n \) \((n \geq 0)\)

  tal que hay existencia \( \sigma = \text{mgu}(L, L_0) \)

  aplique \( \sigma \) al objetivo \( \neg P, L \sigma, Q \sigma \).

  Aplique \( \sigma \) a la cláusula

  reemplace \( L \sigma \) por el cuerpo de la cláusula

  \( \neg P \sigma, L_1 \sigma, \ldots, L_n \sigma \) \( Q \sigma \).
**Resolution**

- **Resolution Principle**

  The proof of the goal $G \vdash P, L, Q$.  
  If there exists a clause $L_0 \vdash L_1, \ldots, L_n$ ($n \geq 0$) such that $\sigma = \text{mgu}(L, L_0)$ can be reduced to proving $\vdash P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma$.

- **Graphical illustration of resolution by “derivation trees”**

  Initial goal $\rightarrow \vdash P, L, Q$.  
  Copy of clause with renamed variables (different from variables in goal!) $\rightarrow L_0 \vdash L_1, L_2, \ldots, L_n$.  
  Unifier of selected literal and clause head $\rightarrow \sigma = \text{mgu}(L, L_0)$.  
  Derived goal $\rightarrow \vdash P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma$. 
Resolution reduces goals to subgoals

For
we also say
or
and write

"Goal₂ results from Goal₁ by resolution"
"Goal₂ is derived from Goal₁"
"Goal₁ is reducible to Goal₂"
"Goal₁ |--Goal₂"

Goal / Goal₁  \[\rightarrow\]  ?- P, L, Q.

Derivation / reduction step  \[\rightarrow\]  
\[L₀ :- L₁, L₂, \ldots, Lₙ\]
\[σ = mgu(L, L₀)\]

Subgoal / Goal₂  \[\rightarrow\]  ?- Pσ, L₁σ, \ldots, Lₙσ, Qσ.
Resolution in Prolog versus Horn Clauses

Resolution of a subgoal literal with the head literal of a Prolog clause corresponds to resolution of a negative literal with the positive literal of a Horn Clause (two sections ago).

\[ \text{Goal} / \text{Goal}_1 \quad \rightarrow \quad \text{Derivation} / \text{reduction step} \quad \rightarrow \quad \text{Subgoal} / \text{Goal}_2 \]

\[ \begin{align*}
? - & \quad P, L, Q. \\
L_0 :& = L_1, L_2, \ldots, L_n \\
\sigma & = \text{mgu}(L, L_0) \\
? - & \quad P\sigma, L_1\sigma, \ldots, L_n\sigma, Q\sigma.
\end{align*} \]
# SLD Resolution

<table>
<thead>
<tr>
<th>General Resolution Principle</th>
<th>SLD Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unrestricted selection functions</td>
<td><strong>Constrained selection functions</strong></td>
</tr>
<tr>
<td>1. any literal from the goal</td>
<td>1. First literal from the goal</td>
</tr>
<tr>
<td>2. any clause from the program</td>
<td>2. First clause from the program</td>
</tr>
</tbody>
</table>

**Problem**

For a practical implementation we have to specify *which* literal / clause to select.

**Remember**

According to the way we write literals in goals and clauses in a program you can remember that SLD-resolution works

1. **Left-to-right** and
2. **Top-to-bottom**

---

Prolog uses SLD Resolution
SLD-Resolution Example: Program and Goal

- **Goal**

```?- isGrandmaOf(maria, Granddaughter).```

- **Program**

```isMotherOf(maria, klara).
isMotherOf(maria, paul).
isMotherOf(eva, anna).

isMarriedTo(paul, eva).

isGrandmaOf(G, E) :- isMotherOf(G, M), isMotherOf(M, E).
isGrandmaOf(G, E) :- isMotherOf(G, V), isFatherOf(V, E).
isFatherOf(V, K) :- isMarriedTo(V, M), isMotherOf(M, K).
...
```
?- isGrandmaOf(maria, Granddaughter).

isGrandmaOf(G1, E1) :- isMotherOf(G1, V1), isFatherOf(V1, E1).
σ₁ = {G1 ← maria, E₁ ← Granddaughter}

?- isMotherOf(maria, V1), isFatherOf(V1, Granddaughter).

isMotherOf(maria, paul).
σ₂ = {V₁ ← paul}

?- isFatherOf(paul, Granddaughter).

isFatherOf(V2, K2) :- isMarriedTo(V2, M2), isMotherOf(M2, K2).
σ₃ = {V₂ ← paul, K₂ ← Granddaughter}

?- isMarriedTo(paul, M2), isMotherOf(M2, Granddaughter).

isMarriedTo(paul, eva).
σ₄ = {M₂ ← eva}

?- isMotherOf(eva, Granddaughter).

isMotherOf(eva, anna).
σ₅ = {Granddaughter ← anna}
SLD-Resolution Example: Result?

?- isGrandmaOf(maria,Granddaughter).

\[ \sigma_1 = \{ G1 \leftarrow maria, E1 \leftarrow Granddaughter \} \]

\[ \sigma_2 = \{ V1 \leftarrow paul \} \]

\[ \sigma_3 = \{ V2 \leftarrow paul, K2 \leftarrow Granddaughter \} \]

\[ \sigma_4 = \{ M2 \leftarrow eva \} \]

\[ \sigma_5 = \{ Granddaughter \leftarrow anna \} \]

So what is the result?

\[ \Rightarrow \text{the last substitution?} \]

\[ \Rightarrow \text{the substitution(s) for the variable(s) of the goal?} \]
SLD-Resolution Example: Derivation with different variable bindings

?- isGrandmaOf(maria, Granddaughter).

isGrandmaOf(G1, E1) :- isMotherOf(G1, V1), isFatherOf(V1, E1).
\[ \sigma_1 = \{ G1 \leftarrow \text{maria}, \text{Granddaughter} \leftarrow E1 \} \]

?- isMotherOf(maria, V1), isFatherOf(V1, E1).

isMotherOf(maria, paul).
\[ \sigma_2 = \{ V1 \leftarrow \text{paul} \} \]

?- isFatherOf(paul, E1).

isFatherOf(V2, K2) :- isMarriedTo(V2, M2), isMotherOf(M2, K2).
\[ \sigma_3 = \{ V2 \leftarrow \text{paul}, E1 \leftarrow K2 \} \]

?- isMarriedTo(paul, M2), isMotherOf(M2, K2).

isMarriedTo(paul, eva).
\[ \sigma_4 = \{ M2 \leftarrow \text{eva} \} \]

?- isMotherOf(eva, K2).

isMotherOf(eva, anna).
\[ \sigma_5 = \{ K2 \leftarrow \text{anna} \} \]
SLD-Resolution Example: Result revisited

?- isGrandmaOf(maria, Granddaughter).

\[ \sigma_1 = \{G1 \leftarrow \text{maria}, \text{Granddaughter} \leftarrow E1\} \]

\[ \sigma_2 = \{V1 \leftarrow \text{paul}\} \]

\[ \sigma_3 = \{V2 \leftarrow \text{paul}, E1 \leftarrow K2\} \]

\[ \sigma_4 = \{M2 \leftarrow \text{eva}\} \]

\[ \sigma_5 = \{K2 \leftarrow \text{anna}\} \]

Observation

- The result is not
- the last substitution
- the substitution(s) for the variable(s) of the goal
- We need to „compose“ the substitutions!
The result is the *composition* of all substitutions computed along a derivation path ...

\[\sigma_1 = \{G1 \leftarrow \text{maria}, \text{Granddaughter} \leftarrow \text{E1}\}\]
\[\sigma_2 = \{V1 \leftarrow \text{paul}\}\]
\[\sigma_3 = \{V2 \leftarrow \text{paul}, \text{E1} \leftarrow \text{K2}\}\]
\[\sigma_4 = \{M2 \leftarrow \text{eva}\}\]
\[\sigma_5 = \{K2 \leftarrow \text{anna}\}\]

\[\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 = \{G1 \leftarrow \text{maria}, \text{Granddaughter} \leftarrow \text{E1}, V1 \leftarrow \text{paul}, V2 \leftarrow \text{paul}, \text{E1} \leftarrow \text{anna}, M2 \leftarrow \text{eva}, K2 \leftarrow \text{anna}\}\]

... restricted to the bindings for variables from the initial goal

\[\sigma = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \mid \text{Vars( isGrandmaOf(maria, Granddaughter) )}\]

\[= \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \mid \{ \text{Granddaughter} \}\]

\[= \{\text{Granddaughter} \leftarrow \text{anna}\}\]
The result is the **composition** of all substitutions computed along a derivation path …

\[
\sigma_1 = \{ G1 \leftarrow \text{maria}, \ E1 \leftarrow \text{Granddaughter} \} \\
\sigma_2 = \{ V1 \leftarrow \text{paul} \} \\
\sigma_3 = \{ V2 \leftarrow \text{paul}, \ K2 \leftarrow \text{Granddaughter} \} \\
\sigma_4 = \{ M2 \leftarrow \text{eva} \} \\
\sigma_5 = \{ \text{Granddaughter} \leftarrow \text{anna} \}
\]

\[
\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 = \{ G1 \leftarrow \text{maria}, \ E1 \leftarrow \text{anna}, \ V1 \leftarrow \text{paul}, \ V2 \leftarrow \text{paul}, \ K2 \leftarrow \text{anna}, \ M2 \leftarrow \text{eva}, \ \text{Granddaughter} \leftarrow \text{anna} \}
\]

… restricted to the bindings for variables from the initial goal

\[
\sigma = \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \mid \text{Vars( isGrandmaOf(\text{maria}, \text{Granddaughter}) )} \\
= \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \mid \{ \text{Granddaughter} \} \\
= \{ \text{Granddaughter} \leftarrow \text{anna} \}
\]

Wouldn’t that yield another result? Different bindings than on the previous page! Does that mean we get a different result?? No! Same result substitution as on the previous page!
Resolution Result Defined

Let $\sigma_1, \ldots, \sigma_n$ be the mgu's computed along a successful derivation path for the goal $G$ and let $\text{Vars}(G)$ be the set of variables in $G$.

- Then the result substitution is $\sigma_1 \ldots \sigma_n / \text{Vars}(G)$

- Informally: The result substitution for a successful derivation path (= a proof) of goal $G$ is obtained by
  a) Composing all substitutions computed during the proof of the goal
  b) …and restricting the composition result to the variables of the goal.
Restriction Defined

Let \( \sigma = \{ v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n \} \) be a substitution and \( V \) be a set of variables.

- Then \( \sigma | V = \{ v_i \leftarrow t_i \mid (v_i \in V) \land (v_i \leftarrow t_i \in \sigma) \} \)

- Terminology: \( \sigma | V \) is called the restriction of \( \sigma \) to \( V \)

- Informally: The restriction \( \sigma | V \) is obtained by eliminating from \( \sigma \) all bindings for variables that are not in \( V \)
Composition Defined (Repeated)

Let $\sigma_1 = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ and $\sigma_2 = \{w_1 \leftarrow u_1, \ldots, w_m \leftarrow u_m\}$ be two substitutions.

- Then $\sigma_1 \sigma_2$ (called the composition of $\sigma_1$ and $\sigma_2$) is the substitution obtained from the set

$$\{v_1 \leftarrow t_1 \sigma_2, \ldots, v_n \leftarrow t_n \sigma_2, w_1 \leftarrow u_1, \ldots, w_m \leftarrow u_m\}$$

by deleting any trivial binding $v_i \leftarrow t_i \sigma_2$ for which $v_i = t_i \sigma_2$ and by deleting any conflicting binding $w_i \leftarrow u_i$ for which $w_i \in \{v_1, \ldots, v_n\}$.

- Operationally, we get the composition $\sigma_1 \sigma_2$ by
  - applying $\sigma_2$ to each right-hand-side of $\sigma_1$,
  - deleting trivial bindings and
  - appending non-conflicting bindings from $\sigma_2$. 
Composition Defined (Repeated)

- Note the difference
  - $t_1 \sigma_2$ is the application of a substitution to a term
  - $\sigma_1 \sigma_2$ is the composition of two substitutions
SLD-Resolution with Backtracking

OK, we’ve seen how resolution finds one answer. But how to find more answers?

→ Backtracking!
Derivation with Backtracking

?- f(X), g(X), h(X).

#1, X=a  choicepoint: #2

?- g(a), h(a).

#1  choicepoint: #2

?- h(a).

The subgoal h(a) fails because there is no clause whose head unifies with it.

The interpreter backtracks to the last “choicepoint” for g(a)
Derivation with Backtracking

?- f(X), g(X), h(X).

#1, X=a choicepoint: #2

?- g(a), h(a).



The subgoal g(a) fails because there is no remaining clause (at the choicepoint or after it) whose head unifies with it.

The interpreter backtracks to the last “choicepoint” for f(X)
Derivation with Backtracking

\[ f(a). \quad g(a). \]
\[ \checkmark f(b). \quad \checkmark g(b). \quad \checkmark h(b). \]

\[ ?- f(Y), \quad g(Y), \quad h(Y). \]
\[ Y=b; \quad \text{no} \]

\[ ?- f(X), \quad g(X), \quad h(X). \]
\[ \#2, \quad X=b \quad \text{choicepoint: ---} \]

\[ ?- g(b), \quad h(b). \]
\[ \#2 \quad \text{choicepoint: ---} \]

\[ ?- h(b). \]
\[ \#1 \quad \text{choicepoint: ---} \]

\[ ?- \text{true.} \]

- The derivation is successful (it derived the subgoal “true”).
- The interpreter reports the successful substitutions
SLD-Resolution with Backtracking: Summary

- SLD-Resolution always selects the
  - the leftmost literal in a goal as a candidate for being resolved
  - the topmost clause of a predicate definition as a candidate for resolving the current goal

- If a clause’s head is not unifiable with the current goal the search proceeds immediately to the next clause

- If a clause’s head is unifiable with the current goal
  - the goal is resolved with that clause
  - the interpreter remembers the next clause as a choicepoint

- If no clause is found for a goal (= the goal fails), the interpreter undoes the current derivation up to the last choicepoint.

- Then the search for a candidate clause continues from that choicepoint
Box-Model of Backtracking

- **A goal** is a box with four ports: call, succeed, redo, fail

  - A conjunction is a chain of connected boxes
    - the “succeed” port is connected to the “call” port of the next goal
    - the “fail” port is connected to the “redo” port of the previous goal
**Box-Model of Backtracking**

- **Subgoals of a clause** are boxes nested within the clause box, with outer and inner ports of the same kind connected
  - clause’s call to first subgoal’s call
  - last subgoal’s succeed to clause’s succeed
  - clause’s redo to last subgoal’s redo
  - first subgoal’s fail to the fail of the clause
Viewing Backtracking in the Debugger (1)

?- gtrace, simplify_aexpr(a-a+b-b, Simple).

... for this goal.

... call the graphical tracer ...

variable bindings in selected stack frame

reference to next choice point

goals without choice points

goals with choice points

call of “built-in” predicate (has no choicepoint)

the only exception is “repeat”

source code view of goal associated to selected stack frame

No selected variable
Viewing Backtracking in the Debugger (2)

The debugger visualizes the port of the current goal according to the box model.
Recursion

- Prolog predicates may be defined recursively.

- A predicate is recursive if one or more rules in its definition refer to itself.

  \[
  \text{descendant}(C,X) :\!- \! \text{child}(C,X).
  \]

  \[
  \text{descendant}(C,X) :\!- \! \text{child}(C,D), \text{descendant}(D,X).
  \]

- What does the descendant/2 definition mean?
  1. \textit{if} \( C \) is a child of \( X \), \textit{then} \( C \) is a descendant of \( X \)
  2. \textit{if} \( C \) is a child of \( D \), \textit{and} \( D \) is a descendant of \( X \), \textit{then} \( C \) is a descendant of \( X \)
Recursion: Derivation Tree for “descend”

child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X,Y):- child(X,Y).
descend(X,Y):-
    child(X,Z), descend(Z,Y).

?- descend(martha, laura)
yes
Example: Derivation and Recursion

- A program (List membership: Arg1 is a member of the list Arg2)

  
  ```prolog
  member(X, [X|_]). % clause #1
  member(X, [_|R]):- member(X, R). % clause #2
  ```

- A query, its successful substitutions ...

  ```prolog
  ?- member(E, [a,b,c]).
  E = a ; E = b ; E = c ; fail.
  ```

- ... and its derivation tree

  ```prolog
  ?- member(E, [a,b,c])
  ```

  ![Derivation Tree Diagram]

  Alternative derivations for
  ?- member(E, [abc])
Recursion: Successor

Suppose we want to express that

- 0 is a numeral
- If X is a numeral, then succ(X) is a numeral

\[
\text{numeral}(0).
\]
\[
\text{numeral} (\text{succ}(X)) : - \text{numeral}(X).
\]

Let’s see how this behaves:

?- \text{numeral}(X).

\begin{verbatim}
X = 0 ;
X = \text{succ}(0) ;
X = \text{succ}(\text{succ}(0)) ;
X = \text{succ}(\text{succ}(\text{succ}(0))) ;
...
\end{verbatim}
# Two different ways to give meaning to logic programs

<table>
<thead>
<tr>
<th><strong>Operational Semantics</strong></th>
<th><strong>Declarative Semantics</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Proof-based approach</td>
<td>• Model-based semantics</td>
</tr>
<tr>
<td>◆ Algorithm to find a proof</td>
<td>◆ Mathematical structure</td>
</tr>
<tr>
<td>◆ Refutation proof using SLD resolution</td>
<td>◆ Herbrand interpretations and Herbrand models</td>
</tr>
<tr>
<td>◆ Basic step: Derivation</td>
<td>◆ Basic step: Entailment (Logical consequence)</td>
</tr>
<tr>
<td>• To prove a goal prove each of its subgoals</td>
<td>• A formula is true if it is a logical consequence of the program</td>
</tr>
<tr>
<td>• Algorithm = Logic + Control</td>
<td>• Algorithm = Logic + Control</td>
</tr>
<tr>
<td>◆ Logic = Clauses</td>
<td>◆ Logic = Clauses</td>
</tr>
<tr>
<td>◆ Control = Top-down resolution process</td>
<td>◆ Control = Bottom-up fixpoint iteration</td>
</tr>
</tbody>
</table>
Negation

OK, we’ve seen how to prove or conclude what is true. But what about negation?

- Closed world assumption
- Negation as failure
- “Unsafe negation” versus existential variables
Closed World Assumption

- We cannot prove that something is false.
- We can only show that we cannot prove its contrary.

```
isFatherOf(kurt,peter).
```

```
?- isFatherOf(adam,cain).
no. \rightarrow means: we cannot prove that “isFatherOf(adam,cain)” is true
```

- If we **assume** that everything that is true is entailed by the program, we may then **conclude** that what is not entailed / provable is not true.
- This **assumption** is known as the “**Closed World assumption**” (CWA)
- The **conclusion** is known as “**Negation by Failure**” (NF)

```
?- not( isFatherOf(adam,cain) ).
yes.
\rightarrow means: we conclude that “not(isFatherOf(adam,cain) )” is true because we cannot prove that “isFatherOf(adam,cain)” is true
```
Negation with Unbound Variables (1)

Deductive Databases / Declarative Semantics

isFatherOf(kurt,peter).

?- ∀X.isFatherOf(adam,X).
no.

?- ∀X.not(isFatherOf(adam,X)).
← unsafe, infinite result set!

● Deductive databases consider all variables to be universally quantified.
● However, the set of values for X for which isFatherOf/2 fails is infinite and unknown because it consists of everything that is not represented in the program.
● So it is impossible to list all these values!
● Therefore, the above negated query with universal quantification is unsafe.
Prolog treats free variables in negated goals as existentially quantified. So it does not need to list all possible values of X.

It shows that there is some value for which the goal $G$ fails, by showing that \( G \) does not succeed for any value

\[
\exists x. \neg G \iff \neg \forall x. G
\]

This is precisely negation by failure!
Negation with Unbound Variables (3)

Existential variables can also occur in clause bodies:

- The clause
  \[
  \text{single}(X) :\!\!\!:\!\!\!\!: \text{human}(X), \text{not}(\text{married}(X,Y)).
  \]
  \[
  \forall X. \, \text{human}(X) \land \text{not}(\exists Y. \, \text{married}(X,Y)) \rightarrow \text{single}(X)
  \]

- The clause means

Take care: Changing the order of negated subgoals changes the (operational) semantics of the clause:

- The clause
  \[
  \text{single}(X) :\!\!\!:\!\!\!\!: \text{not}(\text{married}(X,Y)), \text{human}(X).
  \]
  \[
  \text{single}(X) :\!\!\!:\!\!\!\!: \text{not}(\text{married}(X_1,Y)), \text{human}(X).
  \]

- The clause is the same as

- Both mean

  \[
  \forall X. \, \text{human}(X) \land \text{not}(\exists X_1. \, \exists Y. \, \text{married}(X_1,Y)) \rightarrow \text{single}(X).
  \]

Remember: Free variables in negated goals are existentially quantified.
- They do not “return bindings” outside of the scope of the negation.
- They are different from variables outside of the negation that accidentally have the same name.
Negation with Unbound Variables (4)

Explanations for the previous slide

- **The clause**
  - **means**
  - **because X is already bound by human(X) when the negation is entered.**

<table>
<thead>
<tr>
<th>single($X$) :- human($X$), not(married($X,Y$)).</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall X$. human($X$) $\land$ not($\exists Y$.married($X,Y$)) $\rightarrow$ single($X$)</td>
</tr>
</tbody>
</table>

- **The clause**
  - **is the same as**
  - **Both mean**

<table>
<thead>
<tr>
<th>single($X$) :- not(married($X,Y$)) human($X$).</th>
</tr>
</thead>
<tbody>
<tr>
<td>single($X$) :- not(married($X_1, Y$)), human($X$).</td>
</tr>
</tbody>
</table>

| $\forall X$. human($X$) $\land$ not($\exists X_1, \exists Y$.married($X_1, Y$)) $\rightarrow$ single($X$). |

- **because the yellow $X$ in the first clause is not bound when the negation is reached. So it is existentially quantified, whereas the blue $X$ is universally quantified. Thus both are actually different variables since the same variable cannot be quantified differently in the same scope (remember the “disjoint variables” normalization step).**
Eliminate accidentally equal names!

Remember: Free variables in negated goals are existentially quantified.

- They do not “return bindings” outside of the scope of the negation.
- They are different from variables outside of the negation that accidentally have the same name.

```
nestedneg1(Y) :- % INST
    q(Y), % -
    not( ( p(X,Y), % -
        not( ( f(X,Z), % +
            g(Z) % +
        )
    )
    ),
    q(X, Z) % +
    )
),
q(X) . % -

nestedneg1(Y) :- % INST
    q(Y), % -
    not( ( p(X,Y), % -
        not( ( f(X,Z), % +
            g(Z) % +
        )
    )
    ),
    q(X, Z1) % +
    )
),
q(X1) . % -
```

Eliminate accidentally equal names
A Test

• Predict what this program does!

\[
\begin{align*}
\text{f}(1,a). \\
f(2,b). \\
f(2,c). \\
f(4,c). \\
\text{q}(1). \\
\text{q}(2). \\
\text{q}(3).
\end{align*}
\]

\[
\begin{align*}
\text{negation}(X) & : - \\
& \quad \text{not} ( \\
& \quad \quad ( \text{f}(X,c), \\
& \quad \quad \quad \text{output}(X), \\
& \quad \quad \quad \text{g}(X) \\
& \quad \quad ) , \\
& \quad \quad \text{q}(X).
\end{align*}
\]

\[
\begin{align*}
\text{output}(X) & : - \\
& \quad \text{format} ('\text{Found f}(\sim\text{a},c) ', \ [X]). \\
\text{output}(X) & : - \\
& \quad \text{format} ('\text{but no g}(\sim\text{a})\sim\text{n}', \ [X]).
\end{align*}
\]

• This is what it does (try it out):

?- negation(X).
Found f(2,c) but no g(2)
Found f(4,c) but no g(4)
X=1 ;
X=2 ;
X=3 ;
fail.

• Homework:

If you don’t understand the result reread the slides about negation (and eventually also those about backtracking if you do not understand why output/1 has two clauses).

format/2 is a built-in predicate. It outputs its first argument to the console, replacing the control elements \sim a or \sim w (in the order of their appearance) by the values of the respective list elements from the second argument. \sim a stands for an element that must be an atom. \sim w stands for an arbitrary term. \sim n is a newline.
SLD-Resolution Example: Program and Goal

● Goal

?- isGrandmaOf(maria,Granddaughter).

● Program

isMotherOf(maria, klara).
isMotherOf(maria, paul).
isMotherOf(eva, anna).
isMarriedTo(paul, eva).

isGrandmaOf(G, E) :- isMotherOf(G, M), isMotherOf(M, E).
isGrandmaOf(G, E) :- isMotherOf(G, V), isFatherOf(V, E).
isFatherOf(V, K) :- isMarriedTo(V, M), isMotherOf(M,K).

...
Incompleteness of SLD-Resolution

Can we prove truth or falsity of every goal?

→ No, unfortunately!
Incompleteness of SLD-Resolution

- **Provability**
  - If a goal can be reduced to the empty subgoal then the goal is provable.

- **Undecidability**
  - There is no automated proof system that always answers yes if a goal is provable from the available clauses and answers no otherwise.
  - Prolog answers yes, no or does not terminate.
Incompleteness of SLD-Resolution

- The evaluation strategy of Prolog is **incomplete**.
  - Because of **non-terminating derivations**, Prolog sometimes only derives a subset of the **logical consequences** of a program.

- **Example**
  - $r$, $p$, and $q$ are logical consequences of this program

  $p : - q.$  \(\%1\)
  $q : - p.$  \(\%2\)
  $p : - r.$  \(\%3\)
  $r.$  \(\%4\)

  - However, Prolog’s evaluation strategy **cannot derive** them. It loops indefinitely:

  ?- $p.$
  |--- 1st clause
  |  $q$
  |--- 2nd clause
  |  $p$
  ... etc.
Practical Implications

- Need to understand both semantics
  - The model-based (declarative) semantics is the “reference”
    - We can apply bottom-up fixpoint iteration to understand the set of logical consequences of our programs
  - The proof-based (operational) semantics is the one Prolog uses to prove that a goal is among the logical consequences
    - SLD-derivations can get stuck in infinite loops, missing some correct results
- Need to understand when these semantics differ
  - When do Prolog programs fail to terminate?
    - Order of goals and clauses
    - Recursion and “growing” function terms
    - Recursion and loops in data
  - Which other problems could prevent the operational semantics match the declarative semantics?
    - The cut!
    - Non-logical features
    - …
General Principles

- Try to match both semantics!
  - Your programs will be more easy to understand and maintain

- Write programs with the model-based semantics in mind!
  - If they do not behave as intended change them so that they do!
Chapter Summary

- Proof-theoretic Semantics
  - Refutation Proofs
  - Resolution Principle
  - SLD-Resolution
  - Horn Clauses
  - Unification
  - Incompleteness / non-termination

- Negation as Failure
  - Closed World Assumption
  - Negation as Failure
  - Existential Variables