IN3070/4070 – Logic – Autumn 2020

Lecture 9: Logic Programming

Martin Giese

15th October 2019

Department of Informatics

University of Oslo

Today's Plan

- Motivation
- SLD Resolution
- Prolog
- Syntax
- Semantics
- Lists & Arithmetic
- Negation/Cut/If-then-else
- Summary

Outline

The First-Order Resolution Calculus

Definition 1.1 (First-Order Resolution Calculus).

$$\frac{}{C_1, \ldots, \{\}, \ldots, C_n} \text{ axiom}$$

$$\frac{C_1, \ldots, C_i \cup \{L_1\}, \ldots, C_j \cup \{L_2\}, \ldots, C_n, C_i\sigma \cup C_j\sigma}{C_1, \ldots, C_i \cup \{L_1\}, \ldots, C_j \cup \{L_2\}, \ldots, C_n} \text{ resolution}$$

with $\sigma(L_1) = \sigma(\overline{L_2})$

The First-Order Resolution Calculus

Definition 1.1 (First-Order Resolution Calculus).

$$\frac{}{C_1, \dots, \{\}, \dots, C_n} \textit{ axiom}$$

$$\frac{C_1, \dots, C_i \cup \{L_1\}, \dots, C_j \cup \{L_2\}, \dots, C_n, C_i\sigma \cup C_j\sigma}{C_1, \dots, C_i \cup \{L_1\}, \dots, C_j \cup \{L_2\}, \dots, C_n} \textit{ resolution}$$

with $\sigma(L_1) = \sigma(\overline{L_2})$

$$\frac{C_1, \dots, C_i \cup \{L_1, \dots, L_m\}, \dots, C_n, C_i\sigma \cup \{L_1\sigma\}}{C_1, \dots, C_i \cup \{L_1, \dots, L_m\}, \dots, C_n} \textit{ factorization}$$

with $\sigma(L_1) = \dots = \sigma(L_m)$

- a resolution proof for a set of clauses S is a derivation of S in the resolution calculus; the substitution σ is local for every rule application; variables in every clause C can be renamed

Logic Programming

- use restricted form of resolution for programming a computation
- program is expressed as a set of "Horn" clauses

Logic Programming

- use restricted form of resolution for programming a computation
- program is expressed as a set of "Horn" clauses

- given a query, "SLD resolution" is used to prove that the query is a logical consequence of the program
- unification is used to calculate a substitution of the variables in the given query

Logic Programming

- use restricted form of resolution for programming a computation
- program is expressed as a set of "Horn" clauses

- given a query, "SLD resolution" is used to prove that the query is a logical consequence of the program
- unification is used to calculate a substitution of the variables in the given query

- in imperative programming languages, computation is explicitly constructed by the programmer (using if-then-else, while, for, ...)
- in logic programming, the program is a declarative specification and the resolution inference engine provides an implicit control

Outline

Horn Clauses and Logic Programs

Definition 2.1 (Horn Clause).

A Horn clause is a clause that contains at most one positive literal (a positive literal is a non-negated literal). A definite clause is a Horn clause that contains a (single) positive literal.

Horn Clauses and Logic Programs

Definition 2.1 (Horn Clause).

A Horn clause is a clause that contains at most one positive literal (a positive literal is a non-negated literal). A definite clause is a Horn clause that contains a (single) positive literal.

Definition 2.2 (Logic Program).

A logic program consists of definite clauses of the form:

- *facts:* $\{A\}$ $\quad (A)$
- *rules:* $\{A, \neg B_1, ..., \neg B_n\}$ $\quad (A \leftarrow B_1 \wedge ... \wedge B_n)$

where $A, B_1, ..., B_n$ *are atomic formulae.*

Horn Clauses and Logic Programs

Definition 2.1 (Horn Clause).

A Horn clause is a clause that contains at most one positive literal (a positive literal is a non-negated literal). A definite clause is a Horn clause that contains a (single) positive literal.

Definition 2.2 (Logic Program).

A logic program consists of definite clauses of the form:

- *facts:* $\{A\}$ $\quad (A)$
- *rules:* $\{A, \neg B_1, \dots, \neg B_n\}$ $\quad (A \leftarrow B_1 \wedge \dots \wedge B_n)$

where $A, B_1, \dots, B_n$ *are atomic formulae.*

Definition 2.3 (Goal or Query).

A goal/query clause has the form $\{\neg B_1, \dots, \neg B_n\}$ *where* $B_1, \dots, B_n$ *are atomic formulae.*

SLD Resolution

SLD resolution (Selective Linear Definite clause resolution) is the inference rule used in logic programming

- it is a refinement of the general resolution rule
- it is sound and complete for Horn clauses

SLD Resolution

SLD resolution (Selective Linear Definite clause resolution) is the inference rule used in logic programming

- it is a refinement of the general resolution rule
- it is sound and complete for Horn clauses

Definition 2.4 (SLD Resolution).

$$\frac{}{C_1, \ldots, \{\}, \ldots, C_n} \; \textit{axiom}$$

$$\frac{C_1, \ldots, C_i \cup \{L_1\}, \ldots, C_j \cup \{L_2\}, \ldots, C_n, C_i\sigma \cup C_j\sigma}{C_1, \ldots, C_i \cup \{L_1\}, \ldots, C_j \cup \{L_2\}, \ldots, C_n} \; \textit{resolution}$$

with $\sigma(L_1) = \sigma(\overline{L_2})$

SLD Resolution

SLD resolution (Selective Linear Definite clause resolution) is the inference rule used in logic programming

- it is a refinement of the general resolution rule
- it is sound and complete for Horn clauses

Definition 2.4 (SLD Resolution).

$$\frac{}{C_1, \ldots, \{\}, \ldots, C_n} \; \textit{axiom}$$

$$\frac{C_1, \ldots, C_i \cup \{L_1\}, \ldots, C_j \cup \{L_2\}, \ldots, C_n, C_i\sigma \cup C_j\sigma}{C_1, \ldots, C_i \cup \{L_1\}, \ldots, C_j \cup \{L_2\}, \ldots, C_n} \; \textit{resolution}$$

with $\sigma(L_1) = \sigma(\overline{L_2})$

- *first step: 1st parent clause* $C_i \cup \{L_1\}$ *is the query clause*
  step $n \geq 2$*: 1st parent clauses* $C_i \cup \{L_1\}$ *is resolvent* $C_i\sigma \cup C_j\sigma$ *of step* $n-1$
- *2nd parent clauses* $C_j \cup \{L_2\}$ *is always a clause of the logic program*

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, \ldots\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', \ldots\}$, ... $\{A_n, \neg B_n, \neg B_n', \ldots\}$ be a logic program.

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$\{\neg Q_1, ..., \neg Q_i, ...\}$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$\{\neg Q_1, ..., \neg Q_i, ...\} \qquad \{A_j, \neg B_j, \neg B_j', ...\}$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$$
\begin{array}{lll}
\{\neg Q_1, ..., \neg Q_i, ...\} & \{A_j, \neg B_j, \neg B_j', ...\} & \text{with } \sigma(Q_i) = \sigma(Aj) \\
\quad\quad\downarrow & \swarrow & \\
\{\neg D_1, ..., \neg D_k, ...\} & &
\end{array}
$$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$$\{\neg Q_1, ..., \neg Q_i, ...\} \quad \{A_j, \neg B_j, \neg B_j', ...\} \quad \text{with } \sigma(Q_i) = \sigma(Aj)$$

$$\downarrow \qquad \swarrow$$

$$\{\neg D_1, ..., \neg D_k, ...\} \quad \{A_l, \neg B_l, \neg B_l', ...\}$$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$$
\begin{array}{lll}
\{\neg Q_1, ..., \neg Q_i, ...\} & \{A_j, \neg B_j, \neg B_j', ...\} & \text{with } \sigma(Q_i) = \sigma(Aj) \\
\quad\downarrow \qquad \swarrow & & \\
\{\neg D_1, ..., \neg D_k, ...\} & \{A_l, \neg B_l, \neg B_l', ...\} & \text{with } \sigma(D_k) = \sigma(A_l) \\
\quad\downarrow \qquad \swarrow & & \\
\{\neg E_1, ..., \neg E_m, ...\} & &
\end{array}
$$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and

$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$$
\begin{array}{lll}
\{\neg Q_1, ..., \neg Q_i, ...\} & \{A_j, \neg B_j, \neg B_j', ...\} & \text{with } \sigma(Q_i) = \sigma(Aj) \\
\quad\quad\downarrow & \swarrow & \\
\{\neg D_1, ..., \neg D_k, ...\} & \{A_l, \neg B_l, \neg B_l', ...\} & \text{with } \sigma(D_k) = \sigma(A_l) \\
\quad\quad\downarrow & \swarrow & \\
\{\neg E_1, ..., \neg E_m, ...\} & & \\
\quad\quad\vdots & & \\
\quad\{\neg F_1\} & &
\end{array}
$$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, ...\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', ...\}$, ... $\{A_n, \neg B_n, \neg B_n', ...\}$ be a logic program.

An SLD resolution derivation has the following form:

$$
\begin{array}{lll}
\{\neg Q_1, ..., \neg Q_i, ...\} & \{A_j, \neg B_j, \neg B_j', ...\} & \text{with } \sigma(Q_i) = \sigma(Aj) \\
\quad\quad\downarrow \quad\quad \swarrow & & \\
\{\neg D_1, ..., \neg D_k, ...\} & \{A_l, \neg B_l, \neg B_l', ...\} & \text{with } \sigma(D_k) = \sigma(A_l) \\
\quad\quad\downarrow \quad\quad \swarrow & & \\
\{\neg E_1, ..., \neg E_m, ...\} & & \\
\quad\quad\vdots & & \\
\quad\{\neg F_1\} \quad \{A_n\} & &
\end{array}
$$

An SLD Resolution Derivation

Let $\{\neg Q_1, \neg Q_2, \ldots\}$ be a query clause and
$\{A_1, \neg B_1, \neg B_1', \ldots\}$, … $\{A_n, \neg B_n, \neg B_n', \ldots\}$ be a logic program.

An SLD resolution derivation has the following form:

$$\{\neg Q_1, \ldots, \neg Q_i, \ldots\} \qquad \{A_j, \neg B_j, \neg B_j', \ldots\} \qquad \text{with } \sigma(Q_i) = \sigma(Aj)$$

$$\downarrow \qquad \swarrow$$

$$\{\neg D_1, \ldots, \neg D_k, \ldots\ldots\} \qquad \{A_l, \neg B_l, \neg B_l', \ldots\} \qquad \text{with } \sigma(D_k) = \sigma(A_l)$$

$$\downarrow \qquad \swarrow$$

$$\{\neg E_1, \ldots, \neg E_m, \ldots\ldots\ldots\}$$

$$\vdots$$

$$\{\neg F_1\} \qquad \{A_n\} \qquad \text{with } \sigma(F_1) = \sigma(A_n)$$

$$\downarrow \quad \swarrow$$

$$\{\,\}$$

Outline

- Motivation
- SLD Resolution
- Prolog
- Syntax
- Semantics
- Lists & Arithmetic
- Negation/Cut/If-then-else
- Summary

The Programming Language Prolog

- Prolog (Programming in Logic) is a declarative programming language invented in the early 1970s by A. Colmerauer, R. Kowalski, and P. Roussel

The Programming Language Prolog

- Prolog (Programming in Logic) is a declarative programming language invented in the early 1970s by A. Colmerauer, R. Kowalski, and P. Roussel

- declarative programming: specify the problem and let the computer solve it
- algorithm = logic + control [Kowalski 1979]

The Programming Language Prolog

- Prolog (Programming in Logic) is a declarative programming language invented in the early 1970s by A. Colmerauer, R. Kowalski, and P. Roussel

- declarative programming: specify the problem and let the computer solve it
- algorithm = logic + control [Kowalski 1979]

- A Prolog program is a logic program, i.e. a set of definite clauses
- the symbol ':-' is used to represent the implication '$\leftarrow$'

The Programming Language Prolog

- Prolog (Programming in Logic) is a declarative programming language invented in the early 1970s by A. Colmerauer, R. Kowalski, and P. Roussel

- declarative programming: specify the problem and let the computer solve it
- algorithm = logic + control [Kowalski 1979]

- A Prolog program is a logic program, i.e. a set of definite clauses
- the symbol ':-' is used to represent the implication '$\leftarrow$'

- A Prolog program is "executed" by the Prolog interpreter (control) that implements SLD resolution
- search strategy: choose leftmost literal in the first parent/goal clause (D_1) and choose second parent clause (D_2) from top to bottom among the program clauses

Prolog – An Example

- An example in Prolog (file `family.pl`)

```
male(thomas).                                  % these are facts
male(rolf).
```

Prolog – An Example

- An example in Prolog (file `family.pl`)

```
male(thomas).                                       % these are facts
male(rolf).
female(anna).
female(maria).
```

Prolog – An Example

- An example in Prolog (file `family.pl`)

```
male(thomas).                                % these are facts
male(rolf).
female(anna).
female(maria).
parent(thomas,anna).
parent(maria,anna).
parent(rolf,maria).
```

Prolog – An Example

- An example in Prolog (file family.pl)

```
male(thomas).                                    % these are facts
male(rolf).
female(anna).
female(maria).
parent(thomas,anna).
parent(maria,anna).
parent(rolf,maria).

father(X,Y) :- parent(X,Y), male(X).             % these are rules
mother(X,Y) :- parent(X,Y), female(X).
```

Prolog – An Example

- An example in Prolog (file family.pl)

```
male(thomas).                                    % these are facts
male(rolf).
female(anna).
female(maria).
parent(thomas,anna).
parent(maria,anna).
parent(rolf,maria).

father(X,Y) :- parent(X,Y), male(X).             % these are rules
mother(X,Y) :- parent(X,Y), female(X).

grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

Prolog – An Example

- An example in Prolog (file family.pl)

```
male(thomas).                                    % these are facts
male(rolf).
female(anna).
female(maria).
parent(thomas,anna).
parent(maria,anna).
parent(rolf,maria).

father(X,Y) :- parent(X,Y), male(X).             % these are rules
mother(X,Y) :- parent(X,Y), female(X).

grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

- start Prolog and type '[family].' to load the program
- Ctrl-C stops Prolog; 'halt.' exits Prolog

Prolog Queries – Examples

- `?- parent(maria,anna).`

Prolog Queries – Examples

- `?- parent(maria,anna).`
  `true.`

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  ```
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  ```

Prolog Queries – Examples

- `?- parent(maria,anna).`
  `true.`
  `?- parent(anna,maria).`
  `false.`
- `?- parent(X,anna).`
  `X = thomas  <press ';' for more solutions>`

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  false.
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  false.
  ```
- ```
  ?- father(X,Y).
  ```
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  false.
  ```
- ```
  ?- father(X,Y).
  X = thomas,
  Y = anna  <press ';' for more solutions>
  ```
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  false.
  ```
- ```
  ?- father(X,Y).
  X = thomas,
  Y = anna  <press ';' for more solutions>
  X = rolf,
  Y = maria.
  ```
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  false.
  ```
- ```
  ?- father(X,Y).
  X = thomas,
  Y = anna  <press ';' for more solutions>
  X = rolf,
  Y = maria.
  ```
- ```
  ?- grandfather(rolf,Y).
  ```

Prolog Queries – Examples

- ```
  ?- parent(maria,anna).
  true.
  ?- parent(anna,maria).
  false.
  ```
- ```
  ?- parent(X,anna).
  X = thomas  <press ';' for more solutions>
  X = maria  <press ';' for more solutions>
  false.
  ```
- ```
  ?- father(X,Y).
  X = thomas,
  Y = anna  <press ';' for more solutions>
  X = rolf,
  Y = maria.
  ```
- ```
  ?- grandfather(rolf,Y).
  Y = anna.
  ```

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

$\{\neg$`grandfather(rolf,Y)`$\}$

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

↓ ↙ with σ(V)=rolf, σ(U)=W

{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

↓ ↙ with σ(V)=rolf, σ(U)=W

{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)} {parent(rolf,maria)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

↓ ↙ with σ(V)=rolf, σ(U)=W

{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)} {parent(rolf,maria)}

↘ ↙ with σ(W)=maria

{¬male(rolf),¬parent(maria,Z)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

```
{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}
        ↓              ↙                     with σ(X)=rolf, σ(Y)=Z
{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}
        ↓                     ↙              with σ(V)=rolf, σ(U)=W
{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)} {parent(rolf,maria)}
          ↘                              ↙      with σ(W)=maria
          {¬male(rolf),¬parent(maria,Z)} {male(rolf)}
```

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

↓ ↙ with σ(V)=rolf, σ(U)=W

{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)} {parent(rolf,maria)}

↘ ↙ with σ(W)=maria

{¬male(rolf),¬parent(maria,Z)} {male(rolf)}

↘ ↙

{¬parent(maria,Z)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

↓ ↙ with σ(V)=rolf, σ(U)=W

{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)} {parent(rolf,maria)}

↘ ↙ with σ(W)=maria

{¬male(rolf),¬parent(maria,Z)} {male(rolf)}

↘ ↙

{¬parent(maria,Z)} {parent(maria,anna)}

SLD Resolution Derivation – Example

program clauses:

```
male(rolf).
parent(maria,anna).
parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

query: `?- grandfather(rolf,Y).`

{¬grandfather(rolf,Y)} {grandfather(X,Z),¬father(X,U),¬parent(U,Z)}

↓ ↙ with σ(X)=rolf, σ(Y)=Z

{¬father(rolf,U),¬parent(U,Z)} {father(V,W),¬parent(V,W),¬male(V)}

↓ ↙ with σ(V)=rolf, σ(U)=W

{¬parent(rolf,W),¬male(rolf),¬parent(W,Z)} {parent(rolf,maria)}

↘ ↙ with σ(W)=maria

{¬male(rolf),¬parent(maria,Z)} {male(rolf)}

↘ ↙

{¬parent(maria,Z)} {parent(maria,anna)}

↘ ↙ with σ(Z)=anna

Y = anna. (= σ(Y)= σ(Z)) { }

Outline

Terms and Predicates

Terms $\langle term \rangle$:

Terms and Predicates

Terms ⟨*term*⟩:

- constants ⟨*constant*⟩: start with lower case letters (e.g. `parent`, `anna`)

Terms and Predicates

Terms $\langle term \rangle$:

- constants $\langle constant \rangle$: start with lower case letters (e.g. `parent`, `anna`)
- numbers: like usual (e.g. `123`, `123.456`)

Terms and Predicates

Terms ⟨*term*⟩:

- constants ⟨*constant*⟩: start with lower case letters (e.g. `parent`, `anna`)
- numbers: like usual (e.g. `123`, `123.456`)
- variables: start with upper case letter or the underscore '`_`'
  (e.g. `X`, `Y`, `Number`, `List`, `_ABC`; '`_`' is anonymous variable)

Terms and Predicates

Terms ⟨*term*⟩:

- constants ⟨*constant*⟩: start with lower case letters (e.g. `parent`, `anna`)
- numbers: like usual (e.g. `123`, `123.456`)
- variables: start with upper case letter or the underscore '_'
  (e.g. `X`, `Y`, `Number`, `List`, `_ABC`; '_' is anonymous variable)
- structures: ⟨*constant*⟩ or ⟨*constant*⟩(*Term*1, ..., *TermN*)
  (e.g. `parent(maria,anna)`)

Terms and Predicates

Terms ⟨*term*⟩:

- constants ⟨*constant*⟩: start with lower case letters (e.g. `parent`, `anna`)
- numbers: like usual (e.g. `123`, `123.456`)
- variables: start with upper case letter or the underscore '_'
  (e.g. `X`, `Y`, `Number`, `List`, `_ABC`; '_' is anonymous variable)
- structures: ⟨*constant*⟩ or ⟨*constant*⟩($Term1, ..., TermN$)
  (e.g. `parent(maria,anna)`)

Predicates ⟨*predicate*⟩:

Terms and Predicates

Terms ⟨*term*⟩:

- constants ⟨*constant*⟩: start with lower case letters (e.g. `parent`, `anna`)
- numbers: like usual (e.g. `123`, `123.456`)
- variables: start with upper case letter or the underscore '_'
  (e.g. `X`, `Y`, `Number`, `List`, `_ABC`; '_' is anonymous variable)
- structures: ⟨*constant*⟩ or ⟨*constant*⟩($Term1, ..., TermN$)
  (e.g. `parent(maria,anna)`)

Predicates ⟨*predicate*⟩:

- ⟨*constant*⟩ or ⟨*constant*⟩($Term1, ..., TermN$)
  (e.g. `thomas`, `parent(maria,anna)`)

Facts, Rules, and Queries

A Prolog program consists of clauses; a clause is either a fact or a rule.
The user can query the Prolog program/database.

Facts, Rules, and Queries

A Prolog program consists of clauses; a clause is either a fact or a rule. The user can query the Prolog program/database.

Facts:

- ⟨*predicate*⟩`.` (observe the '.' at the end)
  (e.g. `male(rolf).` or `parent(maria,anna).`)

Facts, Rules, and Queries

A Prolog program consists of clauses; a clause is either a fact or a rule. The user can query the Prolog program/database.

Facts:

- ⟨*predicate*⟩`.` (observe the '.' at the end)
  (e.g. `male(rolf).` or `parent(maria,anna).`)

Rules:

- ⟨*predicate*⟩ `:-` ⟨*predicate1*⟩, ... , ⟨*predicateN*⟩`.`
  (e.g. `father(X,Y) :- parent(X,Y), male(X).`)

Facts, Rules, and Queries

A Prolog program consists of clauses; a clause is either a fact or a rule. The user can query the Prolog program/database.

Facts:

- ⟨*predicate*⟩`.` (observe the '.' at the end)
  (e.g. `male(rolf).` or `parent(maria,anna).`)

Rules:

- ⟨*predicate*⟩ `:-` ⟨*predicate*1⟩, ... , ⟨*predicateN*⟩`.`
  (e.g. `father(X,Y) :- parent(X,Y), male(X).`)
- rules have the form *Head* `:-` *Body*`.`
- '`:-`' can be read as '$\leftarrow$'; comma '`,`' in the body can be read as '$\wedge$'

Facts, Rules, and Queries

A Prolog program consists of clauses; a clause is either a fact or a rule. The user can query the Prolog program/database.

Facts:

- ⟨*predicate*⟩`.` (observe the '.' at the end)
  (e.g. `male(rolf).` or `parent(maria,anna).`)

Rules:

- ⟨*predicate*⟩ `:-` ⟨*predicate*1⟩, ... , ⟨*predicateN*⟩.
  (e.g. `father(X,Y) :- parent(X,Y), male(X).`)
- rules have the form *Head* `:-` *Body*.
- '`:-`' can be read as '$\leftarrow$'; comma '`,`' in the body can be read as '$\wedge$'

Query:

- ⟨*predicate*1⟩, ... , ⟨*predicateN*⟩.
  (e,g, `parent(maria,anna).` or `grandfather(rolf,Y).`)

Outline

Operational Semantics

- Prolog tries to prove the query using the facts and rules in its database
- it starts trying to fulfil/solve the predicates one after the other

Operational Semantics

- Prolog tries to prove the query using the facts and rules in its database
- it starts trying to fulfil/solve the predicates one after the other

- if an appropriate fact matches, then the predicate/goal succeeds
- if the head of a rule matches, then Prolog continues by trying to fulfil the predicates of the rule's body

Operational Semantics

- Prolog tries to prove the query using the facts and rules in its database
- it starts trying to fulfil/solve the predicates one after the other

- if an appropriate fact matches, then the predicate/goal succeeds
- if the head of a rule matches, then Prolog continues by trying to fulfil the predicates of the rule's body

- the database is searched top to bottom
- if more than one fact or head of a rule matches, then alternative options are considered if the search fails (via backtracking)

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

?- grandfather(X,anna).
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                                 -> parent(maria, anna)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                                 -> parent(maria, anna)
                  male(maria)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                              -> parent(maria, anna)
                  male(maria) -> fail
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                                -> parent(maria, anna)
                  male(maria) -> fail
                                -> parent(rolf, maria)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                                -> parent(maria, anna)
                  male(maria) -> fail
                                -> parent(rolf, maria)
                  male(rolf)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                                -> parent(maria, anna)
                  male(maria) -> fail
                                -> parent(rolf, maria)
                  male(rolf)
   parent(maria,anna)
```

Operational Semantics – Example

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
```

```
?- grandfather(X,anna).
-> father(X,Y) -> parent(X,Y) -> parent(thomas, anna)
                  male(thomas)
   parent(anna,anna) -> fail
                                -> parent(maria, anna)
                  male(maria) -> fail
                                -> parent(rolf, maria)
                  male(rolf)
   parent(maria,anna)
grandfather(rolf,anna) succeeds
X = rolf.
```

- variables are instantiated ("bound") during the unification of terms

Logical Semantics

The semantics of a program is specified by the following formula F.

Logical Semantics

The semantics of a program is specified by the following formula F.

```
fact_1.
...
fact_n.
head_1 :- body_1.
...
head_m :- body_m.
?- query.
```

Logical Semantics

The semantics of a program is specified by the following formula F.

```
fact_1.
...
fact_n.
head_1 :- body_1.
...
head_m :- body_m.
?- query.
```

```
( fact_1
∧ ...
∧ fact_n
∧ head_1 ← body_1
∧ ...
∧ head_m ← body_m )
→ query
```

Logical Semantics

The semantics of a program is specified by the following formula F.

```
fact_1.
...
fact_n.
head_1 :- body_1.
...
head_m :- body_m.
?- query.
```

$$
\begin{array}{l}
(\ \texttt{fact\_1} \\
\wedge\ \ldots \\
\wedge\ \texttt{fact\_n} \\
\wedge\ \texttt{head\_1} \leftarrow \texttt{body\_1} \\
\wedge\ \ldots \\
\wedge\ \texttt{head\_m} \leftarrow \texttt{body\_m}\ ) \\
\rightarrow\ \texttt{query}
\end{array}
$$

The query succeeds iff the Prolog program terminates and F is valid.

Logical Semantics

The semantics of a program is specified by the following formula F.

```
fact_1.                      ( fact_1
...                          ∧ ...
fact_n.                      ∧ fact_n
head_1 :- body_1.            ∧ head_1 ← body_1
...                          ∧ ...
head_m :- body_m.            ∧ head_m ← body_m )
?- query.                    → query
```

The `query` succeeds iff the Prolog program terminates and F is valid.

- variables are quantified in the following way:

  $$\forall X1, ..., Xn \; (\exists Y1, ... Yn \; \texttt{body\_i} \; \rightarrow \; \texttt{head\_i})$$

  for all variables $X1, ..., Xn$ occurring in `head_i` and all variables $Y1, ... Yn$ occurring in `body_i`

Logical Semantics

The semantics of a program is specified by the following formula F.

`fact_1.`	`( fact_1`
`...`	$\wedge$ `...`
`fact_n.`	$\wedge$ `fact_n`
`head_1 :- body_1.`	$\wedge$ `head_1` $\leftarrow$ `body_1`
`...`	$\wedge$ `...`
`head_m :- body_m.`	$\wedge$ `head_m` $\leftarrow$ `body_m )`
`?- query.`	$\rightarrow$ `query`

The `query` succeeds iff the Prolog program terminates and F is valid.

- variables are quantified in the following way:

  $$\forall X1, ..., Xn\ (\exists Y1, ... Yn\ \texttt{body\_i} \rightarrow \texttt{head\_i})$$

  for all variables $X1, ..., Xn$ occurring in `head_i` and all variables $Y1, ... Yn$ occurring in `body_i`
- inference engine is a theorem prover based on SLD resolution (only Horn clauses, depth-first search (incomplete!), no occurs-check (unsound!))

Outline

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

- Example: `[a,b,c,d,e]` can be represented, e.g., as

```
[a|[b, c, d, e]]
[a|[b|[c|[d|[e]]]]]
[a, b|[c, d, e]]
[a, b, c, d|[e]]
```

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

- Example: `[a,b,c,d,e]` can be represented, e.g., as

```
[a|[b, c, d, e]]
[a|[b|[c|[d|[e]]]]]
[a, b|[c, d, e]]
[a, b, c, d|[e]]
```

- `?- [H|T]=[a,b,c,d].`

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

- Example: `[a,b,c,d,e]` can be represented, e.g., as

```
[a|[b, c, d, e]]
[a|[b|[c|[d|[e]]]]]
[a, b|[c, d, e]]
[a, b, c, d|[e]]
```

-
```
?- [H|T]=[a,b,c,d].
H = a,
T = [b, c, d].
```

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

- Example: `[a,b,c,d,e]` can be represented, e.g., as

```
[a|[b, c, d, e]]
[a|[b|[c|[d|[e]]]]]
[a, b|[c, d, e]]
[a, b, c, d|[e]]
```

- ```
  ?- [H|T]=[a,b,c,d].
  H = a,
  T = [b, c, d].
  ```

  ```
  ?- [H1,H2|T]=[a,b,c,d].
  ```
  ```

Prolog Lists

Lists are terms that are represented in the following way:

`[<Head>|<Tail>]`

where `<Head>` is the first element and `<Tail>` is the rest of the list

- Example: `[a,b,c,d,e]` can be represented, e.g., as

```
[a|[b, c, d, e]]
[a|[b|[c|[d|[e]]]]]
[a, b|[c, d, e]]
[a, b, c, d|[e]]
```

-
```
?- [H|T]=[a,b,c,d].
H = a,
T = [b, c, d].

?- [H1,H2|T]=[a,b,c,d].
H1 = a,
H2 = b,
T = [c, d].
```

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
- `?- member(a,[a,b,c]).`

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
-
```
?- member(a,[a,b,c]).
true .
?- member(X,[a,b]).
```

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
-
```
?- member(a,[a,b,c]).
true .
?- member(X,[a,b]).
X = a ;
X = b .
```

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
-
```
?- member(a,[a,b,c]).
true .
?- member(X,[a,b]).
X = a ;
X = b .

?- append([a,b],[c],Z).
```

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
-
```
?- member(a,[a,b,c]).
true .
?- member(X,[a,b]).
X = a ;
X = b .
?- append([a,b],[c],Z).
Z = [a, b, c].
?- append(X,Y,[a,b,c]).
```

Predefined Predicates on Lists

- `member(Element,List)` succeeds iff `Element` occurs in `List`
- `append(List1,List2,List3)` succeeds iff appending `List1` and `List2` results in `List3`
- `length(List,Length)` succeeds iff `List` has length/size `Length`
-
```
?- member(a,[a,b,c]).
true .
?- member(X,[a,b]).
X = a ;
X = b .

?- append([a,b],[c],Z).
Z = [a, b, c].
?- append(X,Y,[a,b,c]).
X = [], Y = [a, b, c] ;
X = [a], Y = [b, c] ;
X = [a, b], Y = [c] ;
X = [a, b, c], Y = [] .
```

Lists – Examples

- delete all identical elements from list

```
delete([],_,[]).
delete([X1|T],X,L) :- X==X1, delete(T,X,L).
delete([X1|T],X,[X1|L]) :- X\==X1, delete(T,X,L).
```

'=='-operator succeeds if both sides are identical without unification)

Lists – Examples

- delete all identical elements from list

```
delete([],_,[]).
delete([X1|T],X,L) :- X==X1, delete(T,X,L).
delete([X1|T],X,[X1|L]) :- X\==X1, delete(T,X,L).
```

'=='-operator succeeds if both sides are identical without unification)

- reverse list

```
reverse([],[]).
reverse([H|T],L):- reverse(T,R), append(R,[H],L).
```

Lists – Examples

- delete all identical elements from list

```
delete([],_,[]).
delete([X1|T],X,L) :- X==X1, delete(T,X,L).
delete([X1|T],X,[X1|L]) :- X\==X1, delete(T,X,L).
```

'=='-operator succeeds if both sides are identical without unification)

- reverse list

```
reverse([],[]).
reverse([H|T],L):- reverse(T,R), append(R,[H],L).

?- reverse([o,l,l,e,h],L).
```

Lists – Examples

- delete all identical elements from list

```
delete([],_,[]).
delete([X1|T],X,L) :- X==X1, delete(T,X,L).
delete([X1|T],X,[X1|L]) :- X\==X1, delete(T,X,L).
```

'=='-operator succeeds if both sides are identical without unification)

- reverse list

```
reverse([],[]).
reverse([H|T],L):- reverse(T,R), append(R,[H],L).

?- reverse([o,l,l,e,h],L).
L = [h,e,l,l,o].
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

  ```
  ?- X=3+5, X=Y+Z.
  ```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted
```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```
- to evaluate an arithmetic term the (predefined) 'is' predicate is used

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted
  ```
  ?- X=3+5, X=Y+Z.
  X = 3+5, Y = 3, Z = 5.
  ```
- to evaluate an arithmetic term the (predefined) '`is`' predicate is used
  ```
  ?- X is 3+5.
  ```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) ‘is’ predicate is used

```
?- X is 3+5.
X = 8.
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) '`is`' predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) 'is' predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

```
?- 8 is X+5.
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) 'is' predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

```
?- 8 is X+5.
uncaught exception: error(instantiation_error,(is)/2)
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted
  ```
  ?- X=3+5, X=Y+Z.
  X = 3+5, Y = 3, Z = 5.
  ```
- to evaluate an arithmetic term the (predefined) ‘is’ predicate is used
  ```
  ?- X is 3+5.
  X = 8.
  ```
- The term has to be fully instantiated:
  ```
  ?- 8 is X+5.
  uncaught exception: error(instantiation_error,(is)/2)
  ```
- arithmetic operators ‘=’, ‘<’, ‘>’, ‘>=’, ‘=<’ are interpreted predicates

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) ‘is’ predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

```
?- 8 is X+5.
uncaught exception: error(instantiation_error,(is)/2)
```

- arithmetic operators ‘=’, ‘<’, ‘>’, ‘>=’, ‘=<’ are interpreted predicates
- $0! = 1$, $n! = n * (n-1)!$ if $n > 0$:

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) 'is' predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

```
?- 8 is X+5.
uncaught exception: error(instantiation_error,(is)/2)
```

- arithmetic operators '=', '<', '>', '>=', '=<' are interpreted predicates
- $0! = 1$, $n! = n * (n-1)!$ if $n>0$:

```
factorial(0,1).
factorial(N,I) :- N>0, N1 is N-1,
                  factorial(N1,I1), I is N*I1.
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) 'is' predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

```
?- 8 is X+5.
uncaught exception: error(instantiation_error,(is)/2)
```

- arithmetic operators '=', '<', '>', '>=', '=<' are interpreted predicates
- $0! = 1$, $n! = n * (n-1)!$ if $n>0$:

```
factorial(0,1).
factorial(N,I) :- N>0, N1 is N-1,
                  factorial(N1,I1), I is N*I1.

?- factorial(5,I).
```

Arithmetic Operations

- numbers and terms with arithmetic operators are not interpreted

```
?- X=3+5, X=Y+Z.
X = 3+5, Y = 3, Z = 5.
```

- to evaluate an arithmetic term the (predefined) 'is' predicate is used

```
?- X is 3+5.
X = 8.
```

- The term has to be fully instantiated:

```
?- 8 is X+5.
uncaught exception: error(instantiation_error,(is)/2)
```

- arithmetic operators '=', '<', '>', '>=', '=<' are interpreted predicates
- $0! = 1$, $n! = n * (n-1)!$ if $n > 0$:

```
factorial(0,1).
factorial(N,I) :- N>0, N1 is N-1,
                  factorial(N1,I1), I is N*I1.

?- factorial(5,I).
N = 120.
```

Example: Ordered Lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Example: Ordered Lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Queries:

```
| ?- ordered([3,4,67,8]).
no
```

Example: Ordered Lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Queries:

```
| ?- ordered([3,4,67,8]).
no
```

```
| ?- ordered([3,4,67, 88]).
yes
```

Example: Ordered Lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Queries:

```
| ?- ordered([3,4,67,8]).
no

| ?- ordered([3,4,67, 88]).
yes

| ? - ordered([3,4,X,88]).
instantiation error: 4=<_30 - arg 2
```

Example: Ordered Lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

Queries:

```
| ?- ordered([3,4,67,8]).
no

| ?- ordered([3,4,67, 88]).
yes

| ? - ordered([3,4,X,88]).
instantiation error: 4=<_30 - arg 2
```

Comparison only works if variables are instantiated to numbers.

Example: Length of Lists

- An intuitive definition:

```
length([],0).
length([_ | Ts], N+1) :- length(Ts,N).
```

Example: Length of Lists

- An intuitive definition:

```
length([],0).
length([_ | Ts], N+1) :- length(Ts,N).
```

- Let's try it:

```
| ?- length([3,5,56,7],X).
X = 0+1+1+1+1
Yes
```

Example: Length of Lists

- An intuitive definition:

```
length([],0).
length([_ | Ts], N+1) :- length(Ts,N).
```

- Let's try it:

```
| ?- length([3,5,56,7],X).
X = 0+1+1+1+1
Yes
```

- Correct definition

```
length([],0).
length([_ | Ts], N) :- length(Ts,M), N is M+1.
```

Example: Length of Lists

- An intuitive definition:

```
length([],0).
length([_ | Ts], N+1) :- length(Ts,N).
```

- Let's try it:

```
| ?- length([3,5,56,7],X).
X = 0+1+1+1+1
Yes
```

- Correct definition

```
length([],0).
length([_ | Ts], N) :- length(Ts,M), N is M+1.
```

- Let's try it:

```
| ?- length([3,5,56,7],X).
X = 4
```

Outline

Negation as Failure

- negation '\+' is implemented as "negation as failure"

Negation as Failure

- negation ‘`\+`’ is implemented as “negation as failure”
- ‘`\+ predicate`’ succeeds iff ‘`predicate`’ fails

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
- ```
  male(thomas). male(rolf). female(anna). female(maria).
  parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
  father(X,Y) :- parent(X,Y), male(X).
  mother(X,Y) :- parent(X,Y), female(X).
  grandfather(X,Z) :- father(X,Y), parent(Y,Z).
  ```
  ```

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
-
```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

?- female(kristine).
```

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
- ```
  male(thomas). male(rolf). female(anna). female(maria).
  parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
  father(X,Y) :- parent(X,Y), male(X).
  mother(X,Y) :- parent(X,Y), female(X).
  grandfather(X,Z) :- father(X,Y), parent(Y,Z).

  ?- female(kristine).
  false.
  ```
  ```

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
-
```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

?- female(kristine).
false.

?- \+ female(kristine).
```

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
-
```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

?- female(kristine).
false.

?- \+ female(kristine).
true.
```

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
-

```
male(thomas). male(rolf). female(anna). female(maria).
parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
grandfather(X,Z) :- father(X,Y), parent(Y,Z).

?- female(kristine).
false.

?- \+ female(kristine).
true.

?- \+ parent(rolf,thomas).
```

Negation as Failure

- negation '\+' is implemented as "negation as failure"
- '`\+ predicate`' succeeds iff '`predicate`' fails
- ```
  male(thomas). male(rolf). female(anna). female(maria).
  parent(thomas,anna). parent(maria,anna). parent(rolf,maria).
  father(X,Y) :- parent(X,Y), male(X).
  mother(X,Y) :- parent(X,Y), female(X).
  grandfather(X,Z) :- father(X,Y), parent(Y,Z).

  ?- female(kristine).
  false.

  ?- \+ female(kristine).
  true.

  ?- \+ parent(rolf,thomas).
  true.
  ```
  ```

Non-monotonic Logics

- Standard "classical" propositional and first-order logic is monotonic.

Non-monotonic Logics

- Standard "classical" propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$

Non-monotonic Logics

- Standard "classical" propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$
- Adding facts will never remove logical consequences

Non-monotonic Logics

- Standard "classical" propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$
- Adding facts will never remove logical consequences
- In a semantics with negation as failure,

$$p, q \models \neg r$$

since r cannot be derived from $\{p, q\}$

Non-monotonic Logics

- Standard “classical” propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$
- Adding facts will never remove logical consequences
- In a semantics with negation as failure,

  $$p, q \models \neg r$$

  since r cannot be derived from $\{p, q\}$
- This is what Prolog does.

Non-monotonic Logics

- Standard “classical” propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$
- Adding facts will never remove logical consequences
- In a semantics with negation as failure,

  $$p, q \models \neg r$$

  since r cannot be derived from $\{p, q\}$
- This is what Prolog does.
- Now add the fact r:

  $$p, q, r \not\models \neg r$$

  since r *can* be derived from $\{p, q, r\}$

Non-monotonic Logics

- Standard "classical" propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$
- Adding facts will never remove logical consequences
- In a semantics with negation as failure,

  $$p, q \models \neg r$$

  since r cannot be derived from $\{p, q\}$
- This is what Prolog does.
- Now add the fact r:

  $$p, q, r \not\models \neg r$$

  since r *can* be derived from $\{p, q, r\}$
- Negation as Failure gives a non-monotonic logic

Non-monotonic Logics

- Standard "classical" propositional and first-order logic is monotonic.
- If $A \subseteq A'$ and $A \models B$, then $A' \models B$
- Adding facts will never remove logical consequences
- In a semantics with negation as failure,

  $$p, q \models \neg r$$

  since r cannot be derived from $\{p, q\}$
- This is what Prolog does.
- Now add the fact r:

  $$p, q, r \not\models \neg r$$

  since r *can* be derived from $\{p, q, r\}$
- Negation as Failure gives a non-monotonic logic
- Very different from our classical notion of logical consequence

The Cut

- the cut '`!`' is used to restrict Prolog's backtracking mechanism

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

- "red cut": does change returned solutions

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

- "red cut": does change returned solutions

```
parent(thomas,anna) :- !.
parent(maria,anna). parent(rolf,maria).
```

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

- "red cut": does change returned solutions

```
parent(thomas,anna) :- !.
parent(maria,anna). parent(rolf,maria).

?- parent(X,anna).
```

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

- "red cut": does change returned solutions

```
parent(thomas,anna) :- !.
parent(maria,anna). parent(rolf,maria).
```

```
?- parent(X,anna).
X = thomas.
```

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

- "red cut": does change returned solutions

```
parent(thomas,anna) :- !.
parent(maria,anna). parent(rolf,maria).
```

```
?- parent(X,anna).
X = thomas.
? grandfather(X,anna).
```

The Cut

- the cut '!' is used to restrict Prolog's backtracking mechanism
- the cut is a predefined predicate that succeeds when it is encountered for the first time; any attempt to re-fulfil it results in the failure of the calling (head) predicate
- "green cut": does not change solutions, only affects efficiency

```
factorial(0,1) :- !.
factorial(I,N) :- I>0,I1 is I-1,factorial(I1,N1),N is I*N1.
```

- "red cut": does change returned solutions

```
parent(thomas,anna) :- !.
parent(maria,anna). parent(rolf,maria).
```

```
?- parent(X,anna).
X = thomas.
? grandfather(X,anna).
false.
```

Example: Siblings

Disjunction and If-then-else

- `predicate :- predicate1 ; predicate2.`
  succeeds if `predicate1` succeeds or `predicate2` succeeds; backtracking over `predicate1` and `predicate2` when re-fulfilling `predicate`

Disjunction and If-then-else

- `predicate :- predicate1 ; predicate2.`
  succeeds if `predicate1` succeeds or `predicate2` succeeds; backtracking over `predicate1` and `predicate2` when re-fulfilling `predicate`

  `grandparent(X,Y) :- grandfather(X,Y) ; grandmother(X,Y).`

  (backtracking over `grandfather(X,Y)` `grandmother(X,Y)`)

Disjunction and If-then-else

- `predicate :- predicate1 ; predicate2.`
  succeeds if `predicate1` succeeds or `predicate2` succeeds; backtracking over `predicate1` and `predicate2` when re-fulfilling `predicate`

  `grandparent(X,Y) :- grandfather(X,Y) ; grandmother(X,Y).`

  (backtracking over `grandfather(X,Y)` `grandmother(X,Y)`)

- `Cond -> Goal1 ; Goal2` succeeds iff Cond succeeds and Goal1 succeeds or Cond fails and Goal2 succeeds; no backtracking within Cond ("implicit cut")

Disjunction and If-then-else

- `predicate :- predicate1 ; predicate2.`
  succeeds if `predicate1` succeeds or `predicate2` succeeds; backtracking over `predicate1` and `predicate2` when re-fulfilling `predicate`

  ```
  grandparent(X,Y) :- grandfather(X,Y) ; grandmother(X,Y).
  ```

  (backtracking over `grandfather(X,Y)` `grandmother(X,Y)`)

- `Cond -> Goal1 ; Goal2` succeeds iff Cond succeeds and Goal1 succeeds or Cond fails and Goal2 succeeds; no backtracking within Cond ("implicit cut")

  ```
  grandparent(X,Y) :-
      male(X) -> grandfather(X,Y) ; grandmother(X,Y).
  ```

  (information given by `male(X)` needs to be complete)

Disjunction and If-then-else

- `predicate :- predicate1 ; predicate2.`
  succeeds if `predicate1` succeeds or `predicate2` succeeds; backtracking over `predicate1` and `predicate2` when re-fulfilling `predicate`

  ```
  grandparent(X,Y) :- grandfather(X,Y) ; grandmother(X,Y).
  ```

  (backtracking over `grandfather(X,Y)` `grandmother(X,Y)`)

- `Cond -> Goal1 ; Goal2` succeeds iff Cond succeeds and Goal1 succeeds or Cond fails and Goal2 succeeds; no backtracking within Cond ("implicit cut")

  ```
  grandparent(X,Y) :-
      male(X) -> grandfather(X,Y) ; grandmother(X,Y).
  ```

  (information given by `male(X)` needs to be complete)

  ```
  grandparent(X,Y) :-
      grandfather(X,Y) -> true ; grandmother(X,Y).
  ```

  (no backtracking over `grandfather`)

Problems with Prolog

- No type system
- No standardized module system
- Non-declarative arithmetic
- Cut needed for efficiency
  - Cut has non-declarative semantics
  - Cut can simulate negation as failure (non-monotonic)
  - Cut can be tricky to use
  - Cut makes automated optimization hard
- IO does not play nice with backtracking

Prolog-like Languages

- Mercury

Prolog-like Languages

- Mercury
  - 'Pure' language with type system

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,...

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,. . .
  - Steep learning curve

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables

Prolog-like Languages

- Mercury
  - ‘Pure’ language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`

Prolog-like Languages

- Mercury
  - ‘Pure’ language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`
  - Applications in planning, scheduling, etc.

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`
  - Applications in planning, scheduling, etc.
- Higher-order logic programming, Lambda prolog

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`
  - Applications in planning, scheduling, etc.
- Higher-order logic programming, Lambda prolog
  - Like Prolog, but λ-terms instead of first-order

Prolog-like Languages

- Mercury
  - ‘Pure’ language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`
  - Applications in planning, scheduling, etc.
- Higher-order logic programming, Lambda prolog
  - Like Prolog, but λ-terms instead of first-order
  - Higher-order unification

Prolog-like Languages

- Mercury
  - 'Pure' language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`
  - Applications in planning, scheduling, etc.
- Higher-order logic programming, Lambda prolog
  - Like Prolog, but λ-terms instead of first-order
  - Higher-order unification
  - *not* a functional language, lambda terms are just data

Prolog-like Languages

- Mercury
  - ‘Pure’ language with type system
  - No cut, functional features (syntax), monad-style IO,...
  - Steep learning curve
- Constraint logic programming
  - Gathers and solves constraints on variables
  - From `X>3, X<6, X\==5` infer `X=4`
  - Applications in planning, scheduling, etc.
- Higher-order logic programming, Lambda prolog
  - Like Prolog, but λ-terms instead of first-order
  - Higher-order unification
  - *not* a functional language, lambda terms are just data
  - Can be handy to implement theorem provers

Outline

Summary

- logic program consists of definite clauses (facts and rules)
- SLD resolution is a sound and complete strategy for Horn clauses

Summary

- logic program consists of definite clauses (facts and rules)
- SLD resolution is a sound and complete strategy for Horn clauses

- Prolog is a declarative programming language
- clear and simple semantics based on first-order logic
- Turing-complete (can simulate a Turing machine)

Summary

- logic program consists of definite clauses (facts and rules)
- SLD resolution is a sound and complete strategy for Horn clauses

- Prolog is a declarative programming language
- clear and simple semantics based on first-order logic
- Turing-complete (can simulate a Turing machine)

- Prolog is used for, e.g, theorem proving, expert systems, term rewriting, automated planning, and natural language processing
- has given rise to a number of other languages
- Prolog is used in, e.g.,
  - IBM Watson (natural language question answering system)
  - Tivoli software (system and service management tools)

Summary

- logic program consists of definite clauses (facts and rules)
- SLD resolution is a sound and complete strategy for Horn clauses

- Prolog is a declarative programming language
- clear and simple semantics based on first-order logic
- Turing-complete (can simulate a Turing machine)

- Prolog is used for, e.g, theorem proving, expert systems, term rewriting, automated planning, and natural language processing
- has given rise to a number of other languages
- Prolog is used in, e.g.,
  - IBM Watson (natural language question answering system)
  - Tivoli software (system and service management tools)

- next week: DPLL (efficient SAT solving)